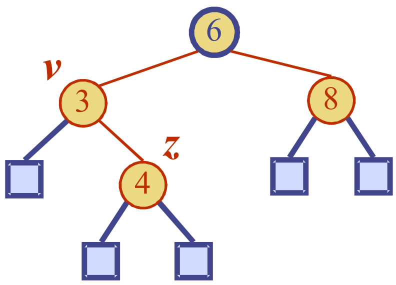
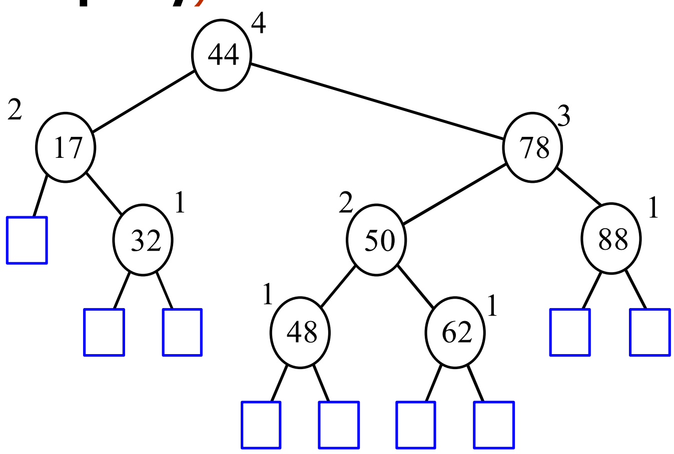
# AVL Trees



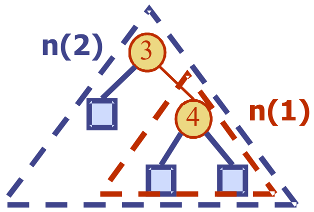



# AVL Tree Definition

- An AVL Tree is a balanced 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 (**Height-Balance Property**)



# Height of an AVL Tree




- Fact: The height of an AVL tree storing n keys is O(log n).
- Proof: Let us bound n(h): the minimum number of internal nodes of an AVL tree of height h.
- We easily see that 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

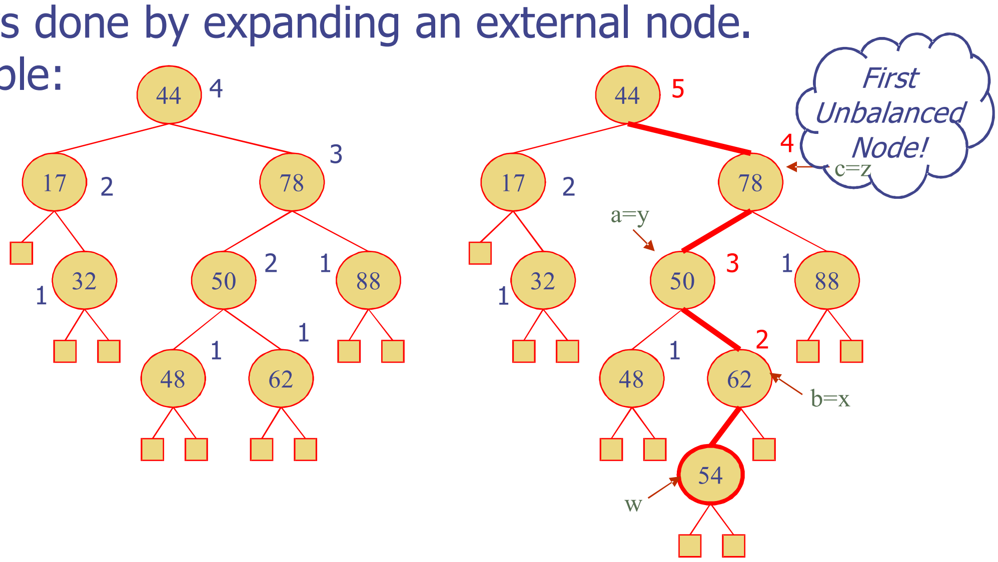
  n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),

  n(h) > $2^i$n(h-2i)
- Solving the base case we get: n(h) > $2^{h/2-1}$
- Taking logarithms: h < 2log 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.
- Example:

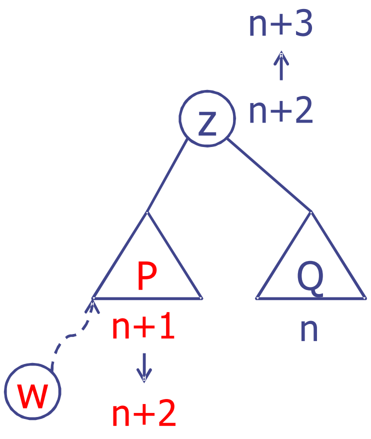


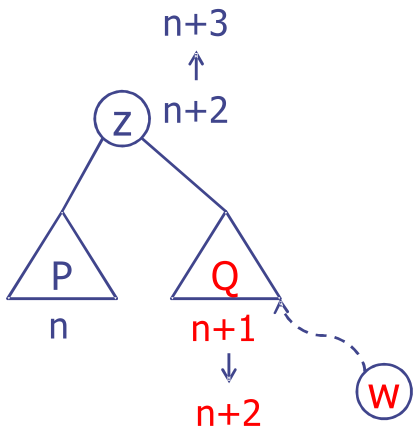

before insertion after insertion

# Violation of HBP

Assume that "z" is the first unbalanced node encountered after insertion

- Case I
- Case II

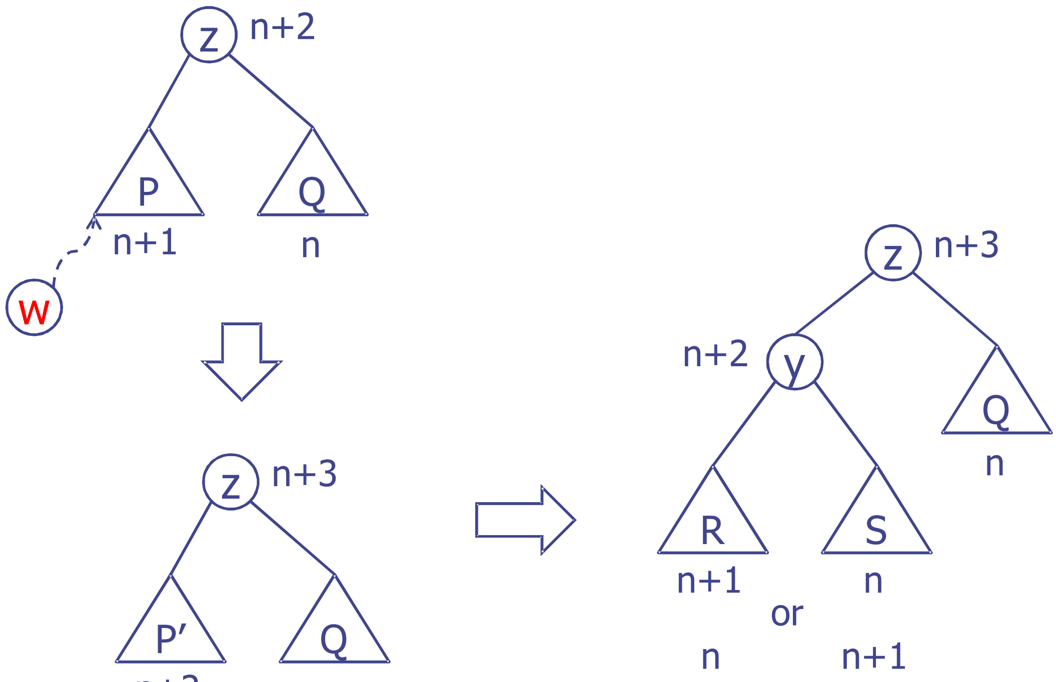



The height of "z" will become n+3 after insertion!

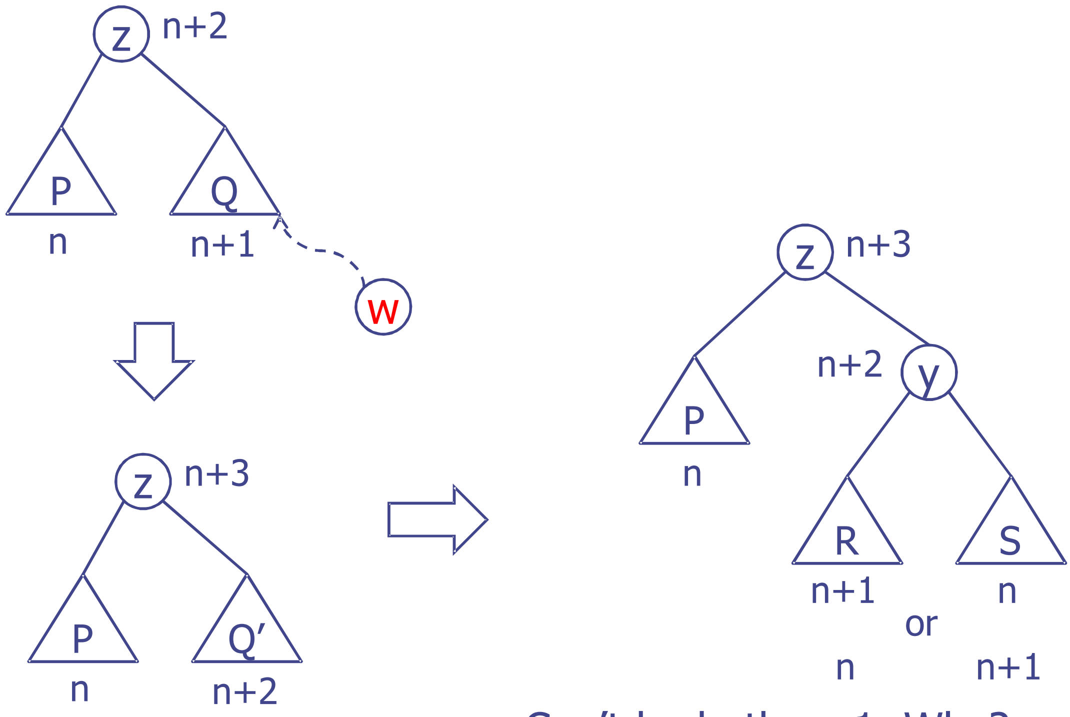


## Case I. Node w is added to P




Can't be both n+1. Why?

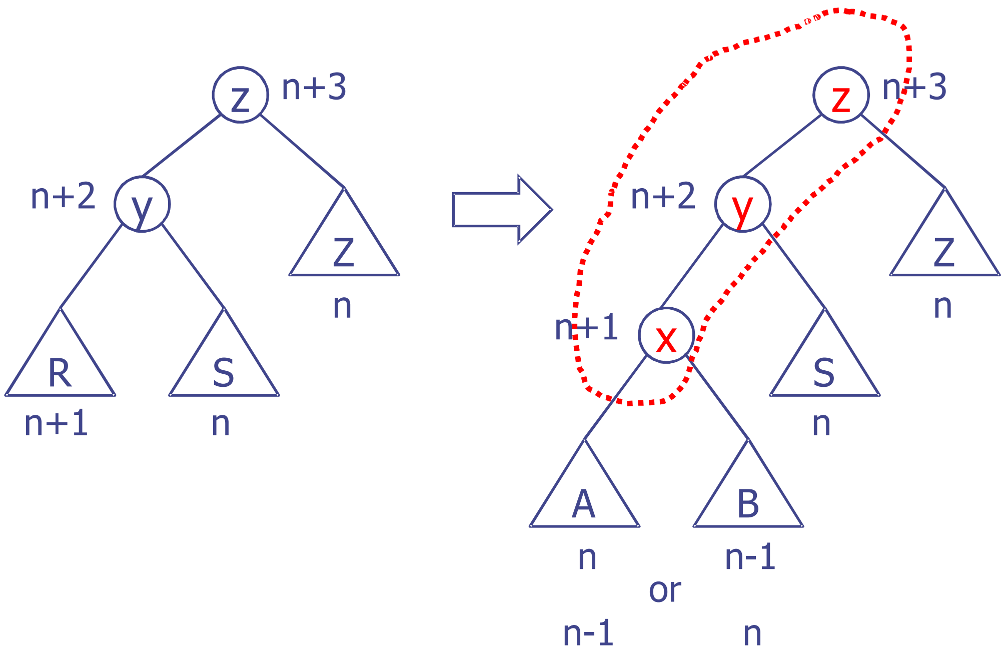
## Case II. Node w is added to Q



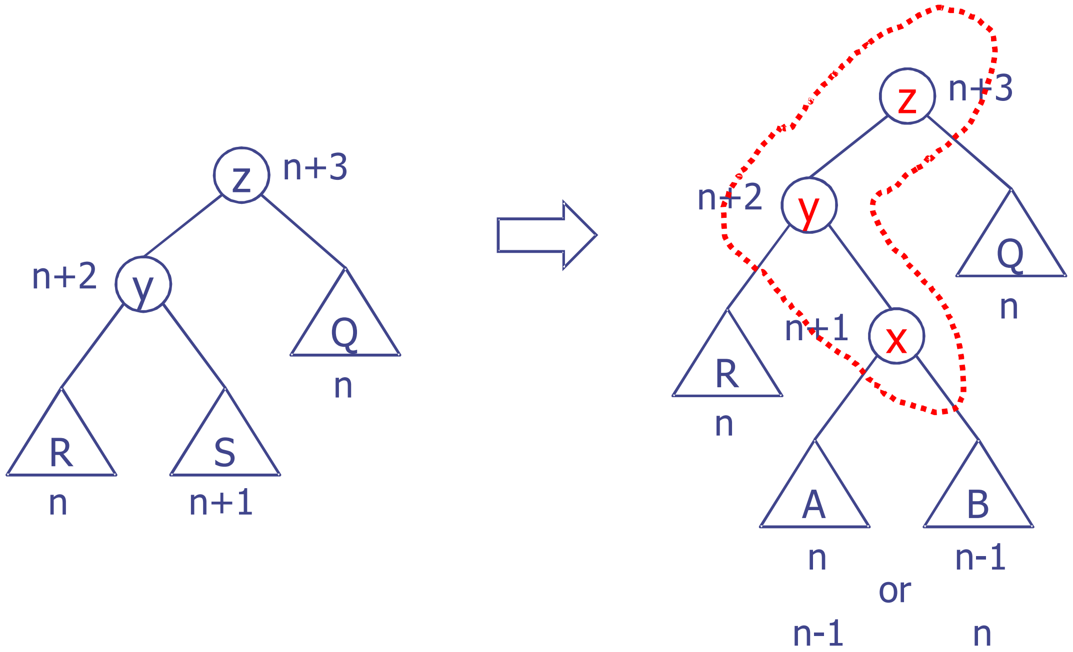

Can’t be both n+1. Why?



## Case I-1: R : n+1, S : n
## (w is added to R)

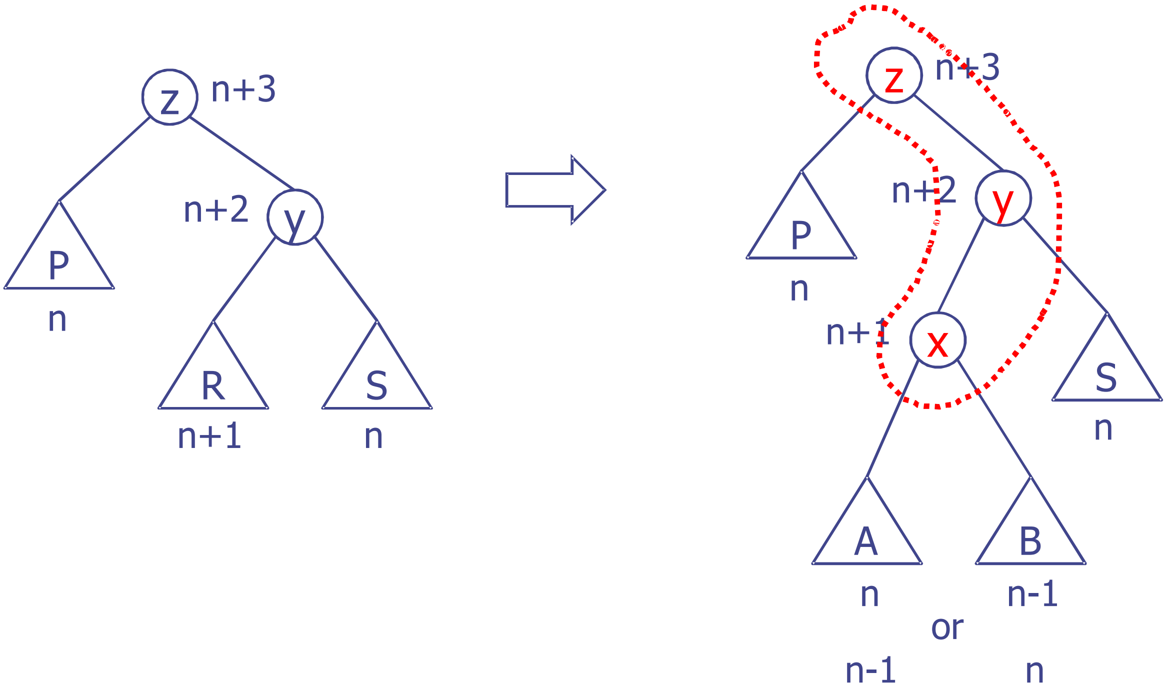


## Case I-2: R : n, S : n+1
## (w is added to S)

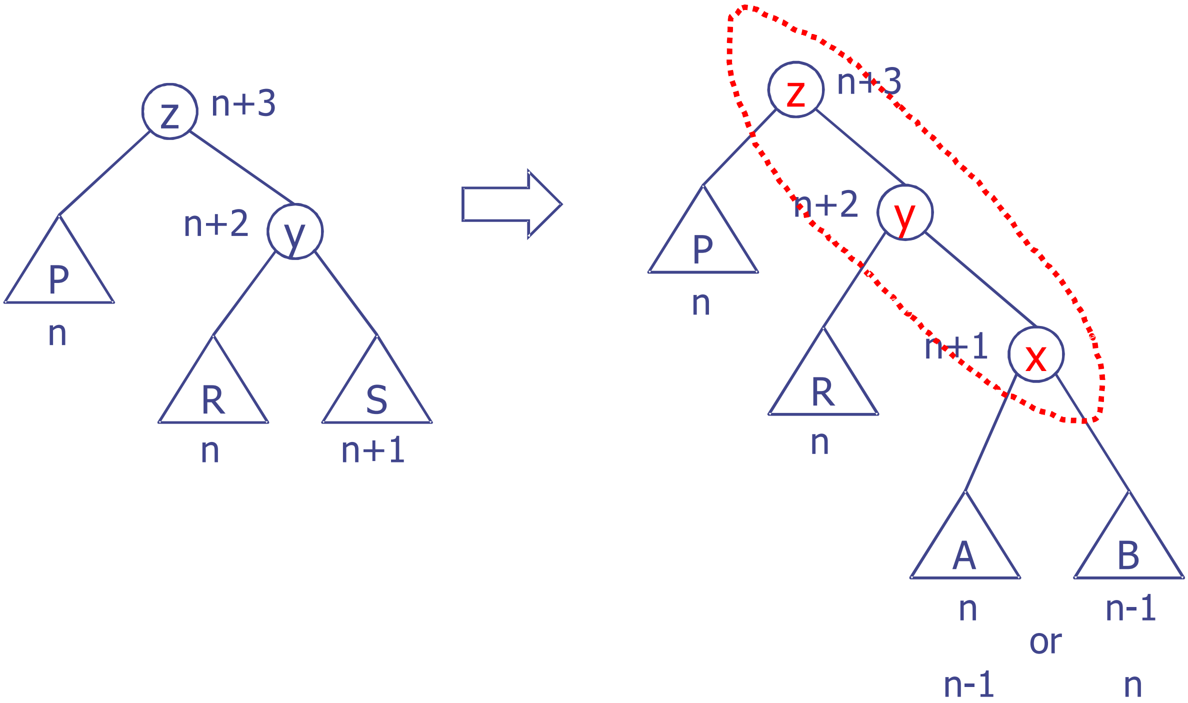





## Case II-1: R : n+1, S : n
## (w is added to R)



Case II-2: R : n, S : n+1
(w is added to S)






# Trinode Restructuring

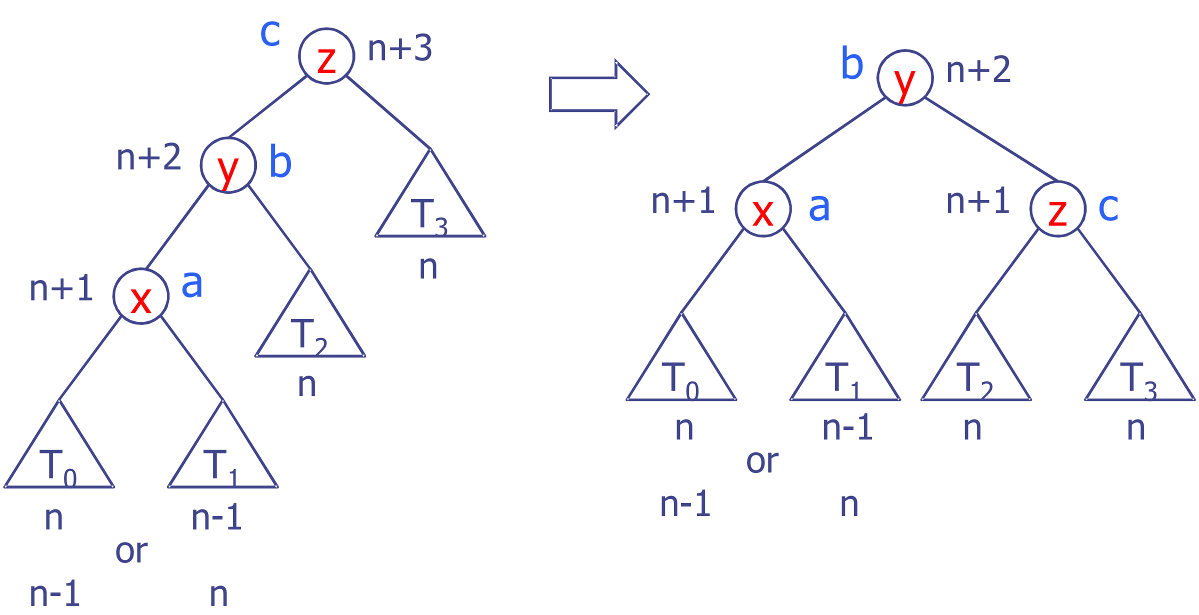
- Let *z* be the first node we encounter in going up from w towards the root
- Let *y* be the child of z with higher height (y must be an ancestor of w)
- Let *x* be the child of y with higher height
  - There cannot be a tie and node x must be an ancestor of w including itself

# Trinode Restructuring (Cont'd)

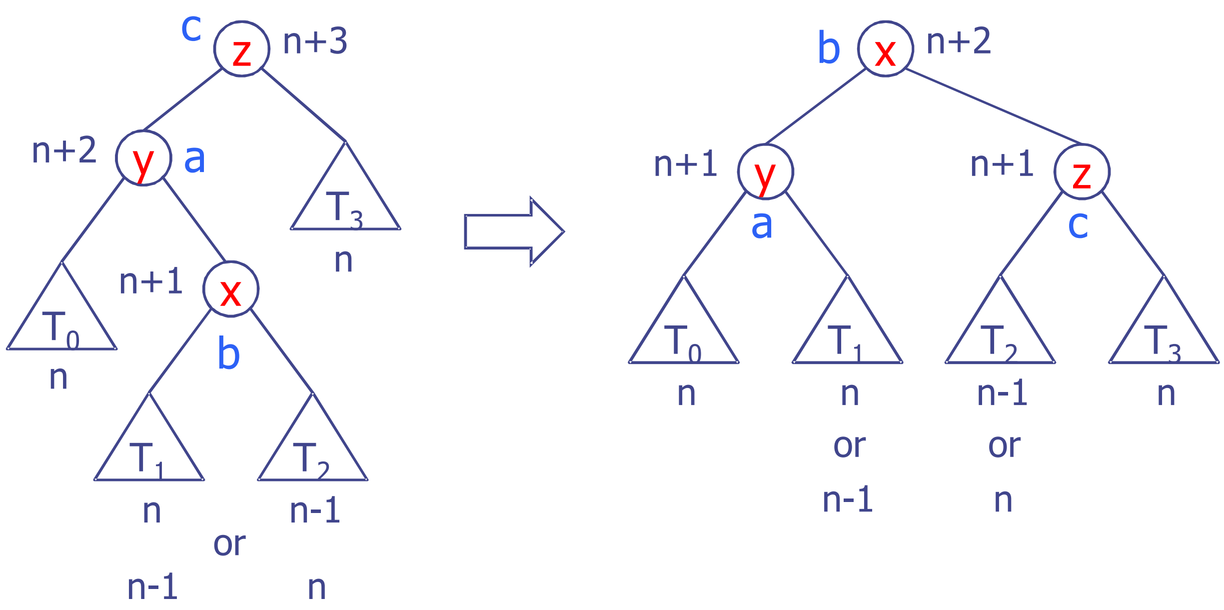
- Let a, b, c be a left-to-right (inorder) listing of the nodes x, y, z.
- Let $T_0$, $T_1$, $T_2$, $T_3$ be a left-to-right (inorder) listing of the four subtrees of x, y, and z
- Make node b as the new root.
- Let a be the left child of b and let $T_0$ and $T_1$ be the left and right child of a, respectively.
- Let c be the right child of b and let $T_2$ and $T_3$ be the left and right child of c, respectively.



Case I-1:
Restructure ➔ Single Rotation Right

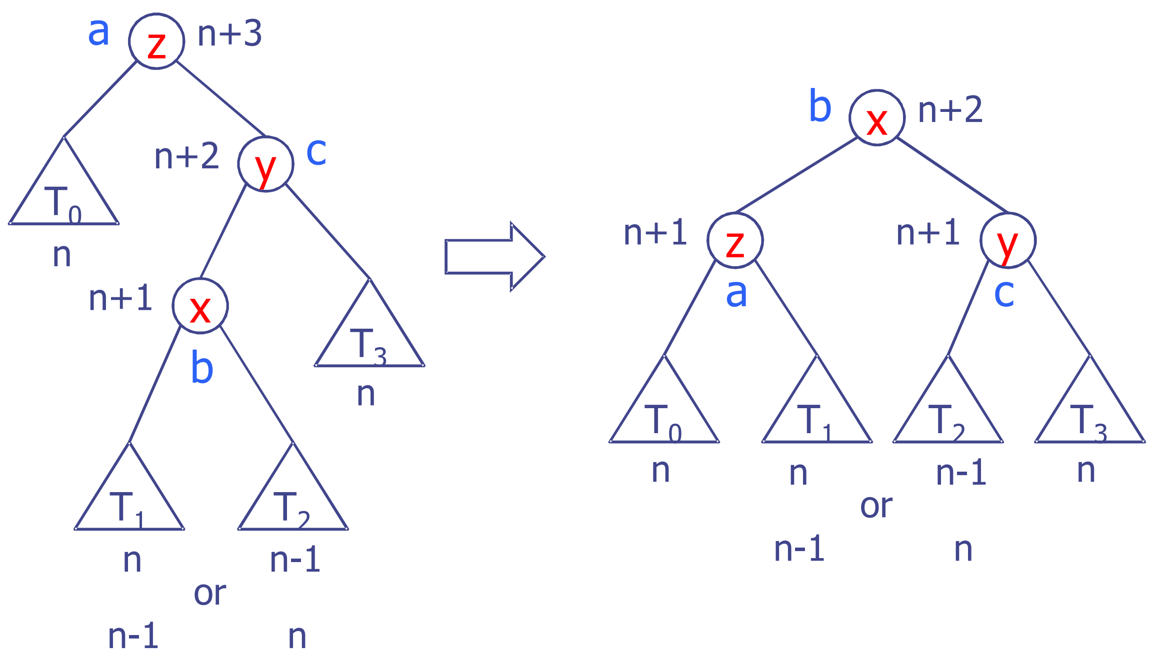


## Case I-2:
## Restructure ➔ Double Rotation Left-Right

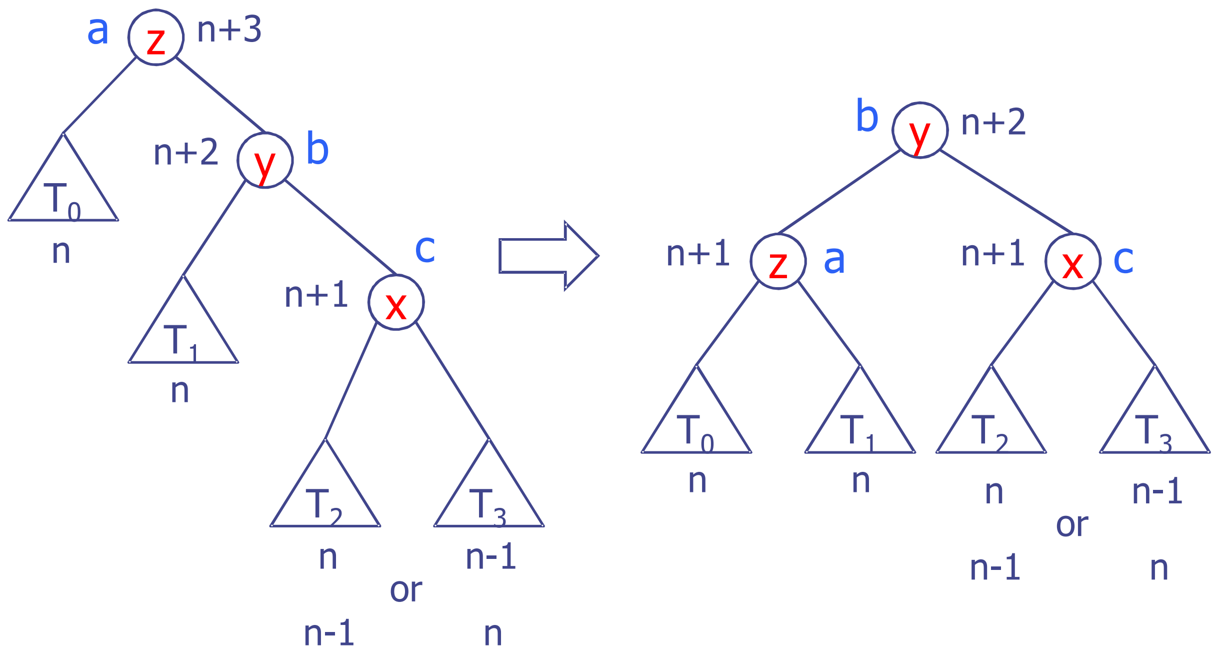





## Case II-1:
## Restructure ➔ Double Rotation Right-Left

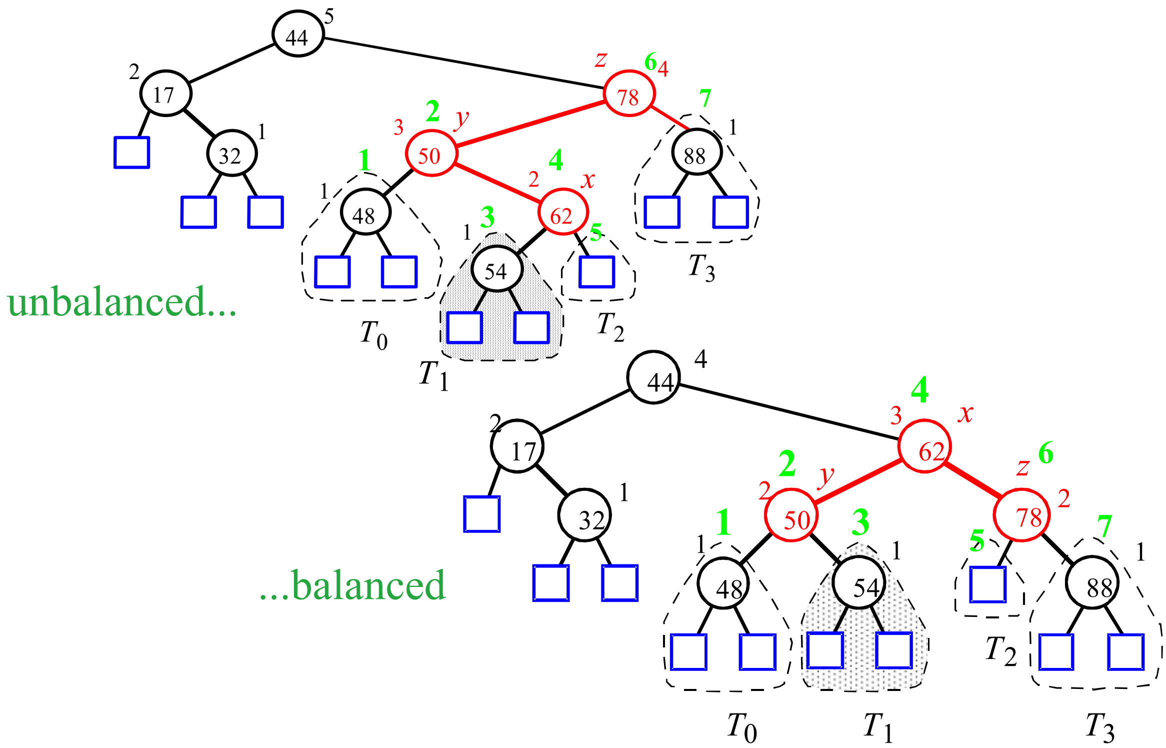


## Case II-2:
## Restructure ➔ Single Rotation Left



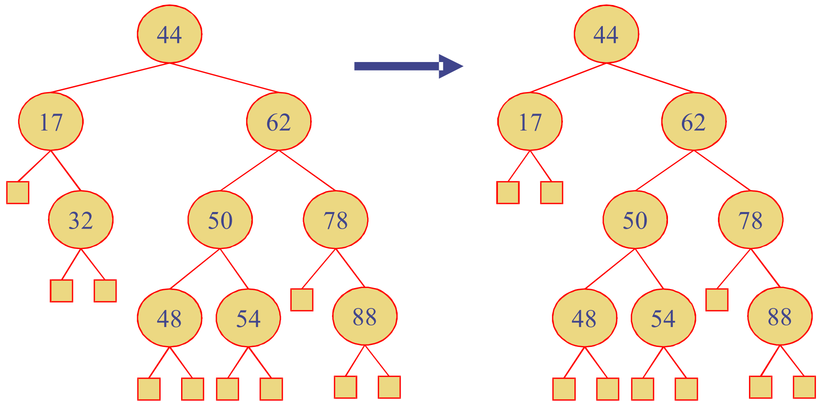



# Insertion Example (Cont'd)



# Removal

- Removal begins as in a binary search tree. However, its parent, w, may cause an imbalance.
- Example:

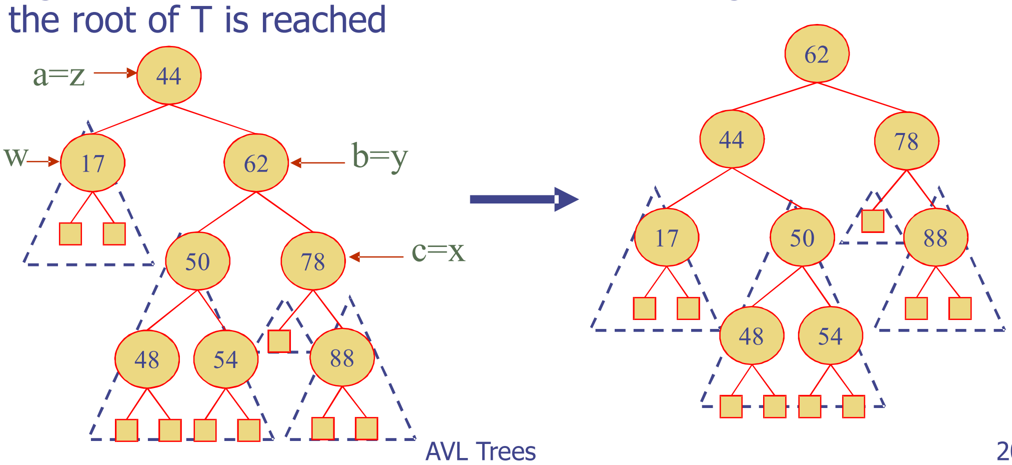



before deletion of 32 after deletion

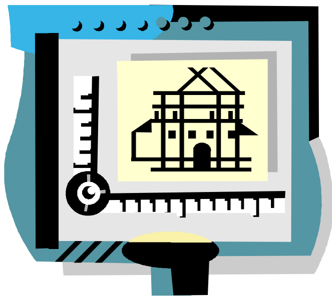


# Rebalancing after a Removal

- 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



# AVL Tree Performance



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