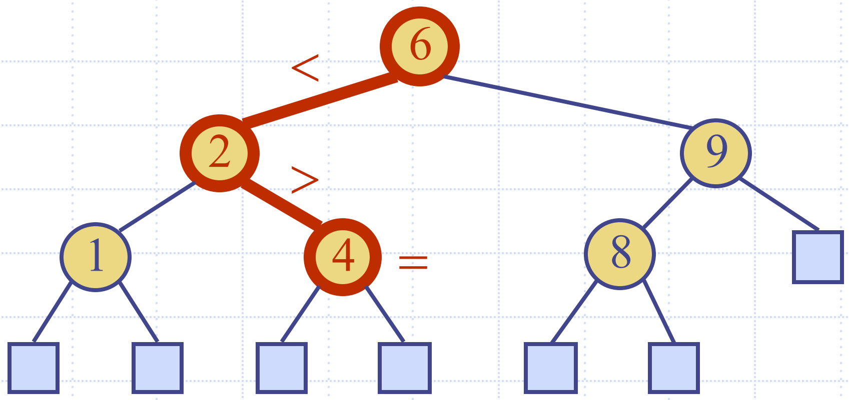
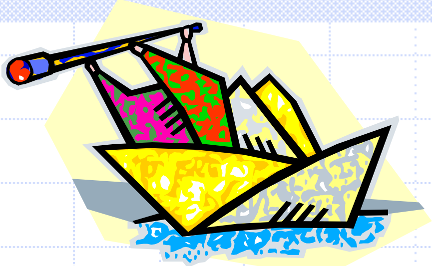
Binary Search Trees

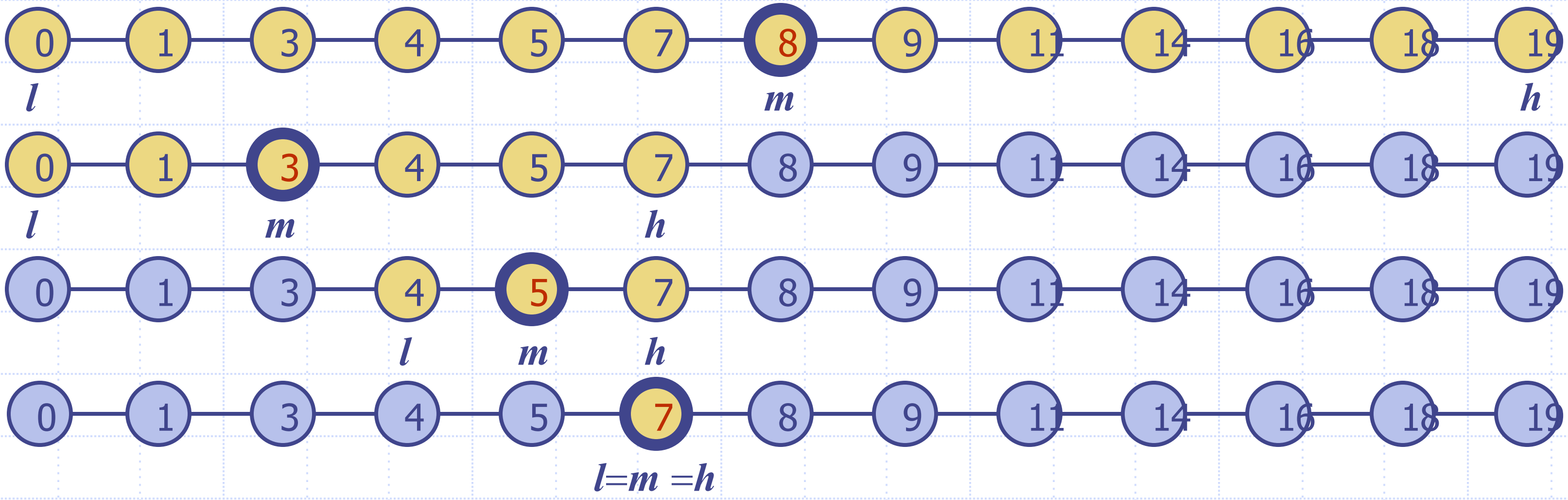
Chapter 11



Binary Search

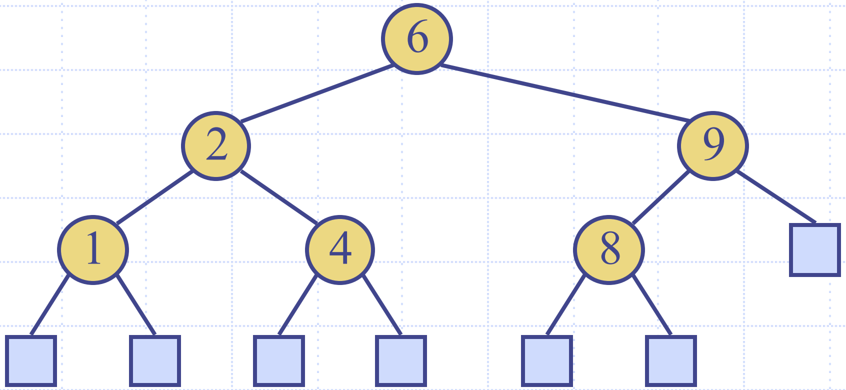


- Binary search can perform nearest neighbor queries on an ordered map that is implemented with an array, sorted by key
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: find(7)



Binary Search Trees

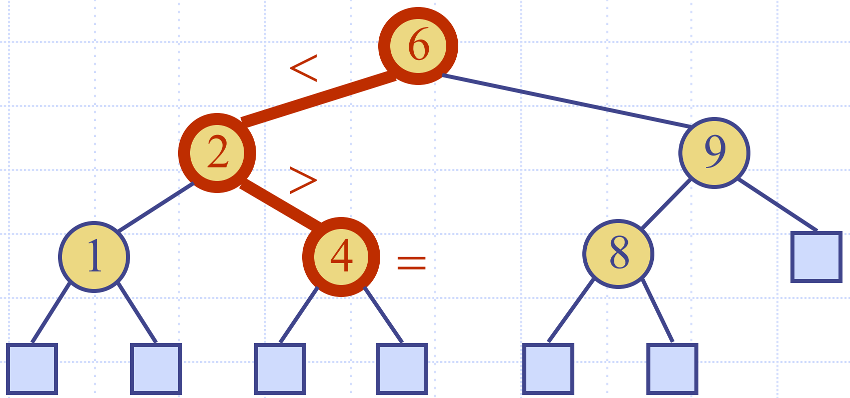
- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items
- An inorder traversal of a binary search tree visits the keys in increasing order



Search

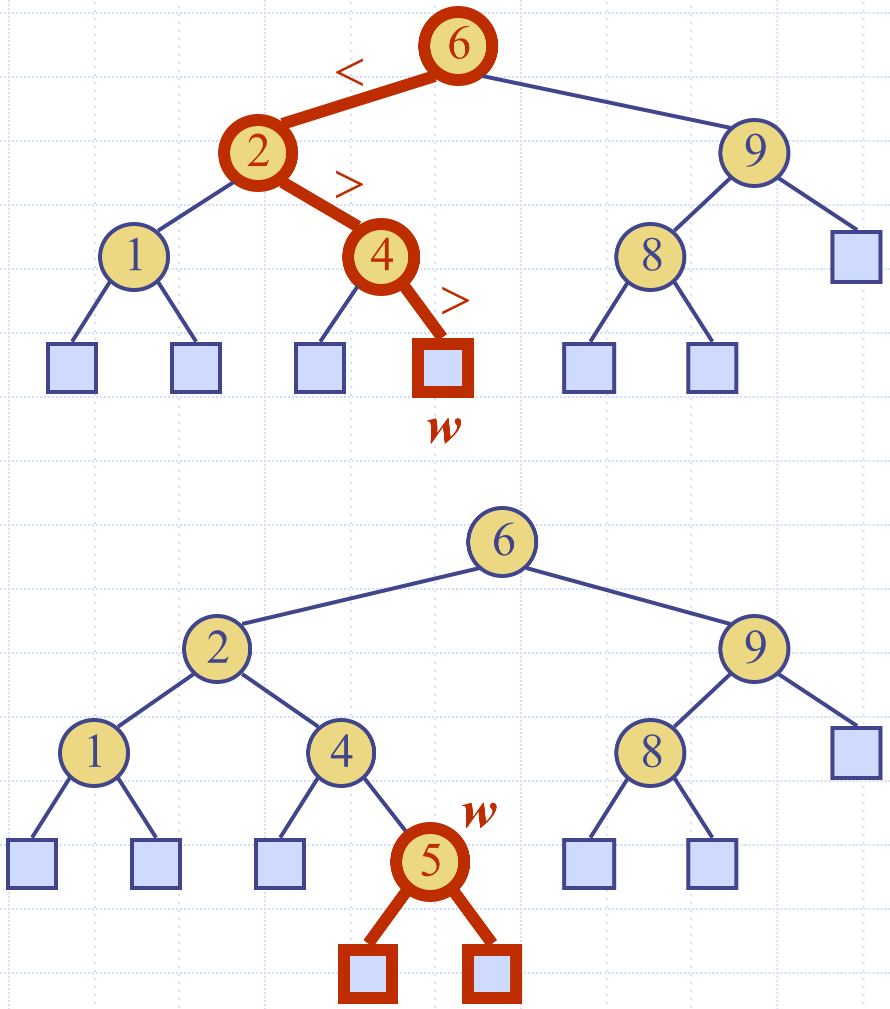
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of k with the key of the current node
- If we reach a leaf, the key is not found
- Example: get(4):
  - Call TreeSearch(4,root)

Algorithm *TreeSearch*(k, v)
  if *T.isExternal* (v)
    return v
if $k < key(v)$
  return *TreeSearch*(k, *left*(v))
else if $k = key(v)$
  return v
else { $k > key(v)$ }
  return *TreeSearch*(k, *right*(v))



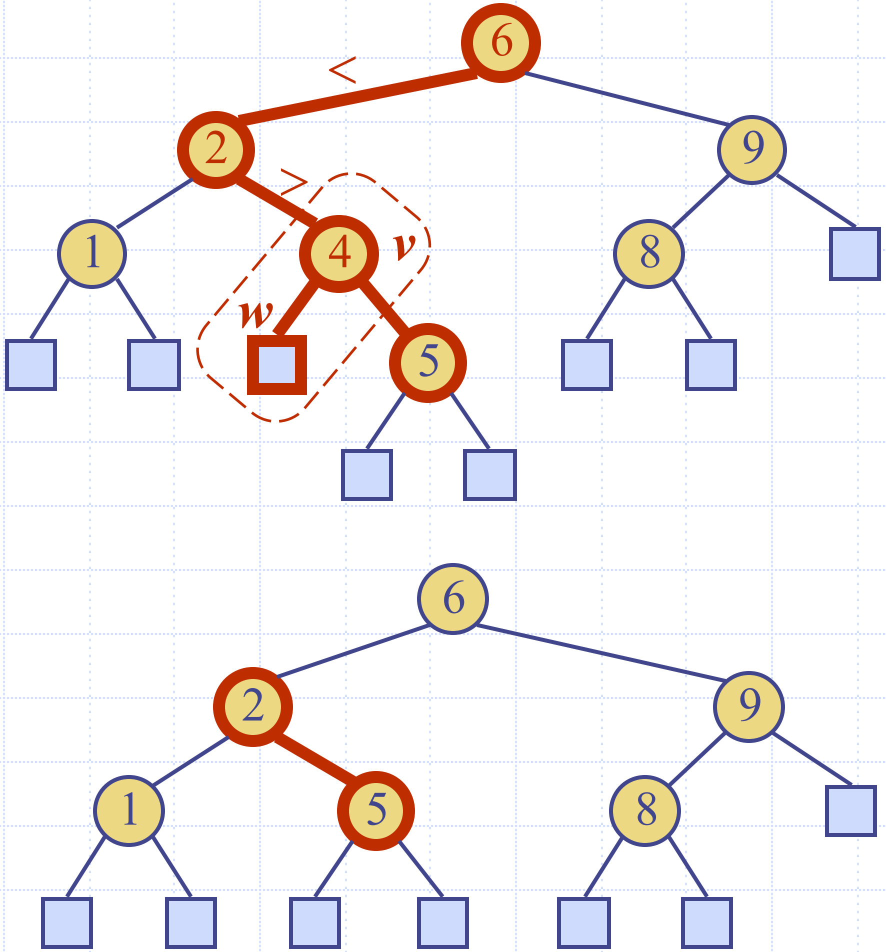
Insertion

- To perform operation put(k, o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5



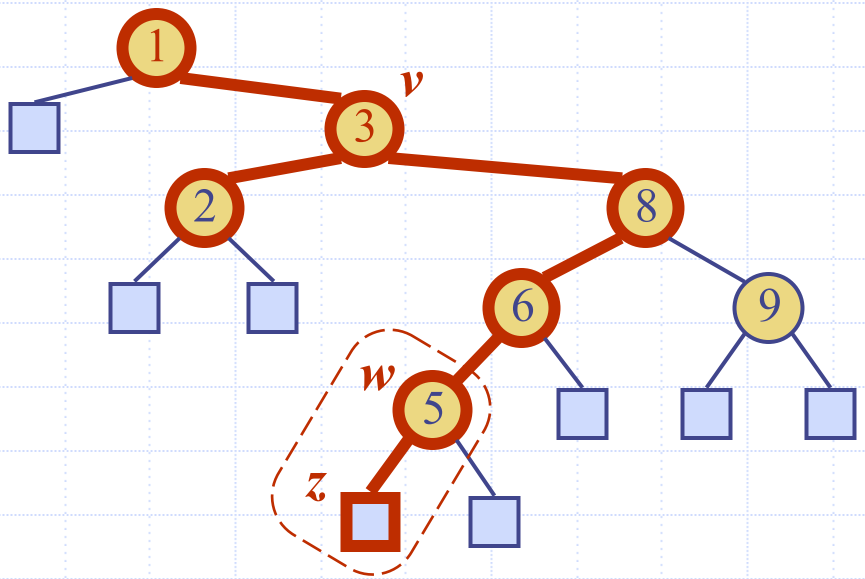
Deletion

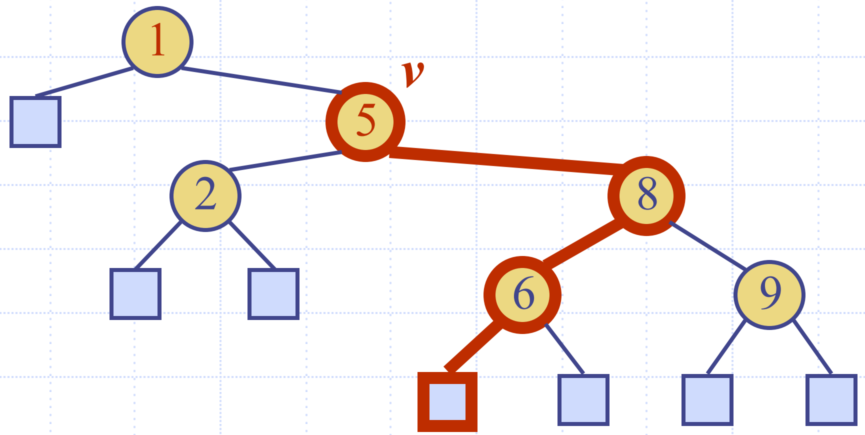
- To perform operation remove(k), we search for key k
- Assume key k is in the tree, and let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent
- Example: remove 4



Deletion (cont.)

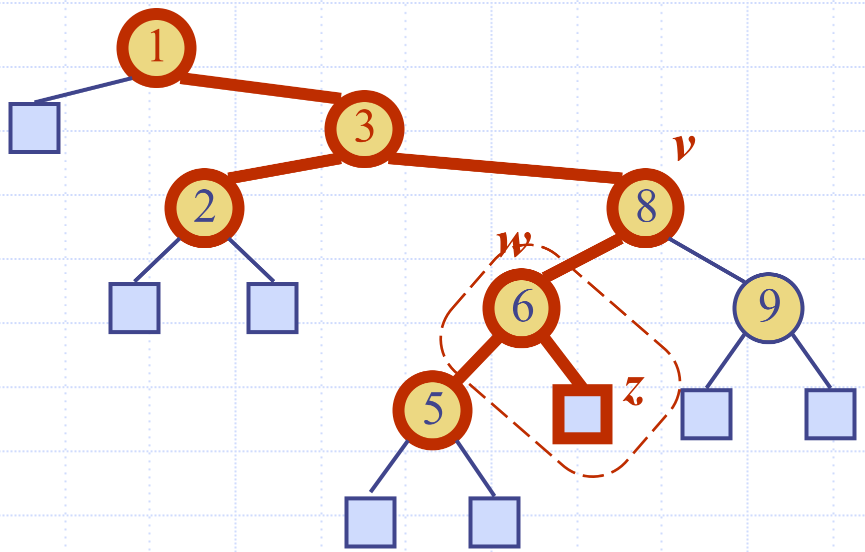
- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that follows v in an inorder traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)
- Example: remove 3

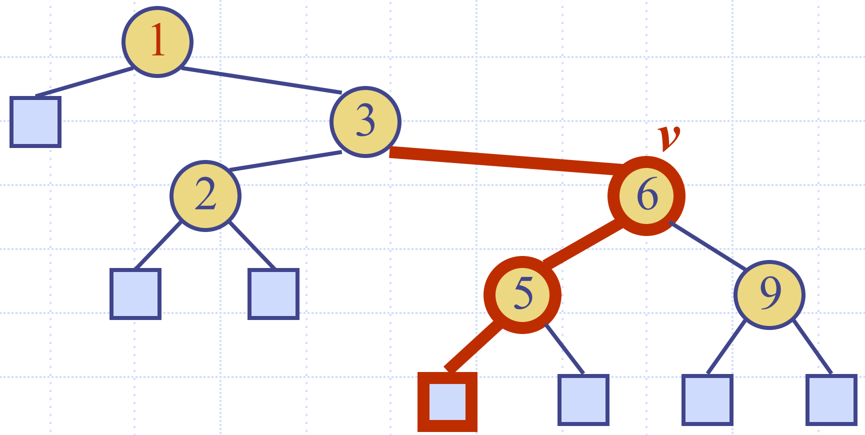




Deletion (cont.)

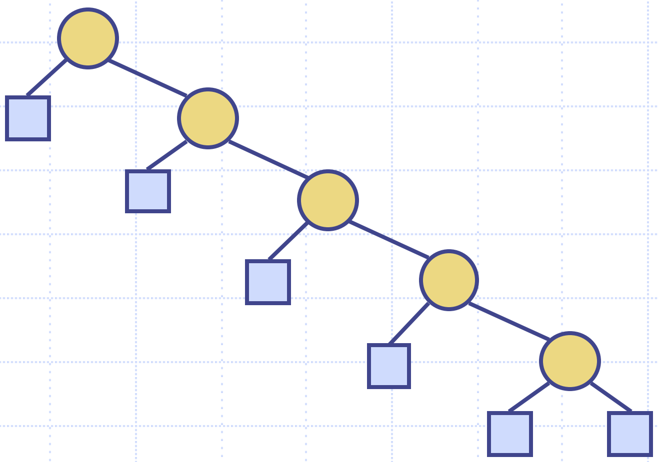
- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that precedes v in an inorder traversal
  - we copy $key(w)$ into node v
  - we remove node w and its right child z (which must be a leaf) by means of operation removeExternal(z)
- Example: remove 8

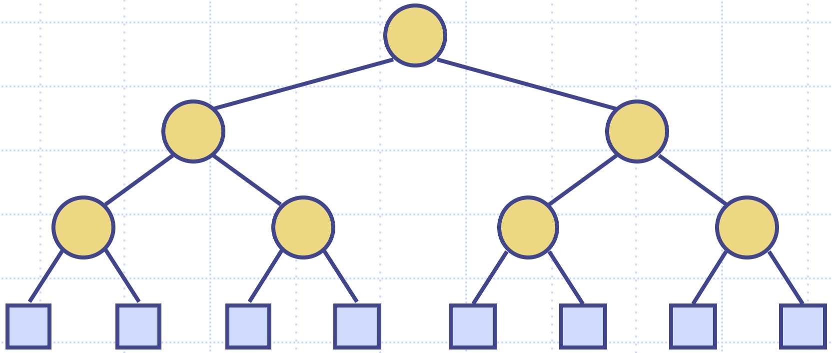




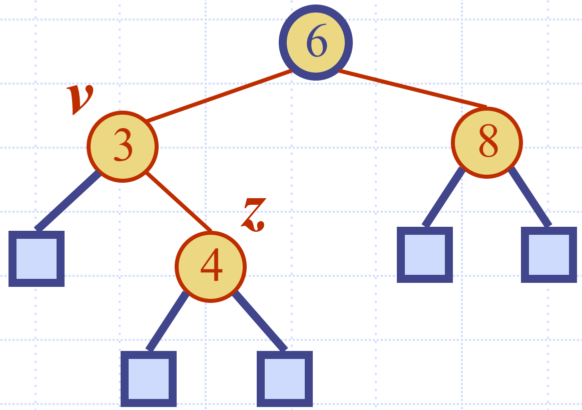
Performance

- Consider an ordered map with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods get, put and remove take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case
- On average, a binary tree generated from random insertions and removals of keys has expected height O(log n)



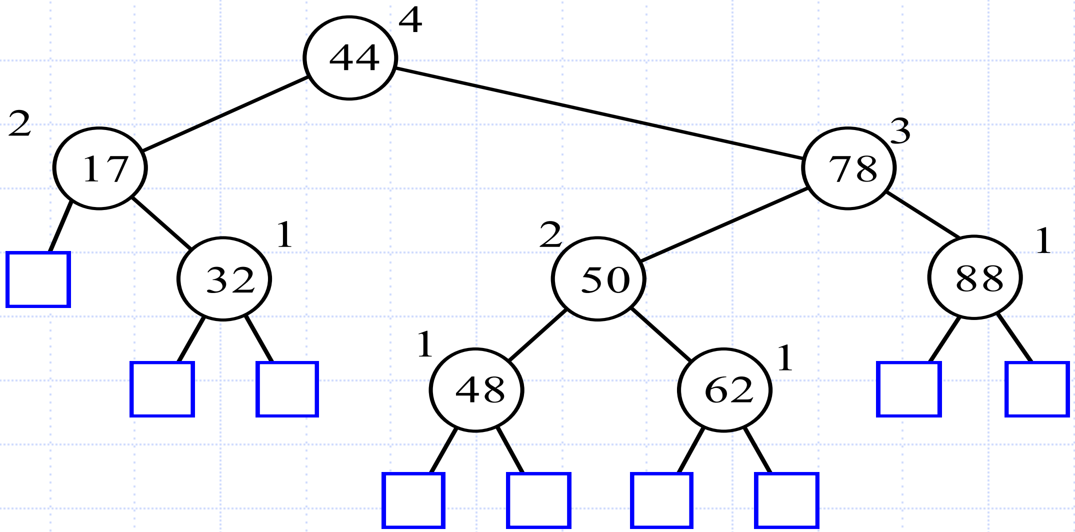


AVL Trees



AVL Tree Definition

- Inventors - Adel'son, Vel'skii and Landis
- 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
- Height-balance property



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

Height of an AVL Tree

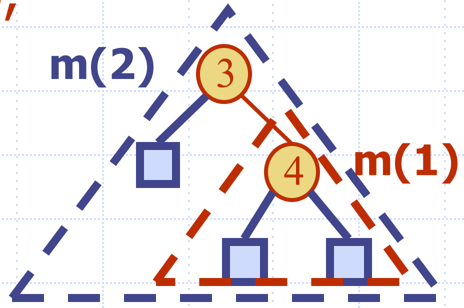
Fact: The height of an AVL tree storing n keys is O(log n).

Proof (by induction): Let us bound m(h): the minimum number of internal nodes of an AVL tree of height h.

- We easily see that m(1) = 1 and m(2) = 2
- For n > 1, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2 (or h-1, but will we pick h-1?).
- That is, m(h) = 1 + m(h-1) + m(h-2)
- Knowing m(h-1) $\geq$ m(h-2), we get m(h) > 2m(h-2). So

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

  m(h) > 2^im(h-2i)
- Solving the base case we get: m(h) > $2^{h-1/2}$
- Taking logarithms: h < 2log m(h)+1
- Thus, the height of an AVL tree is O(log n)



Exercise: Stricter bound- AVL tree with n nodes has height at most $\log_{1.63} n$

Structure of AVL Tree

- Consider an AVL tree on n nodes and the leaf that is closest to the root is at level k. Then
  - The height of the tree is at most 2k-1.

Structure of AVL Tree

- Consider an AVL tree on n nodes and the leaf that is closest to the root is at level k. Then
  - The height of the tree is at most 2k-1.
  - All nodes at levels 1,2,…, k-2 have 2 children.

Structure of AVL Tree

- Consider an AVL tree on n nodes and the leaf that is closest to the root is at level k. Then
  - The height of the tree is at most 2k-1.
  - All nodes at levels 1,2,…, k-2 have 2 children. (proof by contradiction)
    - all levels 1 to k-1 are full (all nodes are present)
    - Hence the tree has at least 2^{k-1} nodes

Structure of AVL Tree

- Consider an AVL tree on n nodes and the leaf that is closest to the root is at level k. Then
  - The height of the tree is at most 2k-1.
  - All nodes at levels 1,2,…, k-2 have 2 children. (proof by contradiction)
    - all levels 1 to k-1 are full (all nodes are present)
    - Hence the tree has at least 2^{k-1} nodes

Structure of AVL Tree

- Consider an AVL tree on n nodes and the leaf that is closest to the root is at level k. Then
  - The height of the tree is at most 2k-1.
    - hence tree has at most 2^{2k-1} nodes.
  - All nodes at levels 1,2,…, k-2 have 2 children.
    - all levels 1 to k-1 are full
    - hence tree has at least 2^{k-1} nodes

Structure of AVL Tree

- Consider an AVL tree on n nodes and the leaf that is closest to the root is at level k. Then
  - The height of the tree is at most 2k-1.
    - hence tree has at most 2^{2k-1} nodes.
  - All nodes at levels 1,2,…, k-2 have 2 children.
    - all levels 1 to k-1 are full
    - hence tree has at least 2^{k-1} nodes
  - Thus $2^{k-1} \leq n \leq 2^{2k-1}$

Structure of AVL Tree

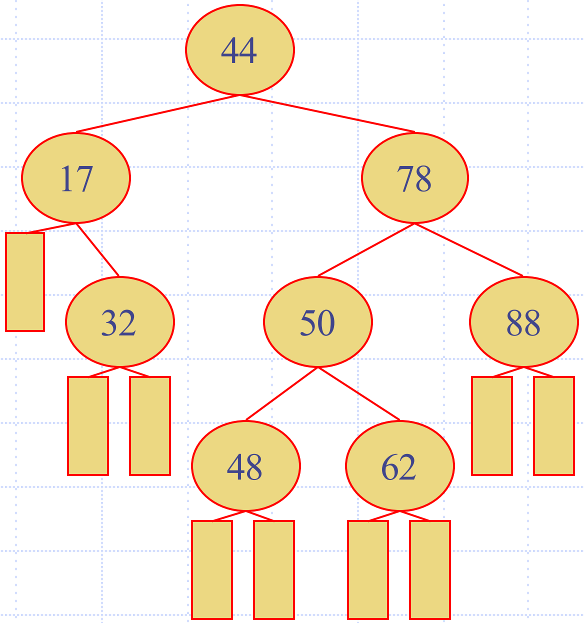
- Consider an AVL tree on n nodes and the leaf that is closest to the root is at level k. Then
  - The height of the tree is at most 2k-1.
    - hence tree has at most 2^{2k-1} nodes.
  - All nodes at levels 1,2,…, k-2 have 2 children.
    - all levels 1 to k-1 are full
    - hence tree has at least 2^{k-1} nodes
  - Thus $2^{k-1} \leq n \leq 2^{2k-1}$
- Substituting h = 2k-1, we get $2^{(h-1)/2} \leq n \leq 2^h$

Summary - Structure of AVL Tree

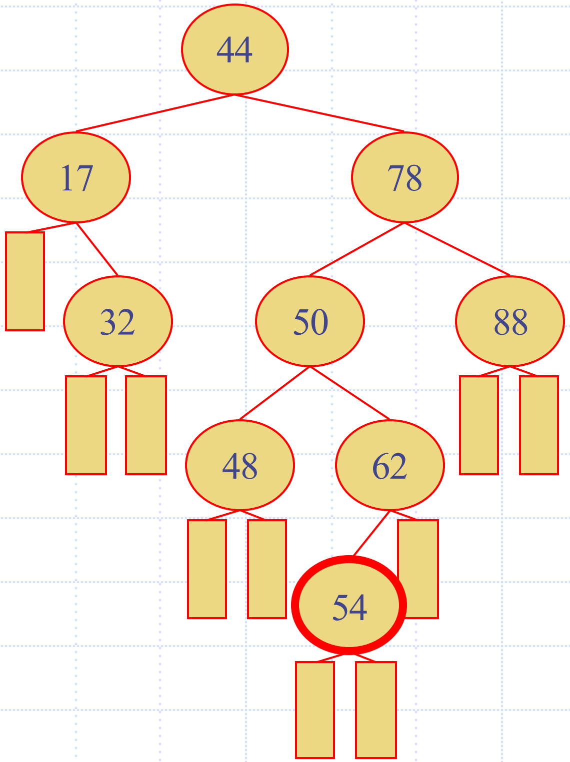
- In an AVL tree of height h, the leaf closest to the root is at level (h+1)/2
- On the first (h-1)/2 levels, the AVL tree is a complete binary tree
  - thins out after (h-1)/2 level
- $2^{(h-1)/2} \leq$ number of nodes $\leq 2^h$

Insertion

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



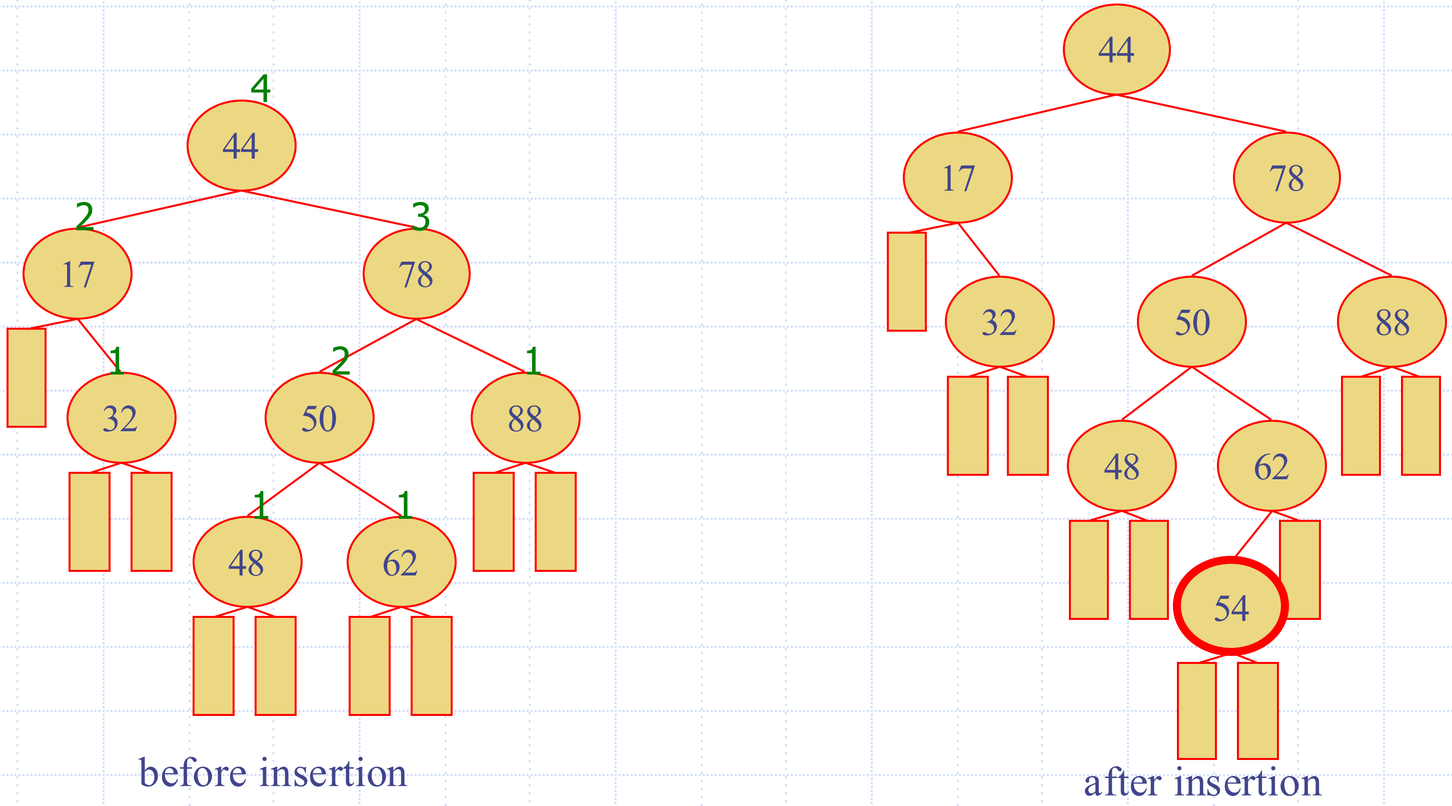
before insertion



after insertion

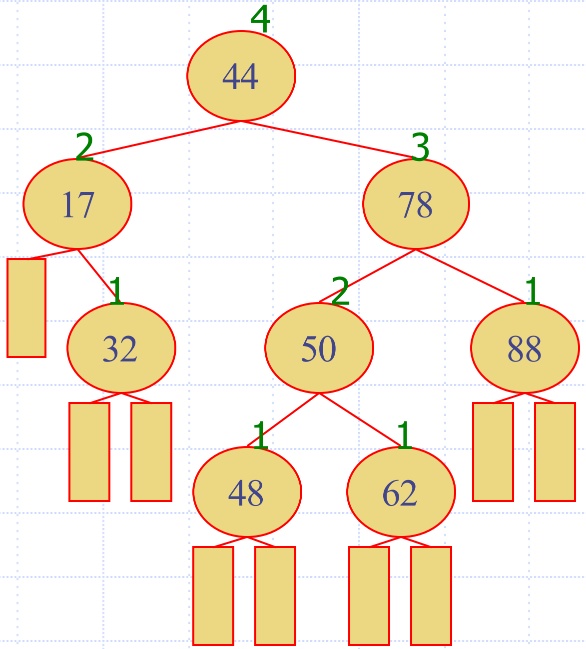
Insertion

- Inserting a node into an AVL tree changes the height of some of the nodes in T.

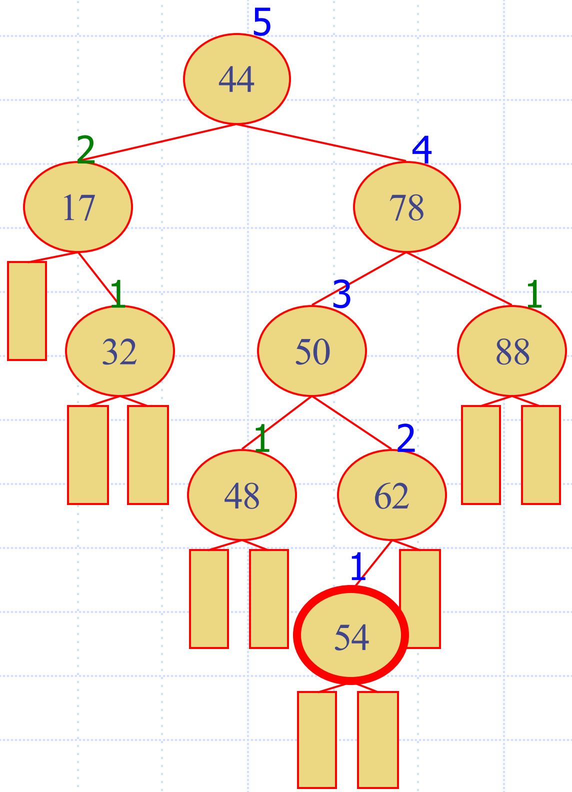


Insertion

- Inserting a node into an AVL tree changes the height of some of the nodes in T.



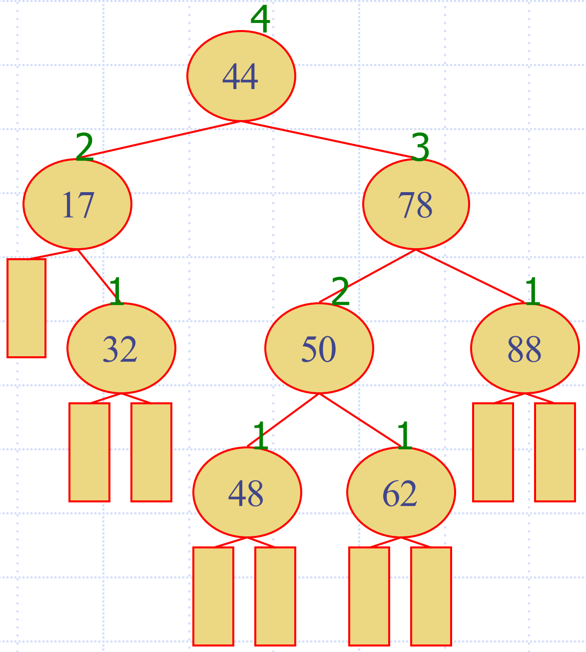
before insertion



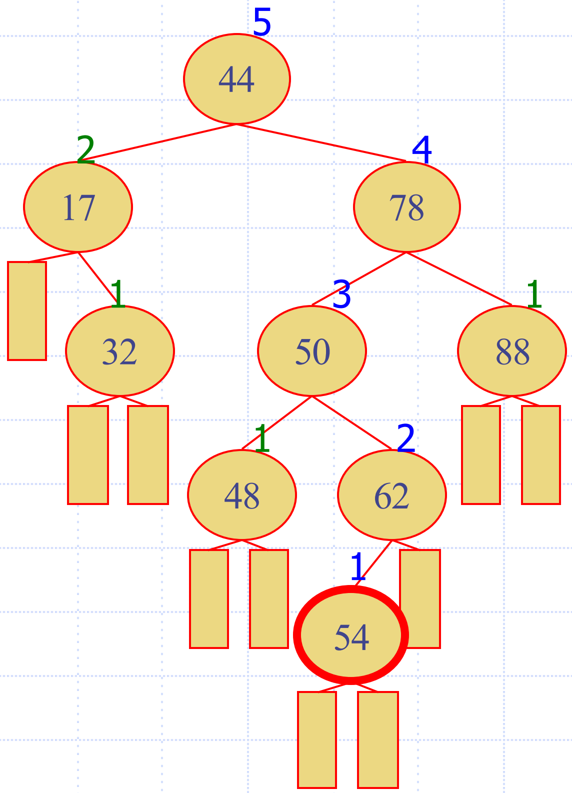
after insertion

Insertion

- Inserting a node into an AVL tree changes the height of some of the nodes in T.
- The only nodes whose heights can increase are the ancestors of inserted node.



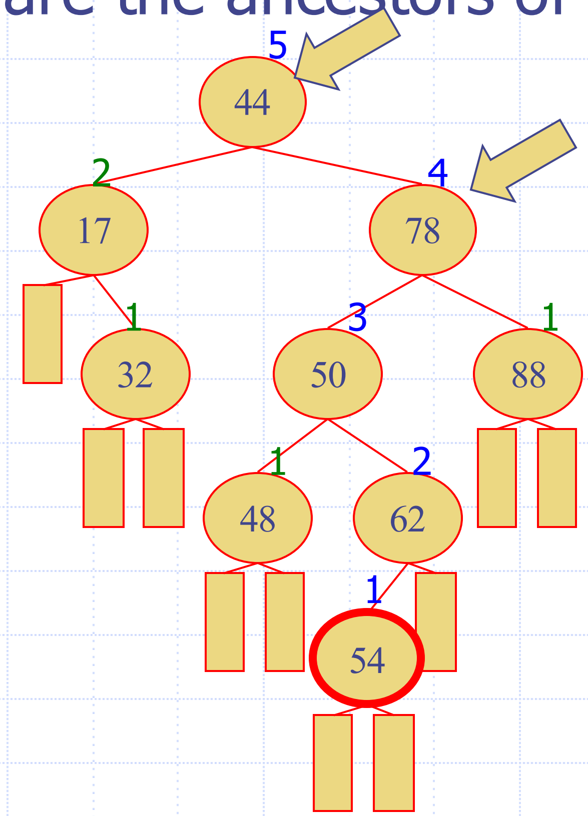
before insertion



after insertion

Insertion

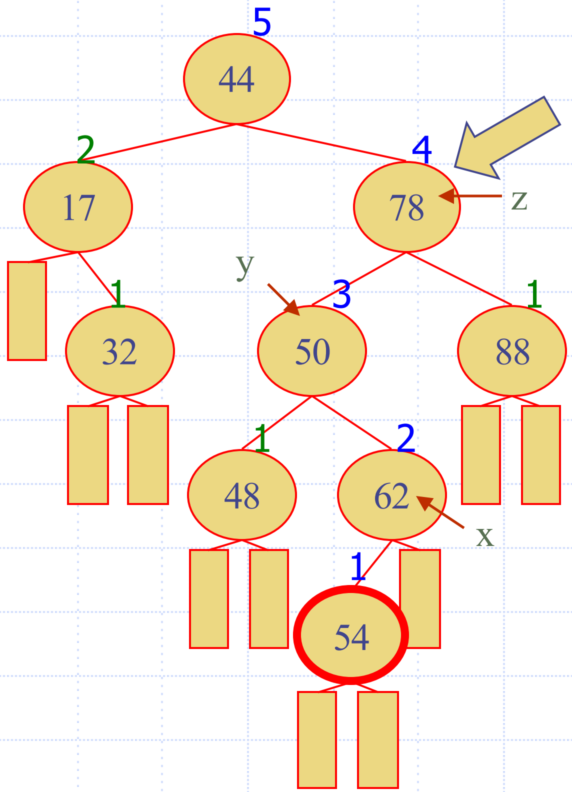
- Inserting a node into an AVL tree changes the height of some of the nodes in T.
- The only nodes whose heights can increase are the ancestors of inserted node.
- If the insertion causes T to become unbalanced, then some ancestor of the inserted node would have a height imbalance.



after insertion

Insertion

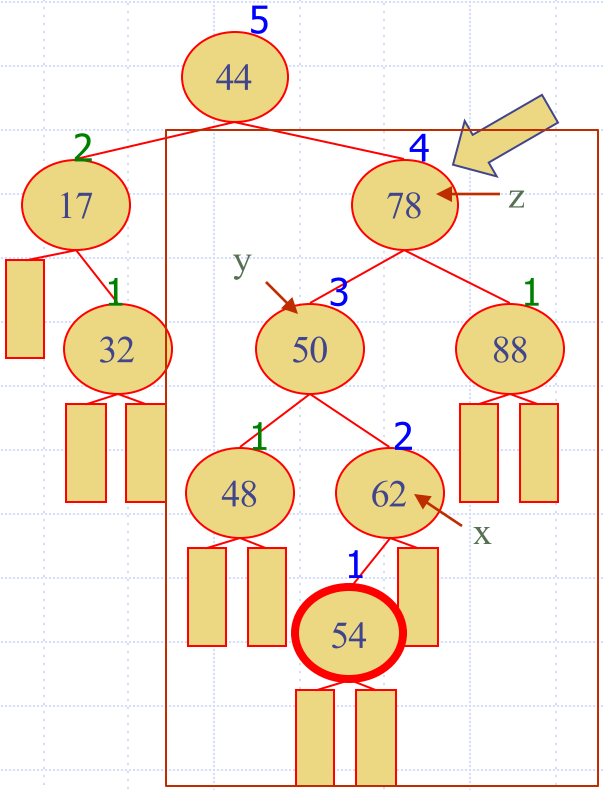
- Inserting a node into an AVL tree changes the height of some of the nodes in T.
- The only nodes whose heights can increase are the ancestors of inserted node.
- If the insertion causes T to become unbalanced, then some ancestor of the inserted node would have a height imbalance.
- We travel up the tree from the inserted node (v), until we find the first node (x) such that its grandparent (z) is unbalanced.
- Let y be the parent of node x.



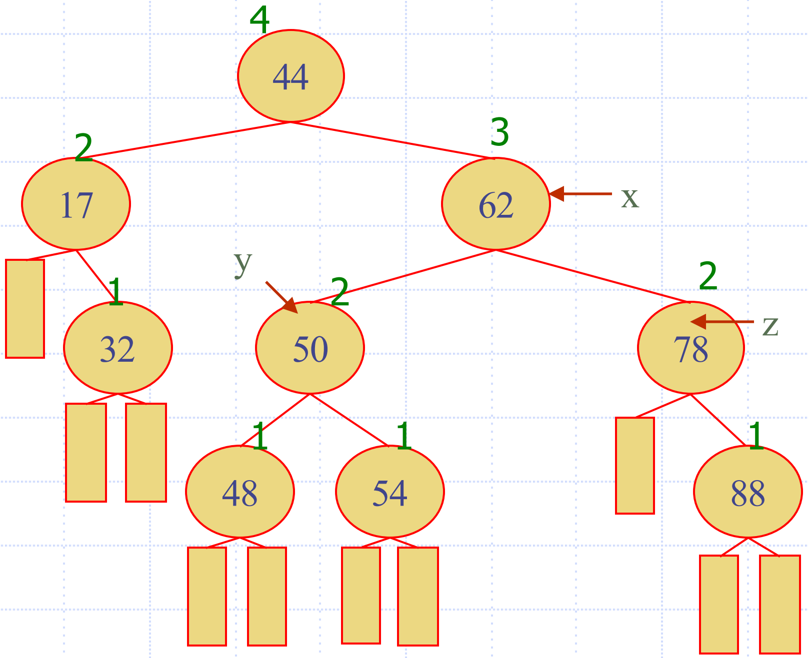
after insertion

Insertion

- To rebalance the subtree rooted at z, we must perform a rotation.



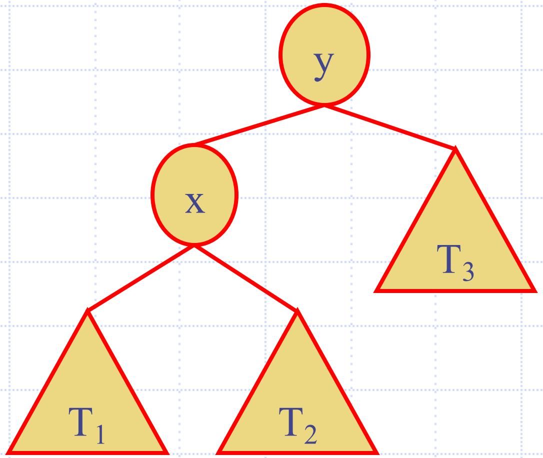
after insertion



after rebalancing

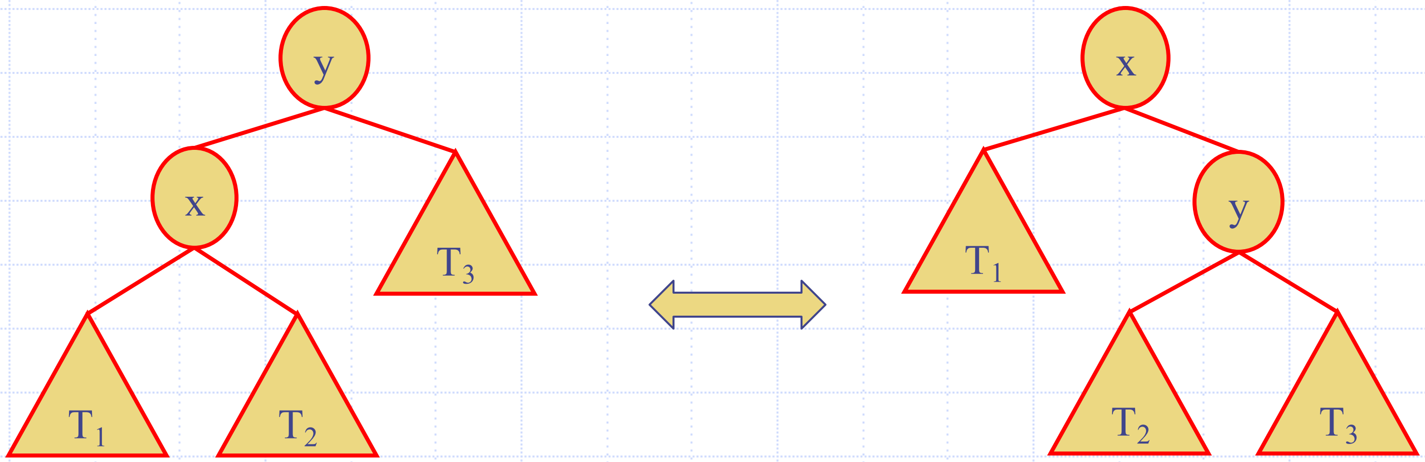
Rotations

- Rotation is a way of locally reorganizing a BST.
- Let x, y be two nodes such that y=parent(x).
- $Keys(T_1)<Key(x)<Keys(T_2)<Key(y)<Keys(T_3)$



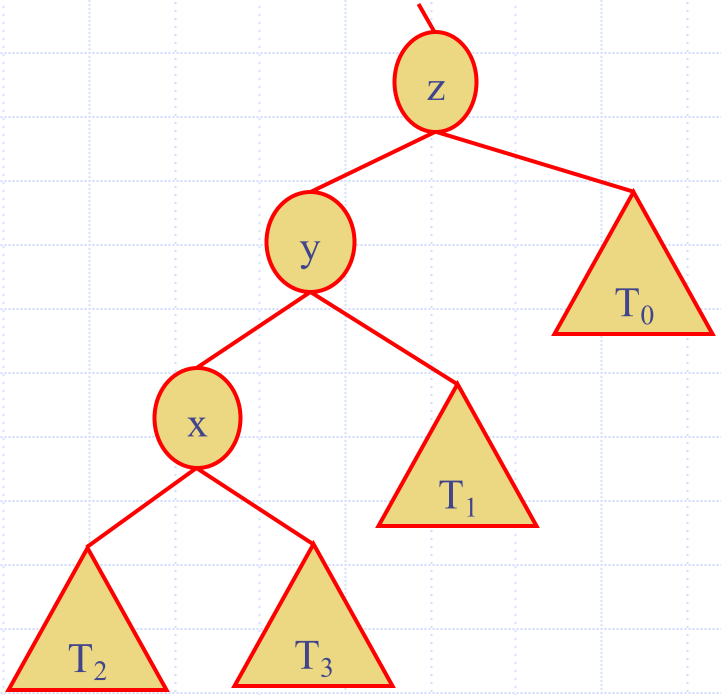
Rotations

- Rotation is a way of locally reorganizing a BST.
- Let x, y be two nodes such that y=parent(x).
- $Keys(T_1)<Key(x)<Keys(T_2)<Key(y)<Keys(T_3)$



Restructuring – Single Rotation

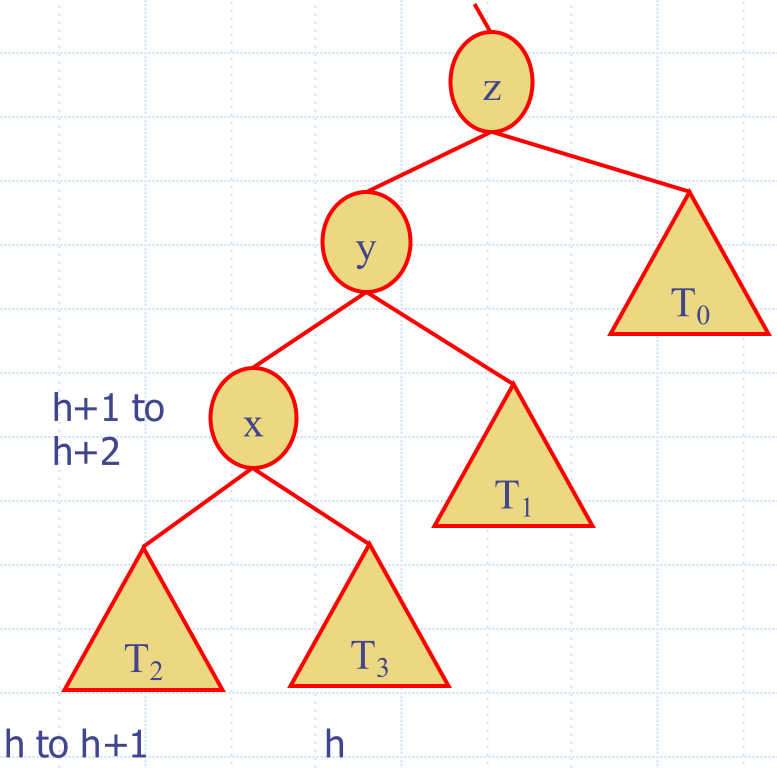
- Rotations to make y the top-most node



- Insertion happens in subtree T_2
- $H(T_2)$ increases from h to h+1
- x remains balanced
  - $H(T_3)$ = h or h+1 or h+2
    - h+2 – then x is originally unbalanced
    - h+1 – then height of x does not increase – z is balanced
    - therefore $H(T_3)$ = h
- H(x) increases from h+1 to h+2

Restructuring – Single Rotation

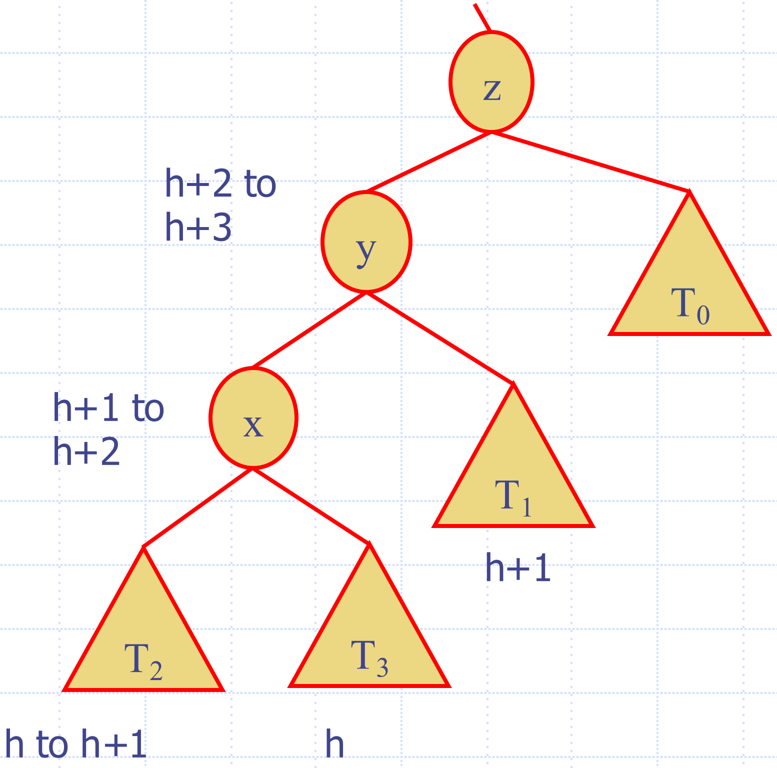
- Rotations to make y the top-most node



- Insertion happens in subtree T_2
- y remains balanced
  - $H(T_1)$ = h+1 or h+2 or h+3
    - h+3 – y is originally unbalanced
    - h+2 – height of y does not increase – z is balanced
    - therefore $H(T_1)$ = h+1
- H(y) = h+2 to h+3.

Restructuring – Single Rotation

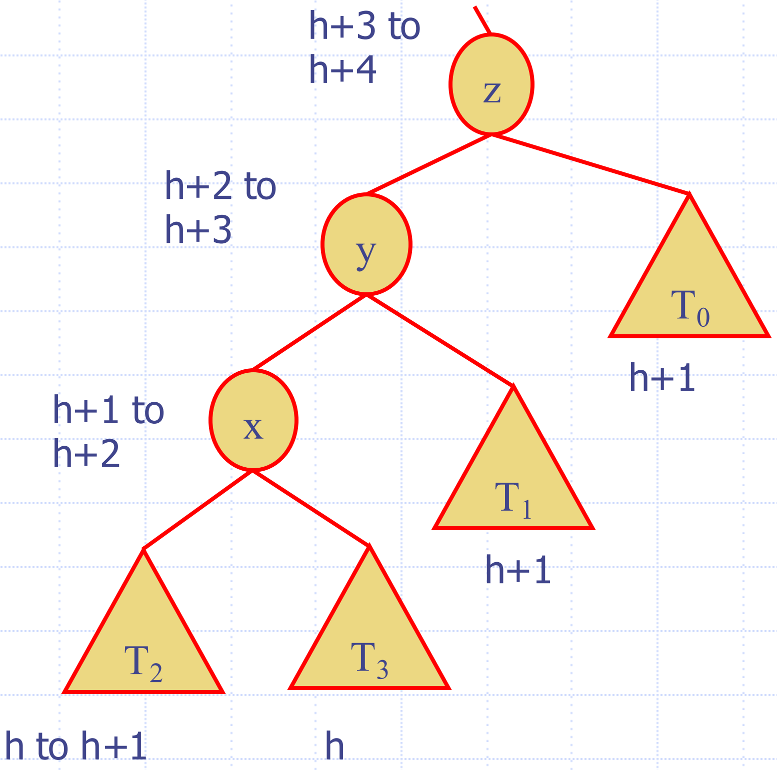
- Rotations to make y the top-most node



- Insertion happens in subtree T_2
- y remains balanced
  - $H(T_1)$ = h+1 or h+2 or h+3
    - h+3 – y is originally unbalanced
    - h+2 – height of y does not increase – z is balanced
    - therefore $H(T_1) = h+1$
- H(y) = h+2 to h+3.
- z is unbalanced
  - $H(T_0)$ = h+1 or h+2 or h+3
    - Since originally z was balanced
    - $H(T_0) = h+1$
- H(z) = h+3 to h+4

Restructuring – Single Rotation

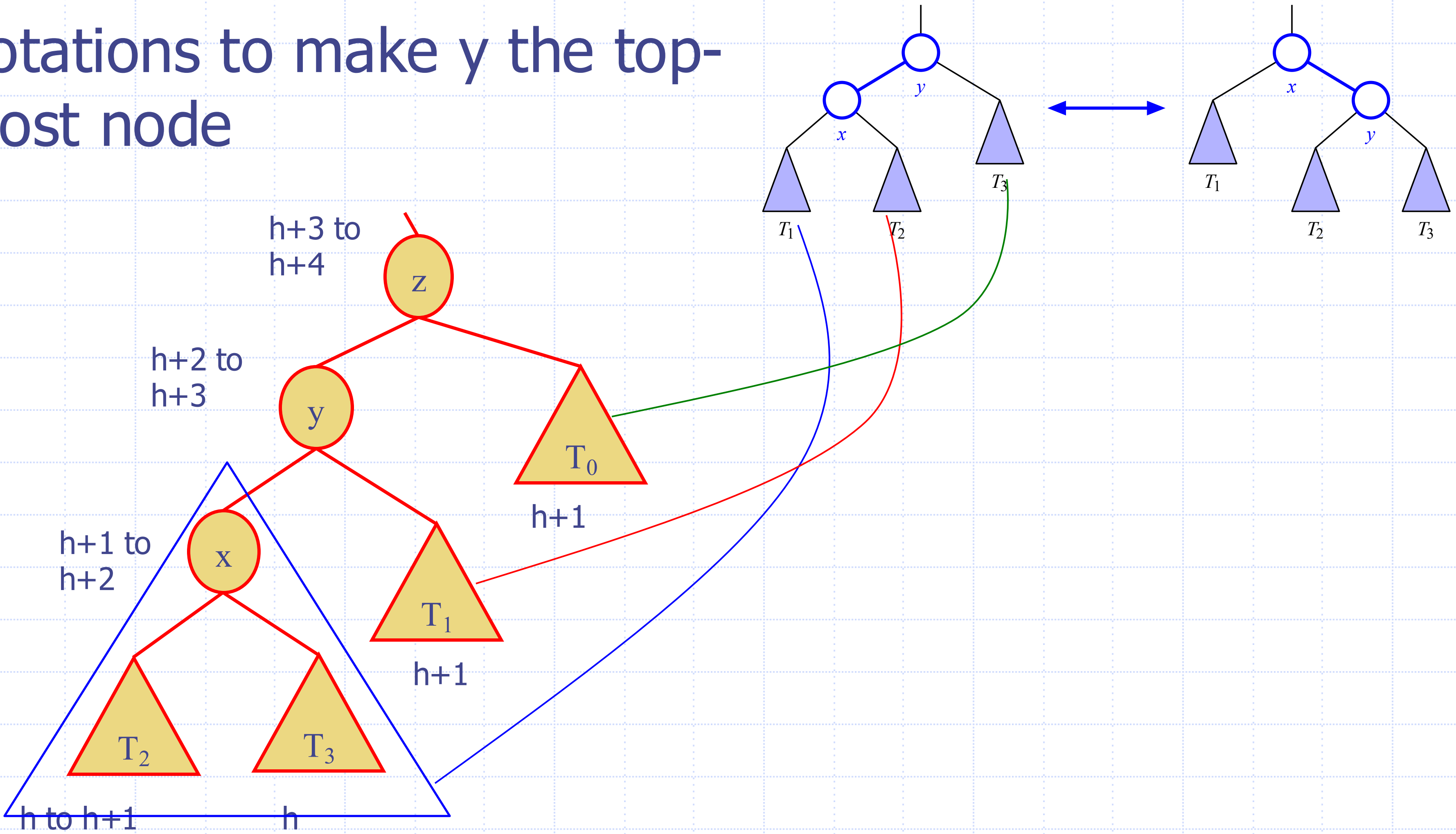
- Rotations to make y the top-most node



- Insertion happens in subtree T_2
- y remains balanced
  - $H(T_1)$ = h+1 or h+2 or h+3
    - h+3 – y is originally unbalanced
    - h+2 – height of y does not increase – z is balanced
    - therefore $H(T_1)$ = h+1
- H(y) = h+2 to h+3.
- z is unbalanced
  - $H(T_0)$ = h+1 or h+2 or h+3
    - Since originally z was balanced
    - $H(T_0)$ = h+1
- H(z) = h+3 to h+4

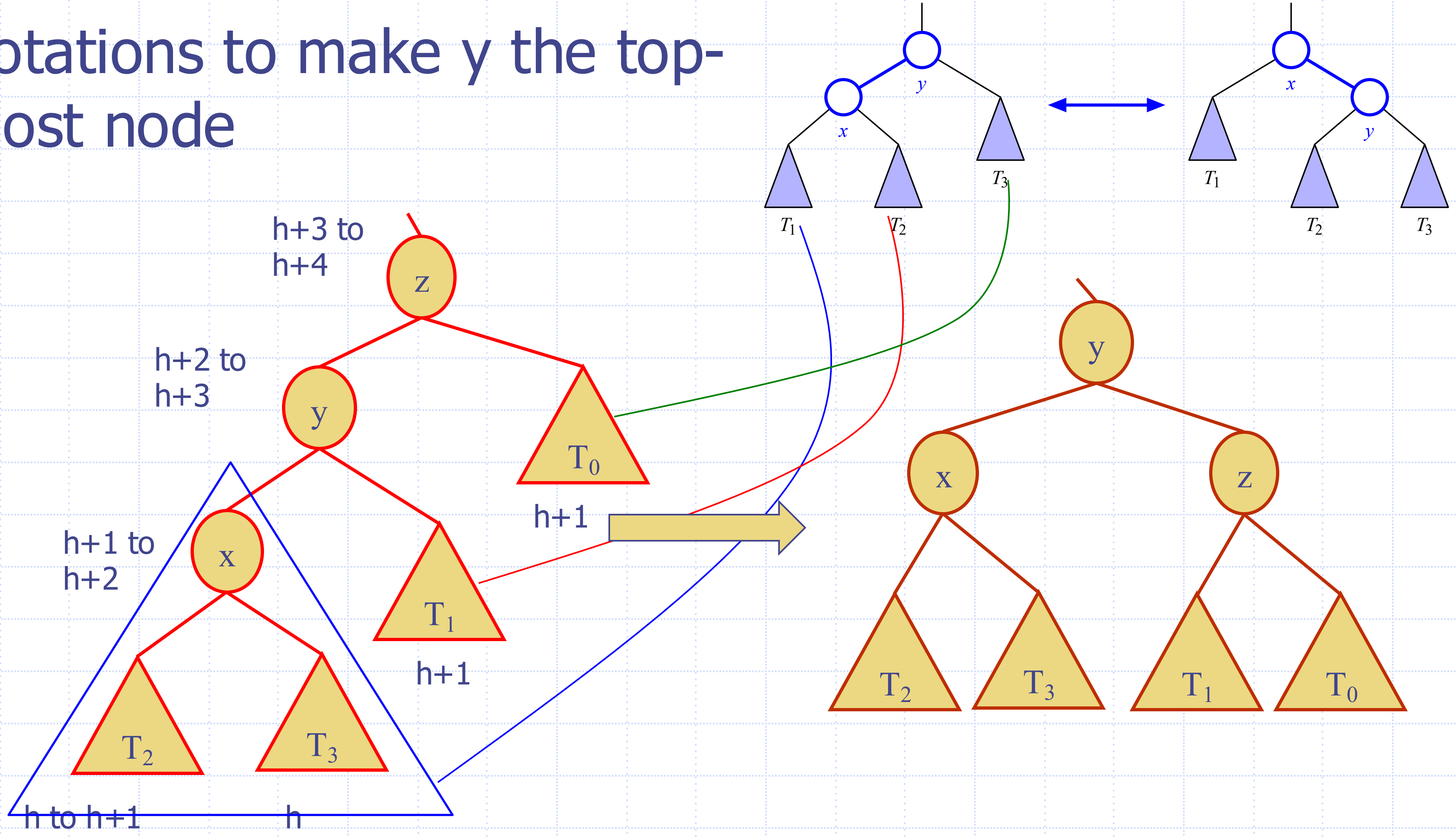
Restructuring – Single Rotation

- Rotations to make y the top-most node



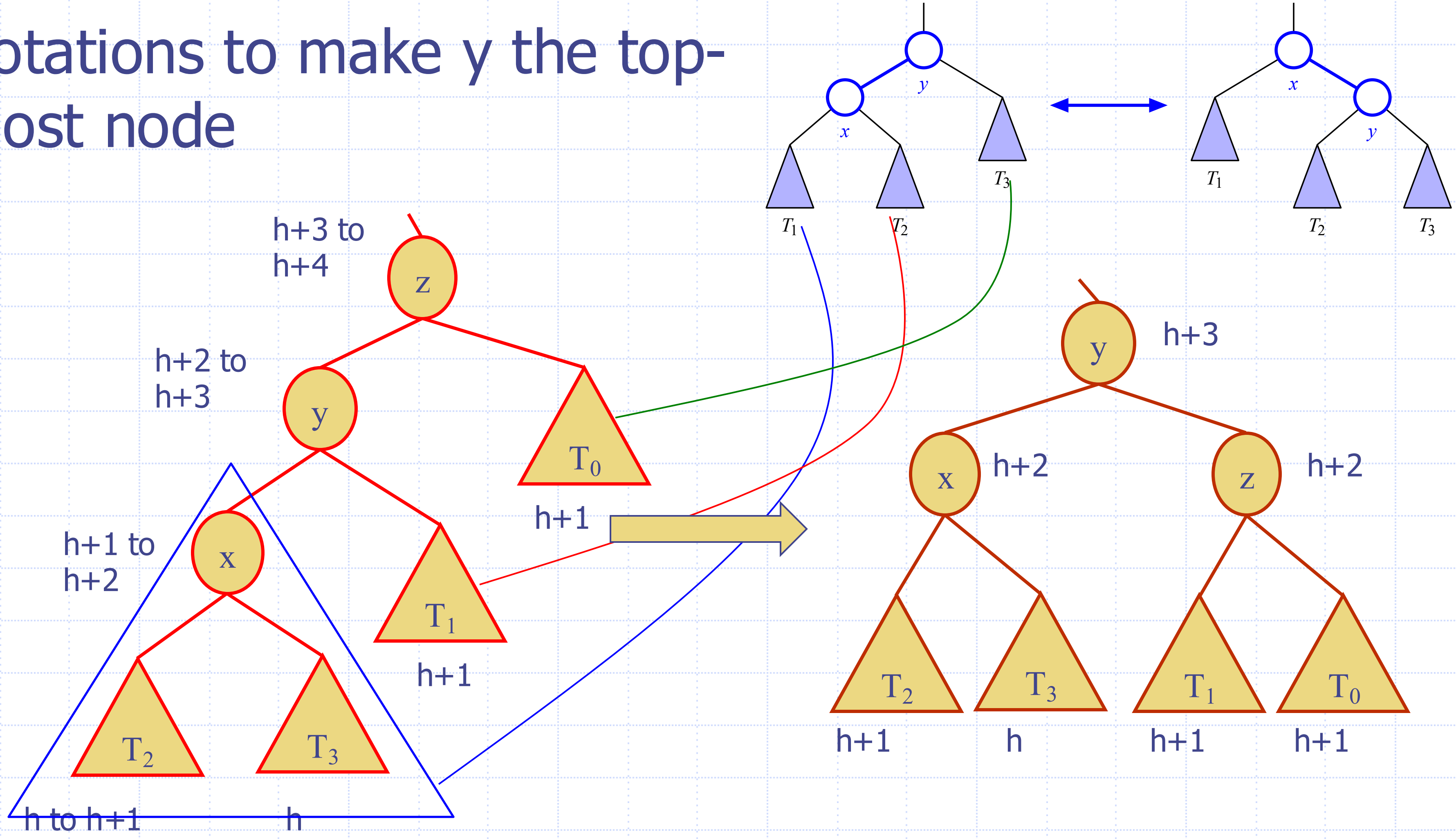
Restructuring – Single Rotation

- Rotations to make y the top-most node



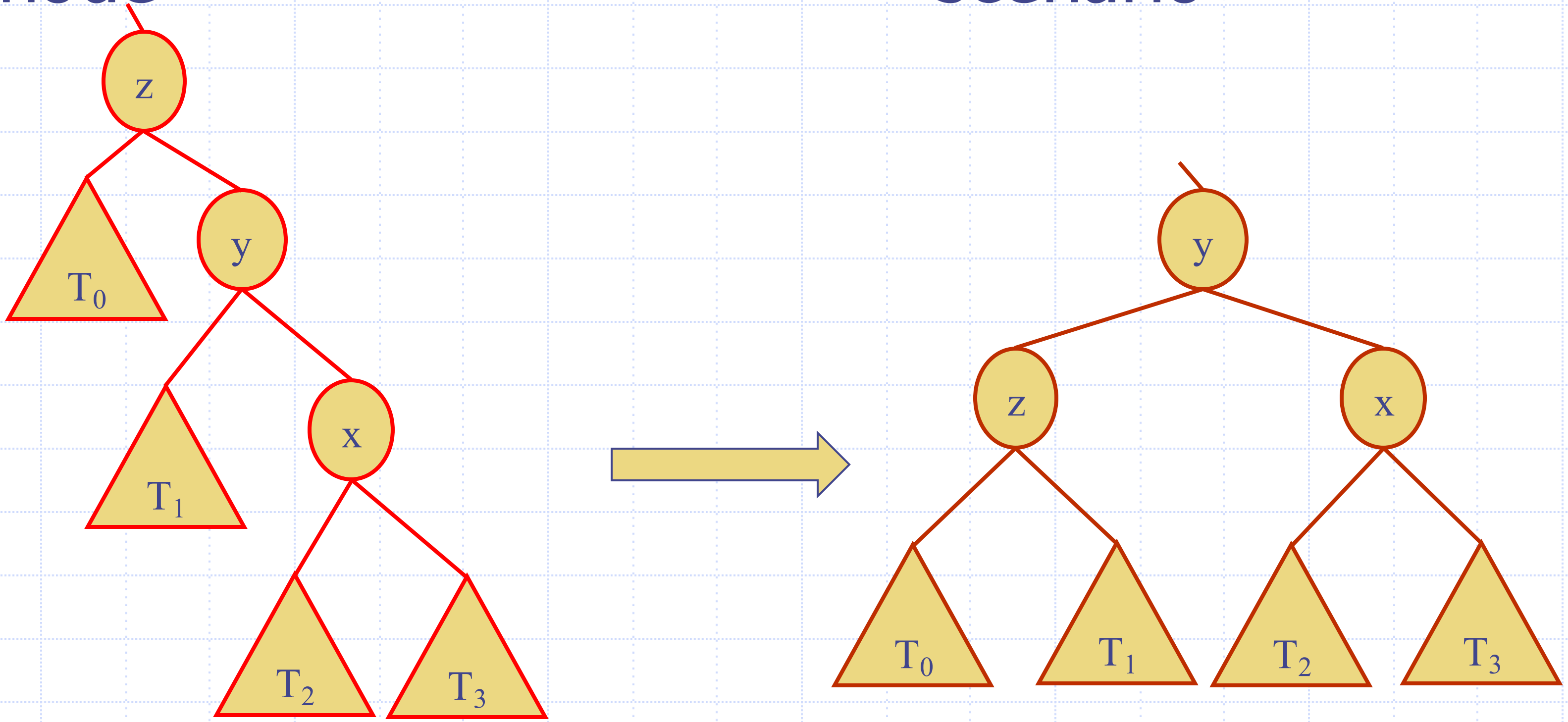
Restructuring – Single Rotation

- Rotations to make y the top-most node



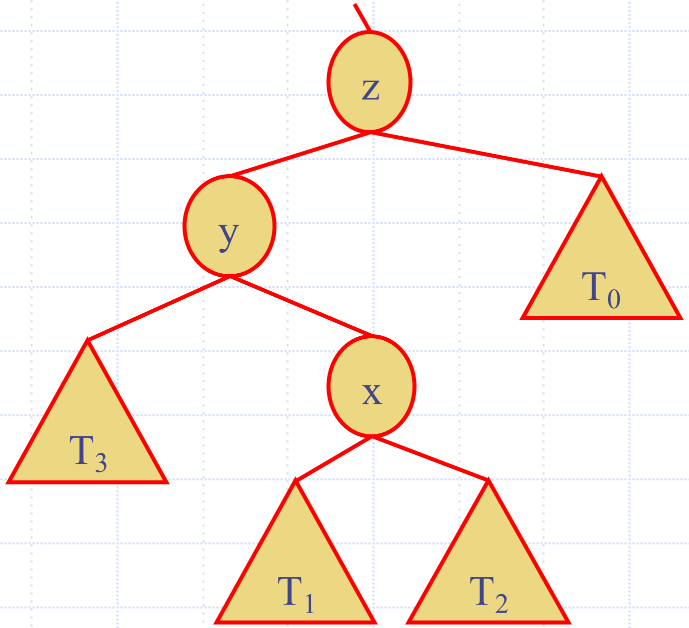
Restructuring – Single Rotation

- Rotations to make y the top-most node
- Symmetric to the previous scenario



Restructuring – Double Rotation

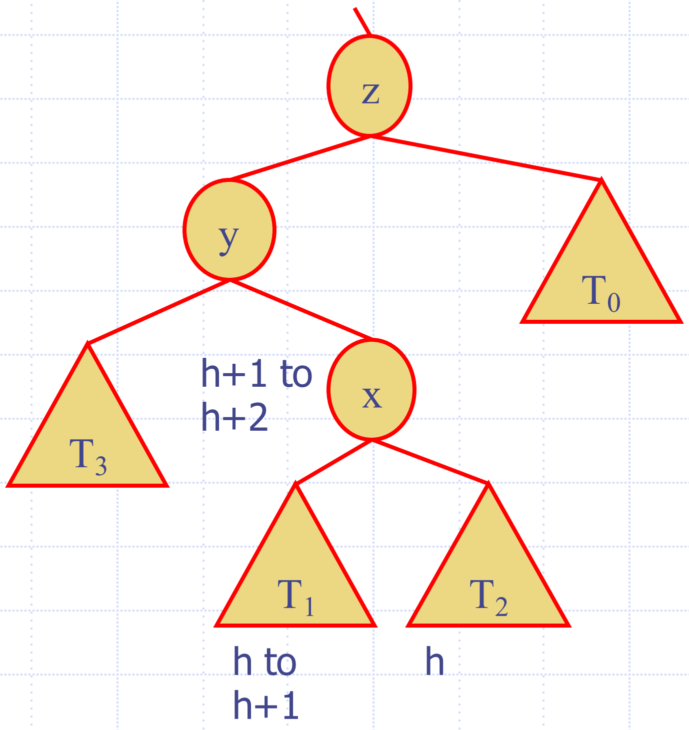
- Perform rotations around y and z to make x the top-most node.



- Insertion happens in subtree T_1
- $H(T_1) = h$ to $h+1$
- x is balanced
  - $H(T_2) = h$ or $h+1$ or $h+2$
    - $h+2$ – x is originally unbalanced
    - $h+1$ – no increase in height of x – z remains balanced
    - therefore $H(T_2) = h$
- $H(x) = h+1$ to $h+2$

Restructuring – Double Rotation

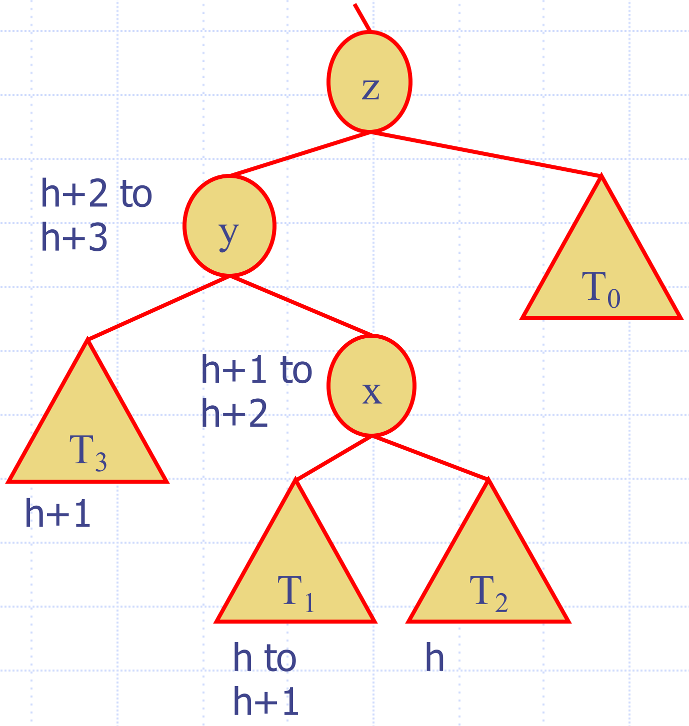
- Perform rotations around y and z to make x the top-most node.



- Insertion happens in subtree T_1
- y remains balanced
  - $H(T_3)$ = h+1 or h+2 or h+3
    - h+3 – y is originally unbalanced
    - h+2 – no increase in height of y – z remains balanced
    - therefore $H(T_3)$ = h+1
- H(y) = h+2 to h+3

Restructuring – Double Rotation

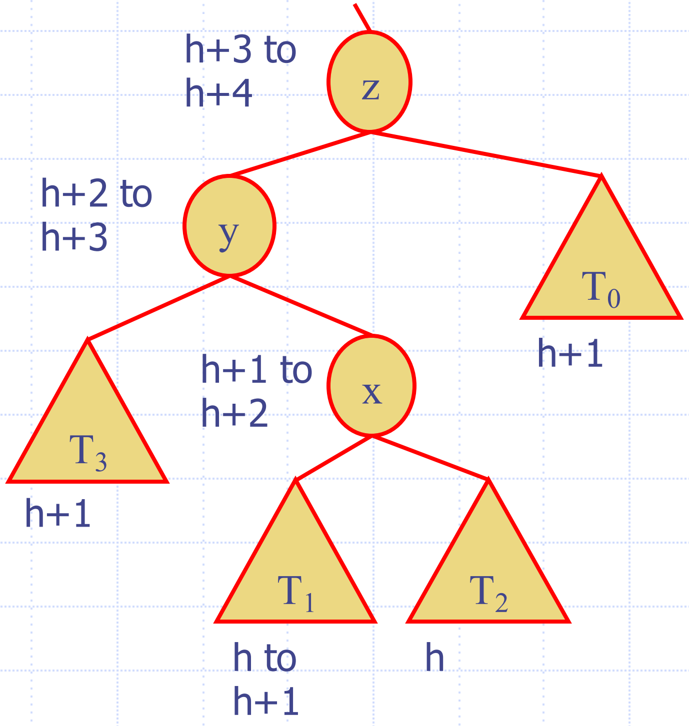
- Perform rotations around y and z to make x the top-most node.



- Insertion happens in subtree T_1
- y remains balanced
  - $H(T_3)$ = h+1 or h+2 or h+3
    - h+3 – y is originally unbalanced
    - h+2 – no increase in height of y – z remains balanced
    - therefore $H(T_3)$ = h+1
- H(y) = h+2 to h+3
- z is unbalanced
  - $H(T_0)$ = h+1 or h+2 or h+3
    - since z was originally balanced
    - $H(T_0)$ = h+1
- H(z) = h+3 to h+4

Restructuring – Double Rotation

- Perform rotations around y and z to make x the top-most node.

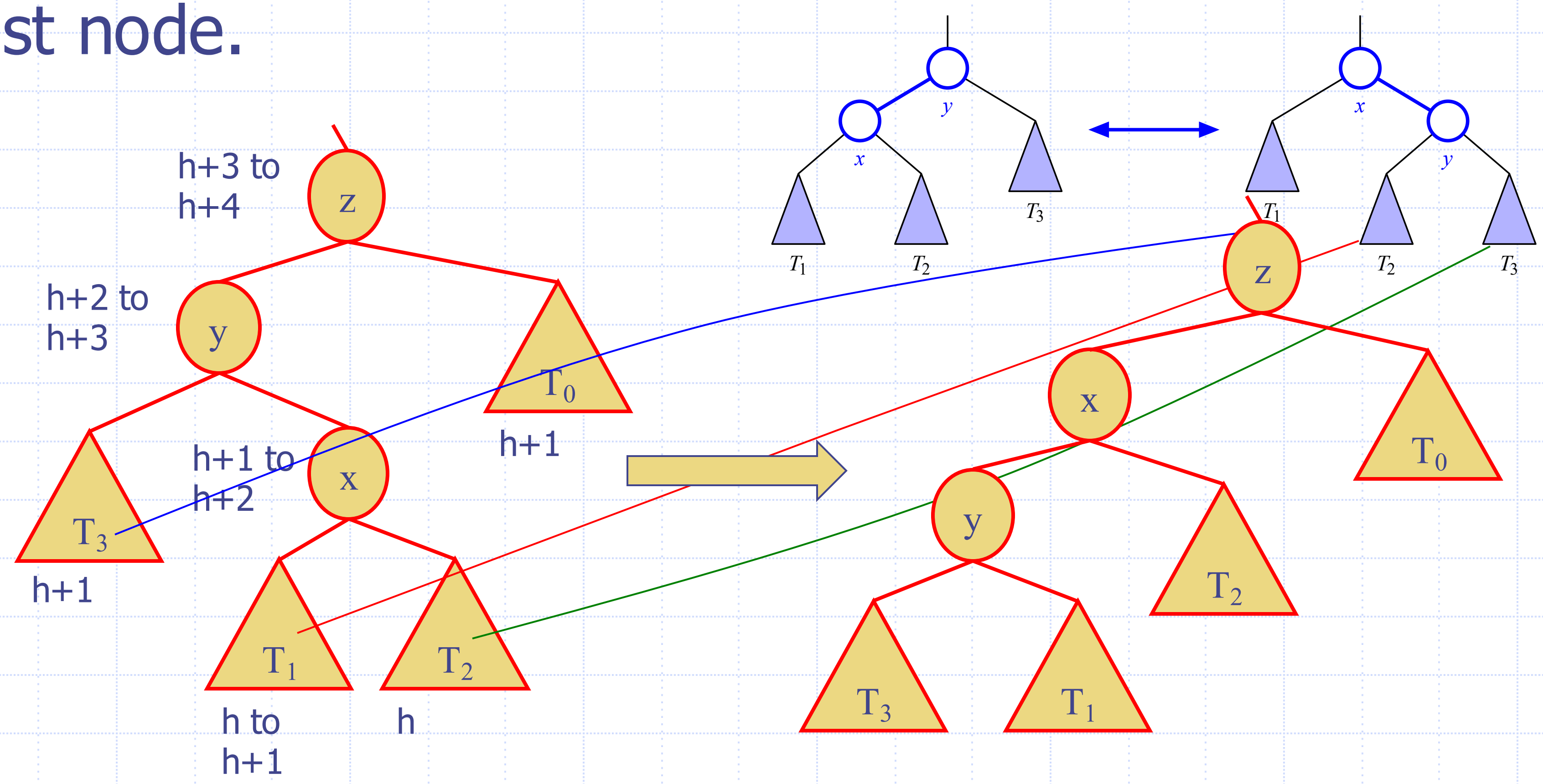


- Insertion happens in subtree T_1
- y remains balanced
  - $H(T_3)$ = h+1 or h+2 or h+3
    - h+3 – y is originally unbalanced
    - h+2 – no increase in height of y – z remains balanced
    - therefore $H(T_3)$ = h+1
- H(y) = h+2 to h+3
- z is unbalanced
  - $H(T_0)$ = h+1 or h+2 or h+3
    - since z was originally balanced
    - $H(T_0)$ = h+1
- H(z) = h+3 to h+4

Restructuring – Double Rotation

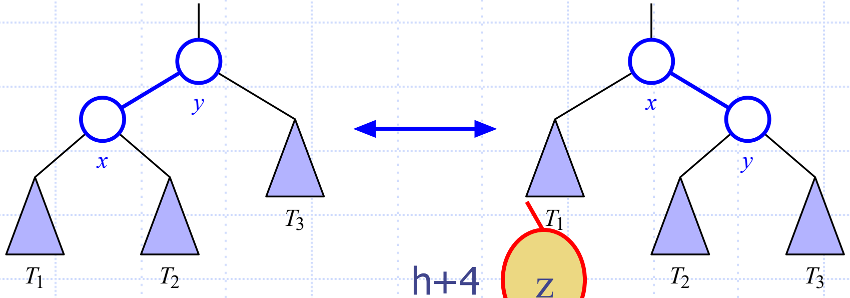
- Perform rotations around y and z to make x the top-most node.

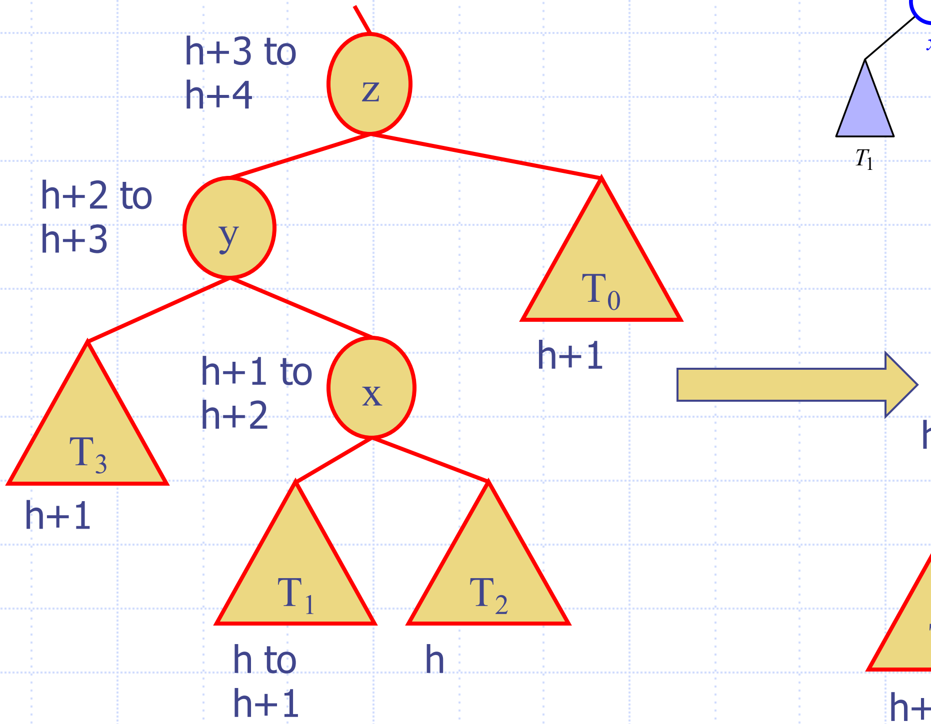
- First step
  - rotate x and y

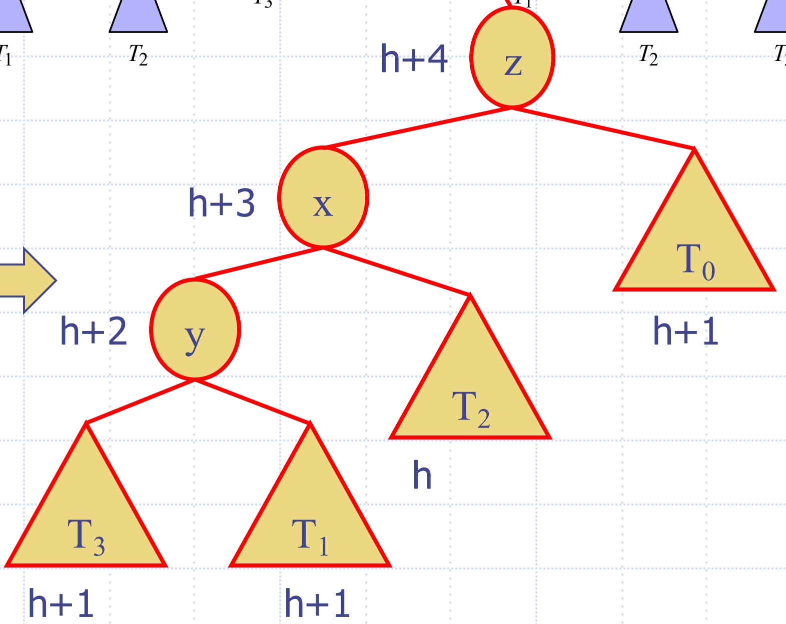


Restructuring – Double Rotation

- Perform rotations around y and z to make x the top-most node.
- First step
  - rotate x and y

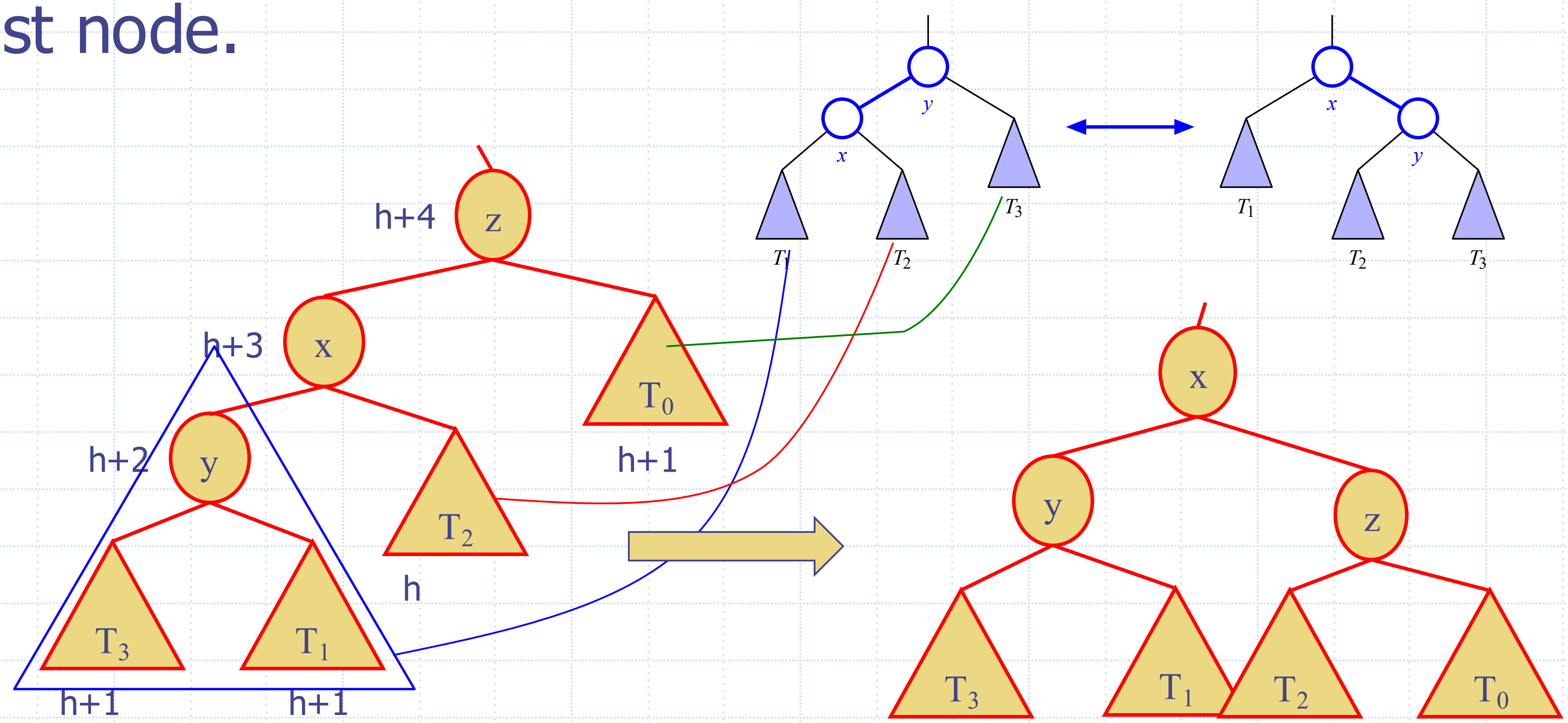






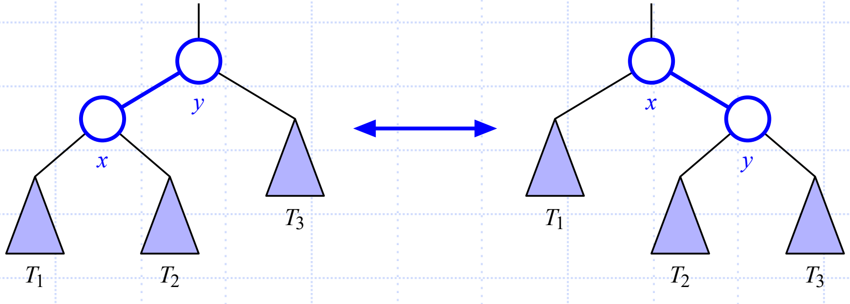
Restructuring – Double Rotation

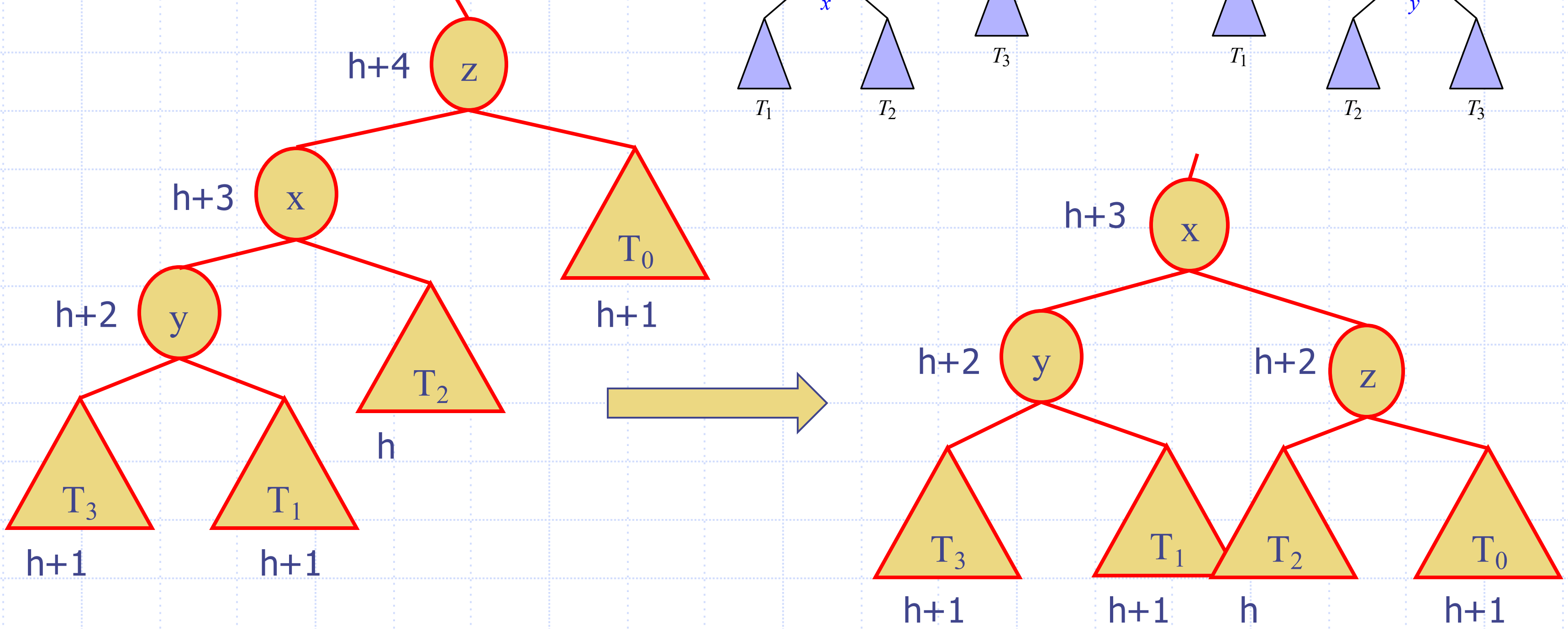
- Perform rotations around y and z to make x the top-most node.
- Second step
  - rotate x and z



Restructuring – Double Rotation

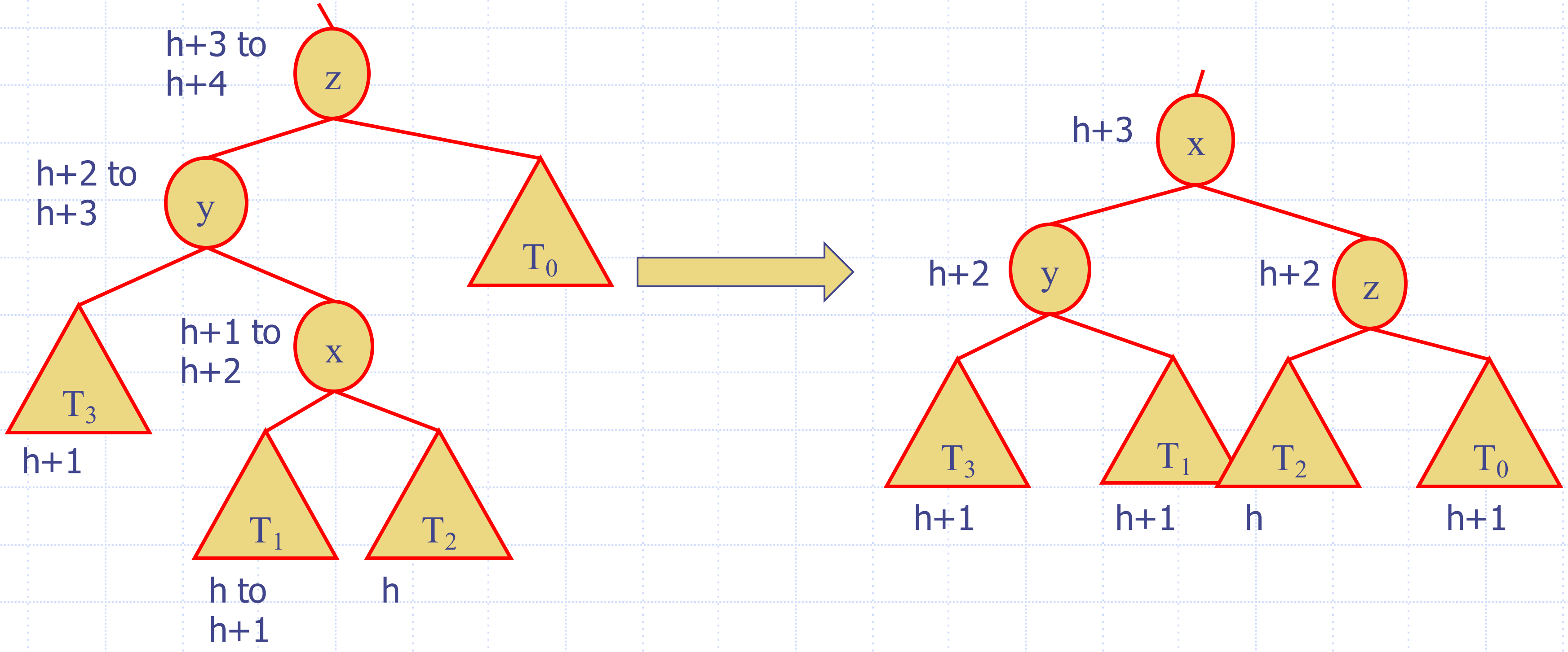
- Perform rotations around y and z to make x the top-most node.
- Second step
  - rotate x and z





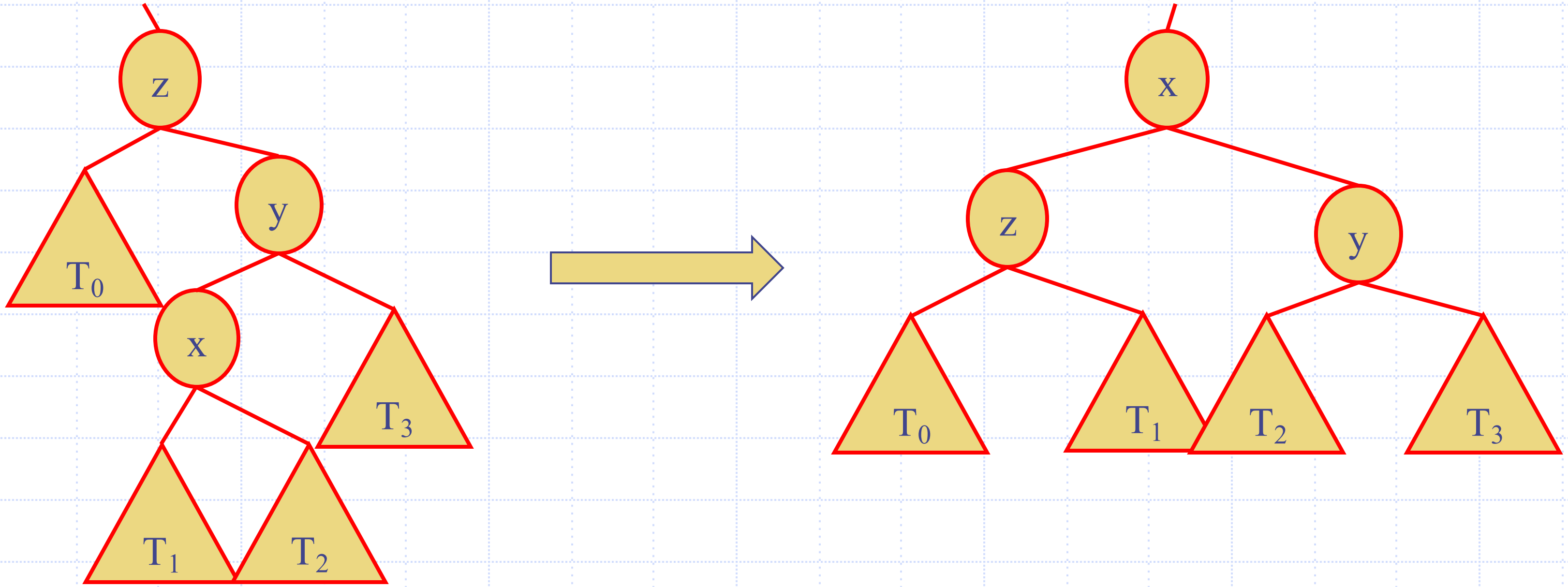
Restructuring – Double Rotation

- Perform rotations around (x,y) and (x,z) to make x the top-most node.

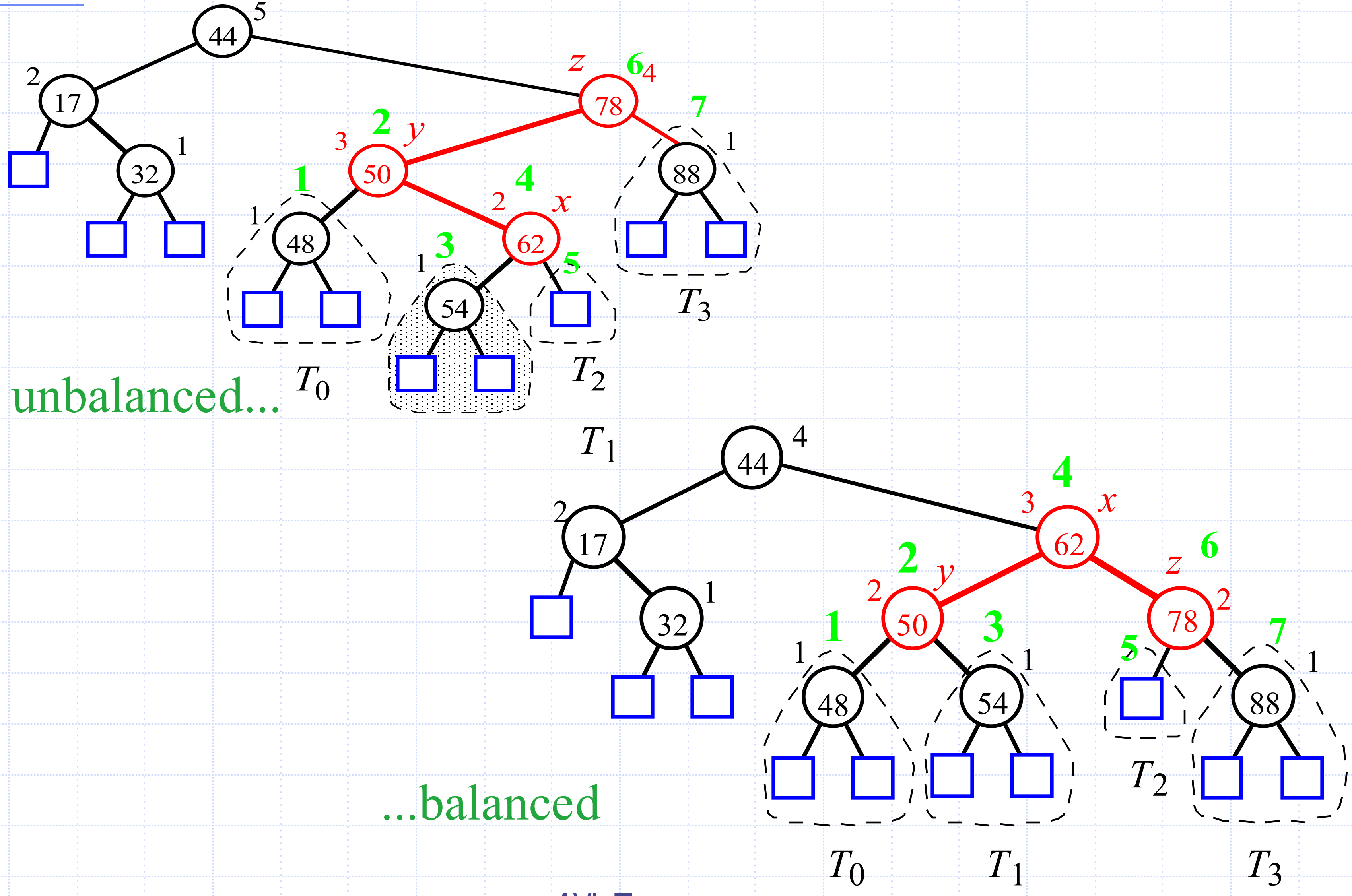


Restructuring – Double Rotation

- Perform rotations around y and z to make x the top-most node.
- symmetric to the previous configuration

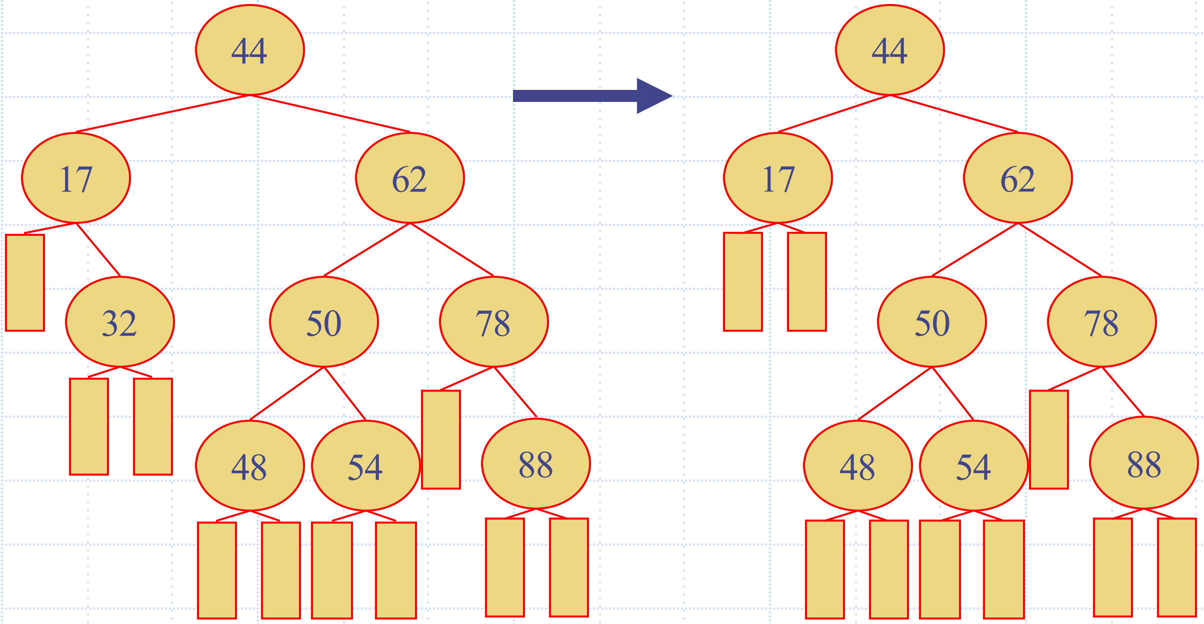


Insertion Example, continued



Removal

- 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:

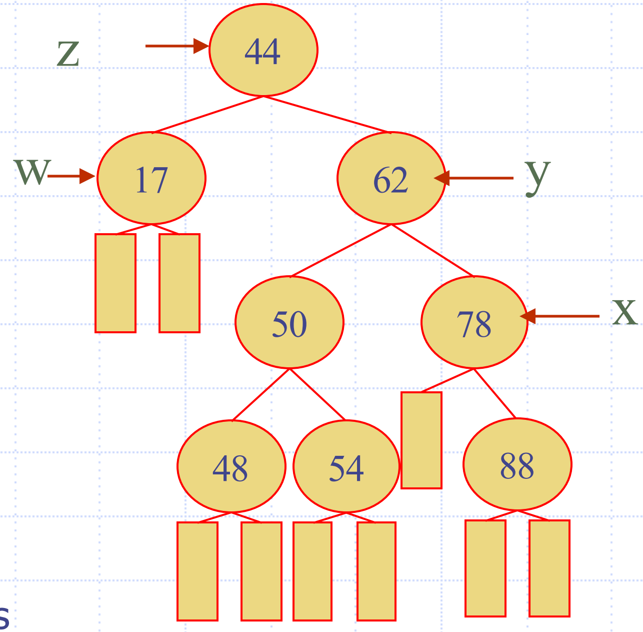


before deletion of 32

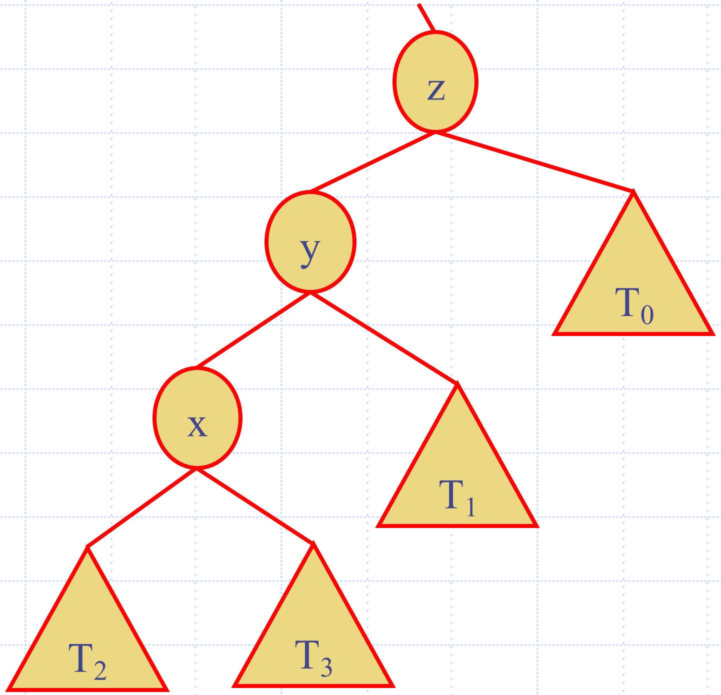
after deletion

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 a trinode restructuring 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

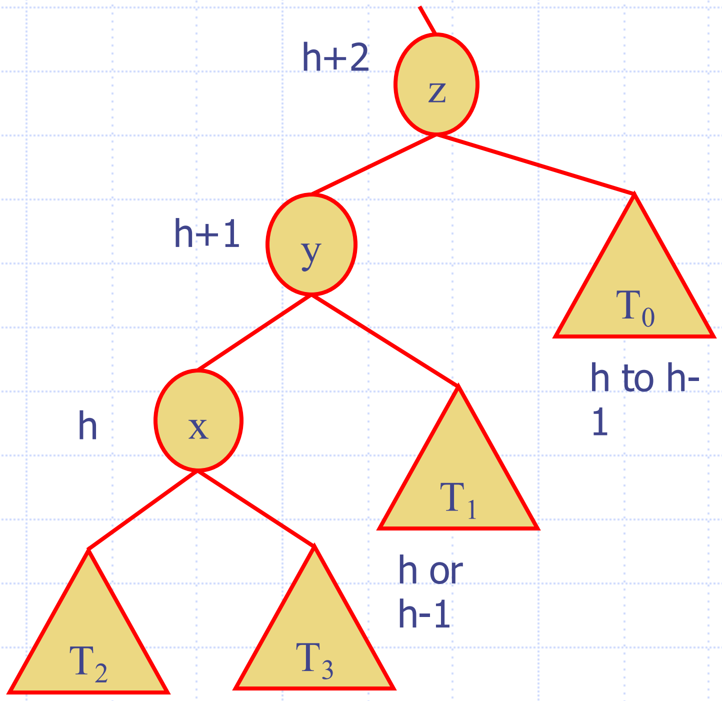


Deletion



- Suppose deletion happens in subtree T_0 and its height reduces from h to h-1
- z was originally balanced and now unbalanced
  - H(y) = h+1
  - H(z) = h+2
- x has larger height than T_1
  - H(x) = h
- y is balanced
  - $H(T_1)$ = h or h-1

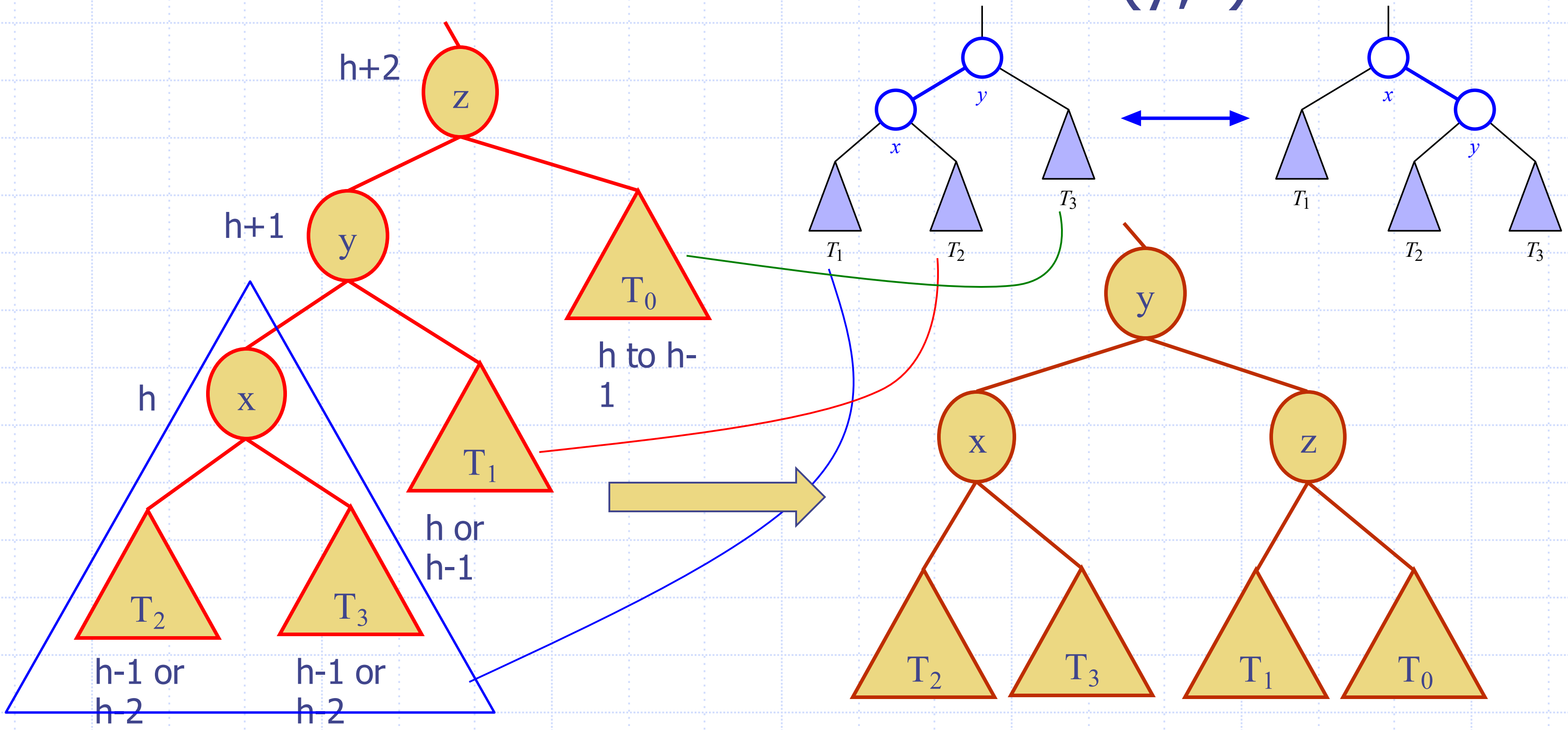
Deletion



- Suppose deletion happens in subtree T_0 and its height reduces from h to h-1
- x is balanced
  - $H(T_2)$, $H(T_3)$ is h-1 or h-2
  - However both T_2 and T_3 cannot have height h-2

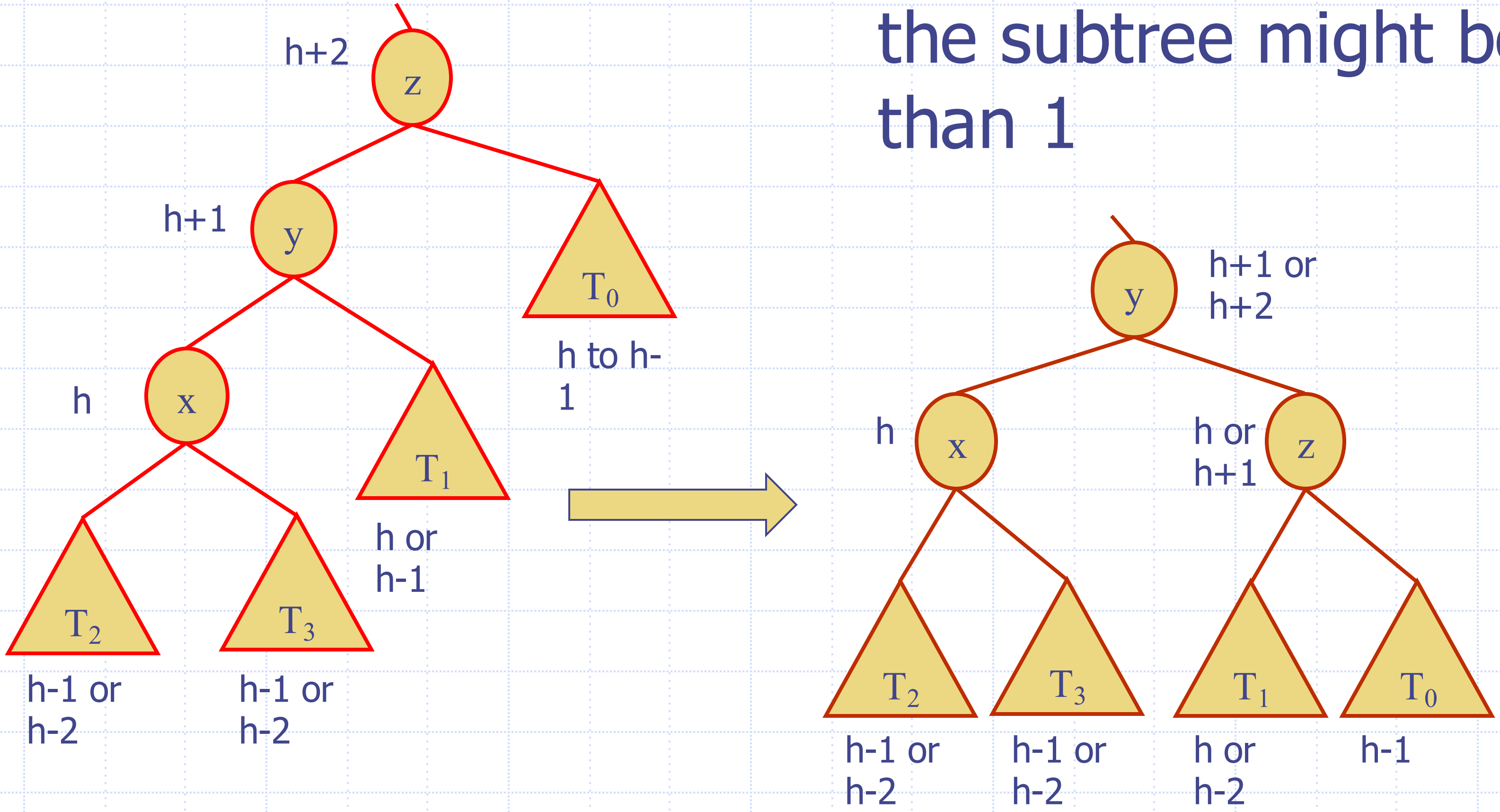
Deletion – Restructuring – Single Rotation

- Rotate (y,z)

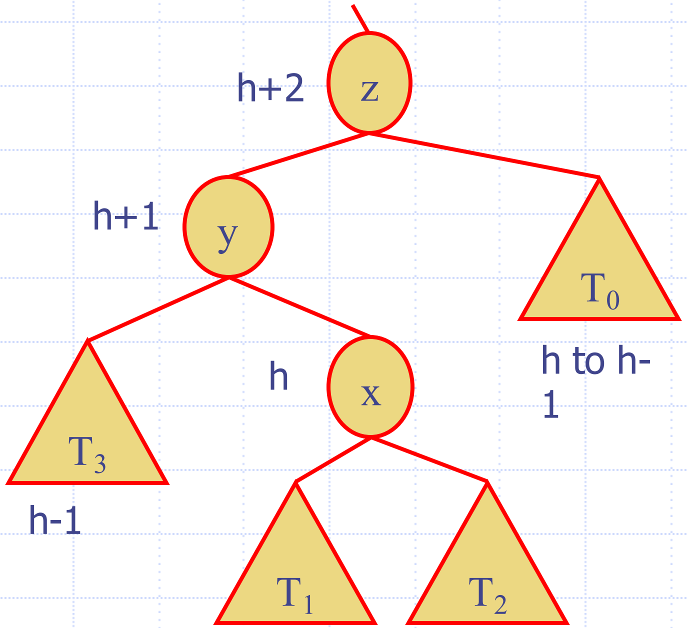


Deletion – Restructuring – Single Rotation

- After rotation the height of the subtree might be less than 1

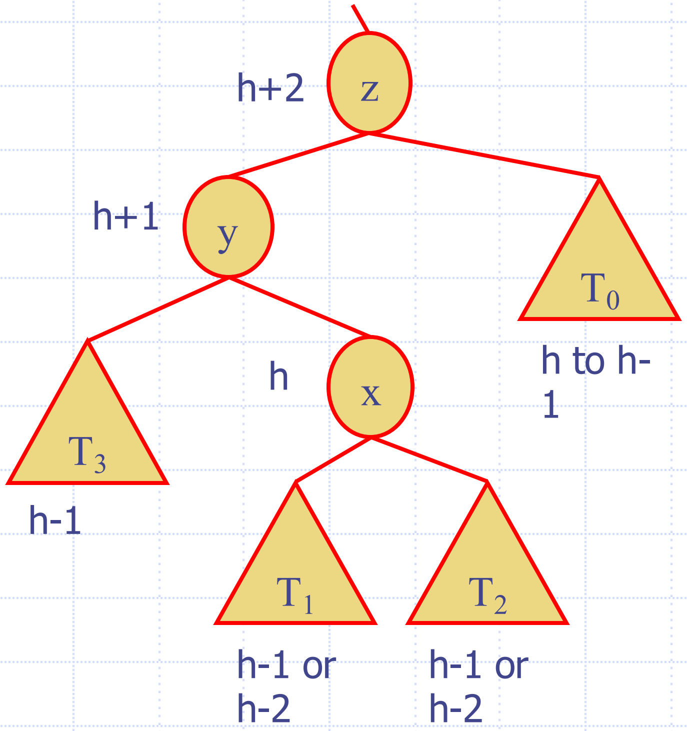


Deletion – Restructuring – Double Rotation



- Suppose deletion happens in subtree T_0 and its height reduces from h to h-1
- z was originally balanced and now unbalanced
  - H(y) = h+1
  - H(z) = h+2
- x has larger height than T_3
  - H(x) = h
- y is balanced
  - $H(T_3)$ = h-1

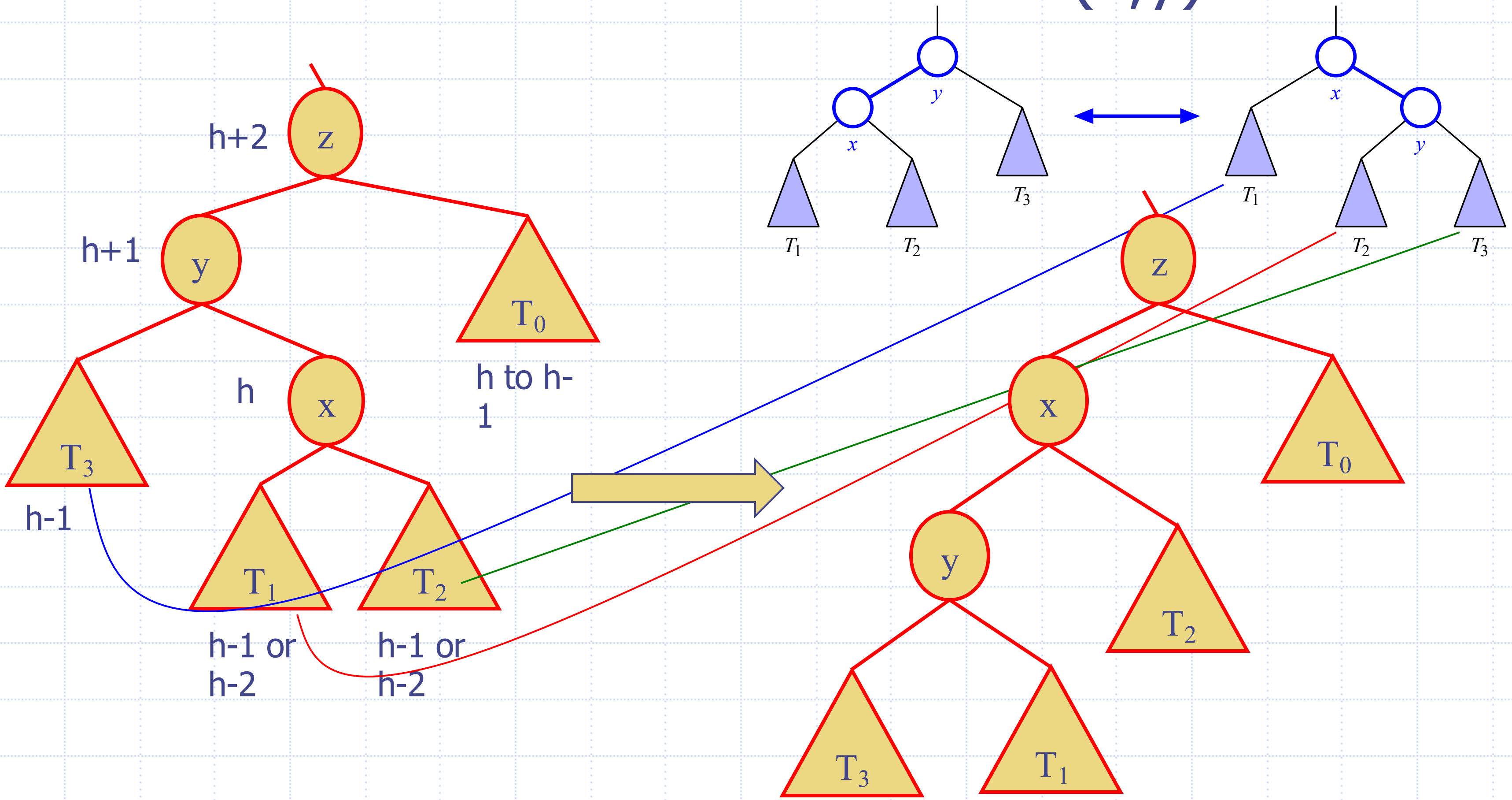
Deletion – Restructuring – Double Rotation



- Suppose deletion happens in subtree T_0 and its height reduces from h to h-1
- x remains balanced
  - $H(T_2)$, $H(T_3)$ is h-1 or h-2
  - However both T_2 and T_3 cannot have height h-2

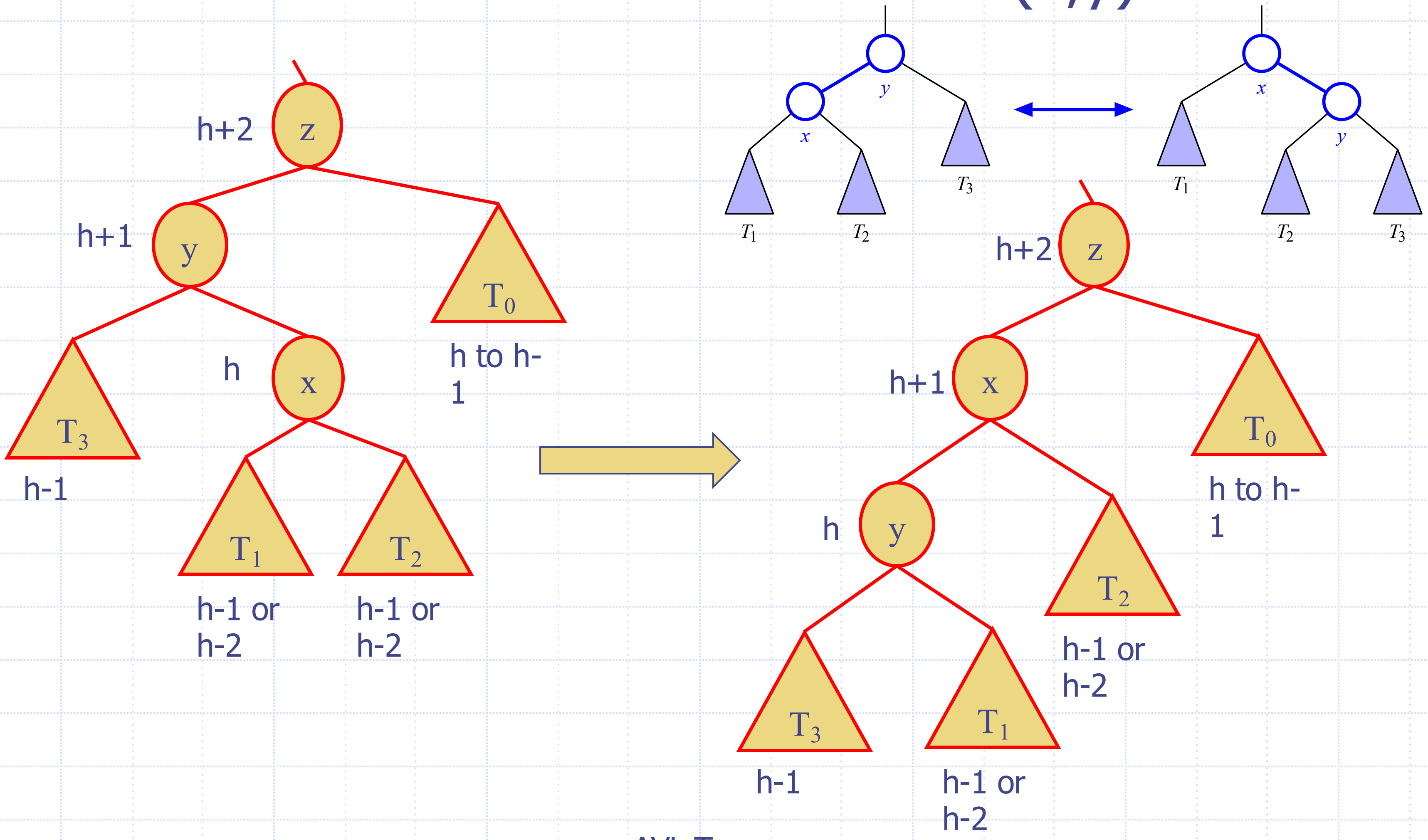
Deletion – Restructuring – Double Rotation

- Step 1
- Rotate (x,y)



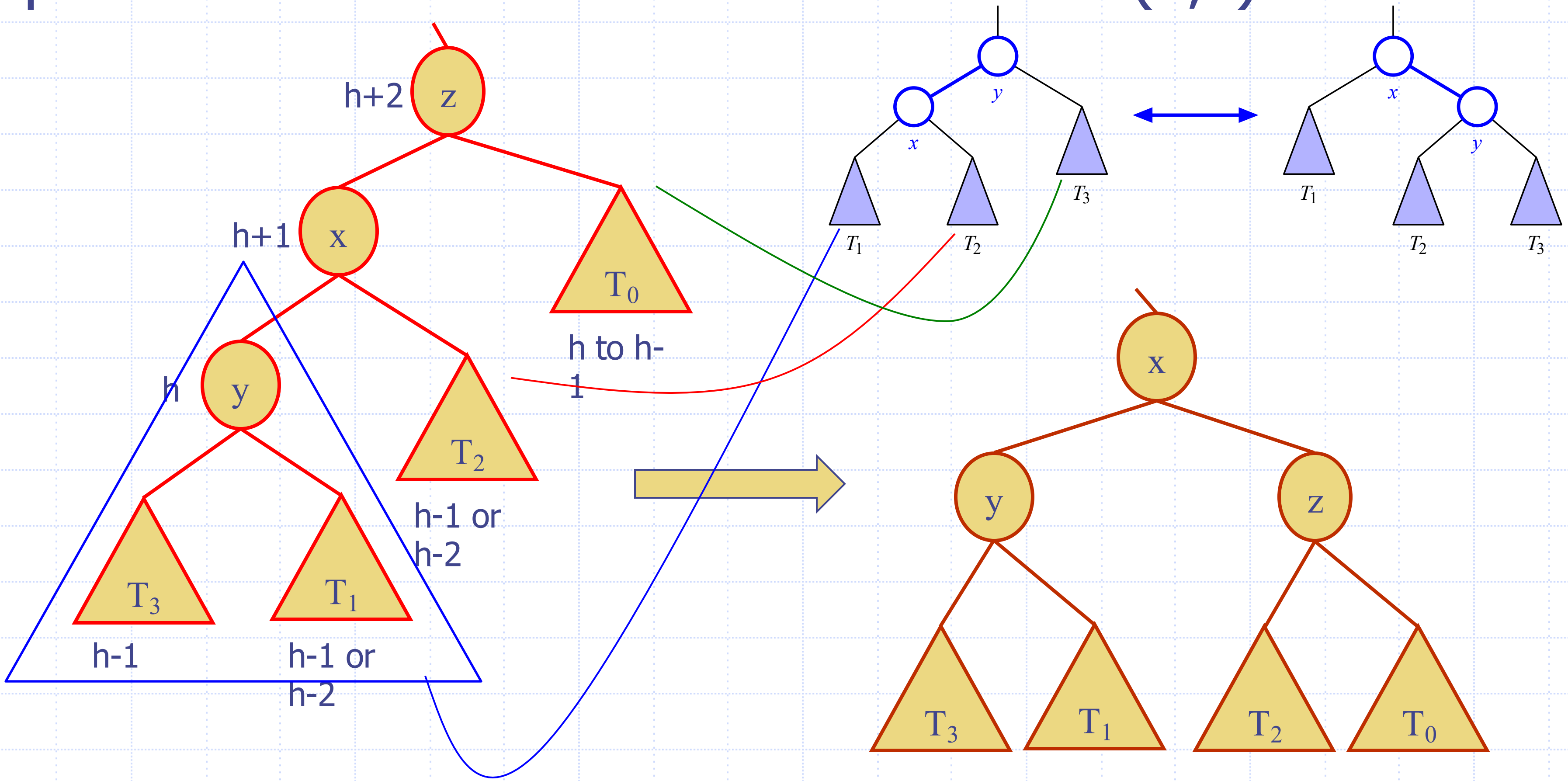
Deletion – Restructuring – Double Rotation

- Step 1
- Rotate (x,y)



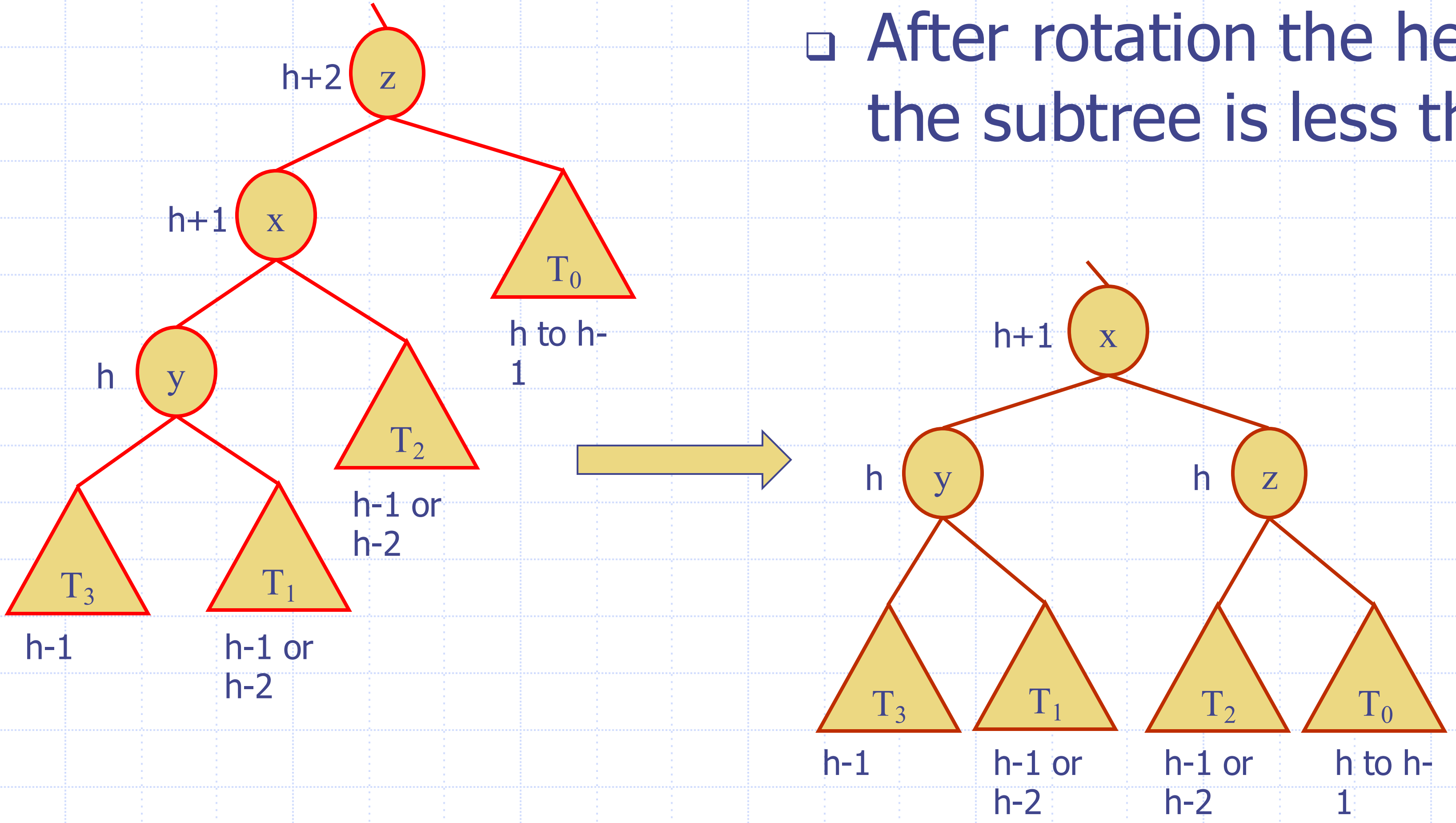
Deletion – Restructuring – Double Rotation

- Step 2
- Rotate (x,z)

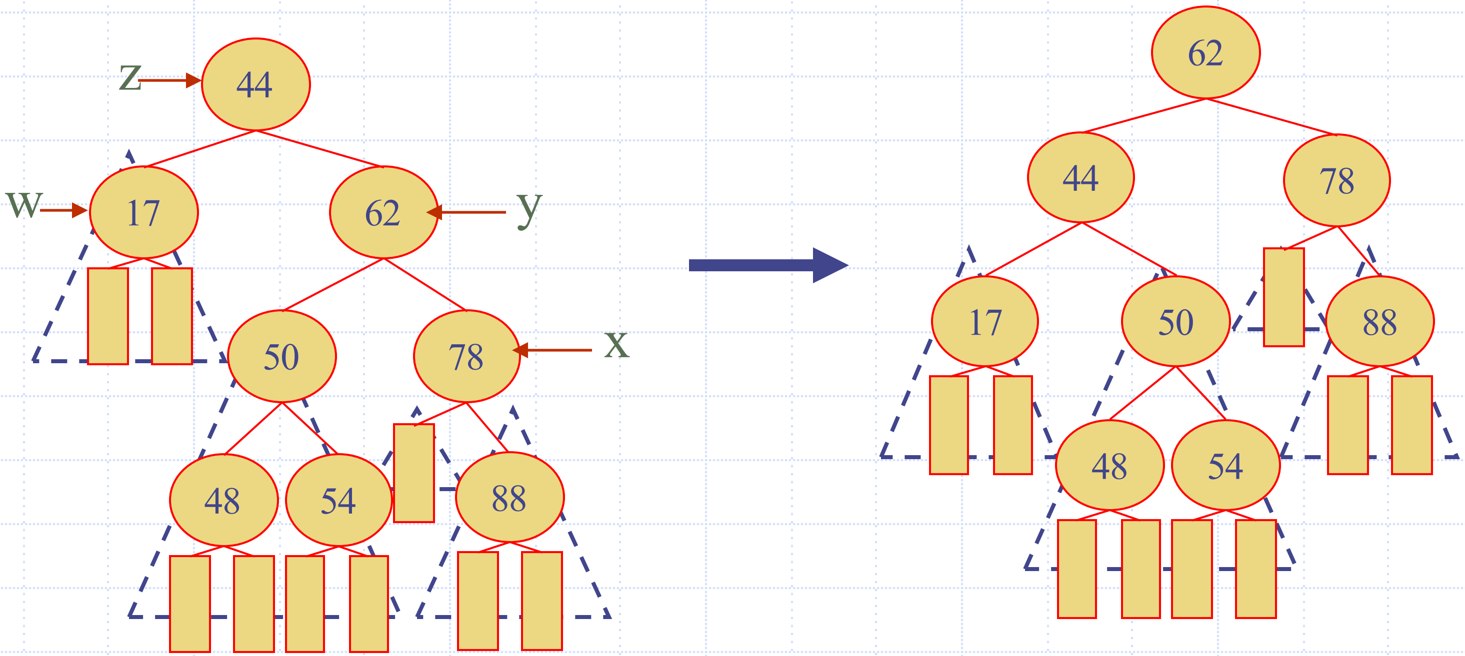


Deletion – Restructuring – Double Rotation

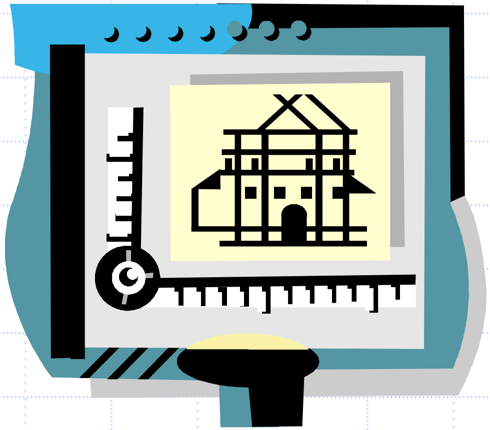
- Step 2
- Rotate (x,z)
- After rotation the height of the subtree is less than 1



Rebalancing after a Removal



AVL Tree Performance



- AVL tree storing n items
  - The data structure uses O(n) space
  - A single restructuring 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)