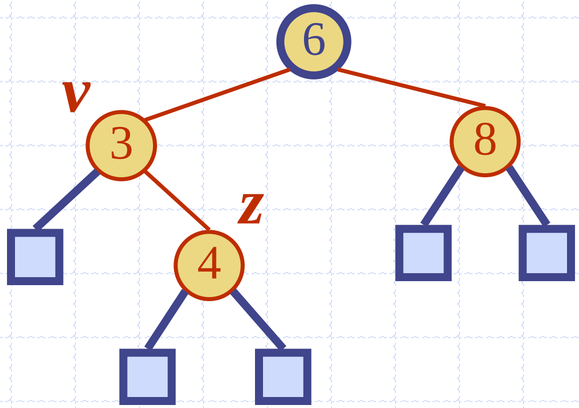
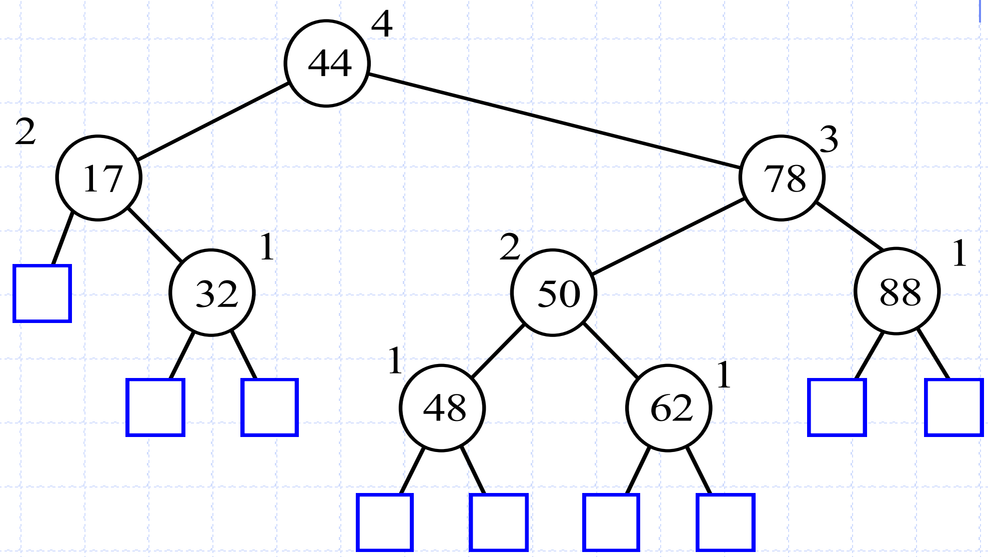
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

AVL Trees



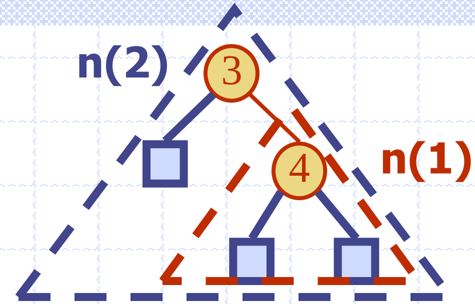
AVL Tree Definition

- **Adelson-Velsky and Landis**
- binary search tree
- balanced
  - each internal node v
    - 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

Height of an AVL Tree



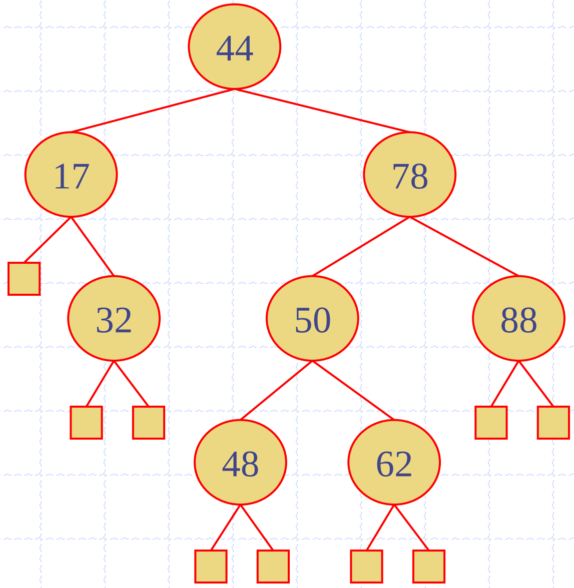
Fact: The height of an AVL tree storing n keys is O(log n).

Proof (by induction): n(h): the minimum number of internal nodes of an AVL tree of height h.

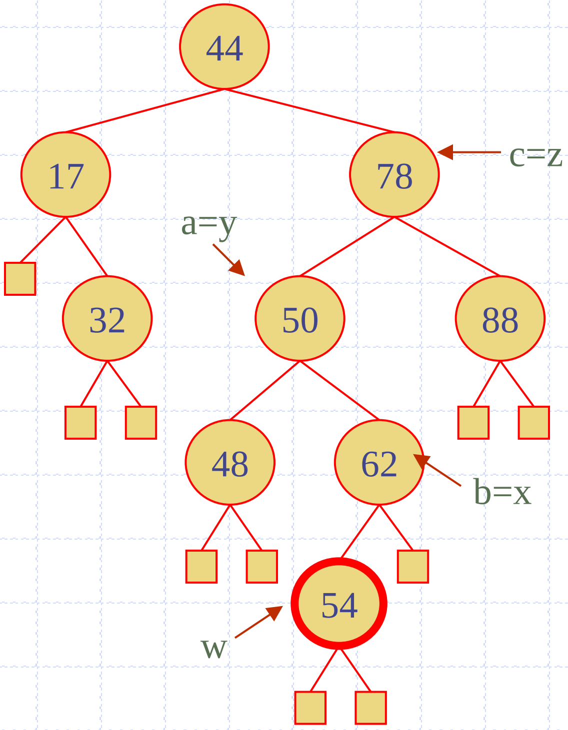
- $n(1) = 1$ and $n(2) = 2$
- For $n > 2$, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another of height n-2.
- That is, $n(h) = 1 + n(h-1) + n(h-2)$
- Knowing $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$. So
  $n(h) > 2n(h-2)$, $n(h) > 4n(h-4)$, $n(h) > 8n(n-6)$, … (by induction),
  $n(h) > 2^i n(h-2i)$
- Solving the base case we get: $n(h) > 2^{h/2 - 1}$
- Taking logarithms: $h < 2\log n(h) + 2$
- Thus the height of an AVL tree is O(log n)

Insertion

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Insert 54:



before insertion

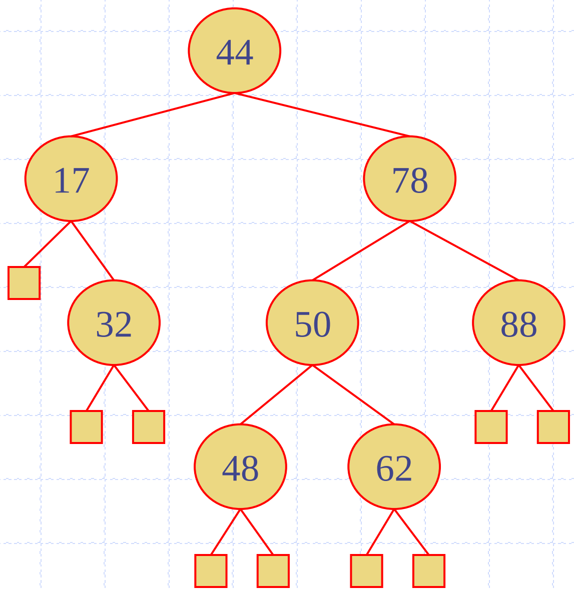


after insertion

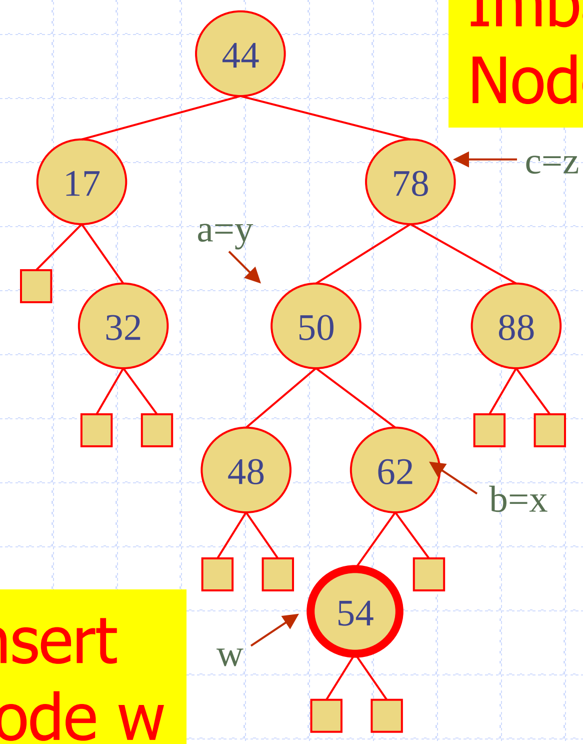
Insertion

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Insert 54:

Imbalance Node z



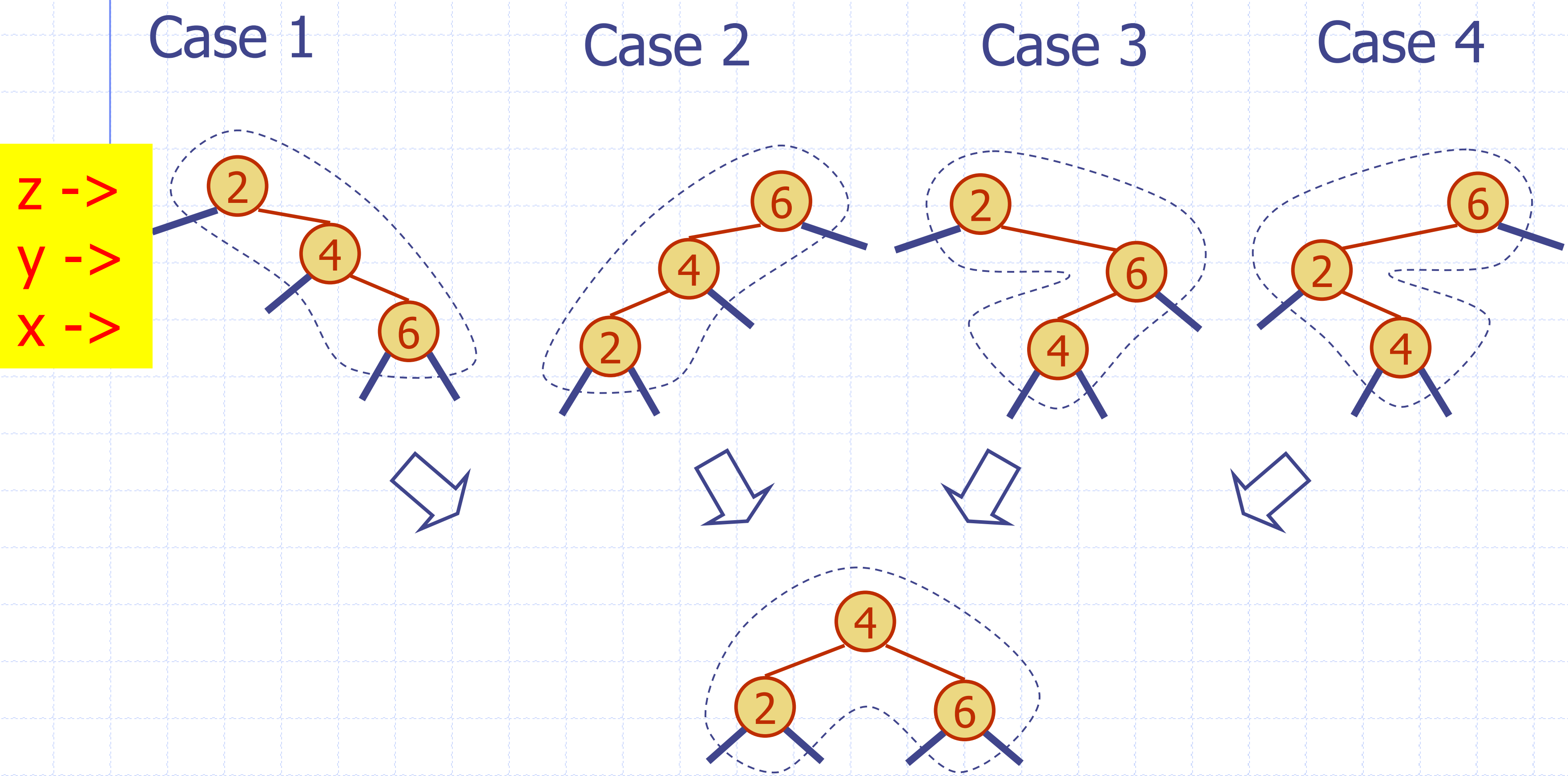
before insertion



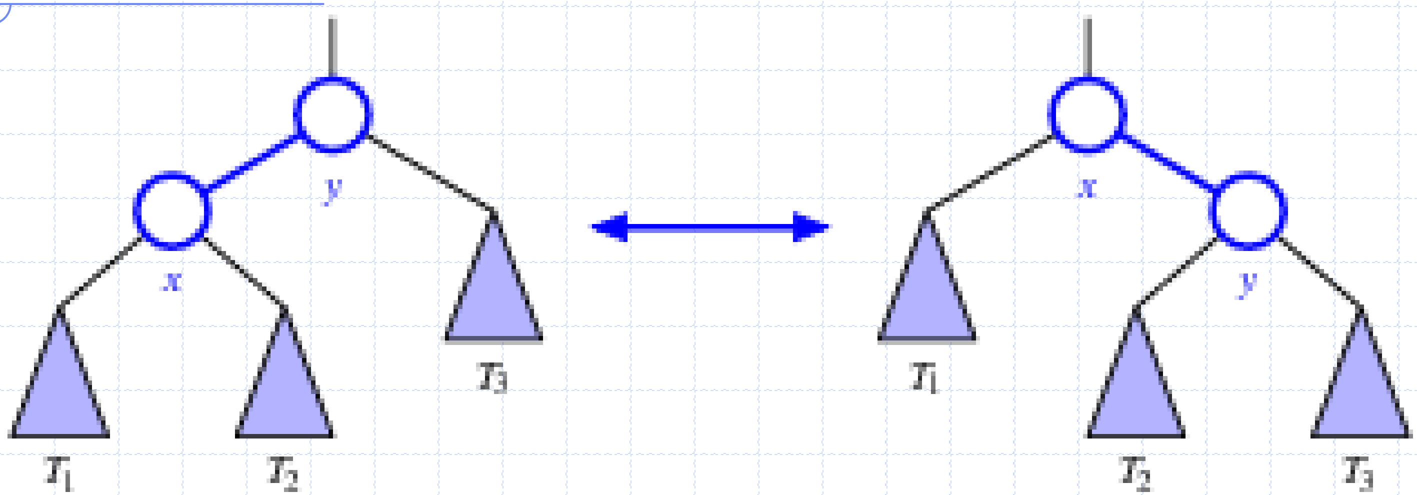
Insert Node w

after insertion

Overview of 4 Cases of Trinode Restructuring



Rotation operation



Consider subTree points to y
and we also have x and y

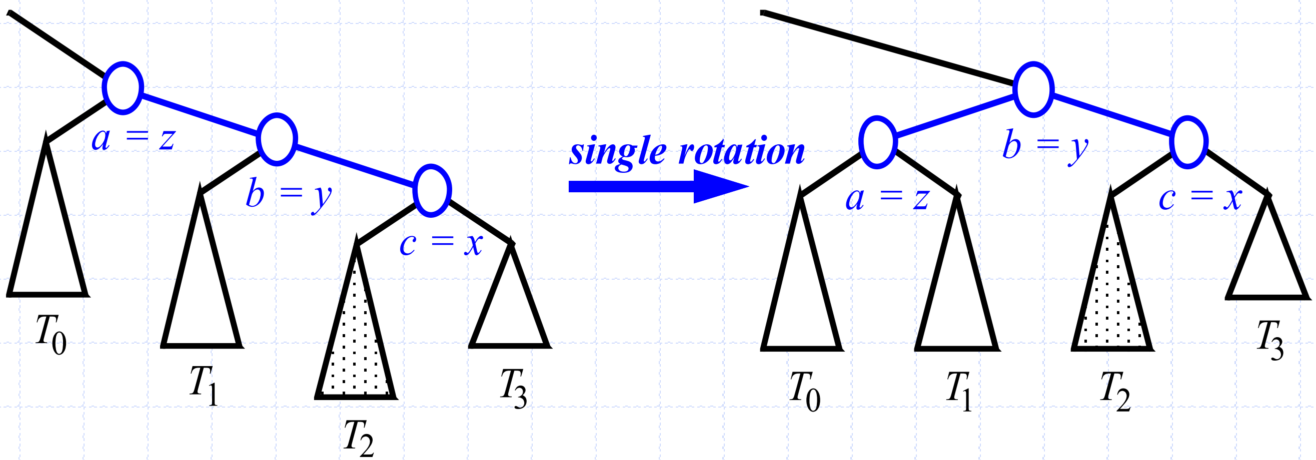
1. y.left = x.right
2. x.right = y
3. subTree = x

With a linked structure
- Constant number of updates
- O(1) time

Trinode Restructuring: Case 1

- Keys: a < b < c
- Nodes: grandparent z is not balanced, y is parent, x is node

Single Rotation:

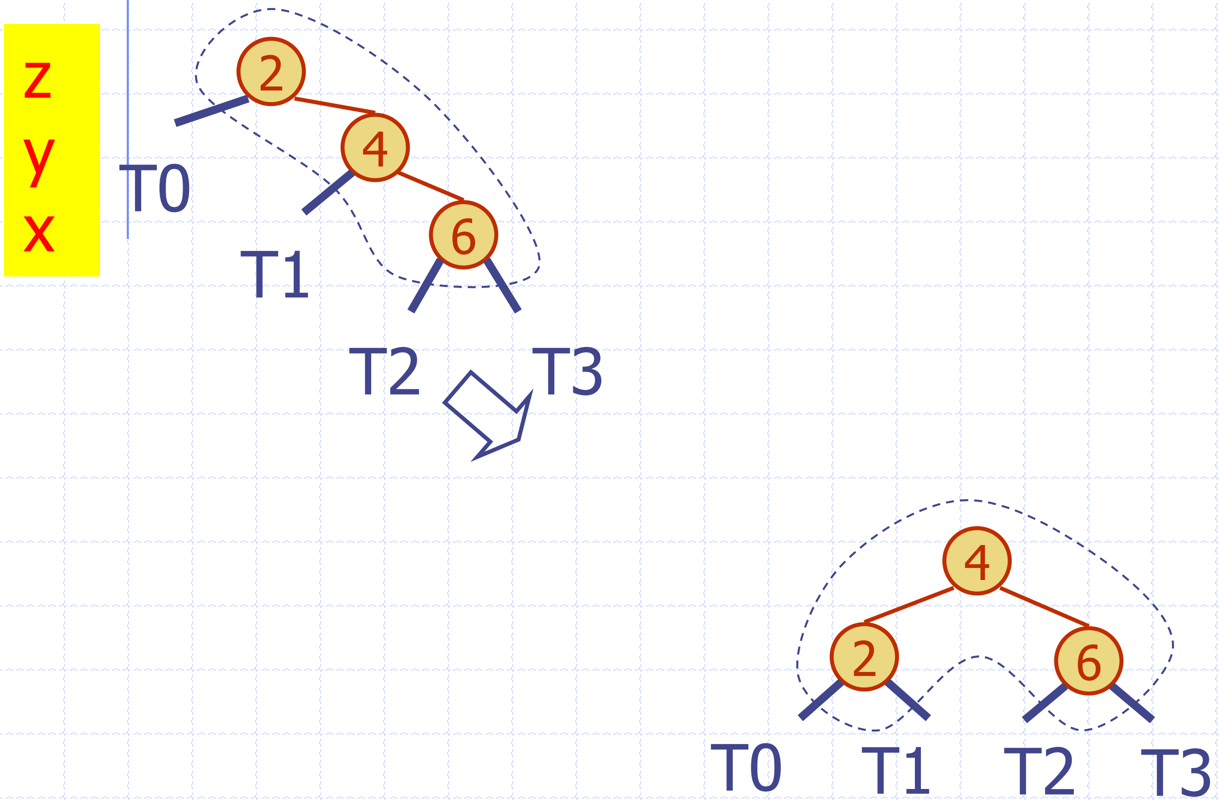


- Not balanced at a, the smallest key
- x has the largest key c

- Result: middle key b at the top

Example for Case 1

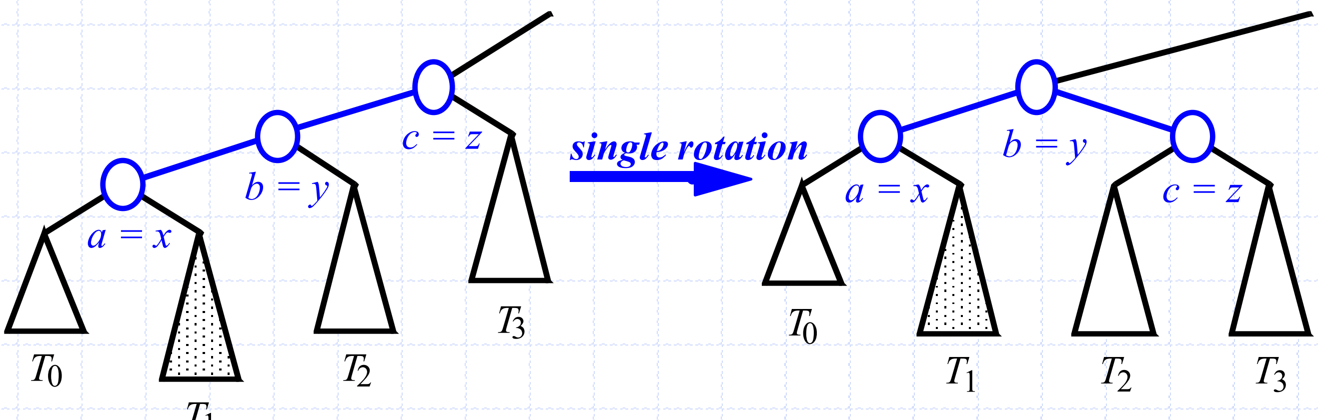
Case 1



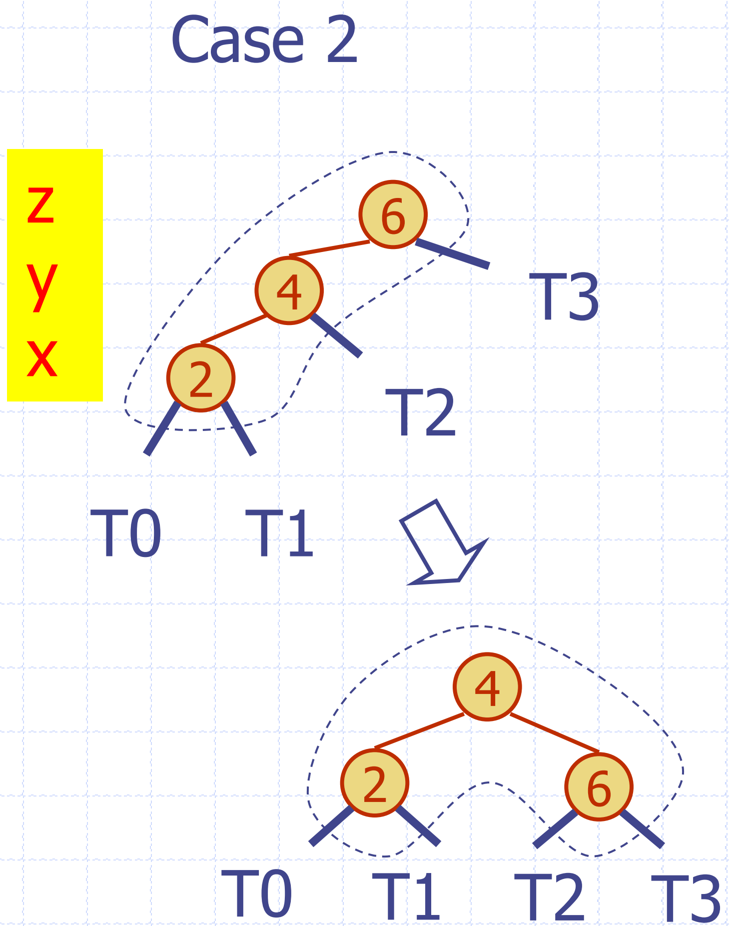
Trinode Restructuring: Case 2

- Keys: a < b < c
- Nodes: grandparent z is not balanced, y is parent, x is node

- Single Rotation:
- Not balanced at c, the largest key
- x has the smallest key a
- Result: middle key b at the top



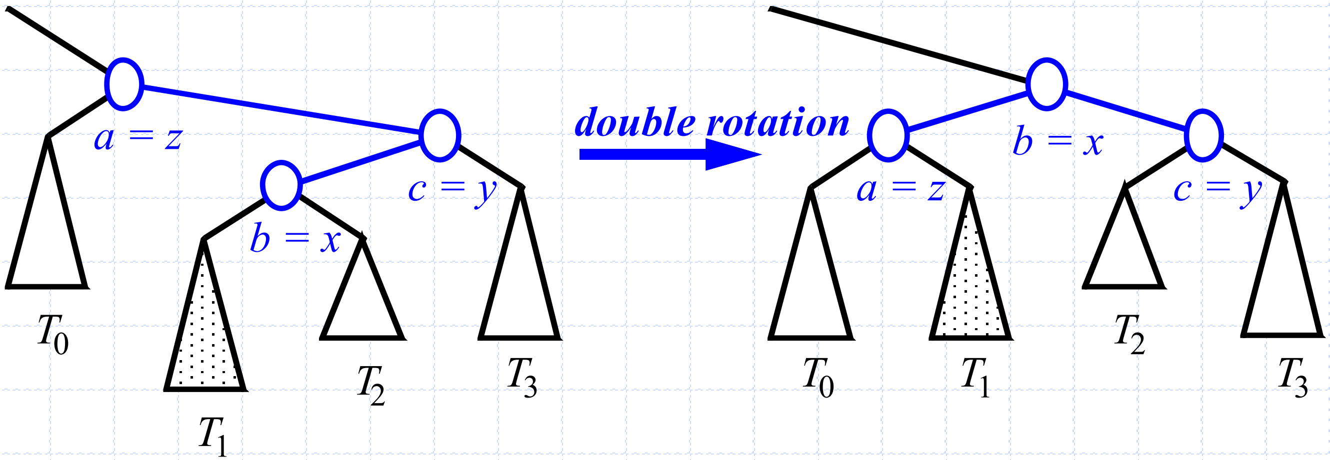
Example for Case 2



Trinode Restructuring: Case 3

- Keys: a < b < c
- Nodes: grandparent z is not balanced, y is parent, x is node

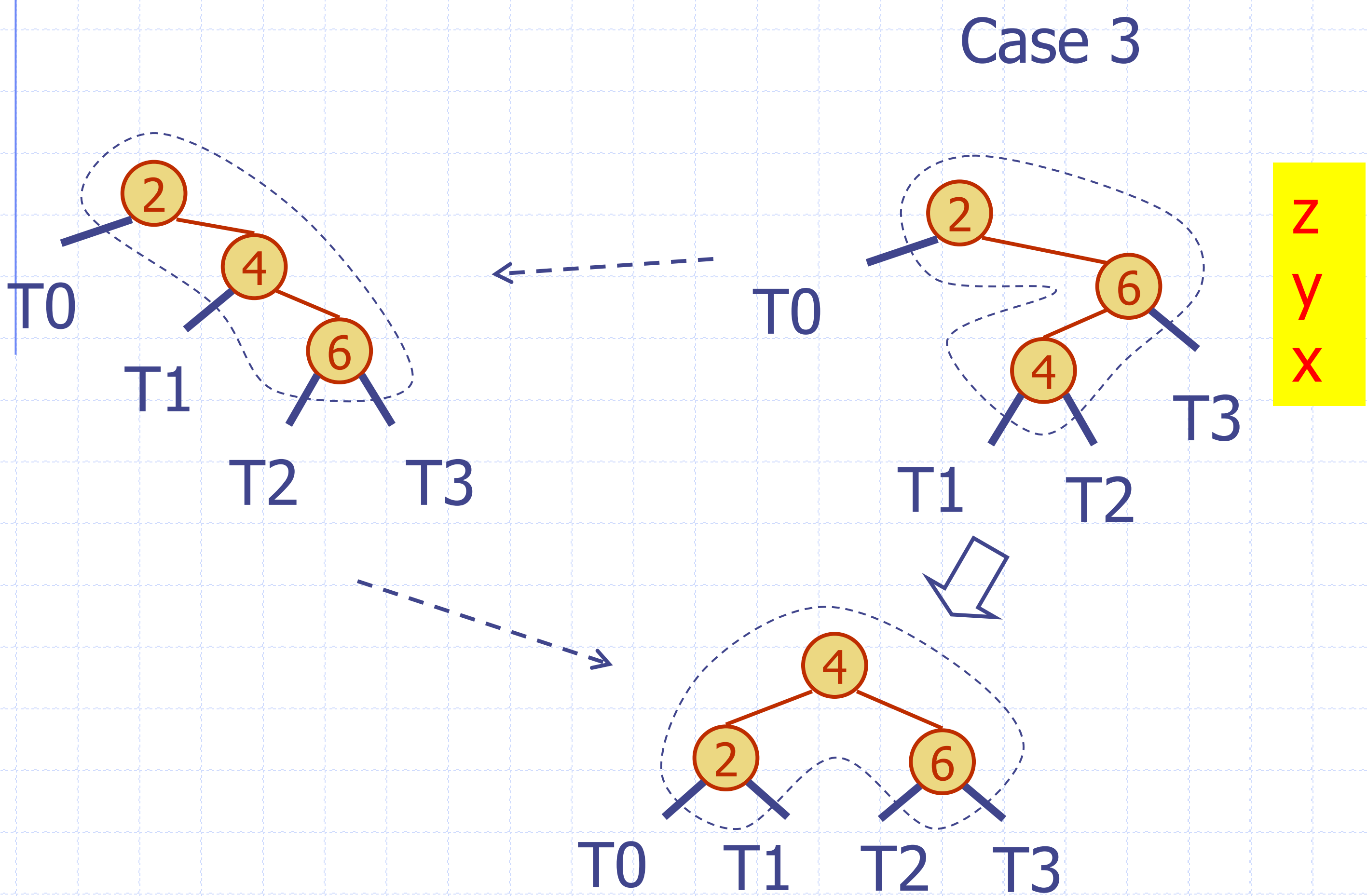
- double rotation:



- Not balanced at a, the largest key
- x has the middle key b
- x is rotated above y
- x is then rotated above z

- Result: middle key b at the top

Example for Case 3

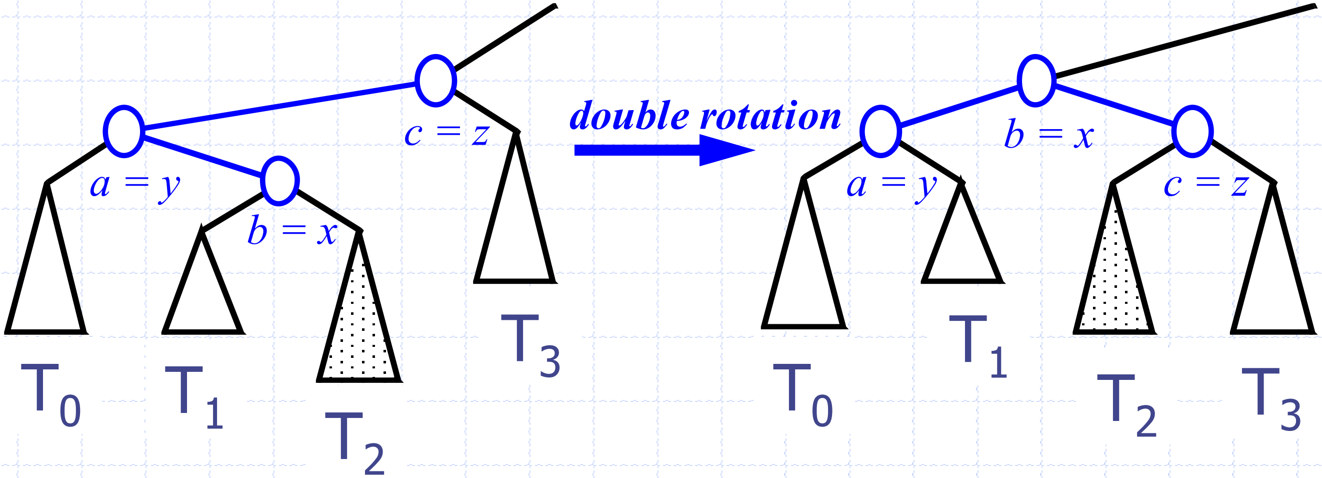


Trinode Restructuring: Case 4

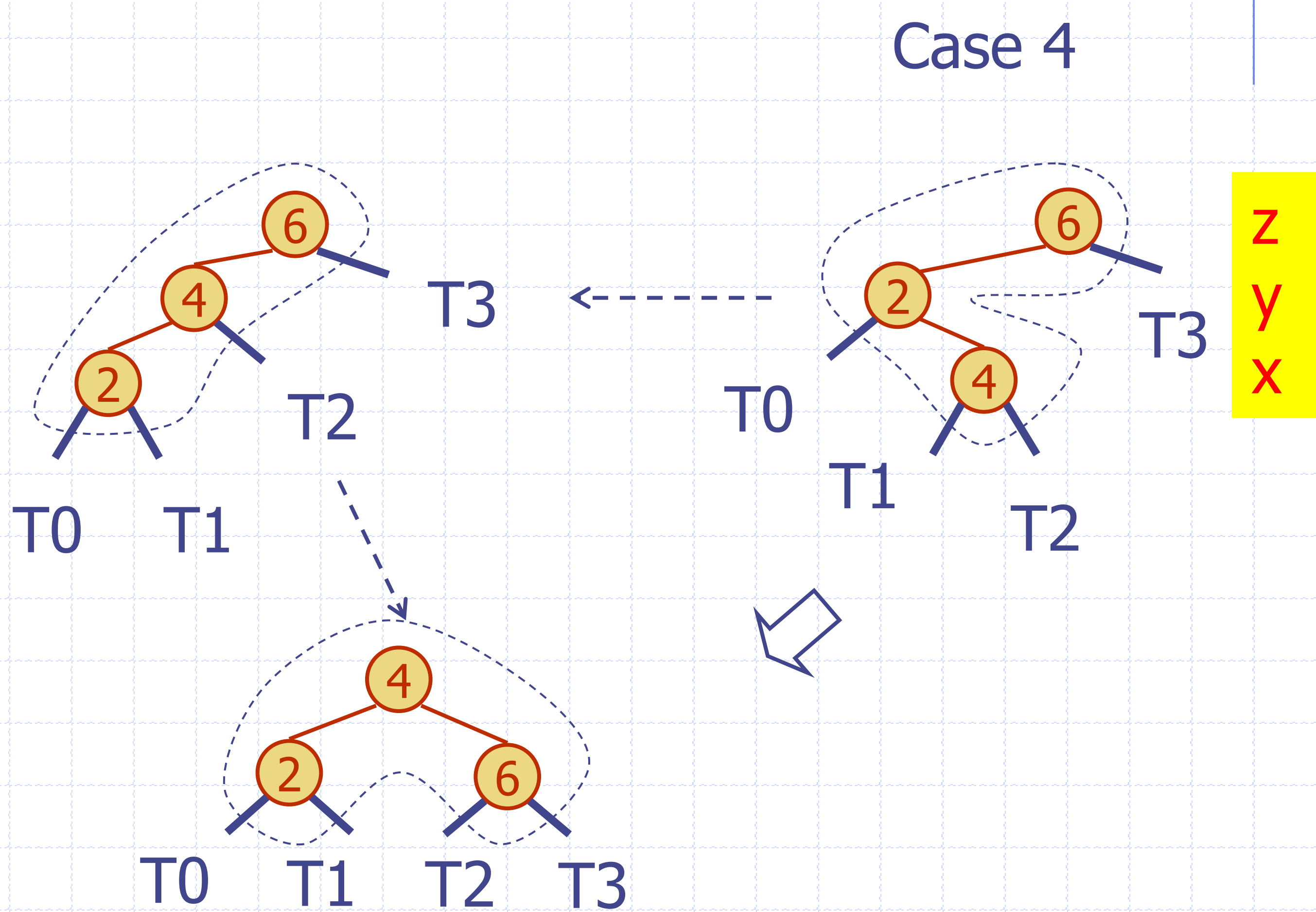
- Keys: a < b < c
- Nodes: grandparent z is not balanced, y is parent, x is node

- double rotation
- Not balanced at c, the largest key
- x has the middle key b
- x is rotated above y
- x is then rotated above x

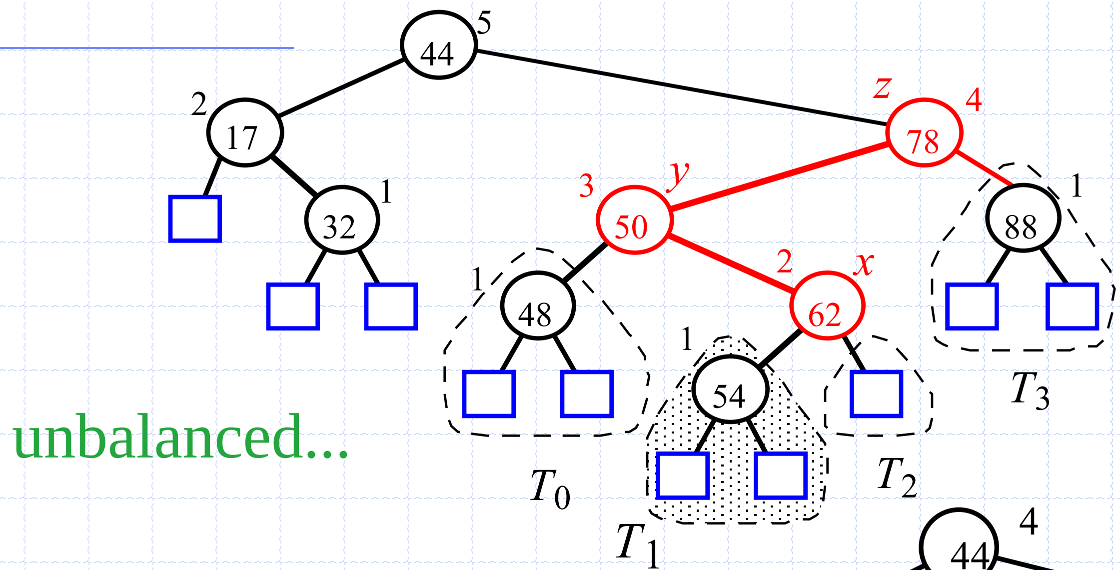
- Result: middle key b at the top



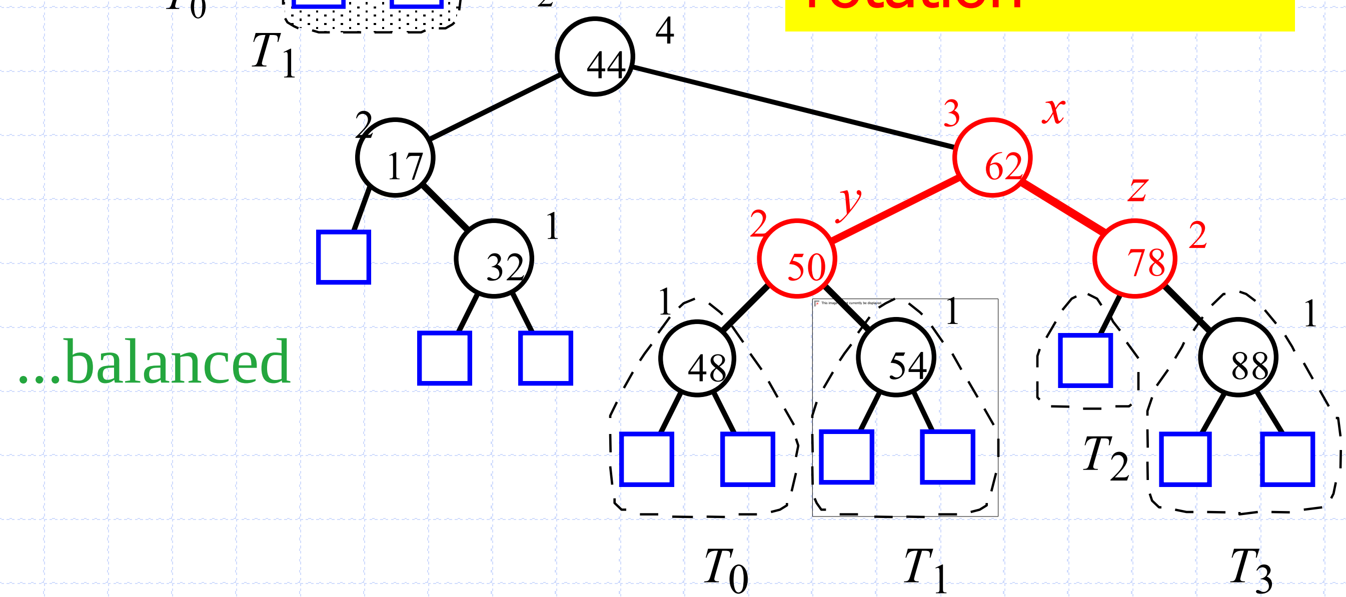
Example for Case 4



Insert 54 (Case 3 or 4?)



Draw the double rotation



Trinode Restructuring summary

Case	imbalance/ grandparent z	Node x	Rotation
1	Smallest key a	Largest key c	single
2	Largest key c	Smallest key a	single
3	Smallest key a	Middle key b	double
4	Largest key c	Middle key b	double

Trinode Restructuring Summary

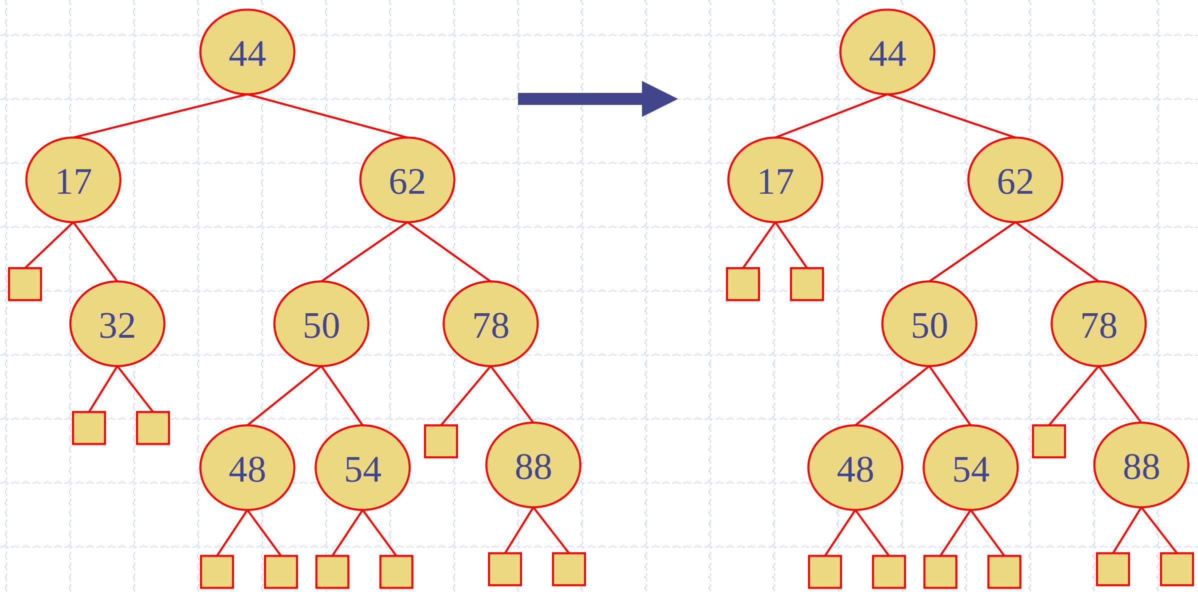
Case	imbalance/ grandparent z	Node x	Rotation
1	Smallest key a	Largest key c	single
2	Largest key c	Smallest key a	single
3	Smallest key a	Middle key b	double
4	Largest key c	Middle key b	double

The resulting balanced subtree has:

- middle key b at the top
- smallest key a as left child
  - T0 and T1 are left and right subtrees of a
- largest key c as right child
  - T2 and T3 are left and right subtrees of c

Removal

- Removal begins as in a binary search tree
  - the node removed will become an empty external node.
  - Its parent, w, may cause an imbalance.
- Remove 32, imbalance at 44

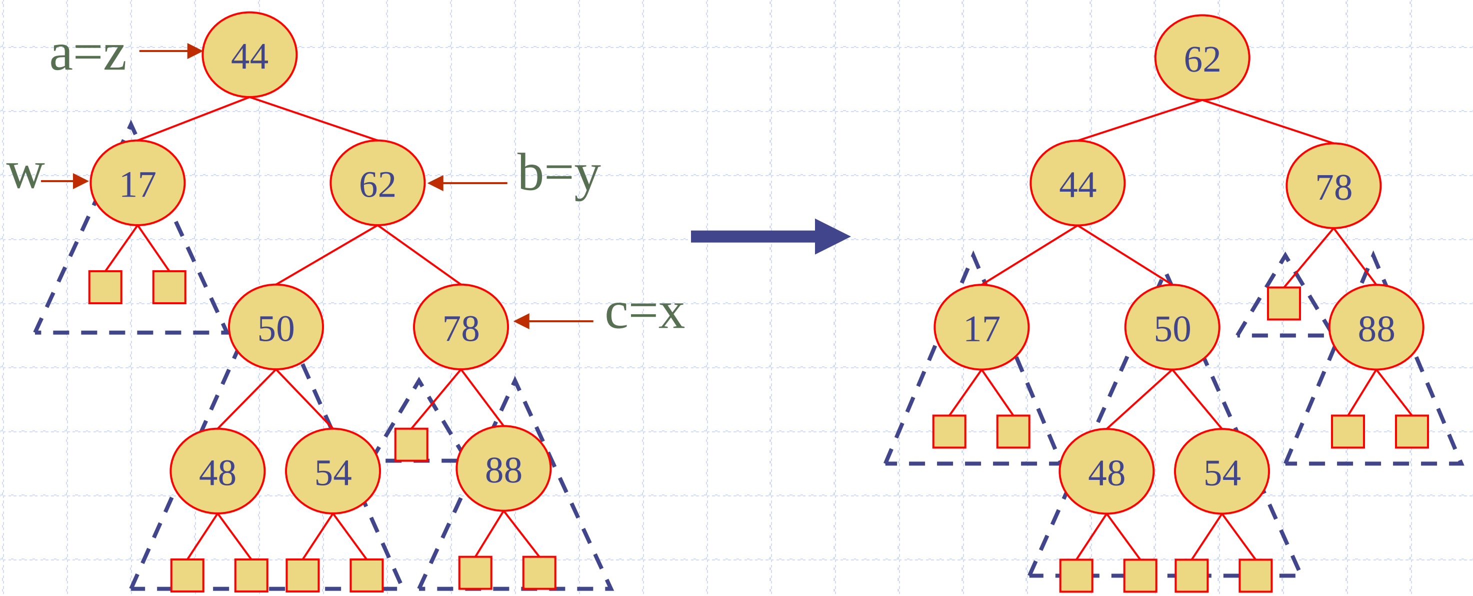


before deletion of 32

after deletion

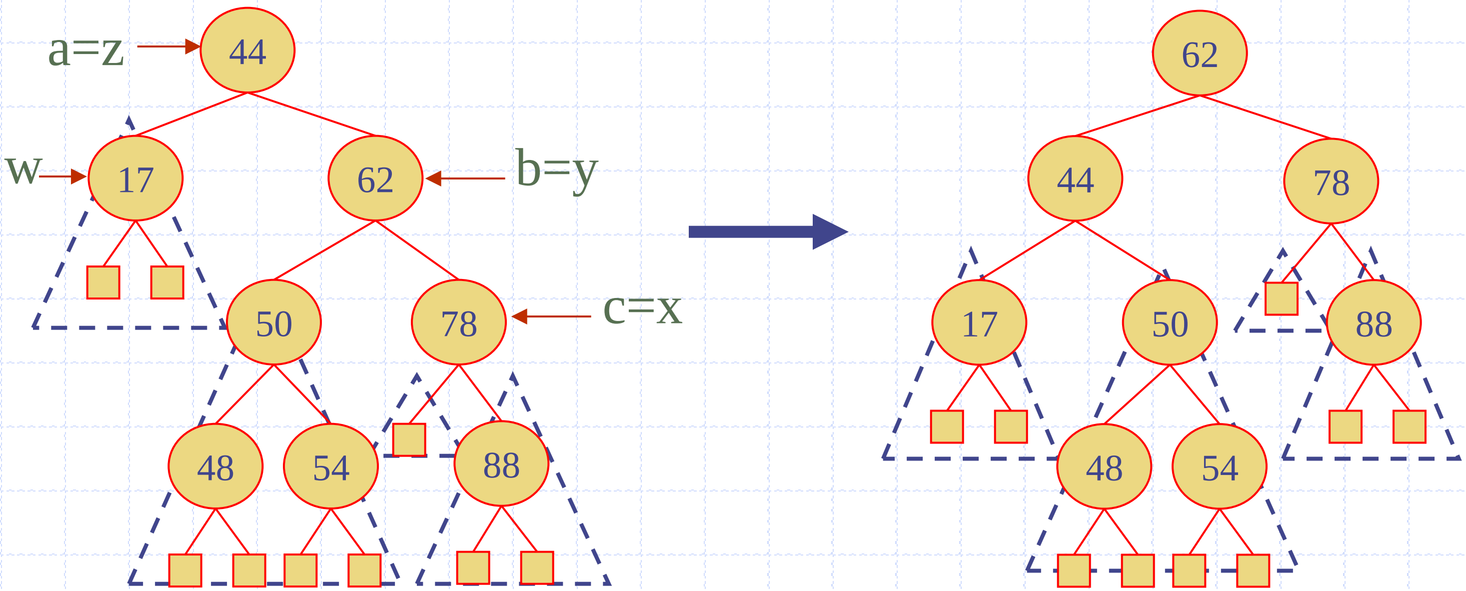
Rebalancing after a Removal

- z = first unbalanced node encountered while travelling up the tree from w.
  - y = child of z with the larger height,
  - x = child of y with the larger height
- trinode restructuring to restore balance at z—Case 1 in example

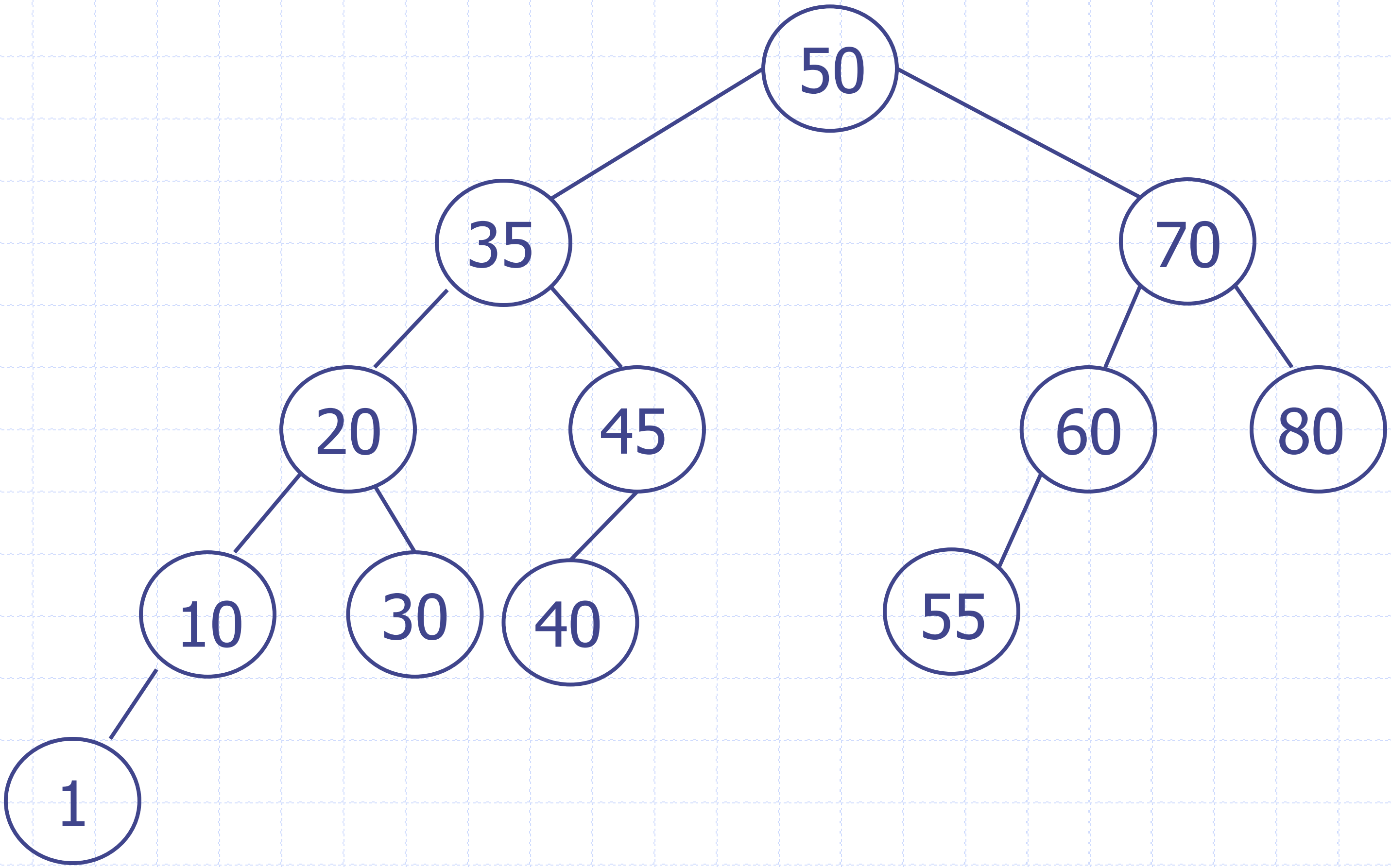


Rebalancing after a Removal

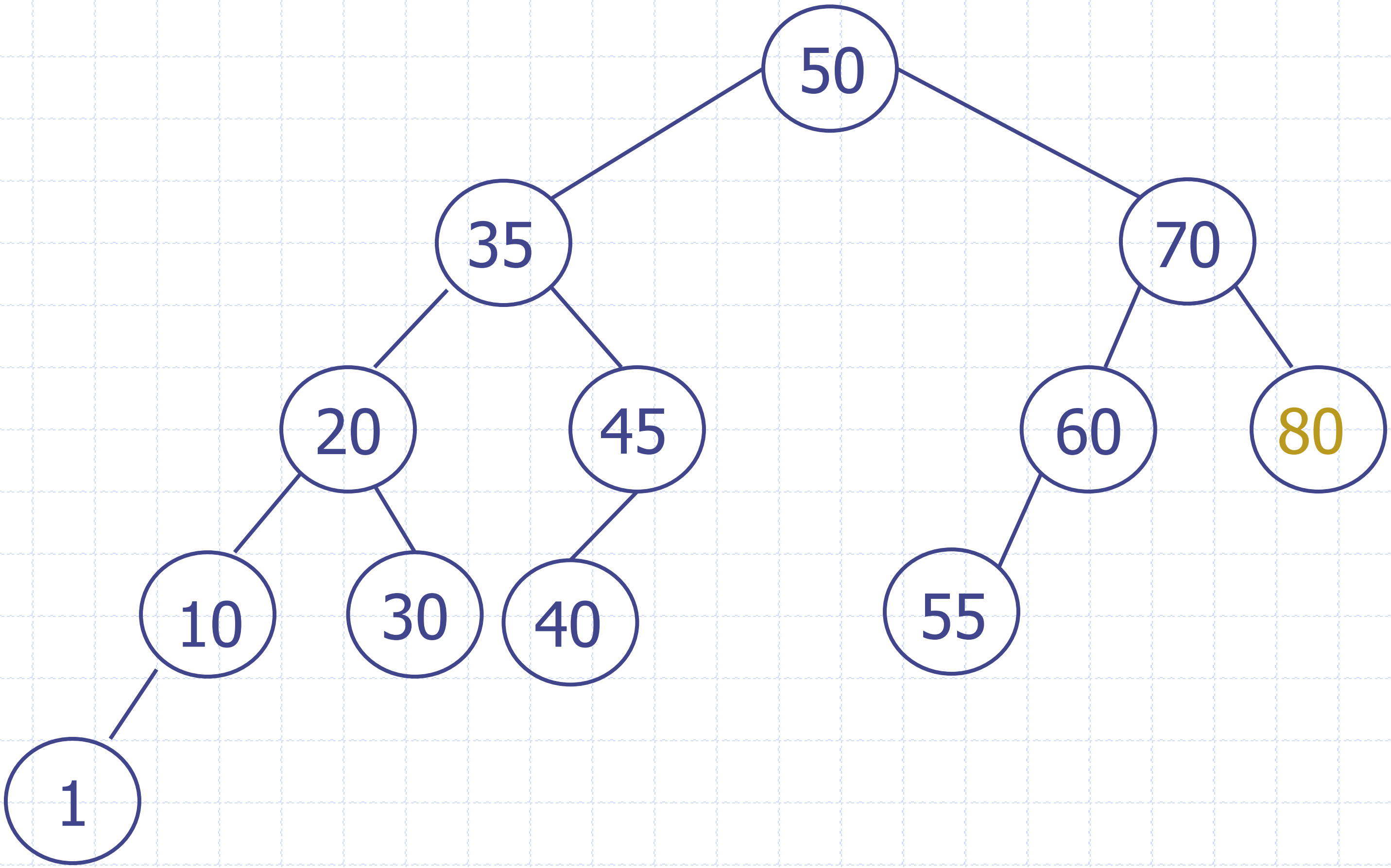
- this restructuring may upset the balance of another node higher in the tree
  - continue checking for balance until the root of T is reached



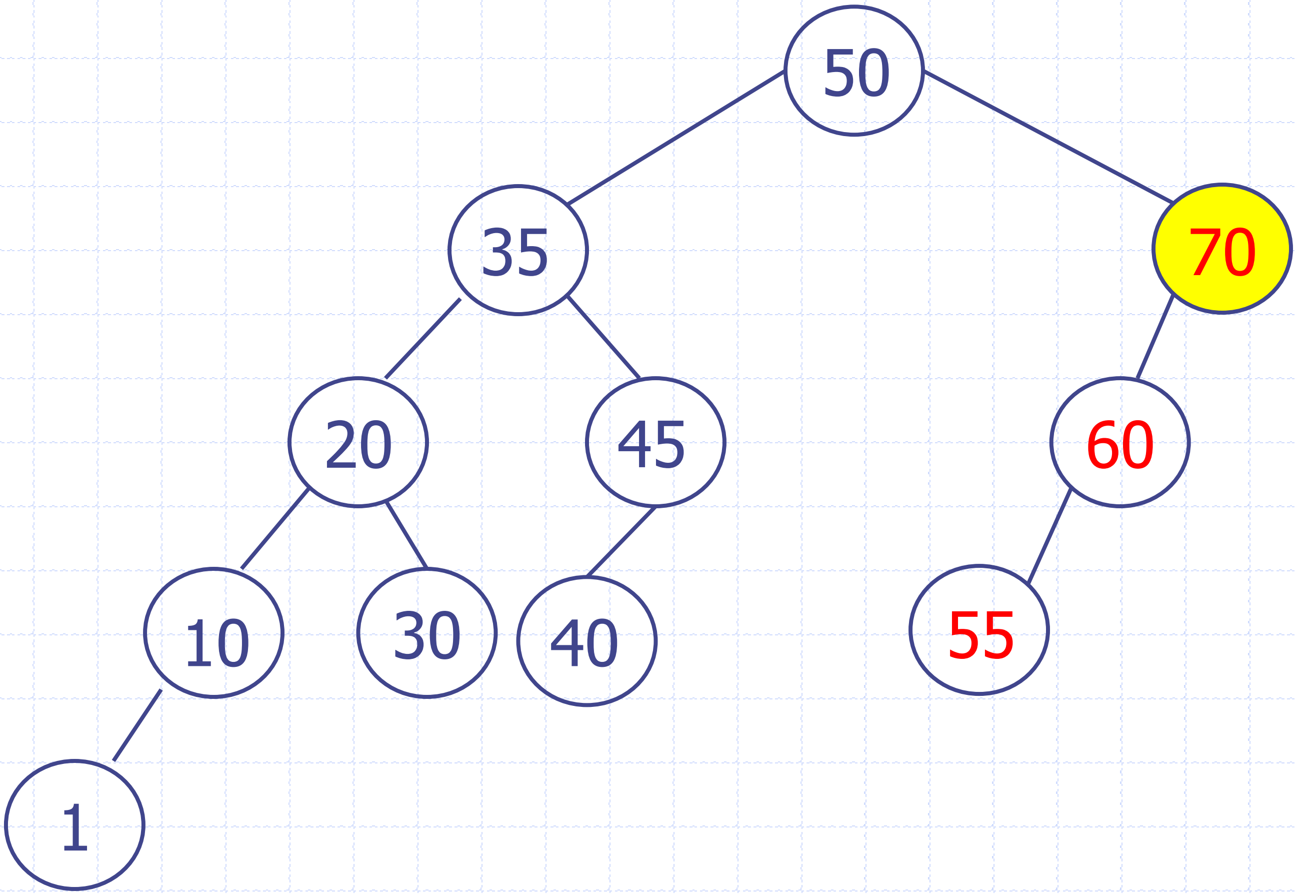
Balanced tree



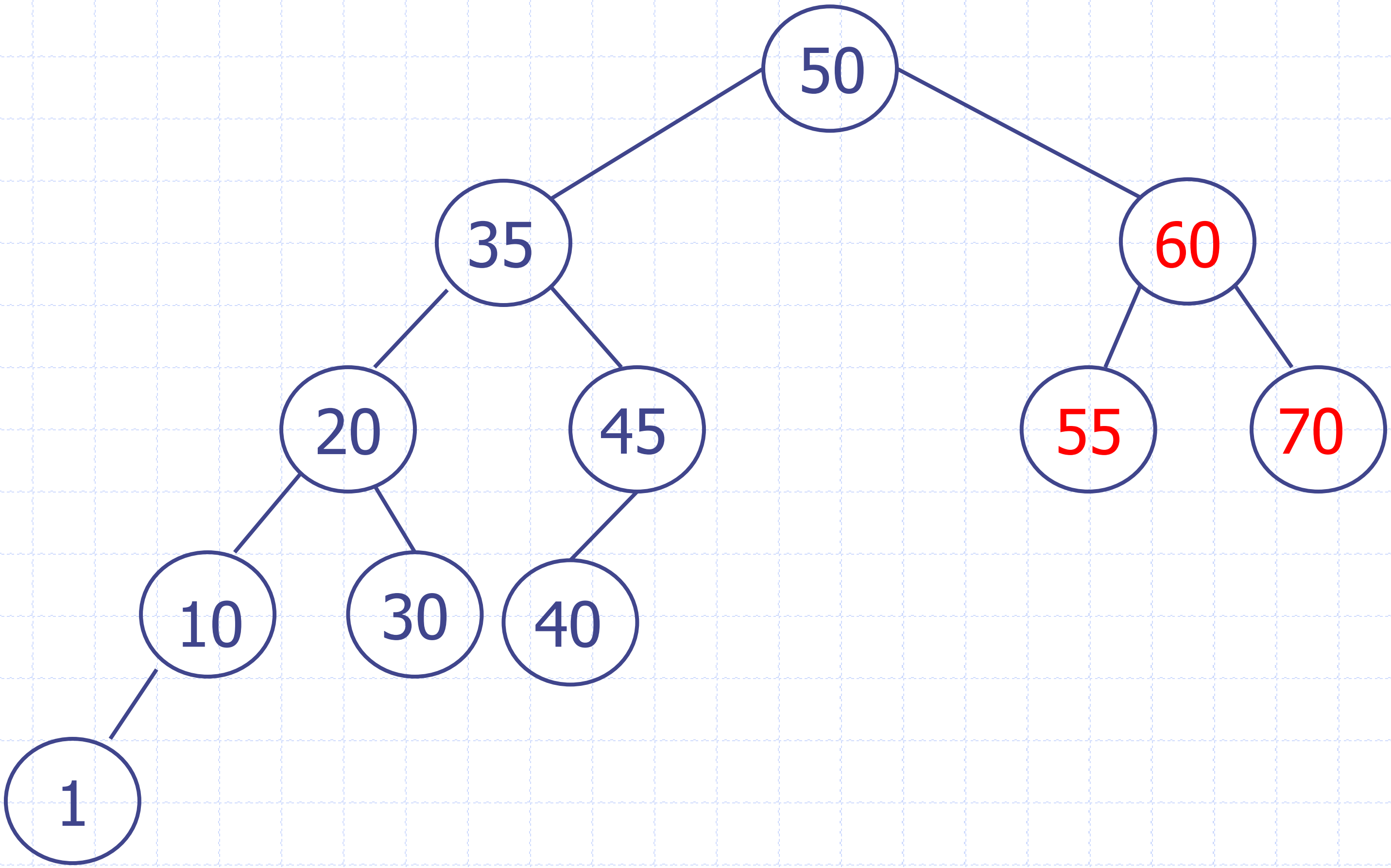
Delete 80



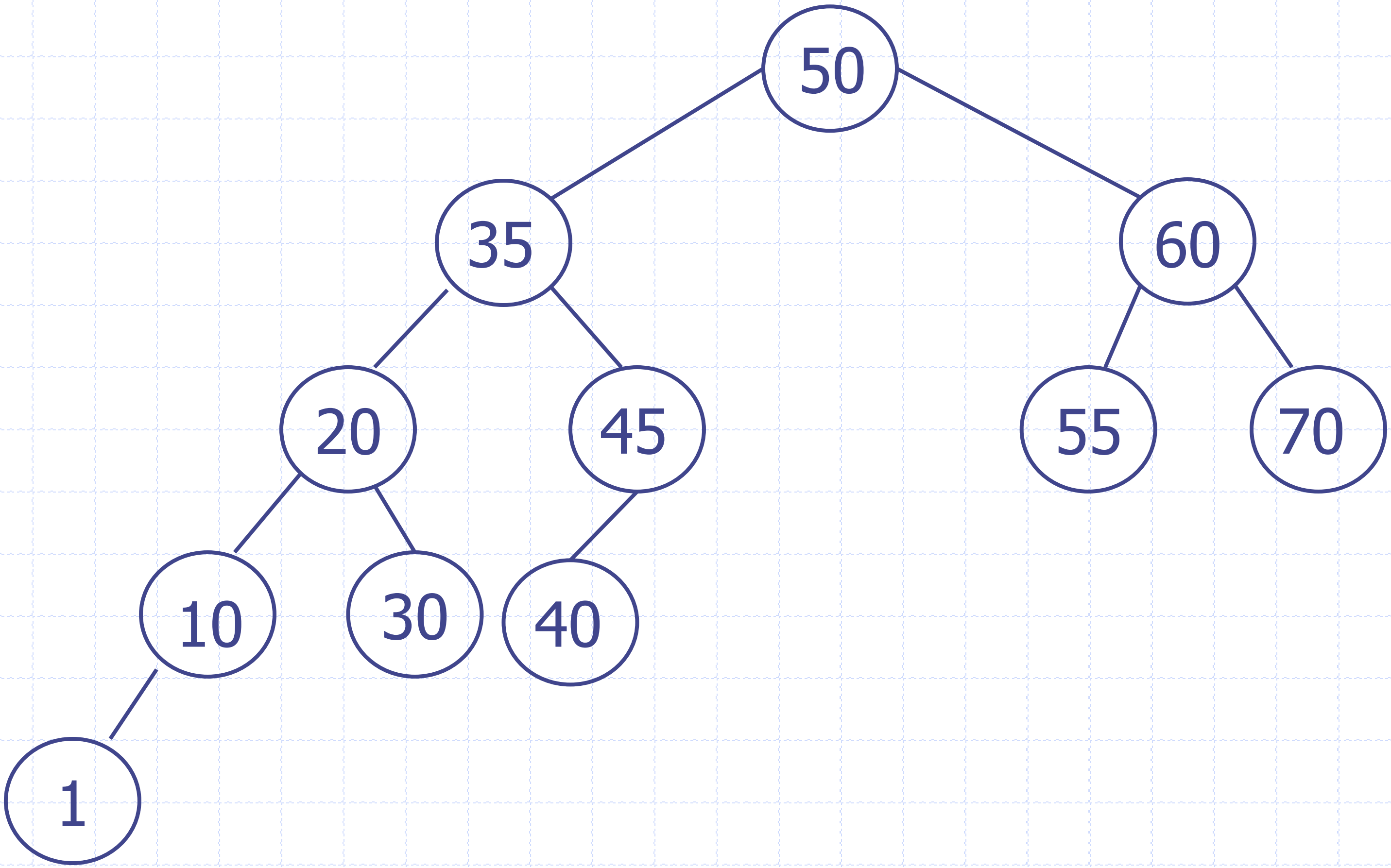
Not balanced at 70



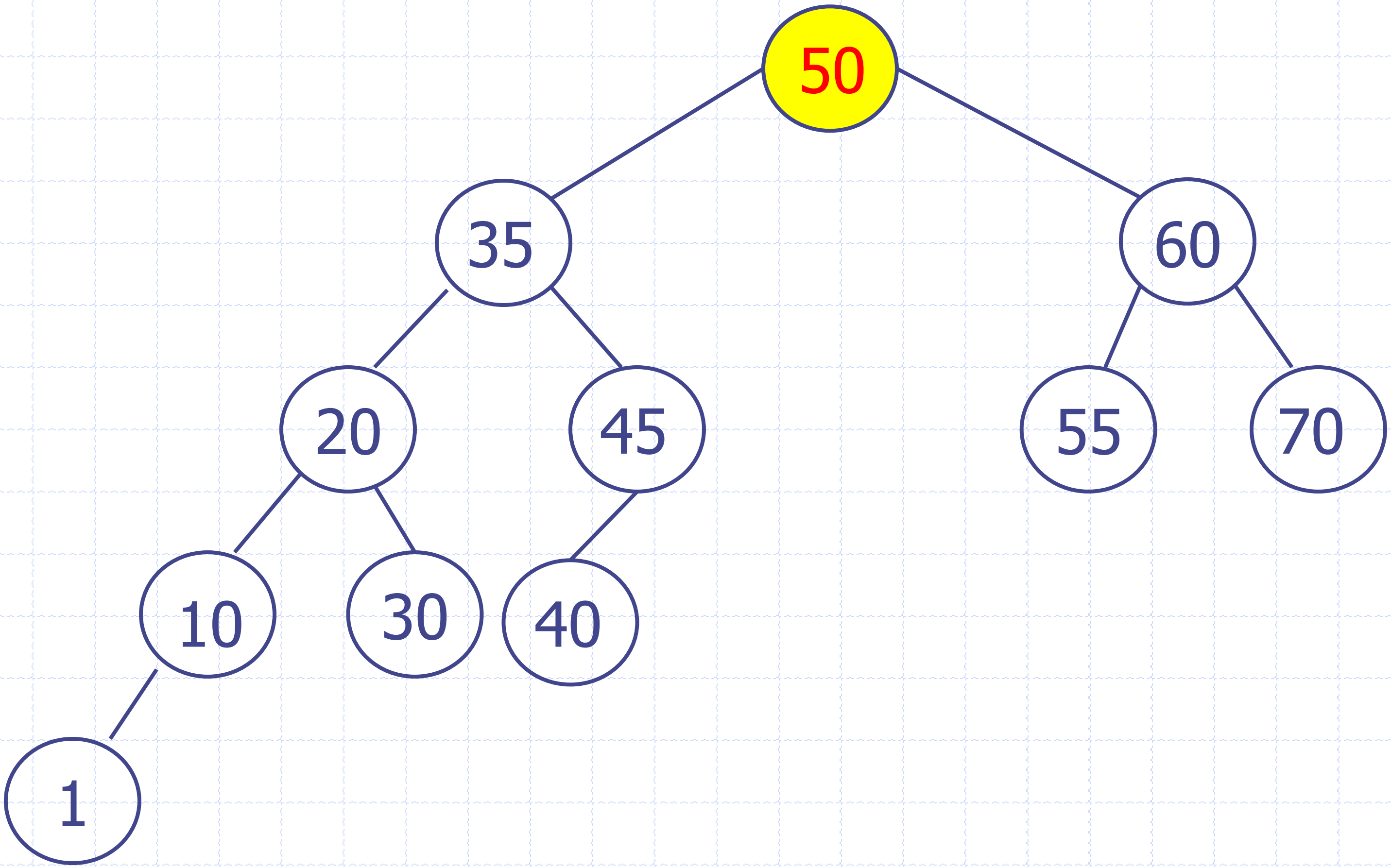
Single rotation



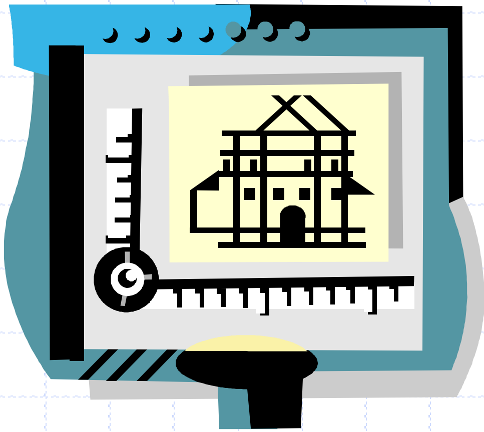
Anything wrong?



Not balanced at 50!



AVL Tree Performance



- n entries
  - O(n) space
  - A single restructuring takes O(1) time
    - using a linked-structure binary tree

Operation	Worst-case Time Complexity	
Get/search	O(log n)	Up to height log n
Put/insert	O(log n)	O(log n): searching & restructuring
Remove/delete	O(log n)	O(log n): searching & restructuring up to height log n

AVL Trees

- balanced Binary Search Tree (BST)
- Insert/delete operations include rebalancing if needed
- Worst-case time complexity: O(log n)
  - expected O(log n) for skip lists
  - No duplicated keys in skip lists
  - No moving a bunch of keys in sorted array