CS F211
Data Structures & Algorithms

DICTIONARY DATA STRUCTURES – SEARCH TREES

Balancing a Search Tree

- Height Balance Property

- AVL Tree

- Example

- Algorithms

-

Height-balance property

- A node **v** in a binary tree is said to be ***height-balanced*** if
  - the difference between the heights of the children of **v** – i.e. its sub-trees – is at most 1.
- **Height Balance Property**:
  - A binary tree is said to be ***height-balanced*** if each of its nodes is height-balanced.
- **Adel'son-Vel'skii and Landis** tree (or **AVL** tree)
  - Any height-balanced binary tree is referred to as an AVL tree.
- *The height-balance property keeps the height minimal*
  - How?

AVL Tree - Height

- Theorem:
  - The minimum number of nodes **n(h)** of an AVL tree of height **h** is $\boldsymbol{\Omega(c^h)}$ for some constant **c >1**.
- Proof (by induction):
  1. $n(1) = 1$ and $n(2) = 2$
  2. For h>2, $n(h) >= n(h-1) + n(h-2) + 1$

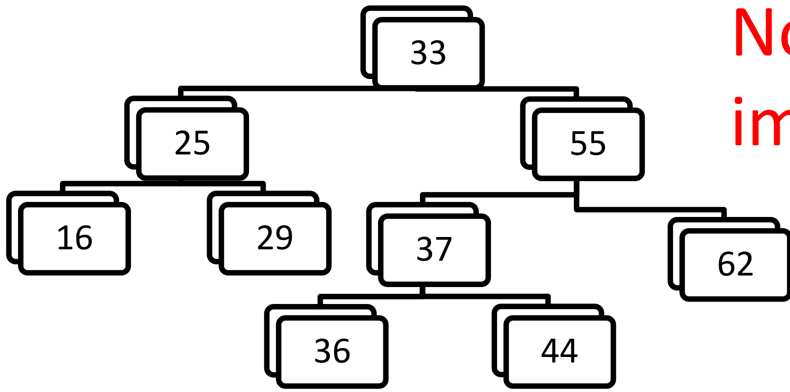
     Why?
  3. Then, n(h) is a monotonic sequence i.e. $n(h) > n(h-1)$. So, $n(h) > 2*n(h-2)$
  4. By, repeated substitution, $n(h) > 2^j * n(h-2*j)$ for $h-2*j >=1$
  5. So, n(h) is $\Omega(2^{h/2})$

AVL TREE - HEIGHT

- Corollary:
  - The height of an AVL tree with n nodes is O(log n).
  - Proof:
    - Obvious from the previous theorem.

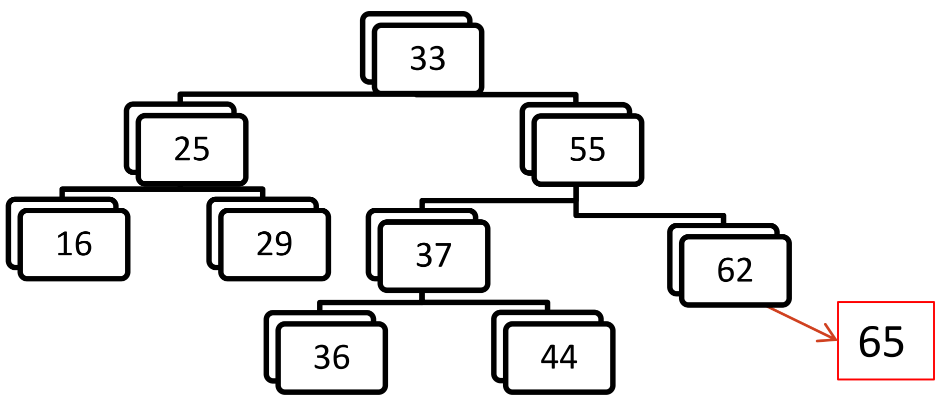
- Thus the cost of a ***find*** operation in an AVL tree with n nodes is O(log n)
  - assuming *insertion* and *deletion* preserve the height-balance property.

AVL Tree – Insertion – Example 1



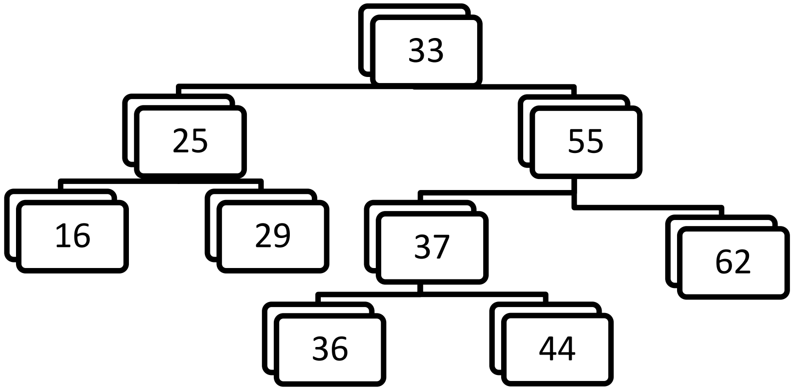
No imbalance

Insert 65:

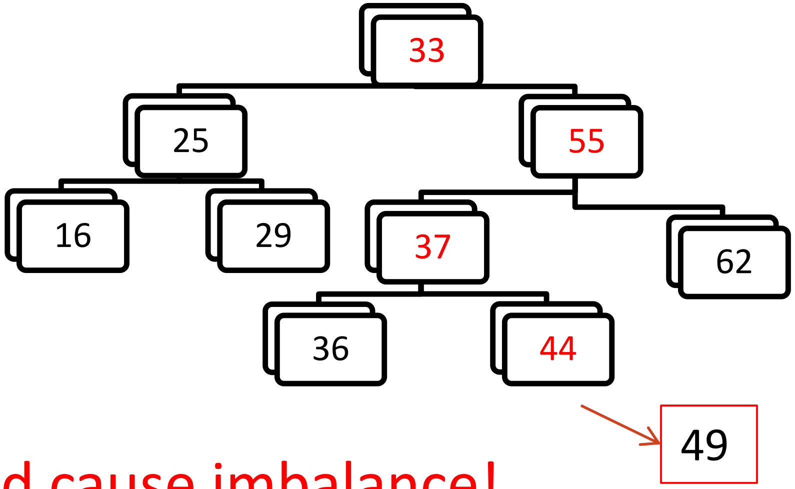


No (re-)balancing needed!

AVL TREE – INSERTION – EXAMPLE 2

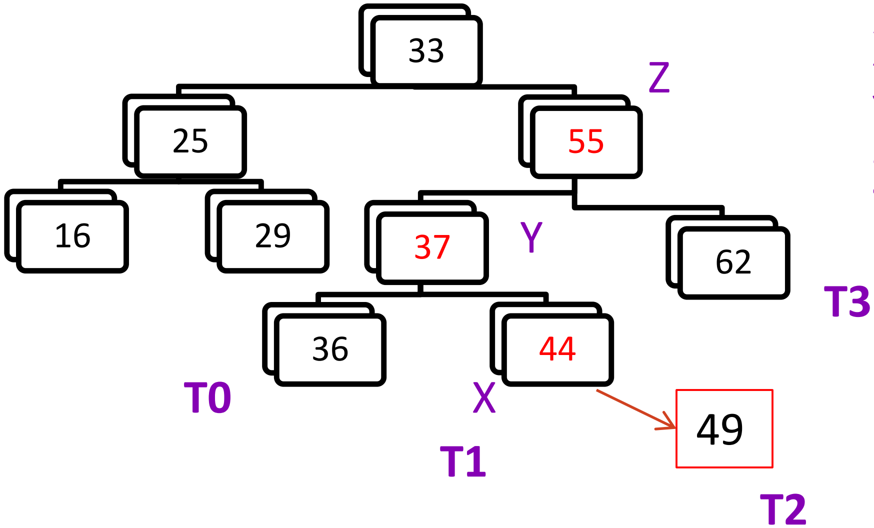


Insert 49



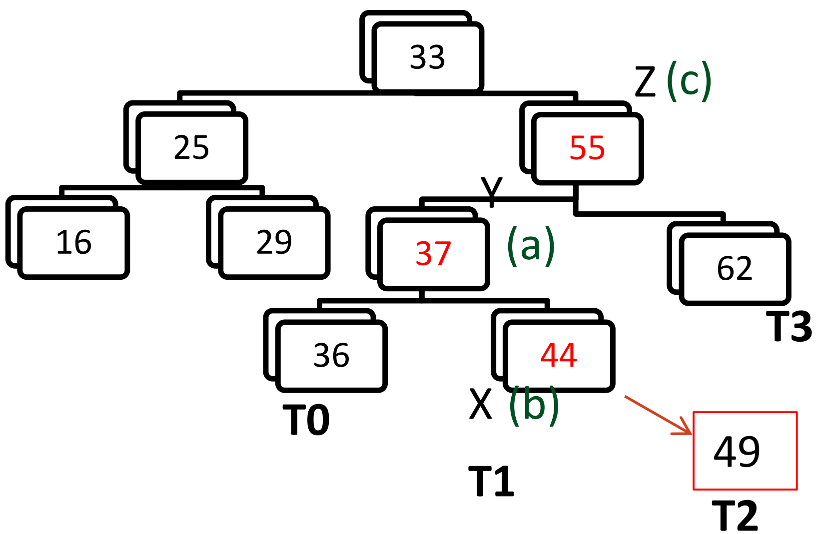
Would cause imbalance!

AVL Tree – Insertion – Example 2



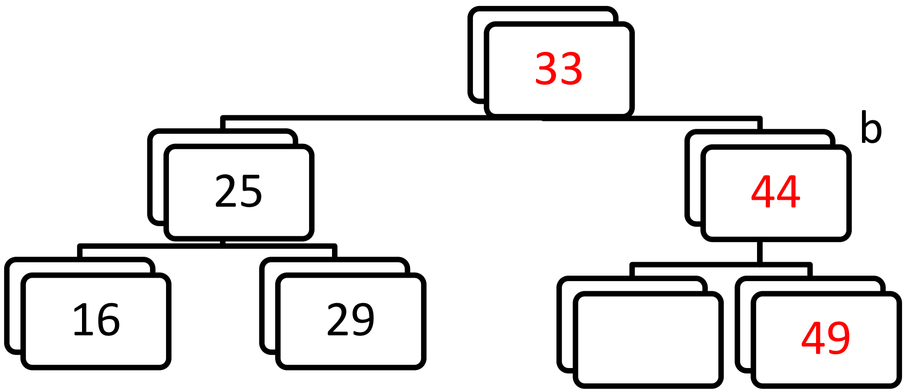
X – point of insertion
Y – parent of X
Z – parent of Y

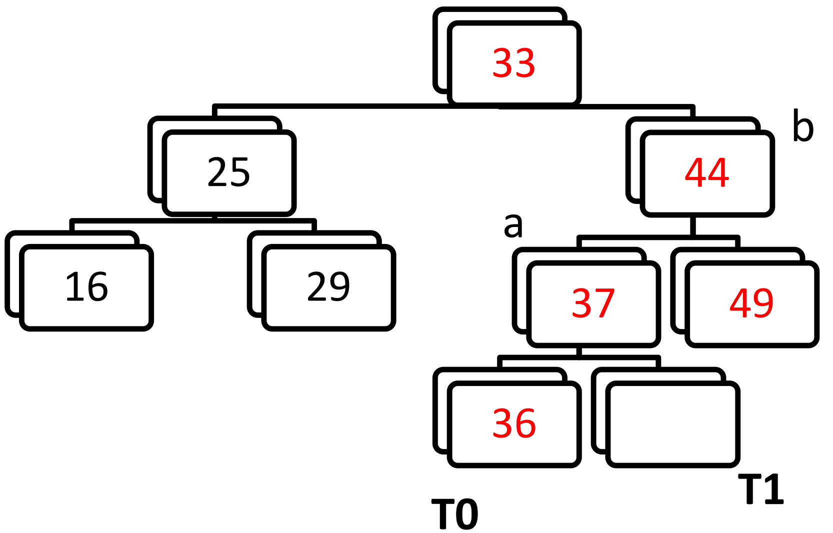
T0 – T3 : left to right listing of other sub-trees involved



(a,b,c) left-to-right listing of (X,Y,Z)

AVL Tree – Insertion e.g.2





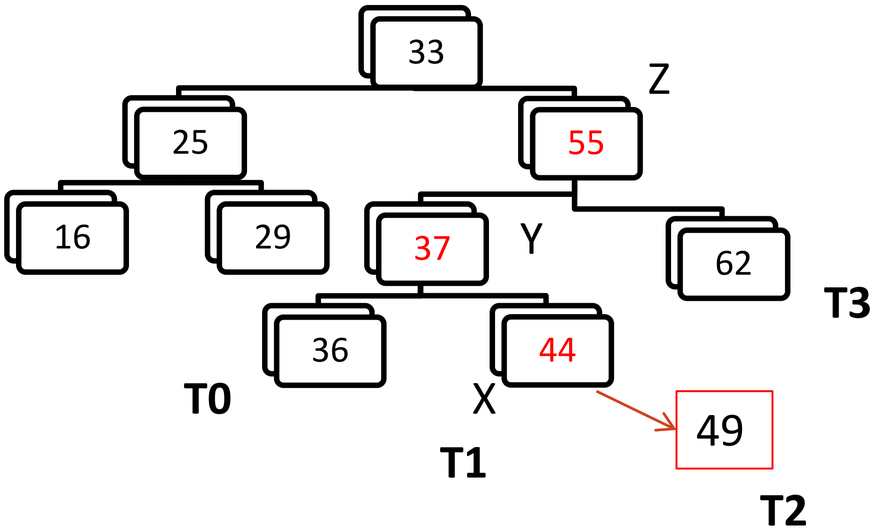
Re-structure:

Input: Z, a , b, c, and T1, T2, T3, T4

1. **Replace subtree at Z with subtree at b**

2. Set a as left subtree of b and set T0 and T1 as left & right subtrees of a

AVL Tree – Insertion – e.g 2



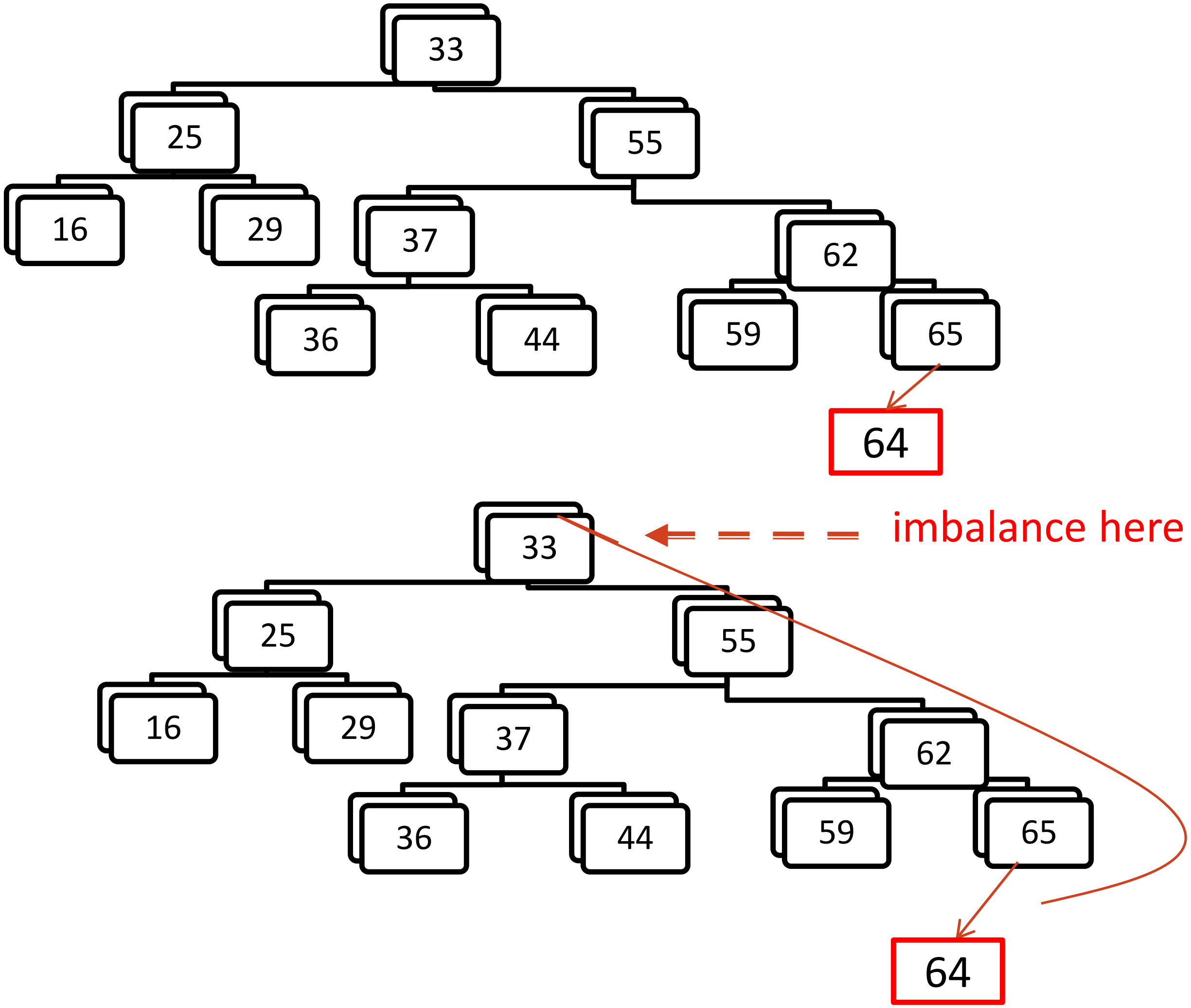
<u>Re-structure</u>:

Input: Z, a , b, c, and T1, T2, T3, T4

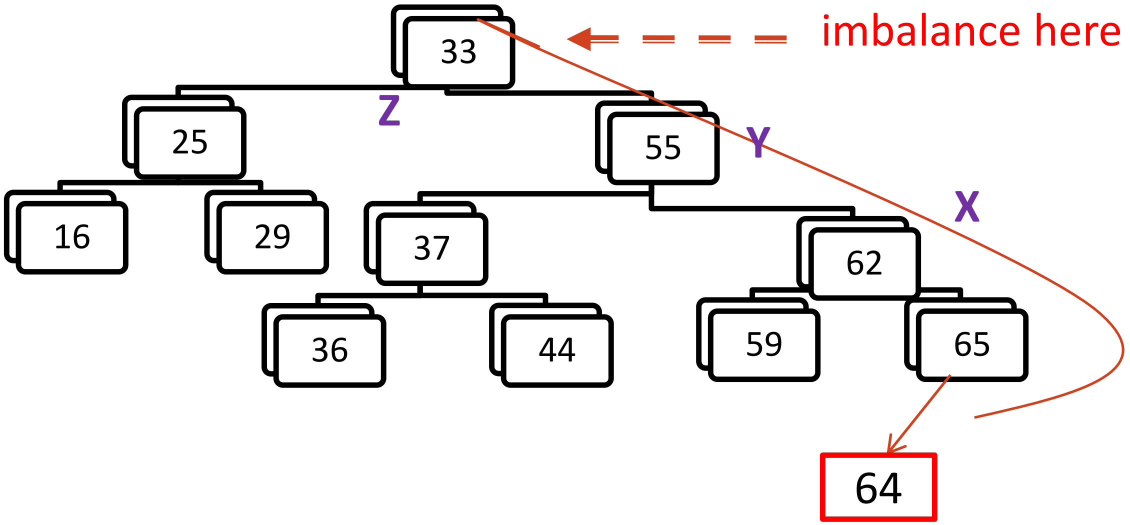
1. **Replace subtree at Z with subtree at b**
2. **Set a as left subtree of b and set T0 and T1 as left & right subtrees of a**
3. **Set c as right subtree of b and set T2 and T3 as left & right subtrees of c**

*This restructuring operation is referred to as a **rotation**.*

AVL Tree – Insertion – e.g. 3



AVL Tree – Insertion - Cases



<u>Generalized rotation</u>:

(*along the path from the inserted node to the root*)

- Let Z be the first unbalanced node.
- Let Y be the child of Z and X be the child of Y.
- Then call rotate with X,Y, and Z.

AVL Tree – ROTATION

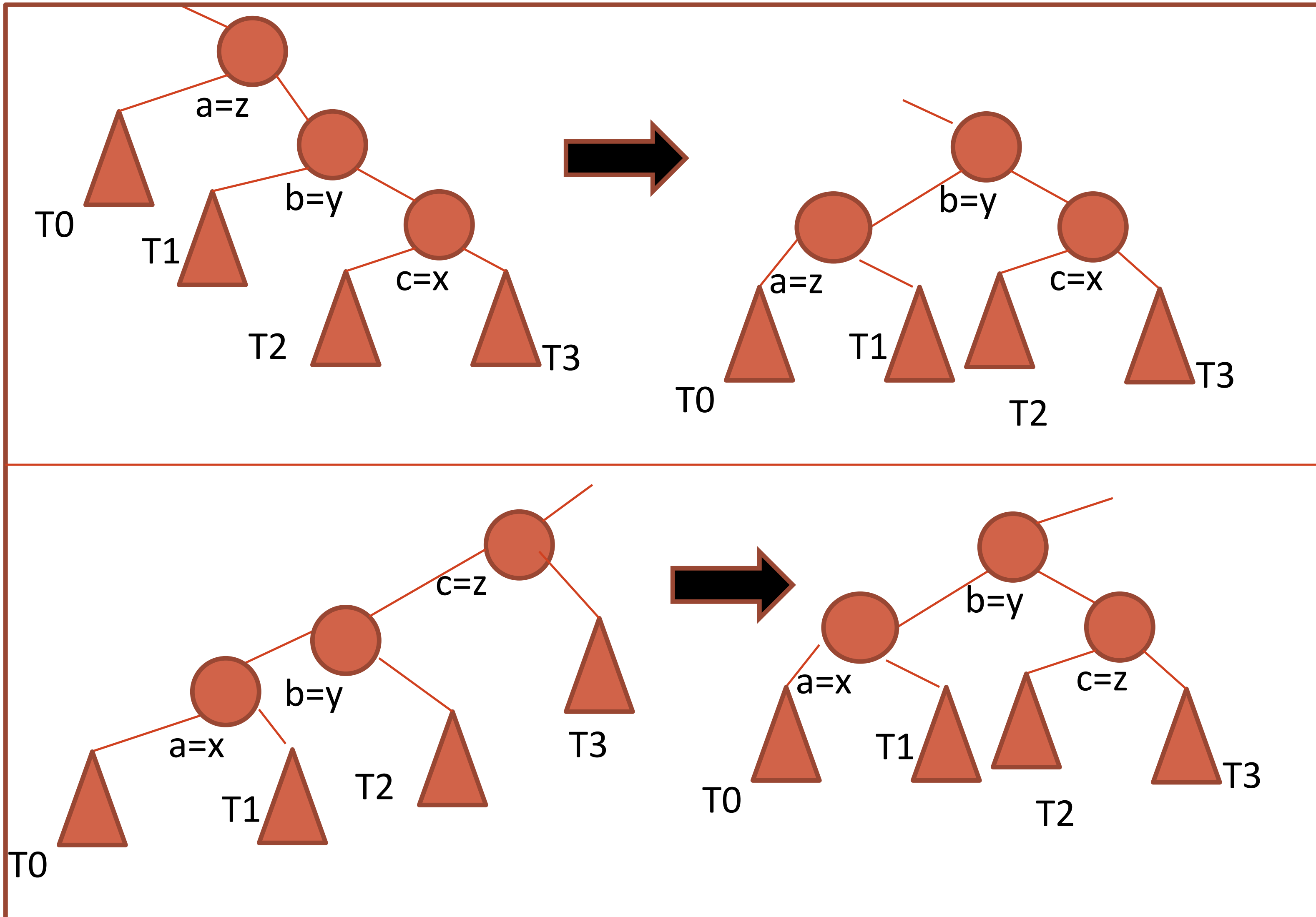
```
rotate (X, Y, Z)
{
    let a, b, c be left-to-right listing of nodes X, Y, and Z
    let T0, T1, T2, T3 be left-to-right listing of other
subtrees of x,y, and z  (i.e. subtrees of X, Y, and Z not
rooted at x or y)
Replace  Z with b;
Set a to be left child of b;
Set T0 and T1 to be left & right subtrees of a;
Set c to be right child of b;
Set T2 and T3 be left & right subtrees of c;
}
```

AVL Tree - Rotation

- The restructuring procedure is referred to as a rotation:
  - “geometric” visualization
- If b==Y then restructuring is referred to as a single rotation
  - i.e. rotating Y over Z
- If b==X then restructuring is referred to as a double rotation
- if b==Z?
  - Argue that this case cannot happen
- Exercise: Draw templates for each possible case. How many of them are there?

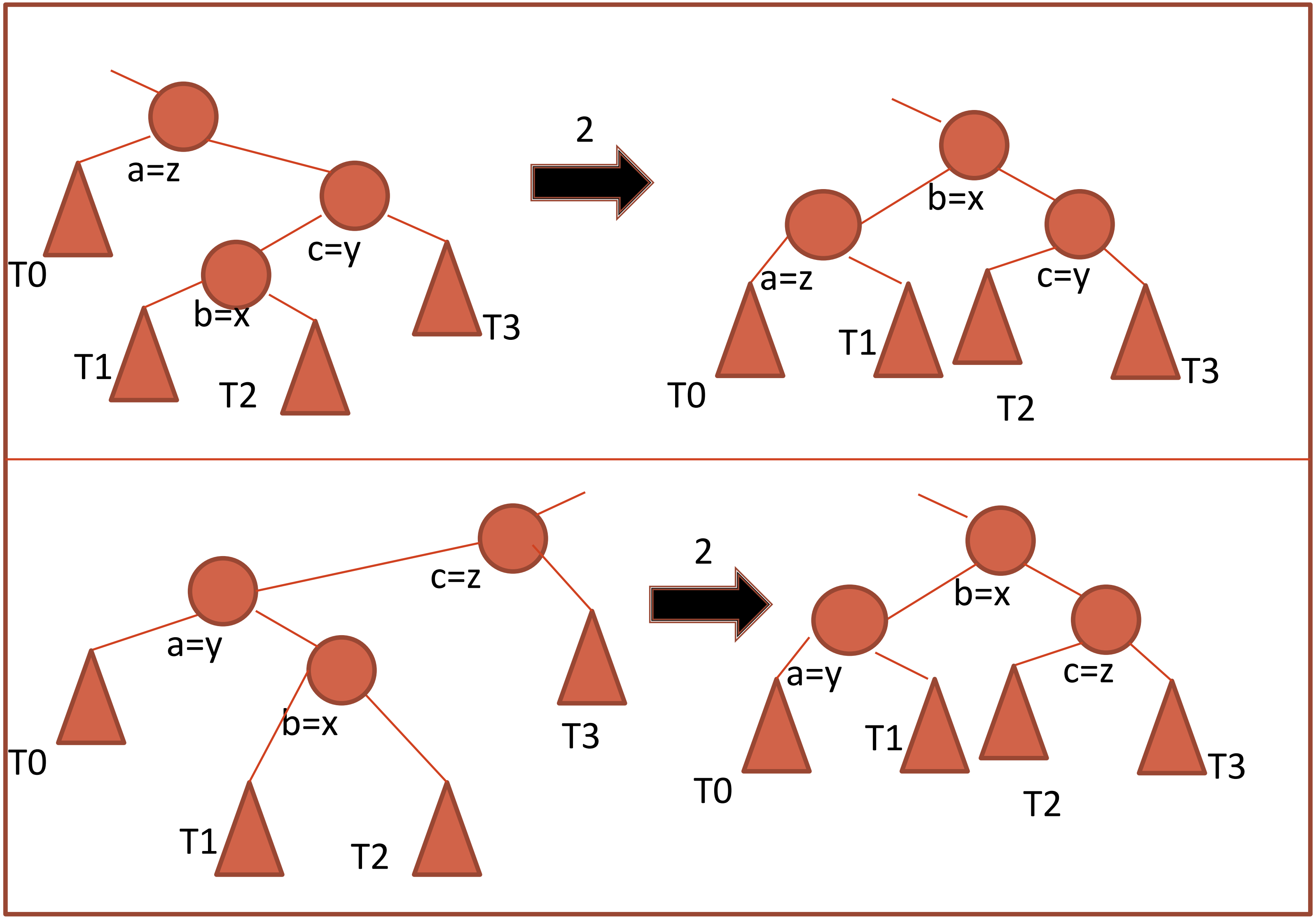
AVL Rotation Cases – Single Rotation

b=y : 2 cases



AVL Rotation Cases – Double Rotation

b=x : 2 cases



AVL Tree - Deletion

- After deletion of node W, – if W is internal, pull one of its descendants up (as in binary search tree).
  - This may result in imbalance (at some ancestor of W)
- Restructuring:
  - Z : first unbalanced node on the path from the deleted node to the root.
  - Y : child of Z with larger height (it won't be an ancestor of W)
  - X : child of Y with larger height (break ties arbitrarily).
  - Then call **rotate(X,Y, Z)**
- Claims:
  - This balances the node Z (locally) – Why?
  - This does not the balance the tree (globally) – Why?

AVL Tree - Deletion

- After deletion of node W:
    1. if W is internal, pull one of its descendants up (as in binary search tree).
    2. Let Z be the first unbalanced ancestral node on the way up. Balance Z by rotation.
    3. Repeat step 2 until the root is balanced.

AVL Tree – Time Complexity

- Time Complexity of
  - Find:
    - O(h) and h is log N
  - Insert:
    - O(h) for finding the right position and O(1) for rotation
    - Total time is O(log N)
  - Delete:
    - O(h) for finding the right node (to be deleted) and O(h) rotations, each rotation taking time O(1).
    - Total time is O(log N)

AVL TREES – IMPLEMENTATION ISSUES

- How do we check for an unbalanced node?
  - Every node maintains a (relative) weight:
    - 0 ==> balanced
    - 1 ==> right sub tree is taller
    - -1 ==> left sub tree is taller
  - On insertion:
    - Weights are to be updated
    - If insertion happens on the right sub tree of node with weight 1 then it *may become unbalanced*
    - Similarly for a left sub tree of node with weight -1

DICTIONARY - COMPARISON

Balanced BST

- Time Complexity:
  - ϴ(logN) - worst case and average case
- Space Complexity
  - ϴ(N) links,
  - ϴ(N) space for counts (height balance info.)

Hashtable

- Time Complexity:
  - ϴ(1) average case and ϴ(N) worst case
- Space Complexity
  - ϴ(N) words – separate chaining (Table and links)
  - ϴ(N) bits – empty/non-empty

AVL Tree –Complexity

- Despite the improved time complexity, Hashtables are preferred to AVL trees in practice:
  - Most often hashtables behave well – O(1) operations with high probability
  - Implementation is complex for AVL trees
  - Rotations in AVL tree destroy locality of memory references.*
    - Why? [Consider the pointer / subtree changes.]
    - Affects caching / paging behavior resulting in bad performance.
  - Update of height balance information results in dirty caches / pages *
    - Virtual Memory performance suffers

AVL TREE –COMPLEXITY [2]

- AVL Trees are preferred only if
  - bound O(log N) is strictly needed OR
  - Ordered operations are needed.
    - E.g. find the minimum element
    - find all elements with key < K in order