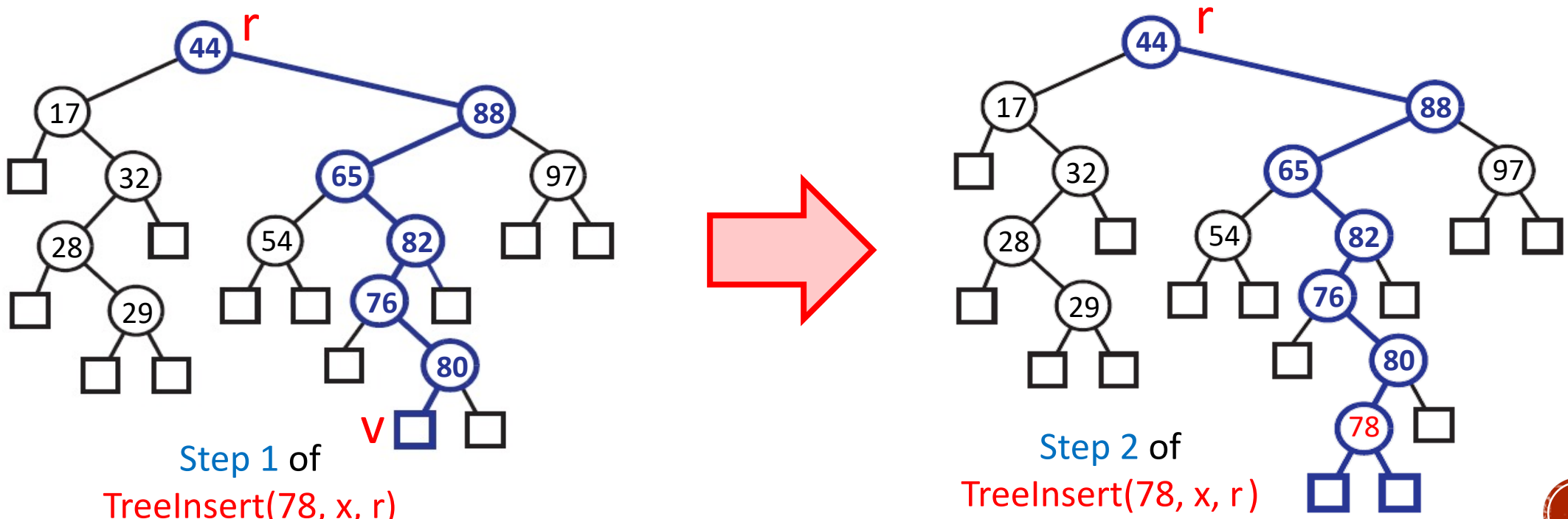
BINARY SEARCH TREE – INSERTION

- Consider this operation: insertAtExternal(v, (k,x)), which **inserts** entry (k,x) at the **external** node v, and **expand** v to become **internal**, having two external children
- This operation is used in TreeInsert(k, x, r) which works in two steps:

  Step 1. **find** the position v to insert entry (k, x) Step 2. insertAtExternal(v, (k,x))



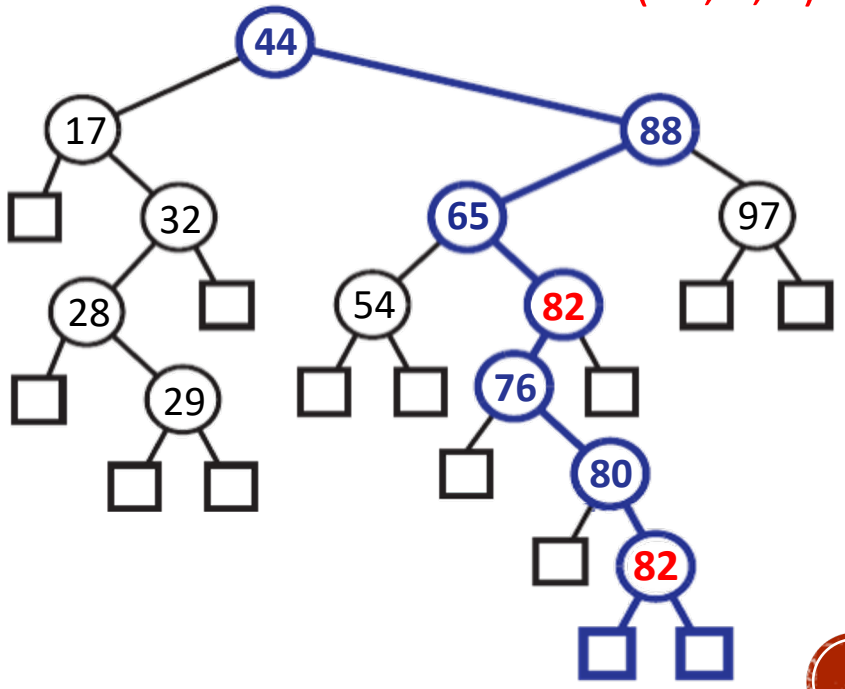
Step 1 of TreeInsert(78, x, r)

Step 2 of TreeInsert(78, x, r)

BINARY SEARCH TREE – INSERTION

- Important: A binary search tree can be used to implement **maps** or **dictionaries**.
- If it will implement a **dictionary**, it should be able to hold multiple entries that have the same key!
- We said that for any node r containing an entry (k,x):
  - Entries in the **left subtree** of r have keys $\leq k$
  - Entries in the **right subtree** of r have keys $\geq k$
- Where do we insert entries whose key $= k$?
  - We either insert them always into the left subtree, or always into the right subtree.
- Let's choose the left subtree, thus:
  - Entries in the **right subtree** of r have keys $> k$

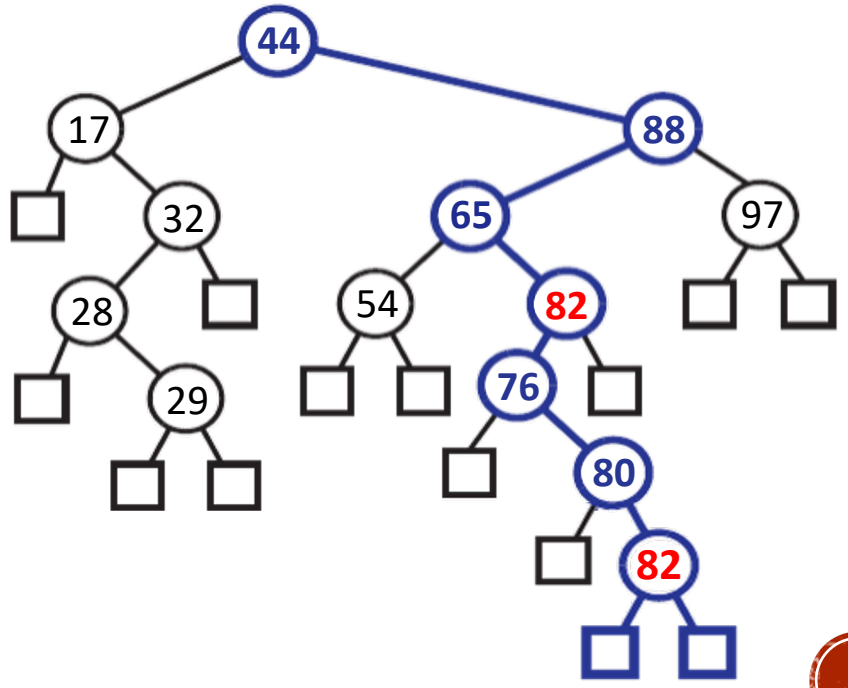
Example:
TreeInsert(82, x, r)



BINARY SEARCH TREE – INSERTION

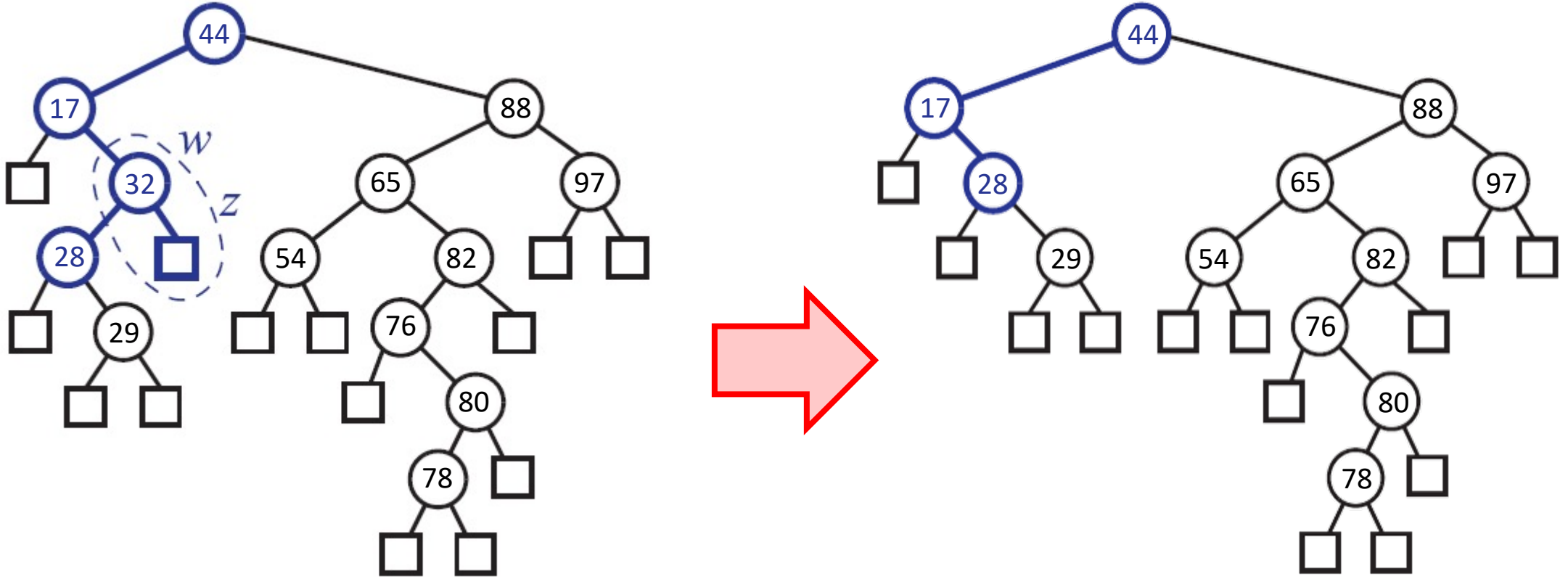
- TreeInsert(*k*,*x*,*r*) is a recursive algorithm that inserts (k,x) in the **subtree rooted at** node r
- TreeInsert(k,x,T.root()) inserts (k,x) in a tree T
- Remember: TreeSearch(k, r) returns either:
  - an **internal** node whose key = k, or
  - an **external** node where we can insert (k,x)
- The if-statement checks if w is **internal**. If so, the tree already contains at least one entry whose key = k, in which case we recursively search the subtree rooted at the **left child** of w
- Note: **Termination** (and **insertion**) only happens at an external node!

Algorithm TreeInsert(*k*,*x*,*r*):
  w ← TreeSearch(*k*,*r*)
  if *T*.isInternal(*w*) **then**
    return TreeInsert(*k*,*x*,*T*.left(*w*))
  T.insertAtExternal(*w*, (*k*,*x*))
  return *w*



BINARY SEARCH TREE – REMOVAL

- When removing nodes, we'll use the following operation:
  - RemoveAboveExternal(z): Removes the **external node** z along with its **parent**, but after replacing that parent with z's **sibling**!



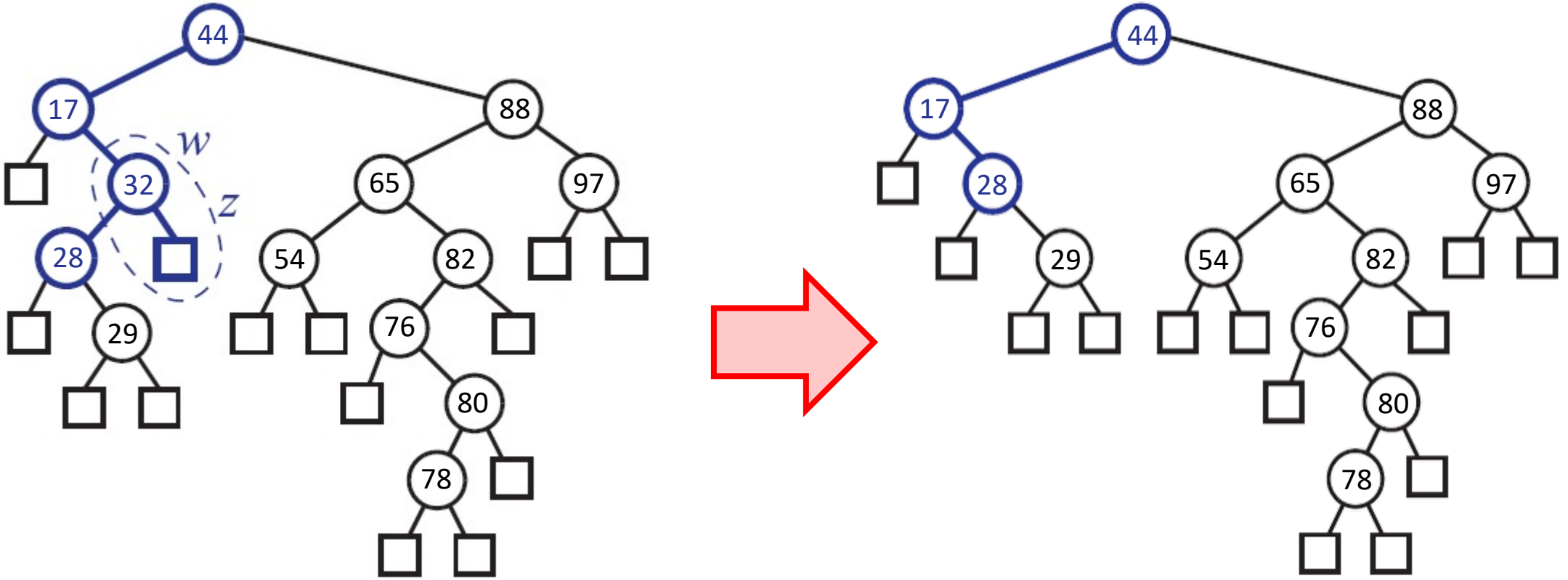
BINARY SEARCH TREE – REMOVAL

- To remove a node, **we first search for its key**, k, using TreeSearch(*k*, *T*.root()), which returns one of the following:
  1. An **external node**, implying that the key was not found.
  2. An **internal node**, w, that has a child that is a leaf
  3. An **internal node**, w, whose children are both internal

- In **Case 1**, there is no need to erase anything, and an error is signaled.

BINARY SEARCH TREE – REMOVAL

- **Case 2**: TreeSearch(*k*, *T*.root()) returns an **internal node** *w* that has a child that is a leaf
  - To remove w, we call removeAboveExternal(z) where z is an external child of w

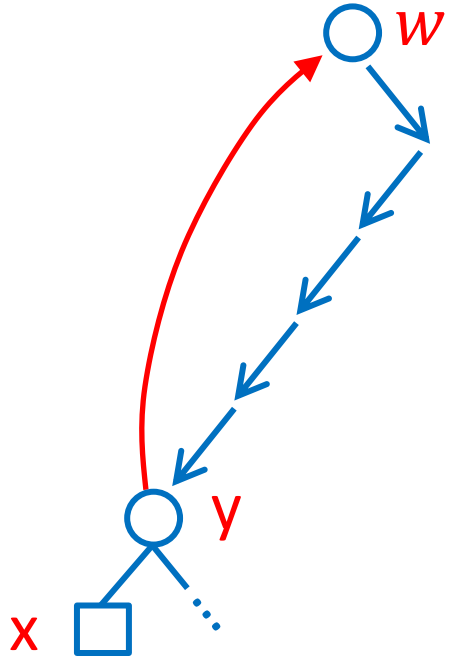


BINARY SEARCH TREE – REMOVAL

- **Case 3**: TreeSearch(*k*, *T*.root()) returns an **internal node** *w* whose children are internal

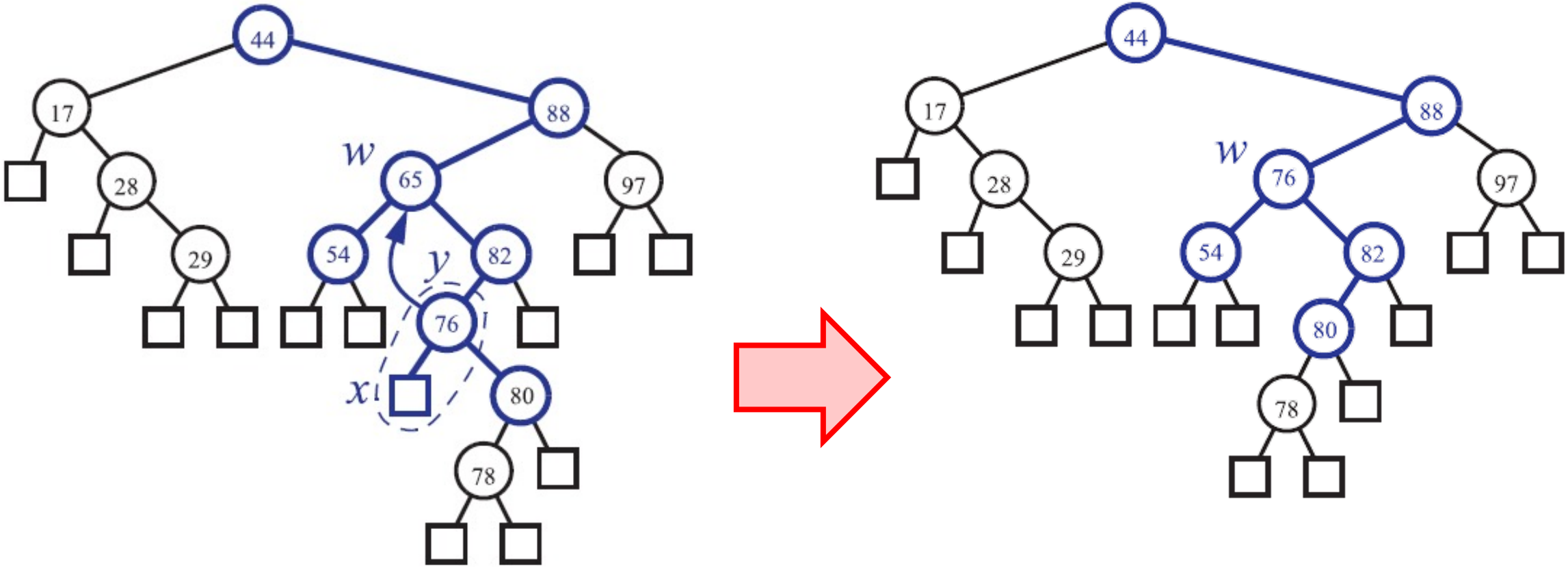
  - To remove w, we may follow three steps:

    1. Find the last **internal** node y that comes after w by going to the **right child** of w and then move down following the **left children**
    2. Move the entry of y into w (which deletes what was stored in w)
    3. Call removeAboveExternal(x) where x is the **left child** of y; this will remove both x and y



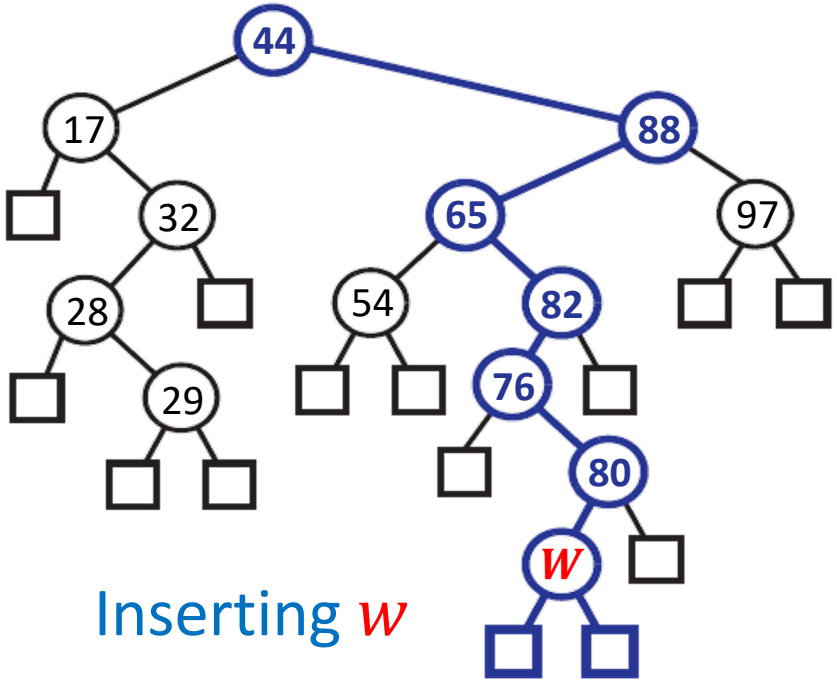
BINARY SEARCH TREE – REMOVAL

- **Case 3**: TreeSearch(*k*, *T*.root()) returns an **internal node** *w* whose children are internal

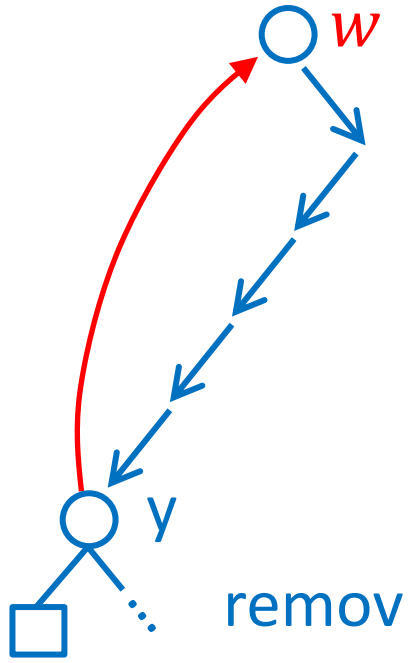


BINARY SEARCH TREE – PERFORMANCE

- Insertion and removal take O(h) = O(log n) time, where h is the **height** of the tree.



Inserting w

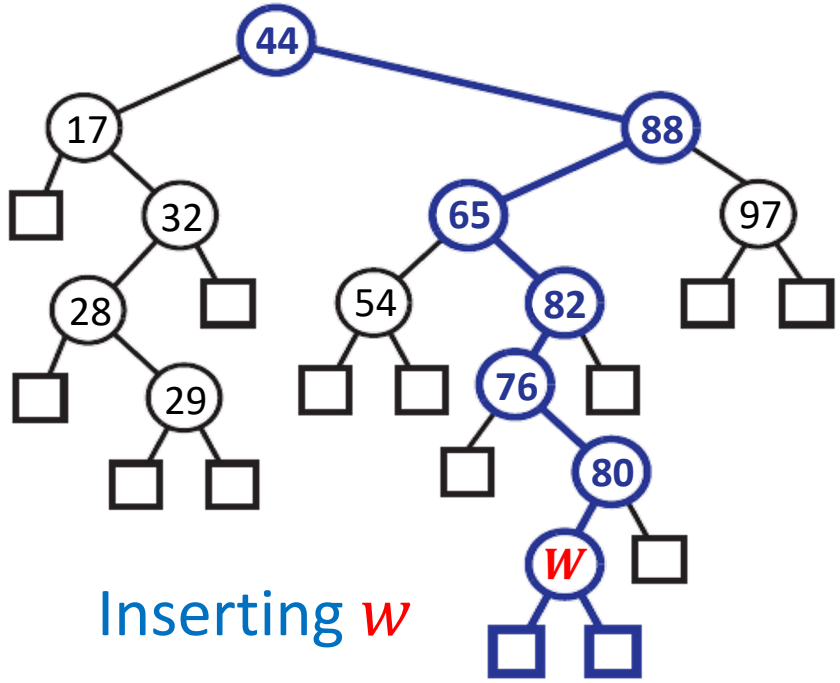


removing w

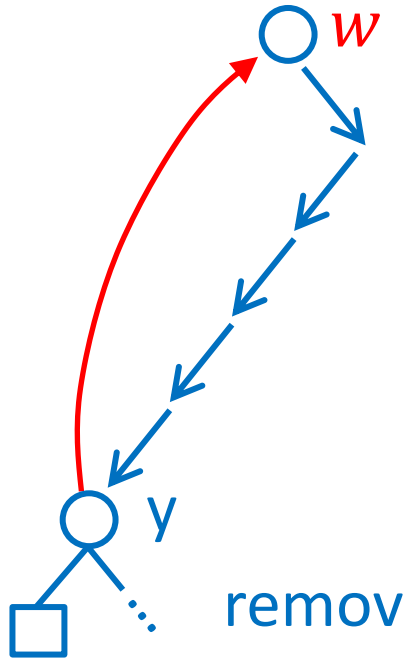
- Given k internal nodes, what is the maximum height of a binary search tree, h?

BINARY SEARCH TREE – PERFORMANCE

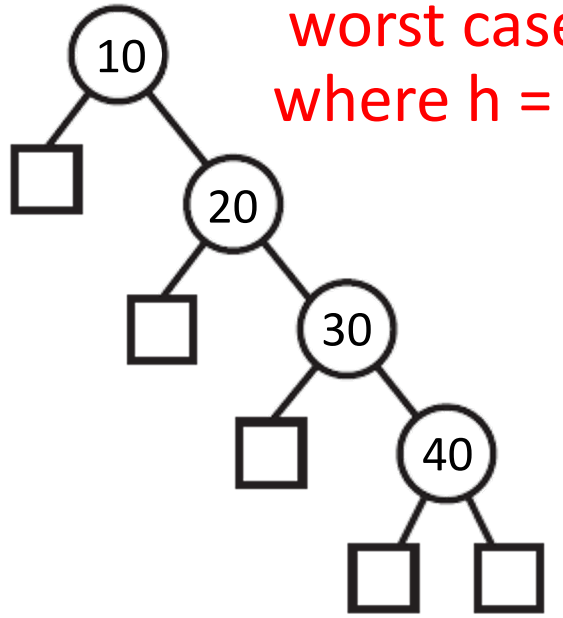
- Insertion and removal take O(h) time, where h is the **height** of the tree.



Inserting w



removing w



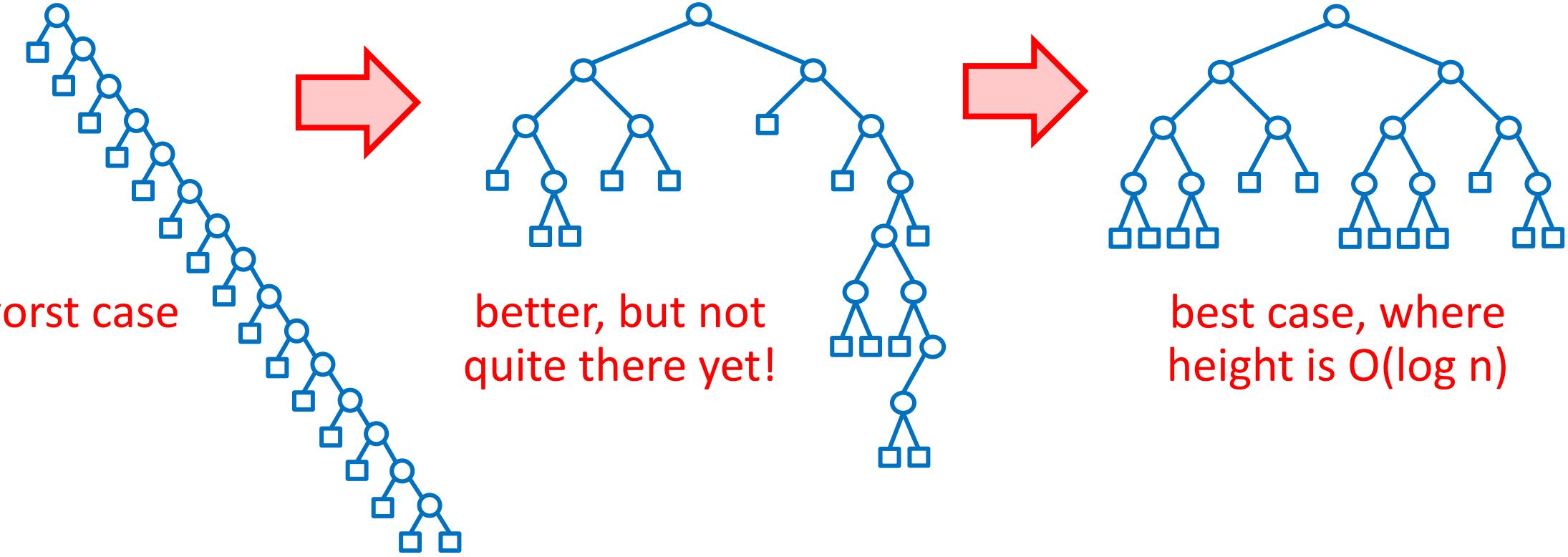
worst case where h = k

- Given k internal nodes, what is the maximum height of a binary search tree, h?

In the **worst case**, h = k, implying that insertion and removal take O(k) time!

BINARY SEARCH TREE – PERFORMANCE

- The height can be very large if the binary search tree is extremely **unbalanced**, e.g. there are nodes whose right-subtree is much taller than the left subtree!



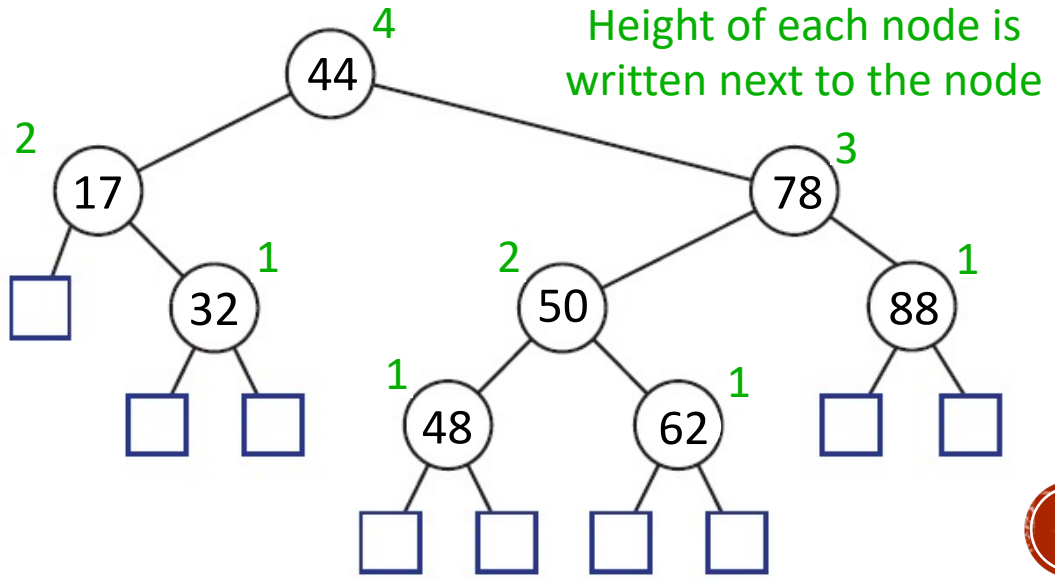
AVL TREES

AVL TREES

- For all the **implementations** of a map or dictionary we've seen so far (be it a list, an array, or a binary search tree (BST)), some operations run in O(n) time still!
- To make all operations run in O(log n) time, all we need is to add one more requirement to the structure of a BST T:

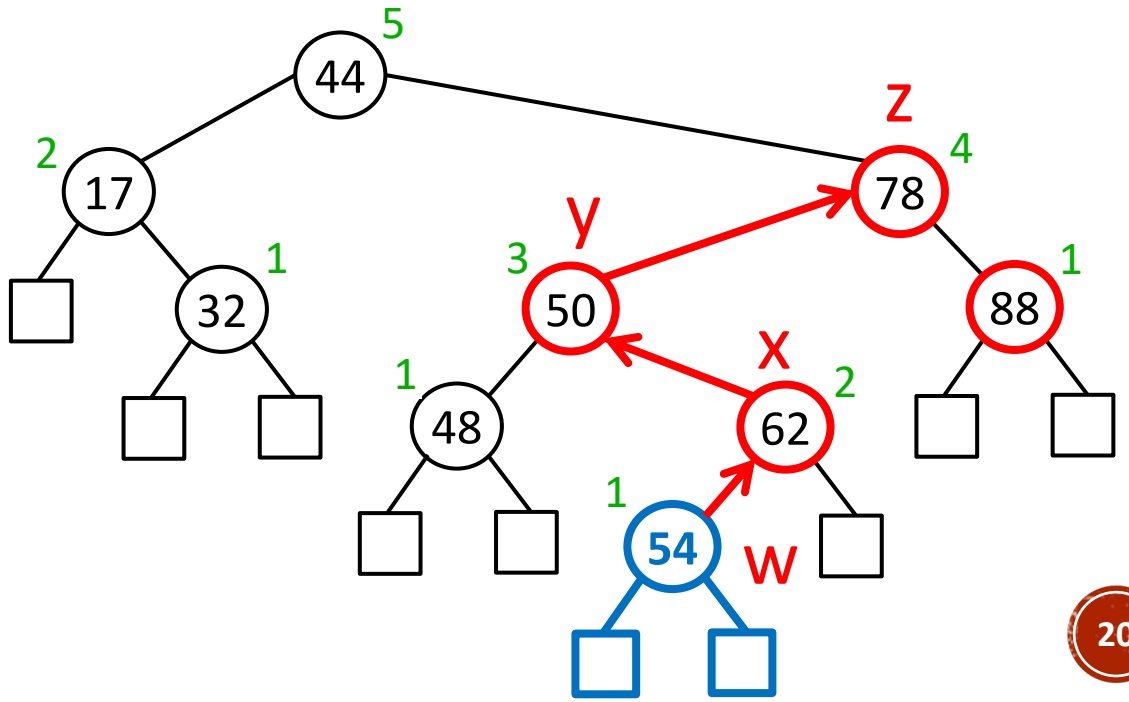
> **Height-Balance Property:** Every internal node v of T is "balanced", i.e., the heights of the children of v differ by **at most 1**.

- Any binary search tree that satisfies this property is an **AVL tree**, named after its inventors (Georgy **A**delson-**V**elsky and Evgenii **L**andis)!
- This property will assure that the **height** of a BST containing n entries is O(log n)



AVL TREES – INSERTION

- Recall that insertion always inserts the new entry at a node w that was previously an **external node**, and makes w an **internal node** with two external children
- Here is an example of an AVL tree that becomes **unbalanced** after inserting a node w whose key is **54**, and we want to determine where to fix the tree.
  - Let z be the **first unbalanced node** we encounter when going from w upwards
- The height of y is **greater by 2** compared to the height of its sibling, making z unbalanced!
- Our **repair strategy** will focus on nodes x, y, and z, using the *trinode restructuring* function

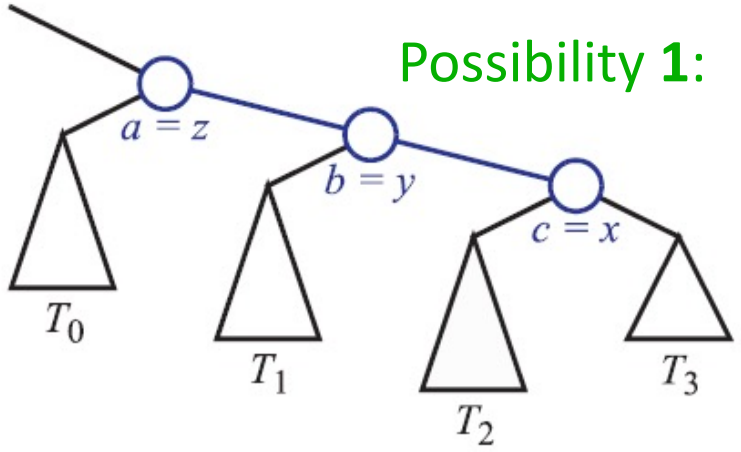


TRINODE RESTRUCTURING

- Here are the 4 possibilities of how z becomes **unbalanced** after an insertion operation:

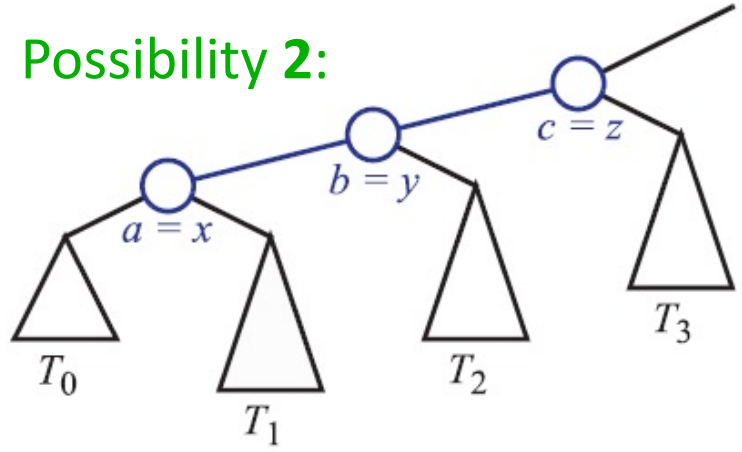
Possibility **1**:

Difference in height between T_0 and T_2 is $\geq$ **1**



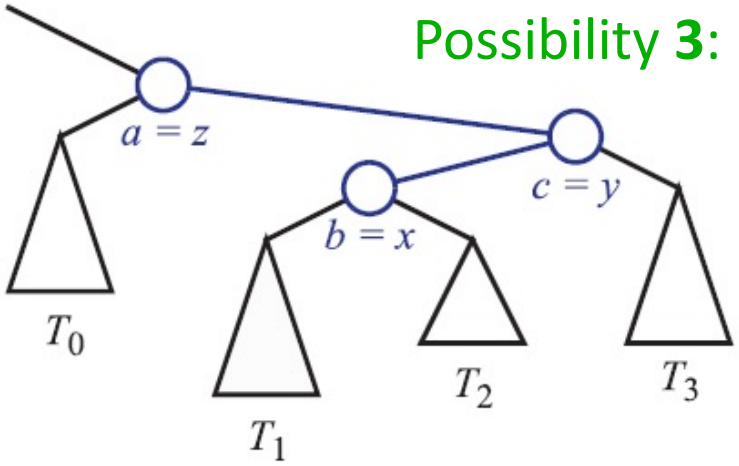
Possibility **2**:

Difference in height between T_1 and T_3 is $\geq$ **1**



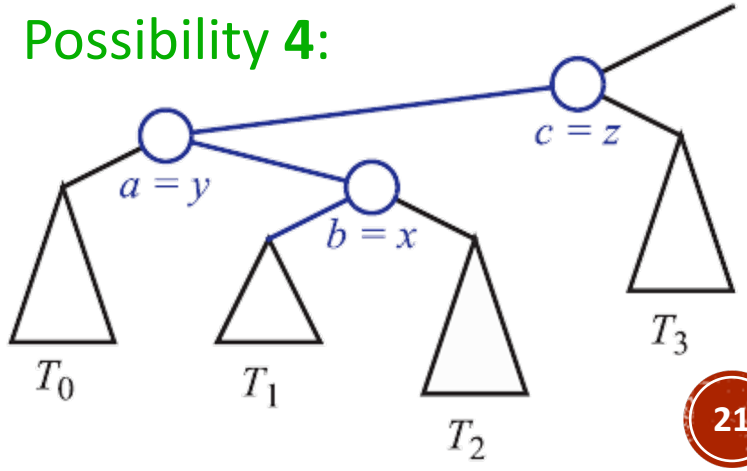
Possibility **3**:

Difference in height between T_0 and T_1 is $\geq$ **1**



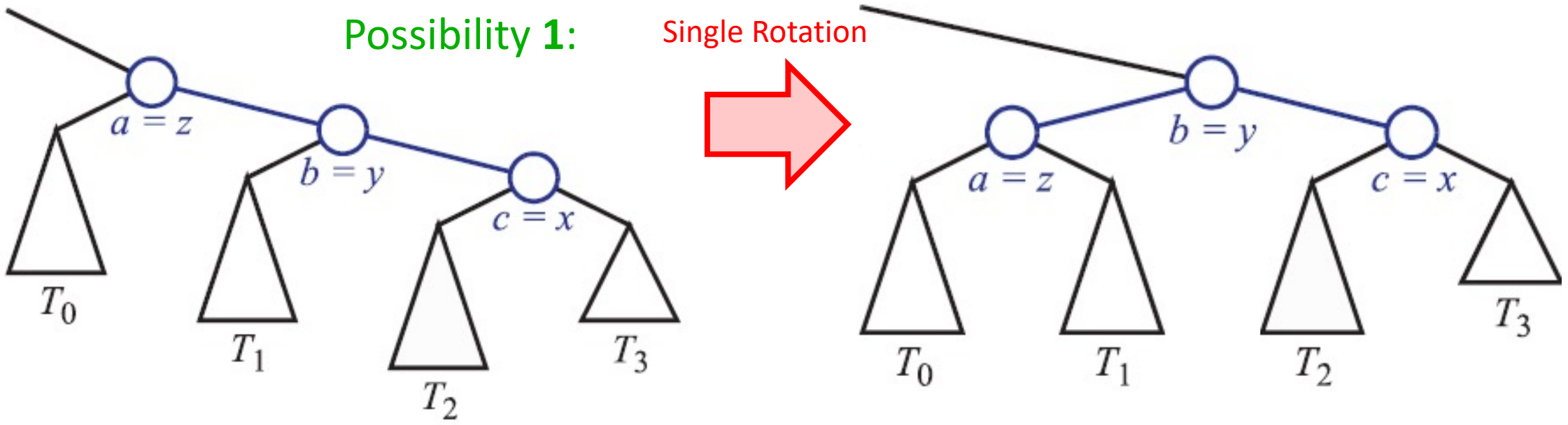
Possibility **4**:

Difference in height between T_2 and T_3 is $\geq$ **1**



TRINODE RESTRUCTURING

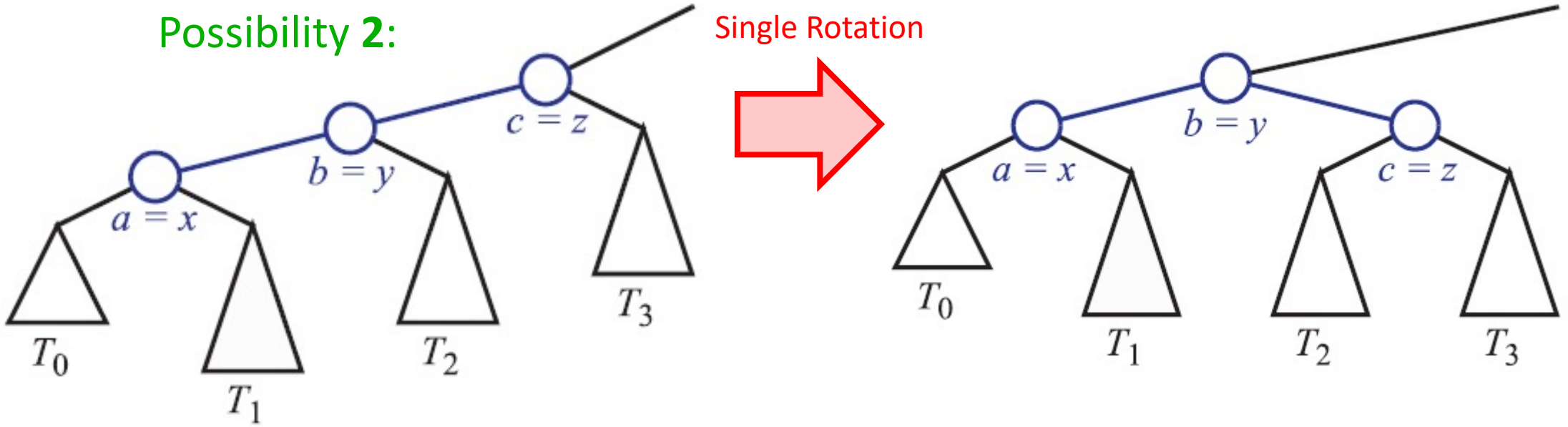
- A trinode restructuring does the following:
  1. Make the b the parent of a and c
  2. Make T_0 and T_1 the left and right subtrees of a, respectively
  3. Make T_2 and T_3 the left and right subtrees of c, respectively



TRINODE RESTRUCTURING

- A trinode restructuring does the following:
  1. Make the b the parent of a and c
  2. Make T_0 and T_1 the left and right subtrees of a, respectively
  3. Make T_2 and T_3 the left and right subtrees of c, respectively

Possibility **2**:

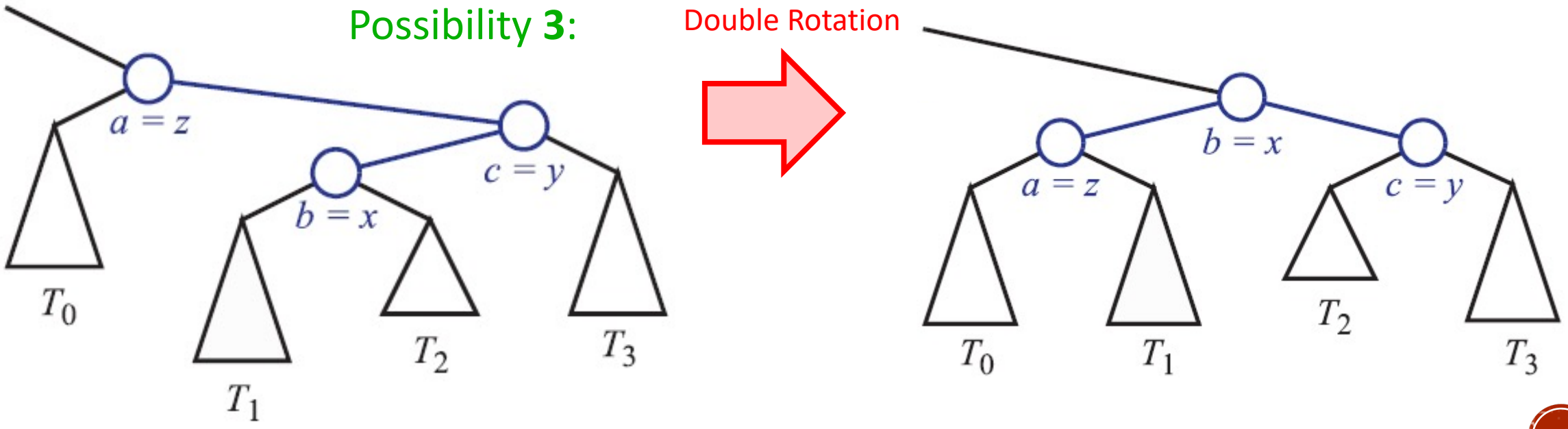


TRINODE RESTRUCTURING

- A trinode restructuring does the following:
  1. Make the b the parent of a and c
  2. Make T_0 and T_1 the left and right subtrees of a, respectively
  3. Make T_2 and T_3 the left and right subtrees of c, respectively

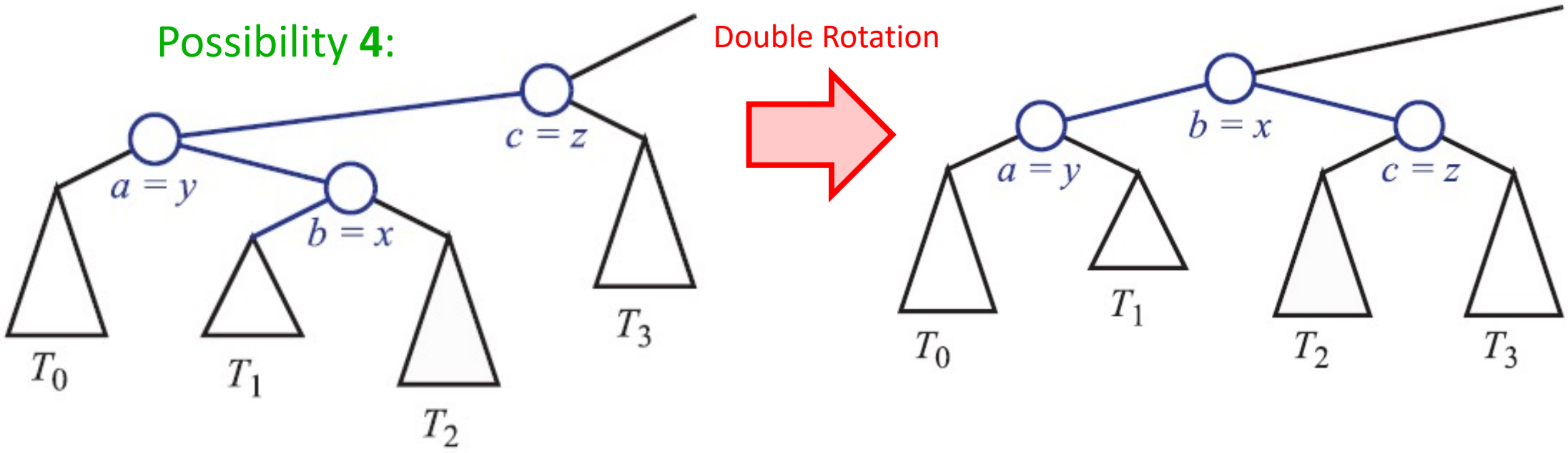
Possibility **3**:

Double Rotation



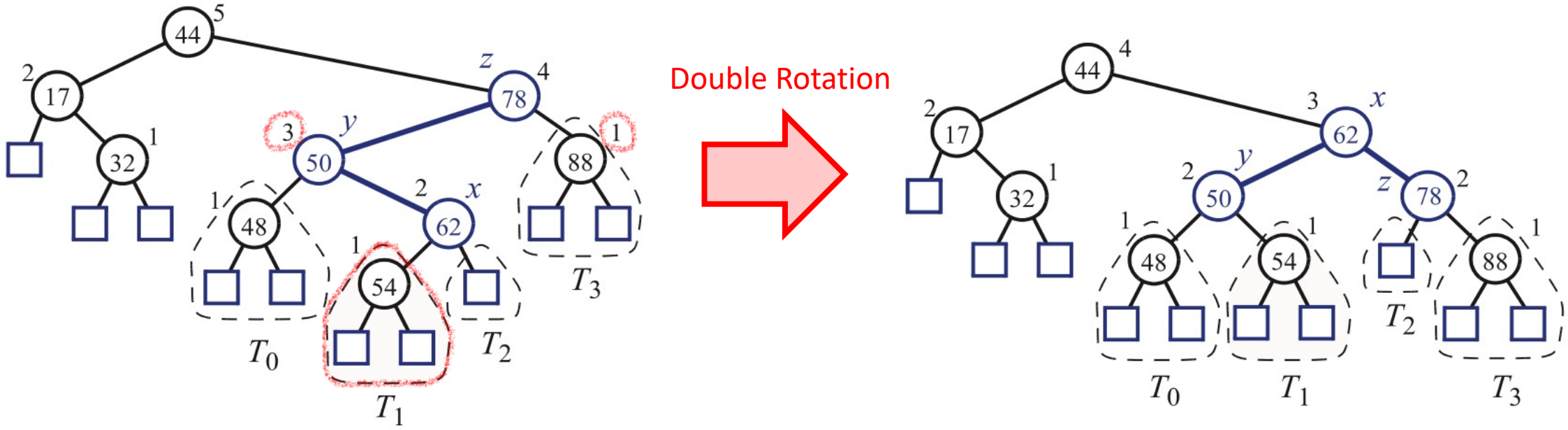
TRINODE RESTRUCTURING

- A trinode restructuring does the following:
  1. Make the b the parent of a and c
  2. Make T_0 and T_1 the left and right subtrees of a, respectively
  3. Make T_2 and T_3 the left and right subtrees of c, respectively



TRINODE RESTRUCTURING

- An example insertion of an entry with key 54 to an AVL tree
  - After adding a new node for key 54, the node storing key 78 becomes unbalanced
  - A trinode restructuring restores the height-balance property.



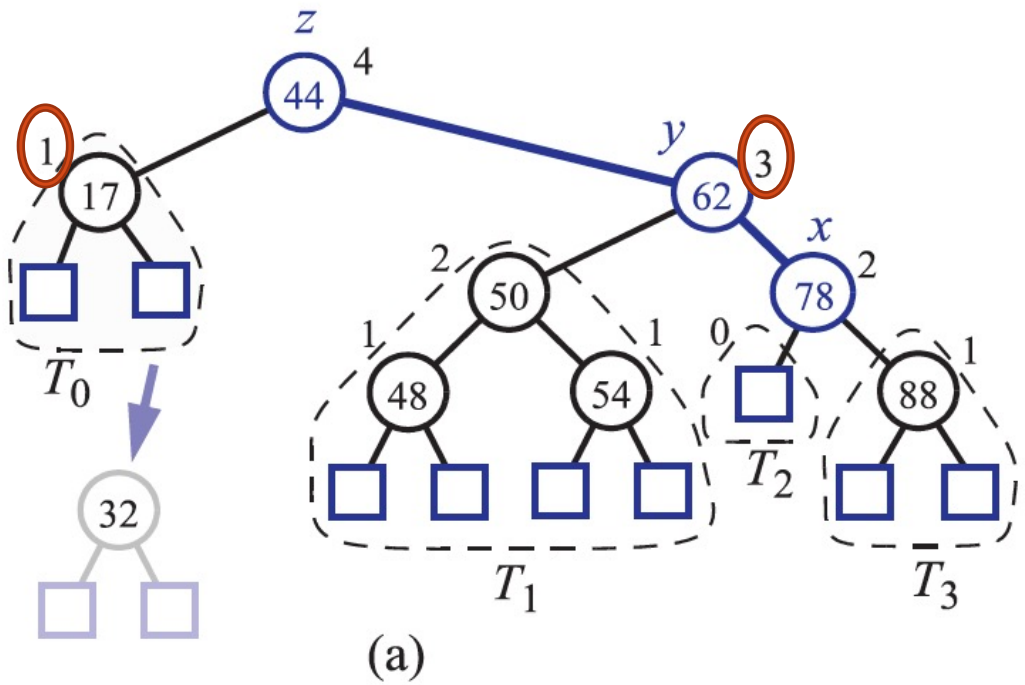
AVL TREES – REMOVAL

- We said insertion in AVL trees consists of **two steps**:
  - **Insert the node** as usual, just like in any binary search tree
  - **Fix any unbalanced nodes** using the trinode restructuring function

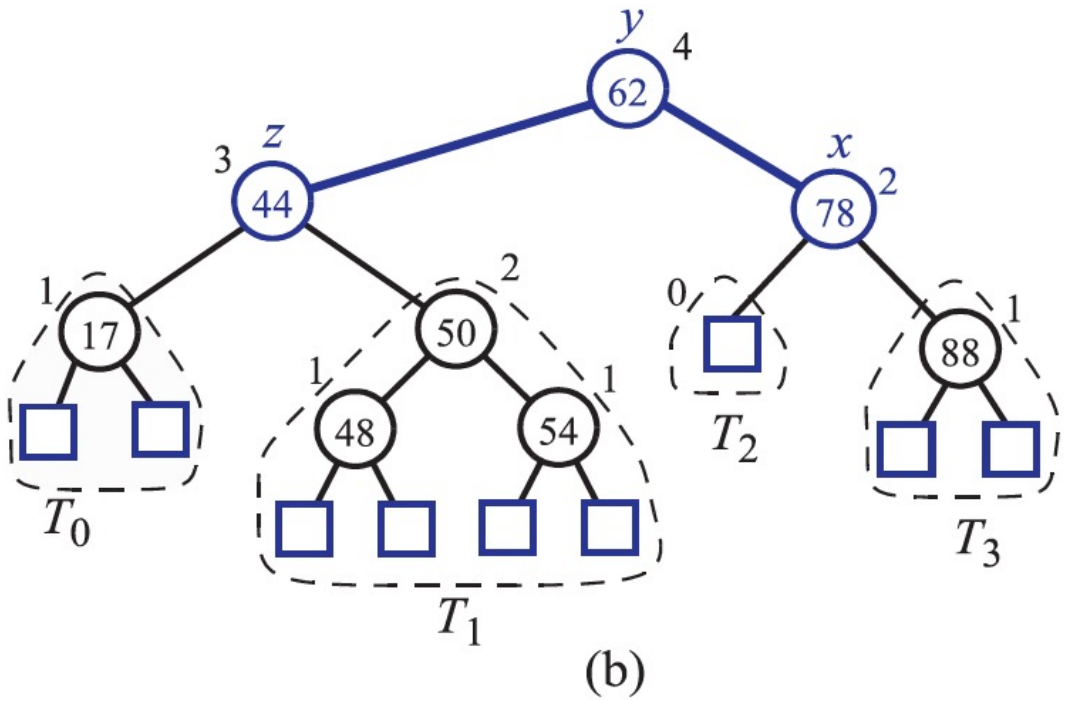
- Removing a node is done in the **same way**:
  - **Remove the node** as usual, just like in any binary search tree via removeAboveExternal( )
  - **Fix any unbalanced nodes** using the trinode restructuring function
  - However, fixing an unbalanced node may make its ancestors unbalanced, thus we need to repeatedly go up the tree, **looking for (and fixing) unbalanced nodes**!

AVL TREES – REMOVAL

Example: we want to remove node with key 32 from the following tree:



Single Rotation



Removal of the entry with key 32

Trinode restructuring restores the height-balance property

AVL TREES – COMPLEXITY

With AVL trees:

- **Searching** for a node takes O(log n) time, because:
  - Searching a binary search tree takes O(h) time, where h is the height of the tree
  - The height of AVL trees is O(log n)
- **Insertion** and **removal** take O(log n) time, because:
  - Insertion and removal require searching, which takes O(log n), as well as trinode restructuring, which takes O(1) (which could be repeated in the case of removal)