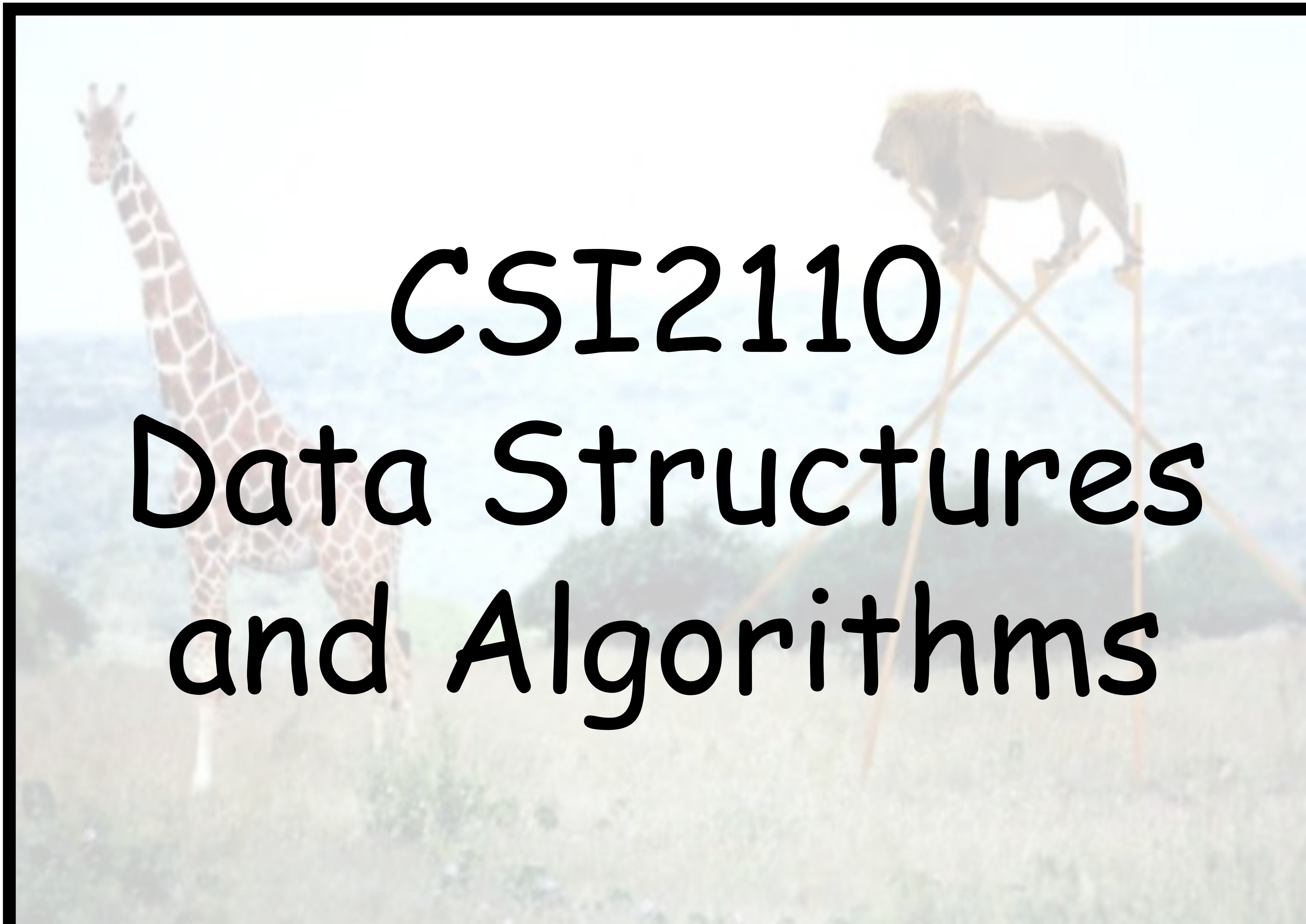
CSI2110 Data Structures and Algorithms



- https://www.youtube.com/watch?v=aQS9DqLWxw4

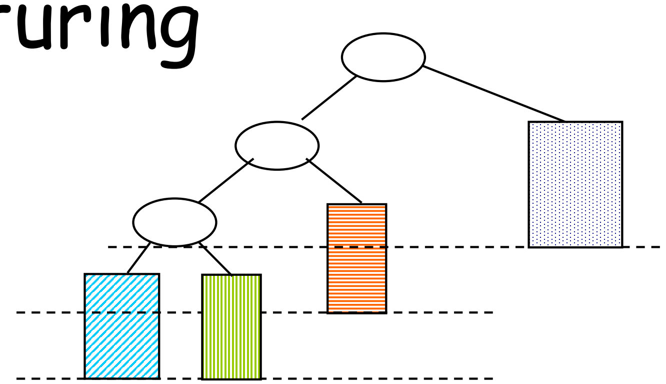
- https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

AVL Trees

Adel'son-Vel'skii and Landis

Data structure that implements MAP ADT

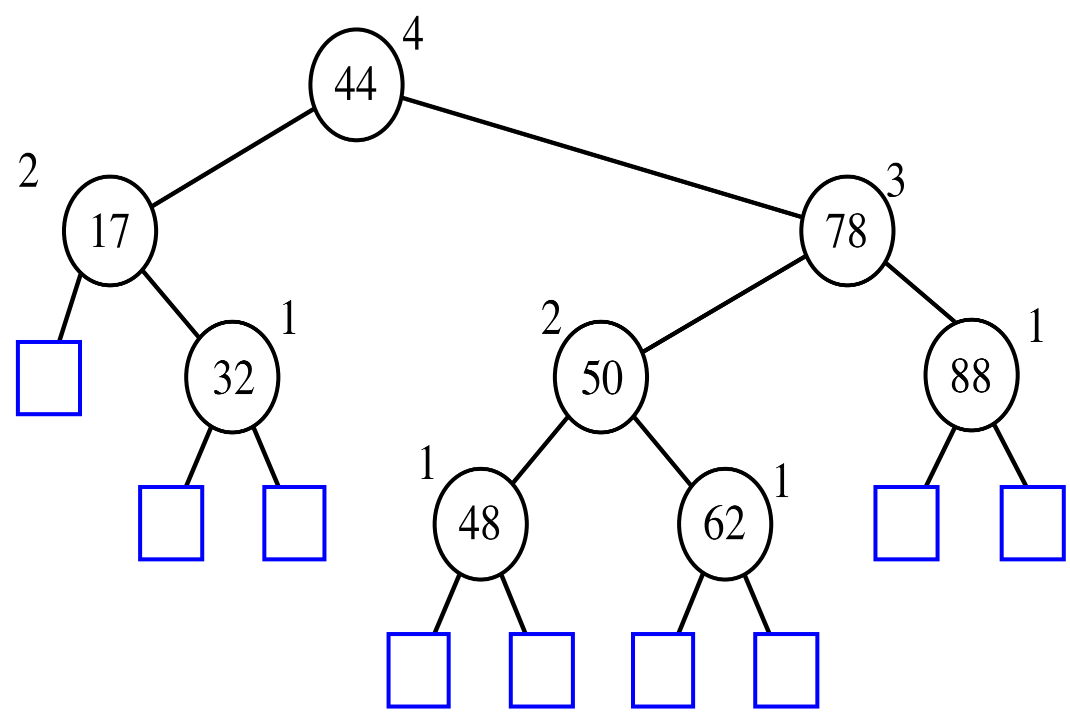
- Height of an AVL Tree
- Insertion and restructuring
- Removal and restructuring
- Costs



AVL Tree

- **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*.

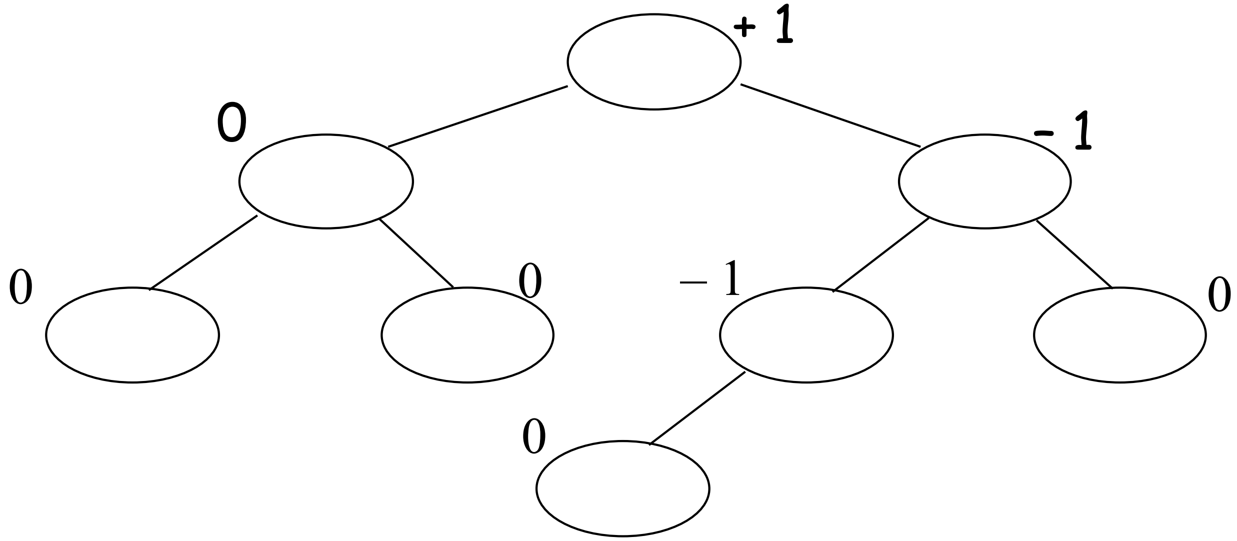
An example of an AVL tree where the heights are shown next to the nodes:



Balancing Factor

height(right subtree) – height(left subtree)

$\in\{-1, 0, 1\}$ for AVL tree



Height of an AVL Tree

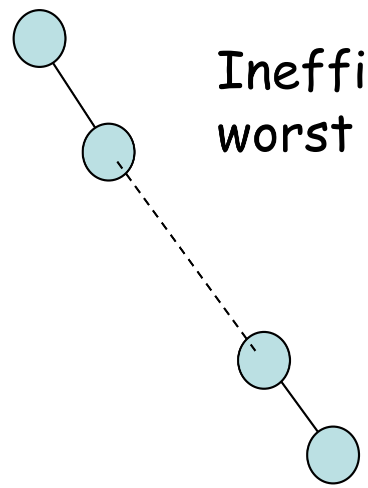
Note: "longest" possible heap with n nodes.

Always O(log n)

But cannot search efficiently

Note: "longest" possible binary tree with n nodes:

O(n)

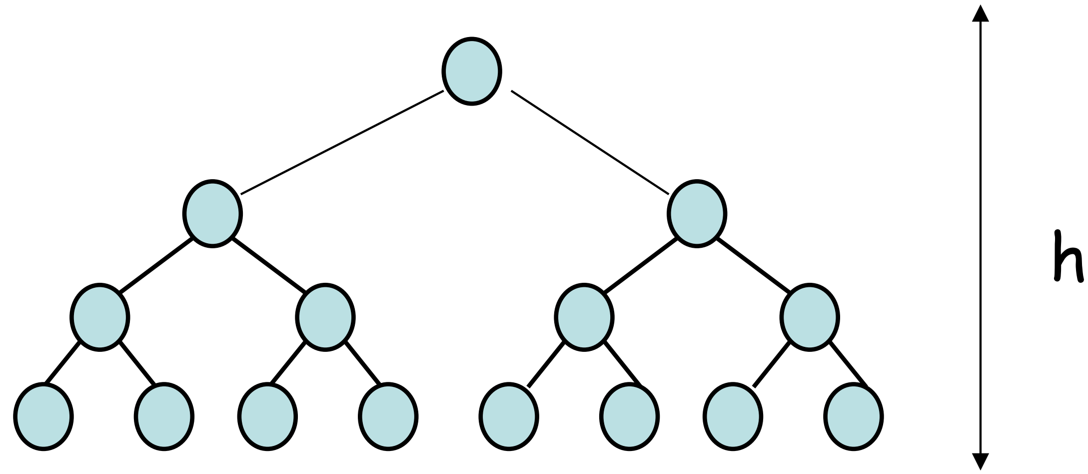


Inefficient to search in the worst case.

We'll now see that the ***height*** of an AVL tree T storing n keys is O(log n).

Note: AVL tree with the highest possible number of internal nodes for a given height h:

Perfect:



$n = 2^h$

So $h = \log n$

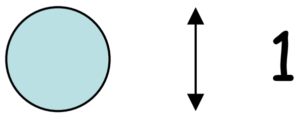
To construct the "longest" possible AVL tree, we look for

the minimum number of nodes of an AVL tree of height h.

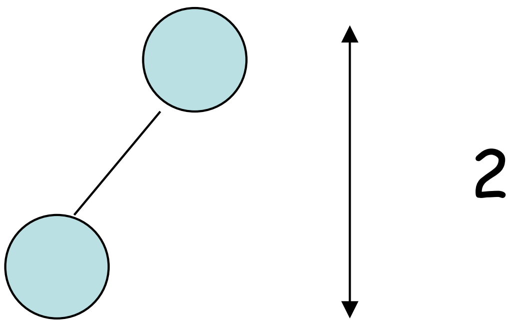
$n(h)$

Easy to see that $n(1) = 1$ and $n(2) = 2$

(please note that dummy nodes are not shown but contributing to the height)



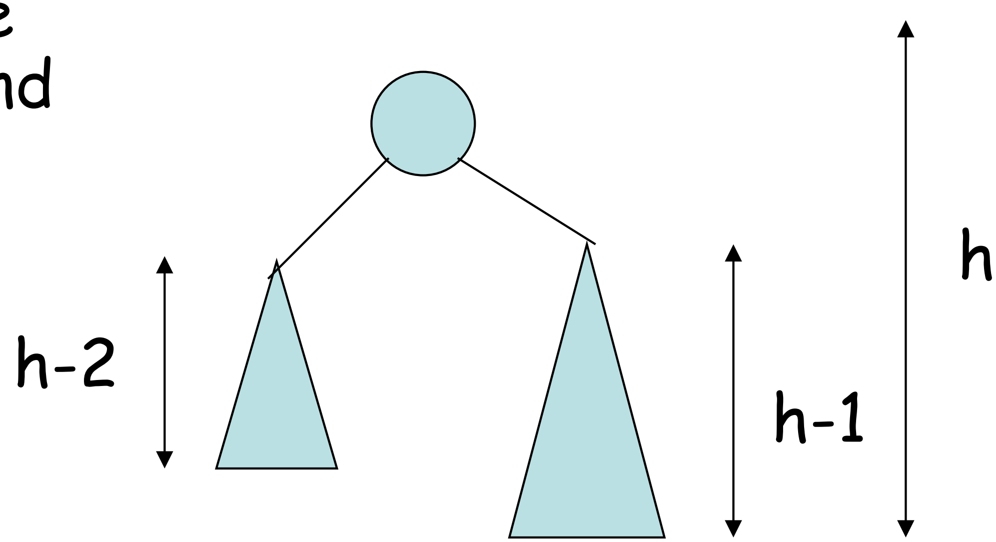
$n(1) = 1$



$n(2) = 2$

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

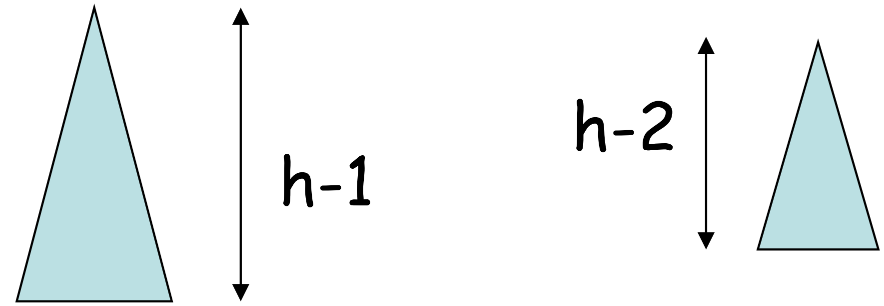
For n ≥ 3, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and the other AVL subtree of height h-2.



$$n(h) = 1 + n(h-1) + n(h-2)$$

Height of an AVL Tree

$$n(h) = 1 + \boxed{n(h-1)} + n(h-2)$$



Clearly: $\boxed{n(h-1) > n(h-2)}$

so: $n(h) > 1 + n(h-2) + n(h-2)$

$n(h) > 1 + 2\,n(h-2) > 2\,n(h-2)$

$n(h) > 2n(h-2)$

for any h

Height of an AVL Tree

So, now we know: $n(h) > 2\ \boxed{n(h-2)}$

but then also: $\boxed{n(h-2)} > 2\ n(h-4)$

$$n(h) > 4\ n(h-4)$$

$n(h) > \boxed{4n(h-4)}$

but then also: $\boxed{n(h-4)} > 2n(h-6)$

$$n(h) > 8n(h-6)$$

We can continue:

$n(h) > 2n(h-2)$

$n(h) > 4n(h-4)$

$n(h) > 8n(h-6)$

...

$n(h) > 2^i n(h-2i)$

$n(h) > 2^i n(\underline{h-2i})$

$h-2i = 2$

And we know that

$n(1) = 1$

$n(2) = 2$

Now we pick I such that h is either 1 or 2

That is pick i= ceil(h/2)-1, and substitute:

for i = ceil(h/2) - 1 $\quad n(h) > 2^{ceil(h/2) - 1}\, n(1)$

$n(h) > 2^{h/2-1}$

$\log n(h) > \log 2^{(h/2)-1}$

$\log n(h) > h/2 - 1$

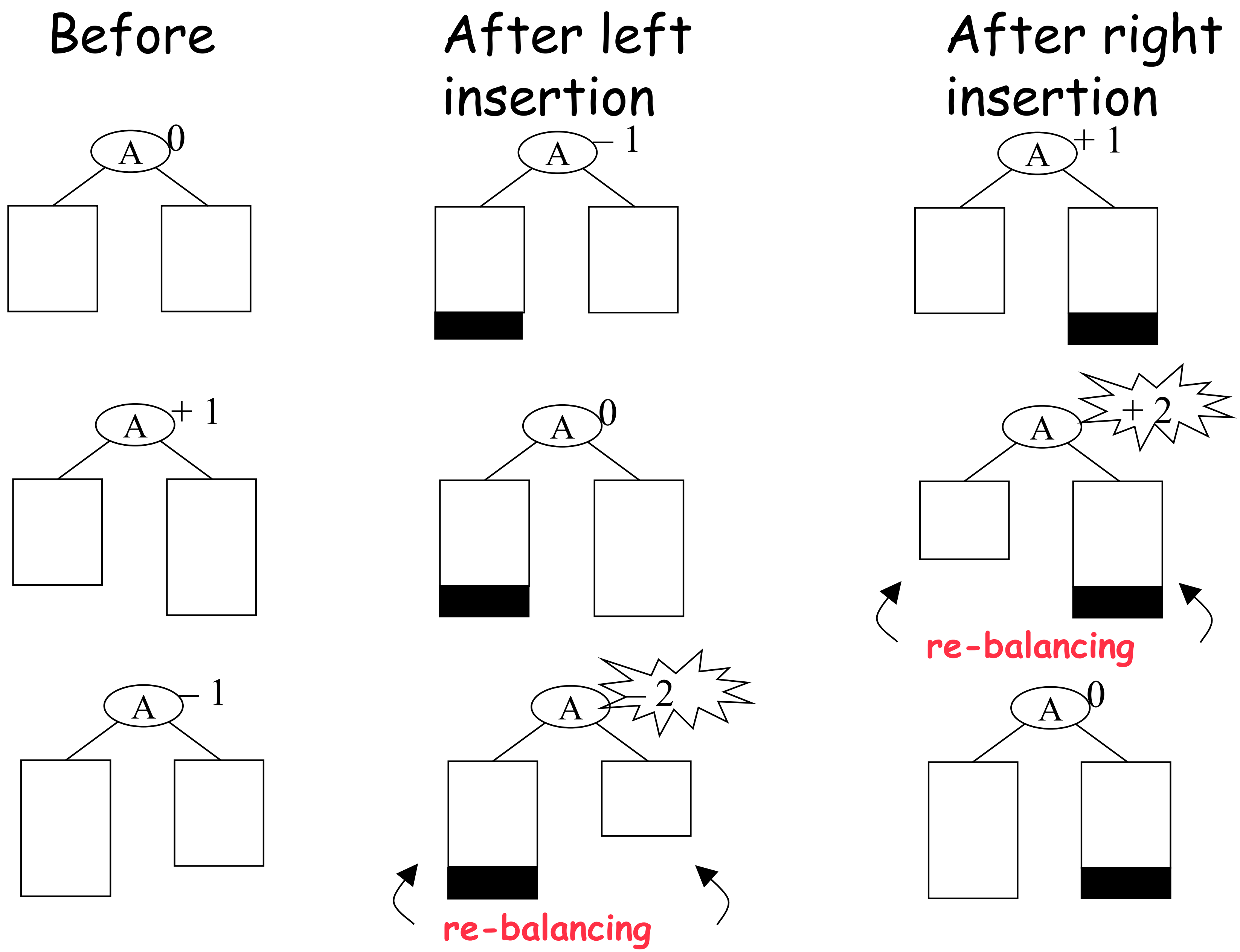
$h < 2 \log n(h) + 2$

which means that h is **O(log n)**

Insertion

- A binary search tree T is called ***balanced*** if for every node v, the height of v's children differ by at most one.
- Inserting a node into an AVL tree involves performing an expandExternal(w) on T, which changes the heights of some of the nodes in T.
- If an insertion causes T to become unbalanced we have to rebalance...

Insertion


Before
After left insertion
After right insertion
A
0
A
– 1
A
+ 1
A
+ 1
A
0
A
+ 2
A
– 1
A
2
A
0
re-balancing
re-balancing

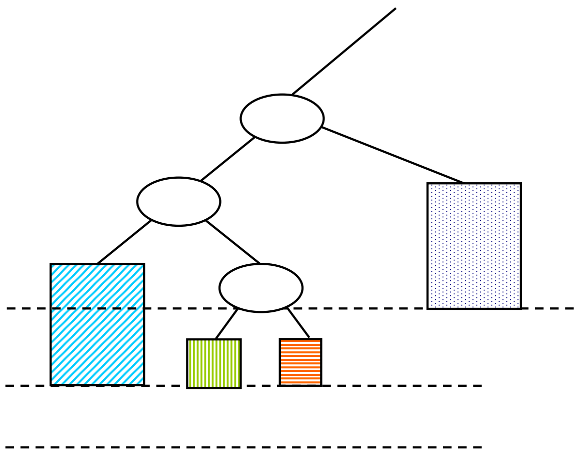
Rebalancing after insertion

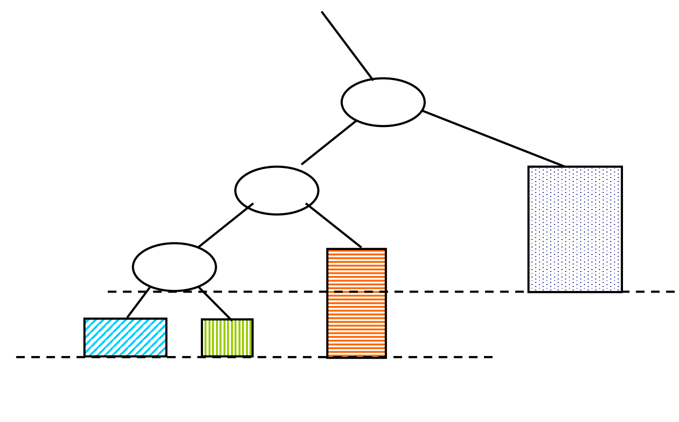
We are going to identify 3 nodes which form a grandparent, parent, child triplet and the 4 subtrees attached to them. We will rearrange these elements to create a new balanced tree.

Rebalancing

Step 1: Trace the path back from the point of insertion to the first node whose grandparent is unbalanced. Label this node x, its parent y, and grandparent z.

Examples .....

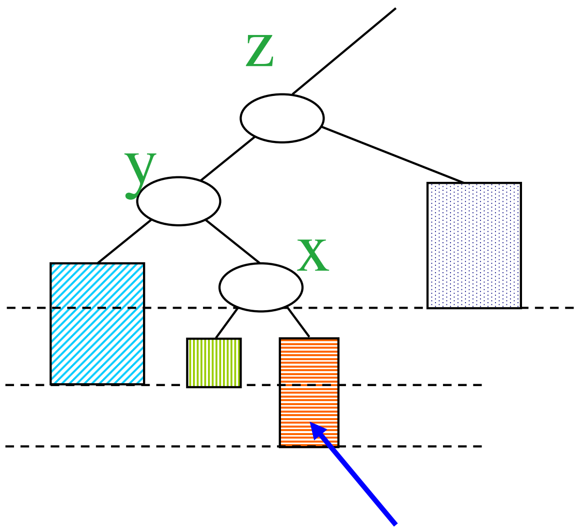


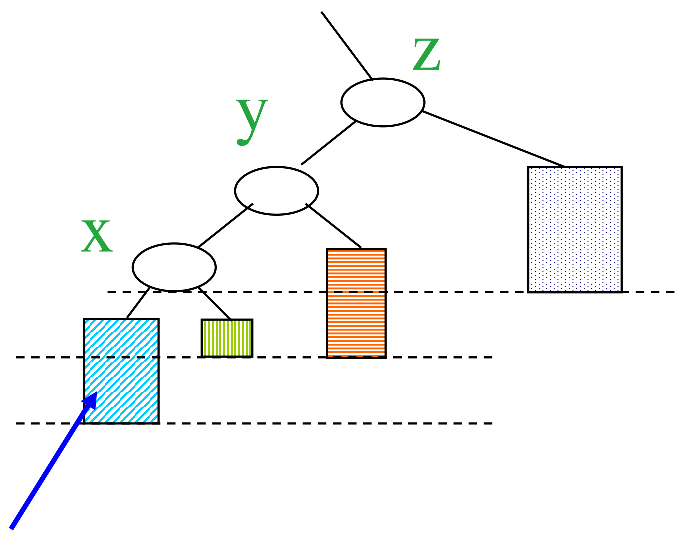


Rebalancing

Step 1: Trace the path back from the point of insertion to the first node whose grandparent is unbalanced. Label this node x, its parent y, and grandparent z.

Examples .....

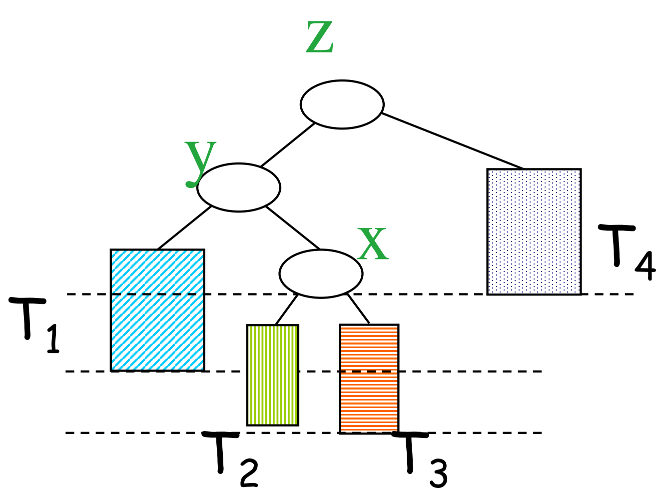


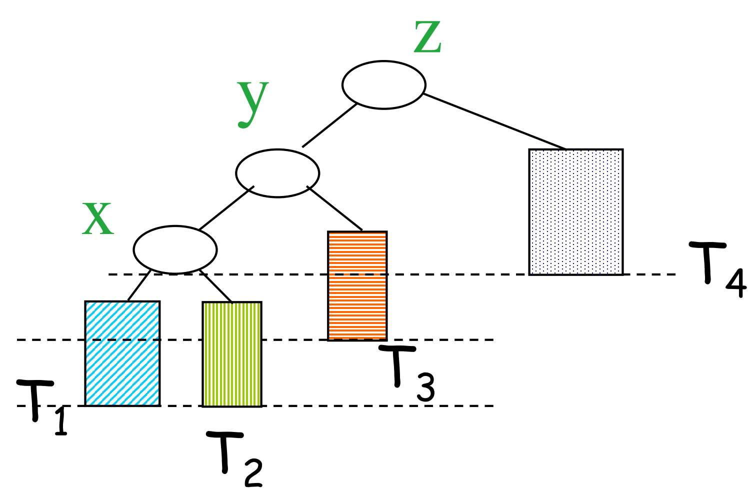


Rebalancing

Step 2: These nodes will have 4 subtrees connected to them. Label them T_1, T_2, T_3, T_4 from left to right.

Examples .....

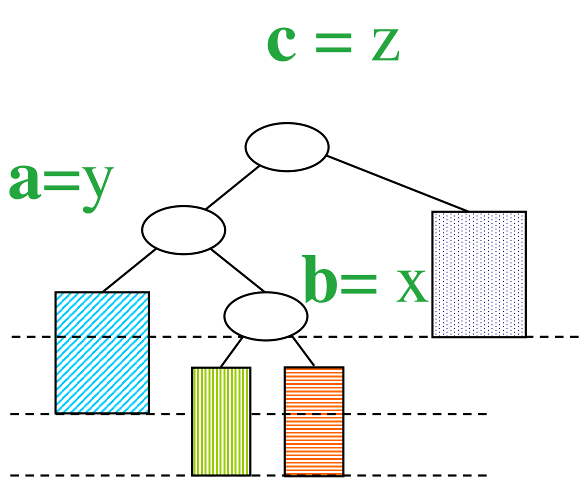




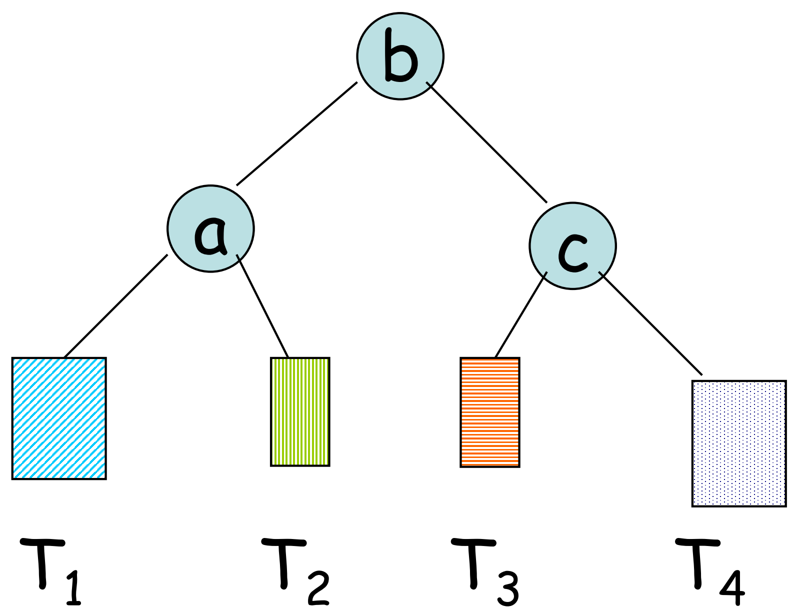
Rebalancing

Step 3: Rename x, y, z to a, b, c according to their inorder traversal i.e. if y,x,z is the relative order of those nodes following the inorder traversal then label y 'a', x 'b' and z 'c'.

Example

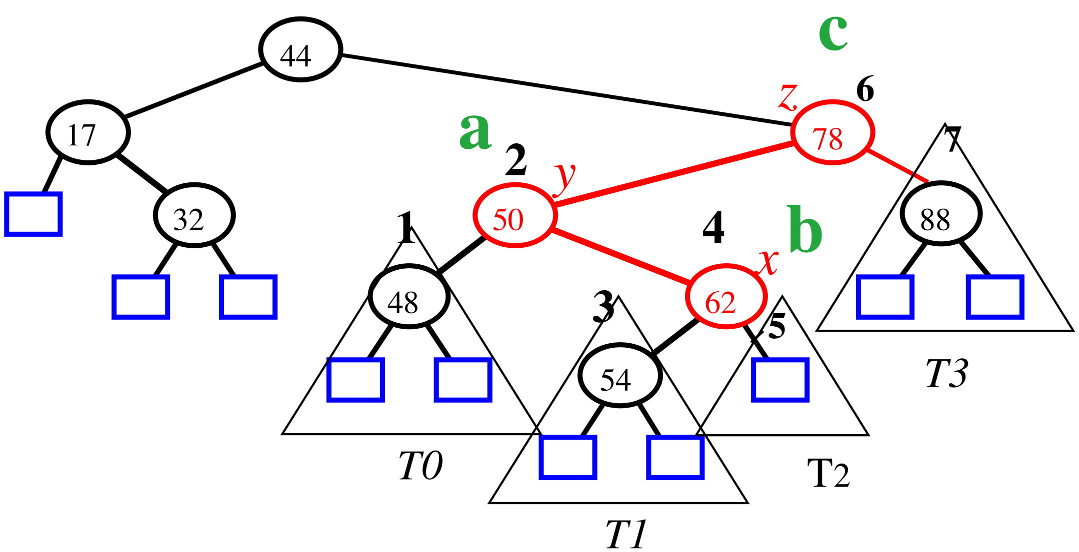


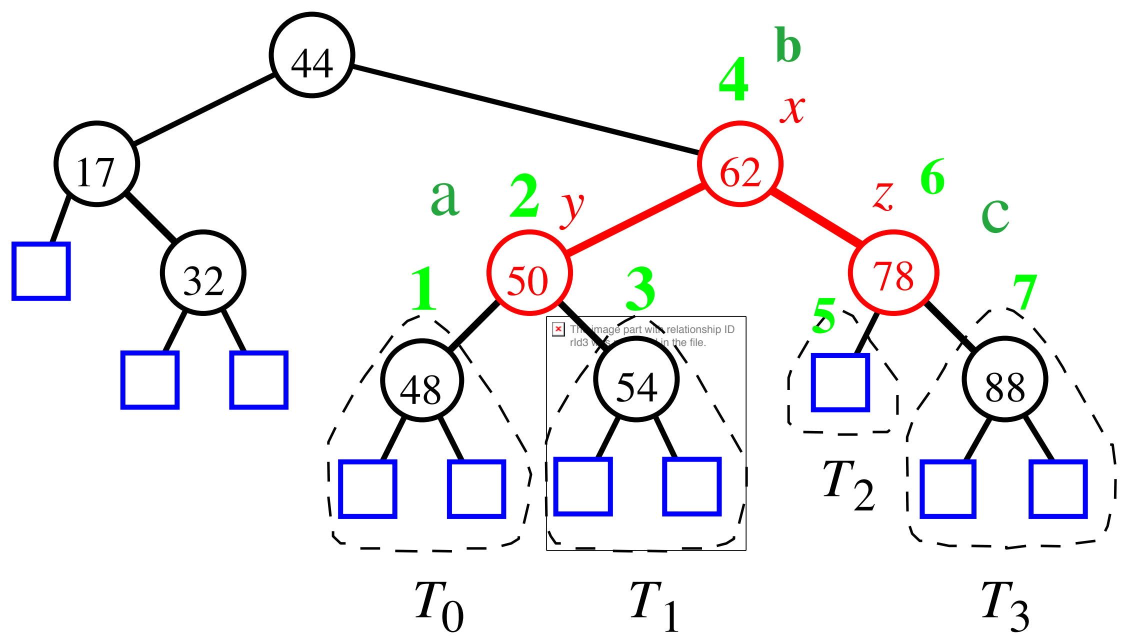
Step 4: Replace the tree rooted at z with the following tree:



Rebalance done!

Example: after inserting 54





Does this really work?

We need to see that the new tree is :

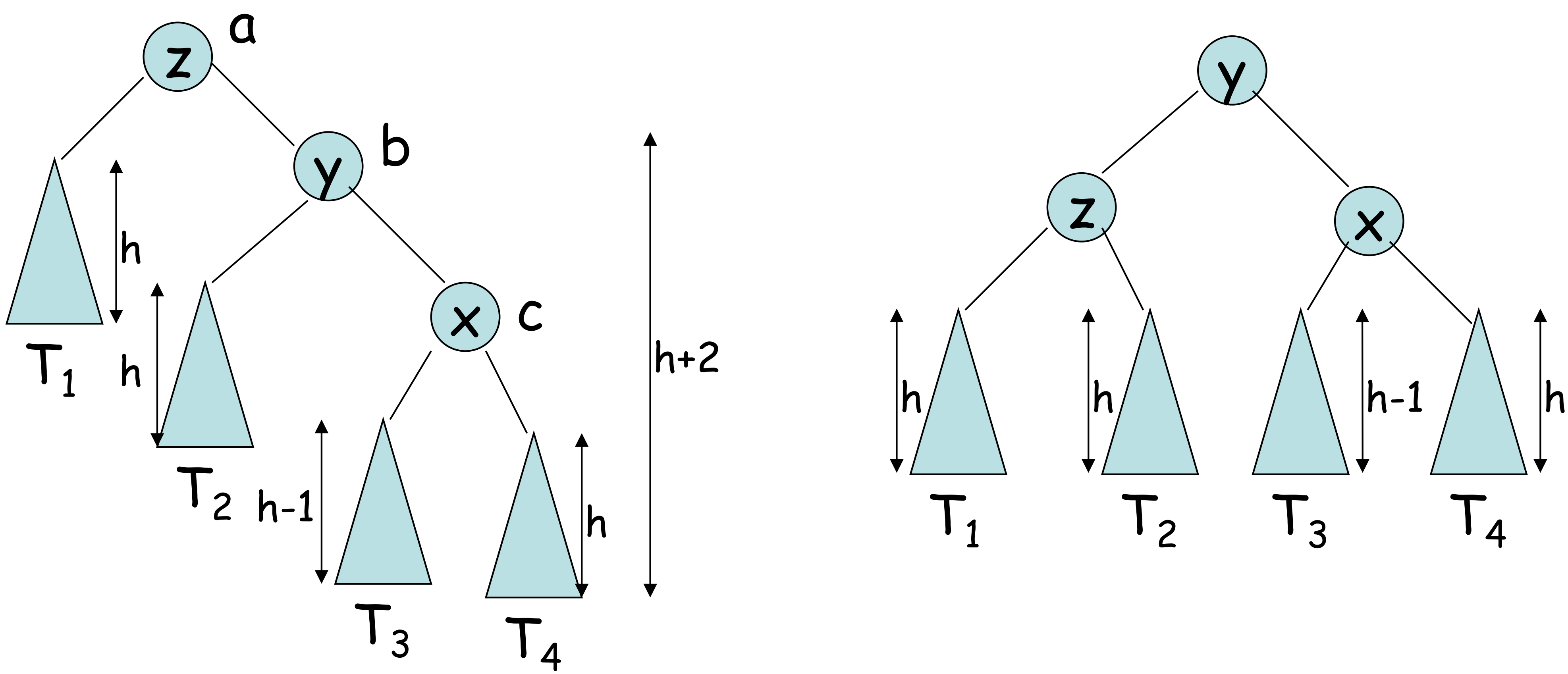
a) A Binary search tree - the inorder traversal of our new tree should be the same as that of the old tree

Inorder traversal: by definition is T1 a T2 b T3 c T4

b) Balanced: have we fixed the problem?

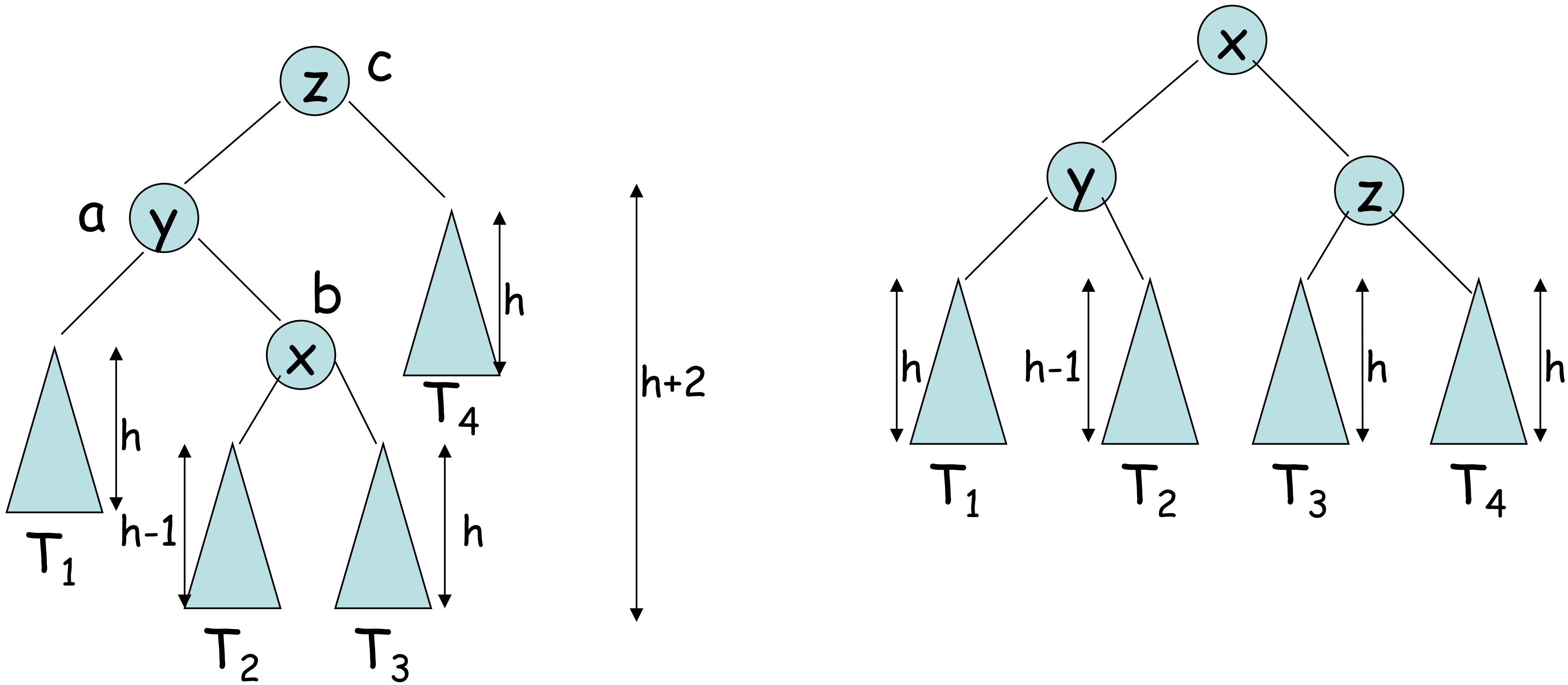
We consider 2 types of examples

Example 1



Inorder: T1 z T2 y T3 x T4

Example 2



Inorder: T1 y T2 x T3 z T4

An Observation…

Notice that in both cases, the new tree rooted at b has the same height that the old tree rooted at z had before insertion.

So.. once we have done one rebalancing act, we are done.

rebalance (v)

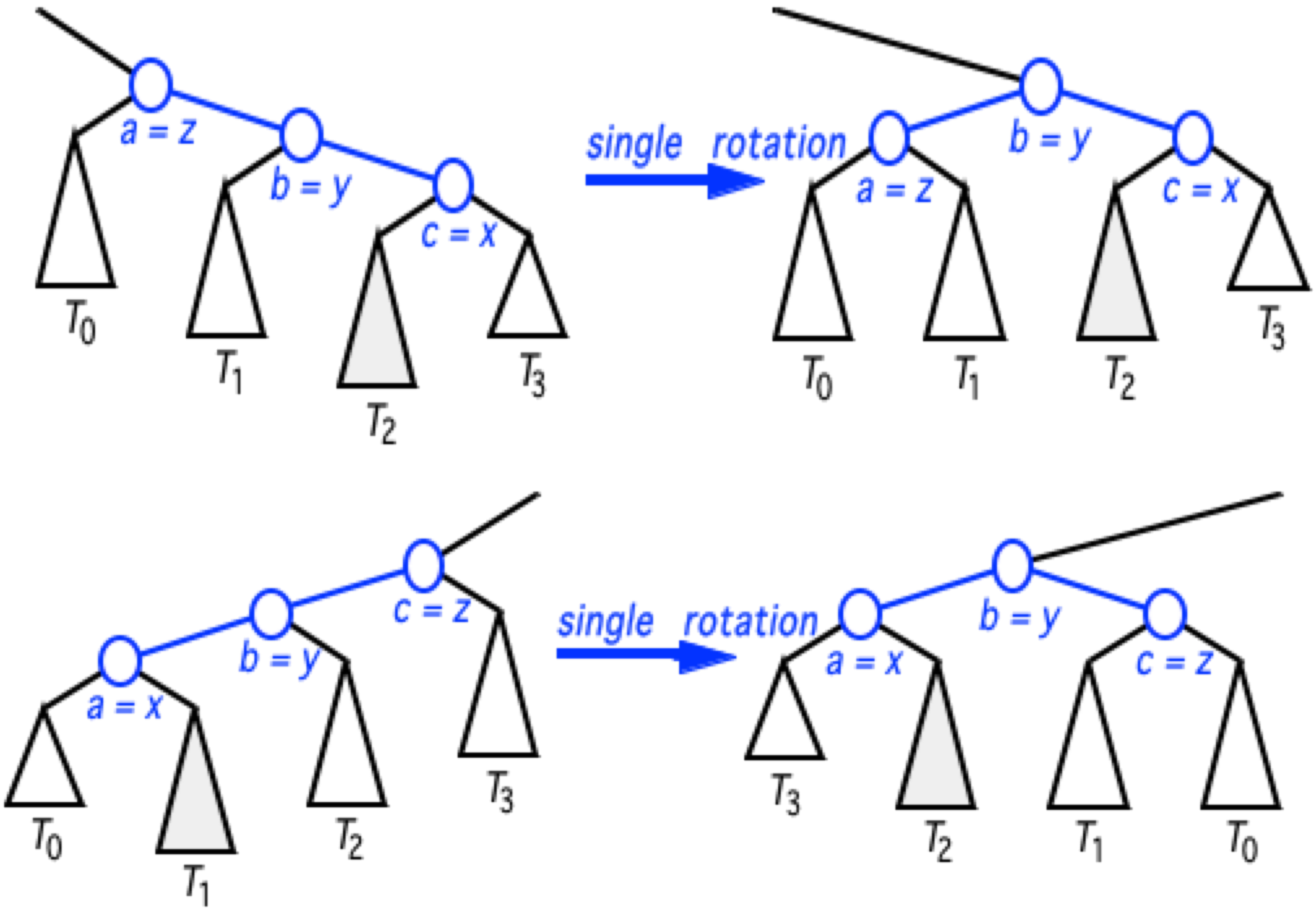
```
x <- v; Y <- x.parent; z <- y.parent
while (z.isBalanced and not(z.isRoot) )
   x <- y; y <- z; z <- z.parent
if ( not z.isBalanced)
   if ( x = y.left)   { x<=y}
      if (y = z.left) {x<=y<=z}
            a <- x; b <- y; c<- z;
            T2 <- x.right; T3 <- y.right;
      else              { z<=x<=y}
            a <- z; b <- x; c <- y;
            T2 <- x.left; T3 <- x.right;
  else                 {y<=x}
      if (y = z.left)  {y<=x<=z}
            a <- y; b <- x; c <- z;
            T2 <- x.left; T3 <- x.right

      else              { z<=y<=x}
            a <- z; b <- y; c <- x;
            T2 <- y.left; T3 <- x.left
```

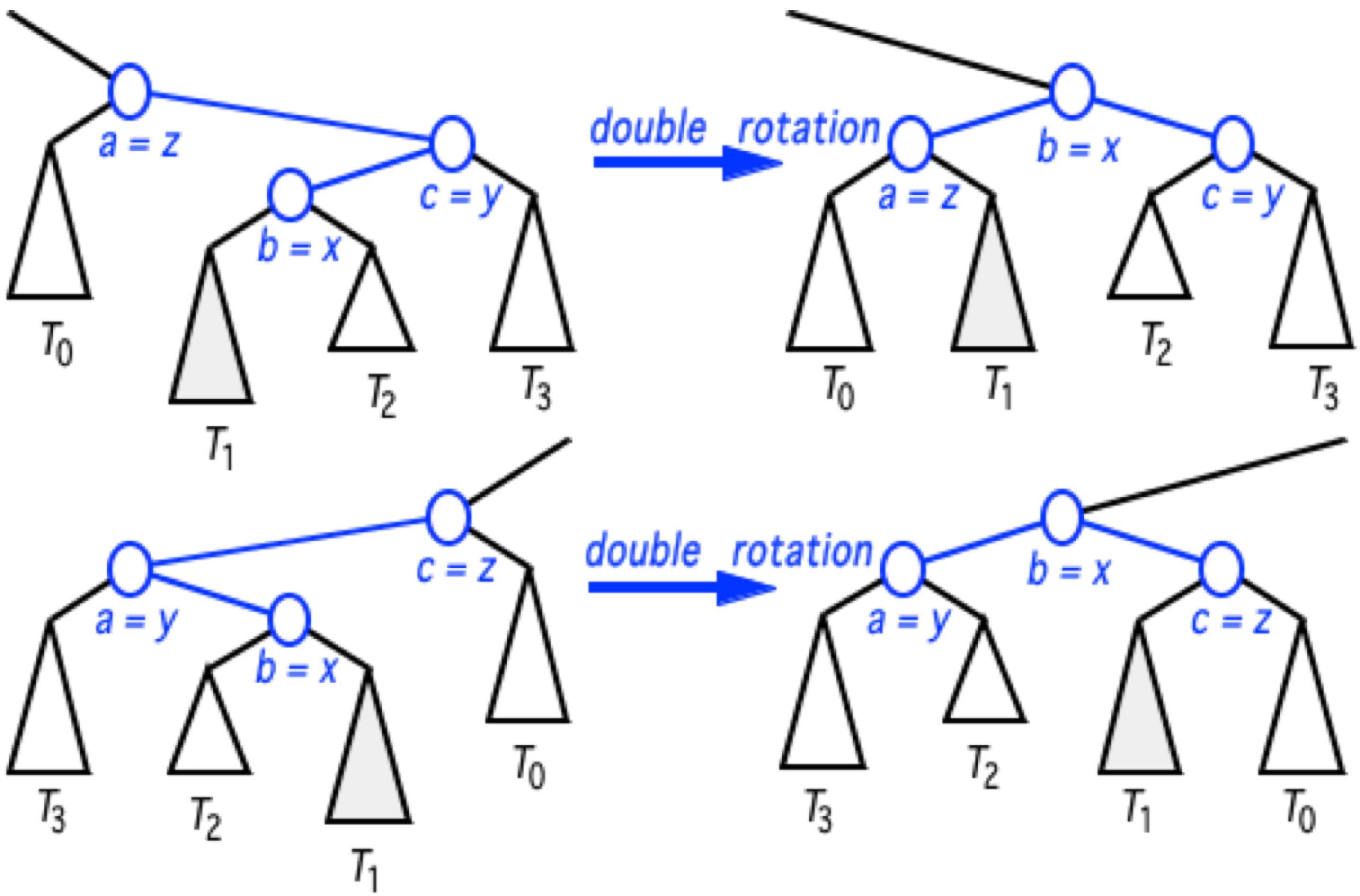
```
 T1 <- a.left; T4 <- c.right
 b.left <- a; b.right <- c
 a.left <- T1; a.right <- T2
 c.left <- T3; c.right <- T4
T1.parent <- a; T2.parent <-a
T3.parent <- b; T3.parent <- c

if (z.isRoot) then
      root <- b
      b.parent <- NULL
else if (z.isLeftChild)
           z.parent.left<-b
        else z.parent.right <- b
 b.parent <- z.parent
 a.parent <- b; c.parent <- b
```

Restructuring (as Single Rotations)



Restructuring (as Double Rotations)



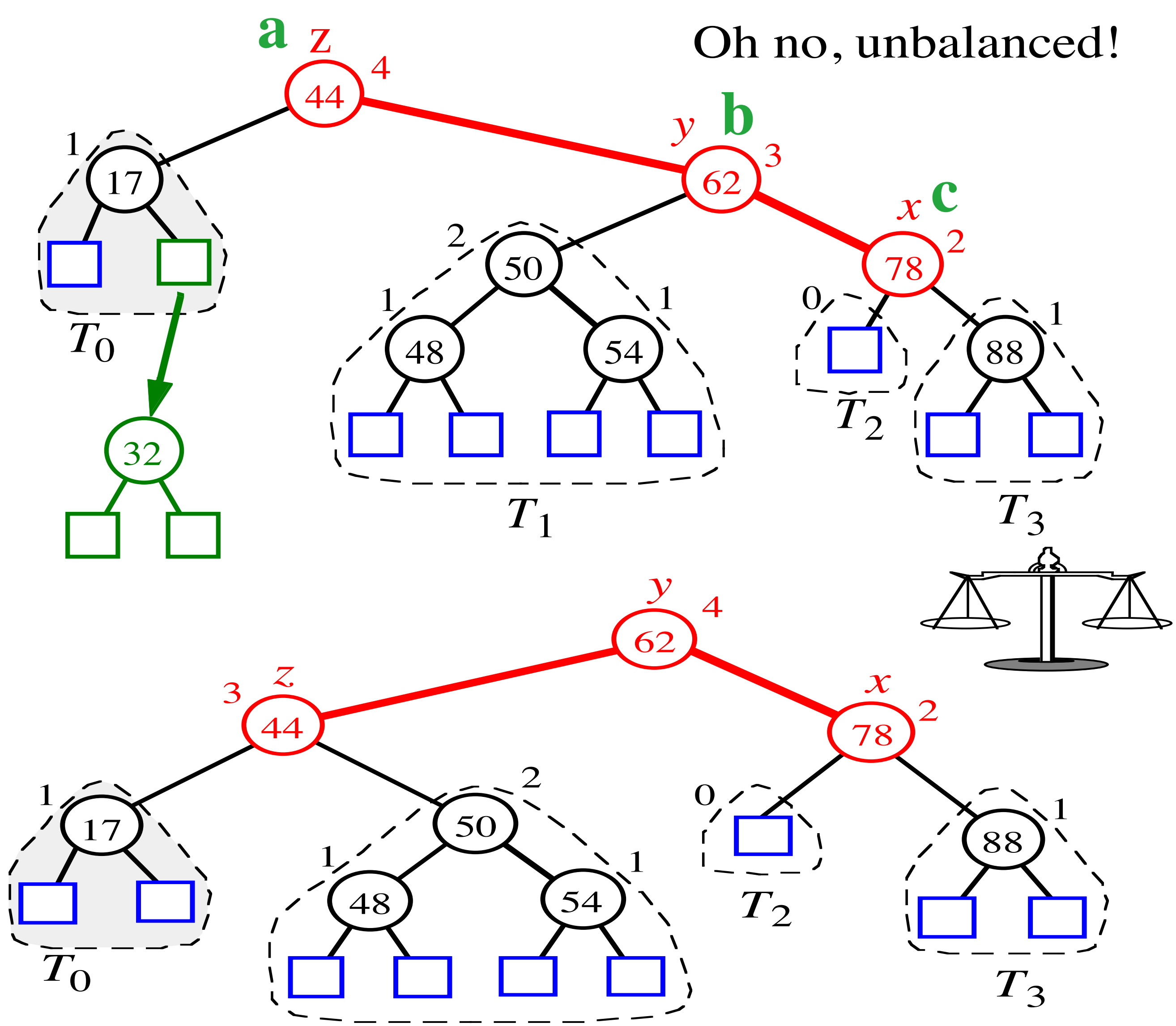
Removal

- We can easily see that performing a removeAboveExternal(w) can cause T to become unbalanced.
- 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 can perform operation restructure(x) to restore balance at the subtree rooted 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

Removal (contd.)

the choice of x is not unique !!!

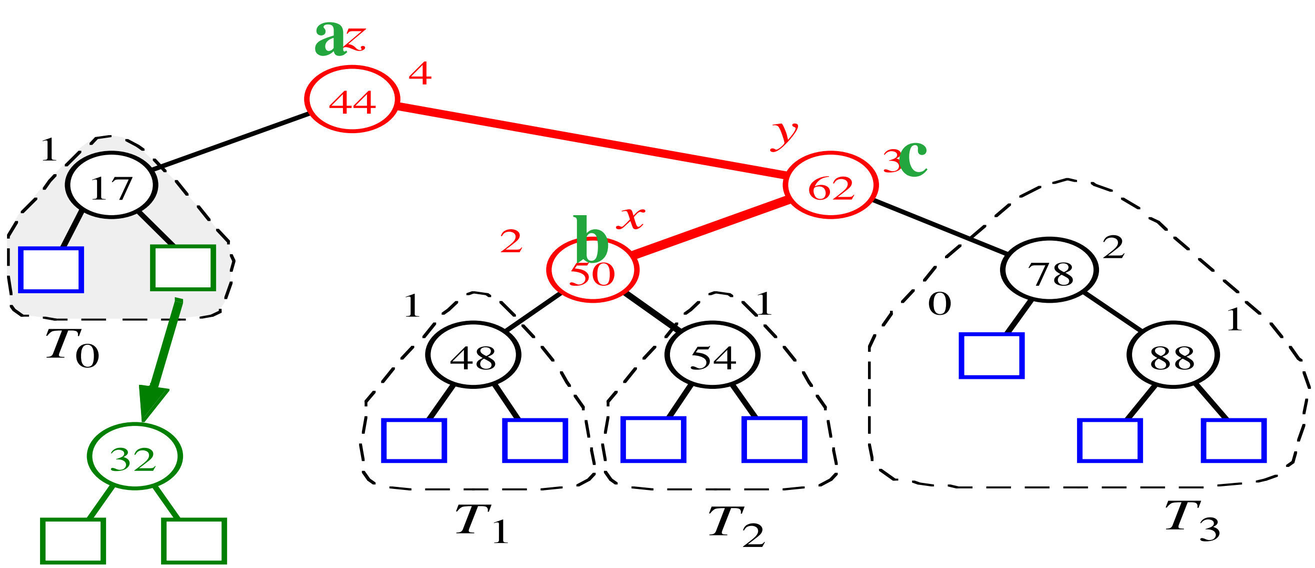
Oh no, unbalanced!



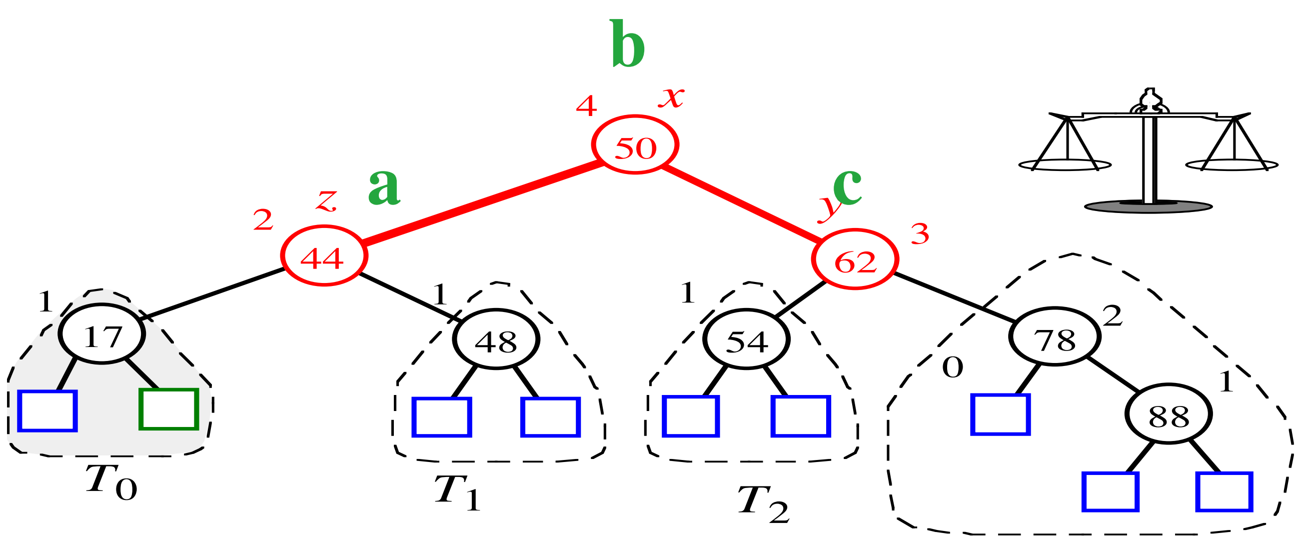
Whew, balanced!

Removal (contd.)

- we could choose a different x:



Oh no, unbalanced!



Whew, balanced!

Again.. x, y, z

z : **the first unbalanced node** encountered while travelling up the tree from w.

y : **the child of z with the larger height**

x : **the child of y with the larger height.** If both children of y have the same height, let x be the child of y on the same side as y.

COMPLEXITY

Searching:	findElement(k):
Inserting:	insertItem(k, o):
Removing:	removeElement(k):

O(log n)

Some implementation details are very important:

The trinode restructure is accomplished using the rotation operation:

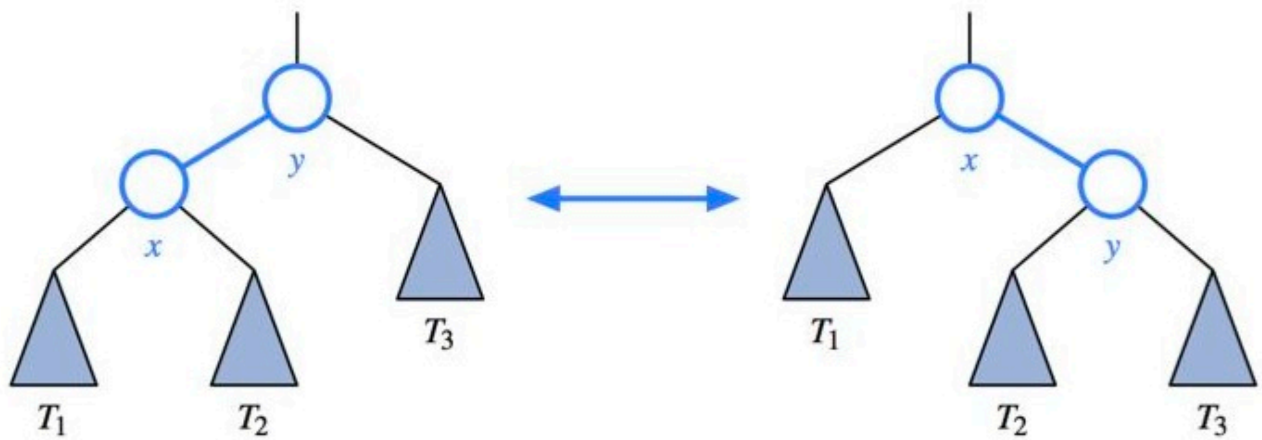


Figure 11.8: A rotation operation in a binary search tree. A rotation can be performed to transform the left formation into the right, or the right formation into the left. Note that all keys in subtree T_1 have keys less than that of position x, all keys in subtree T_2 have keys that are between those of positions x and y, and all keys in subtree T_3 have keys that are greater than that of position y.

Trinode restructuring using rotation operation:

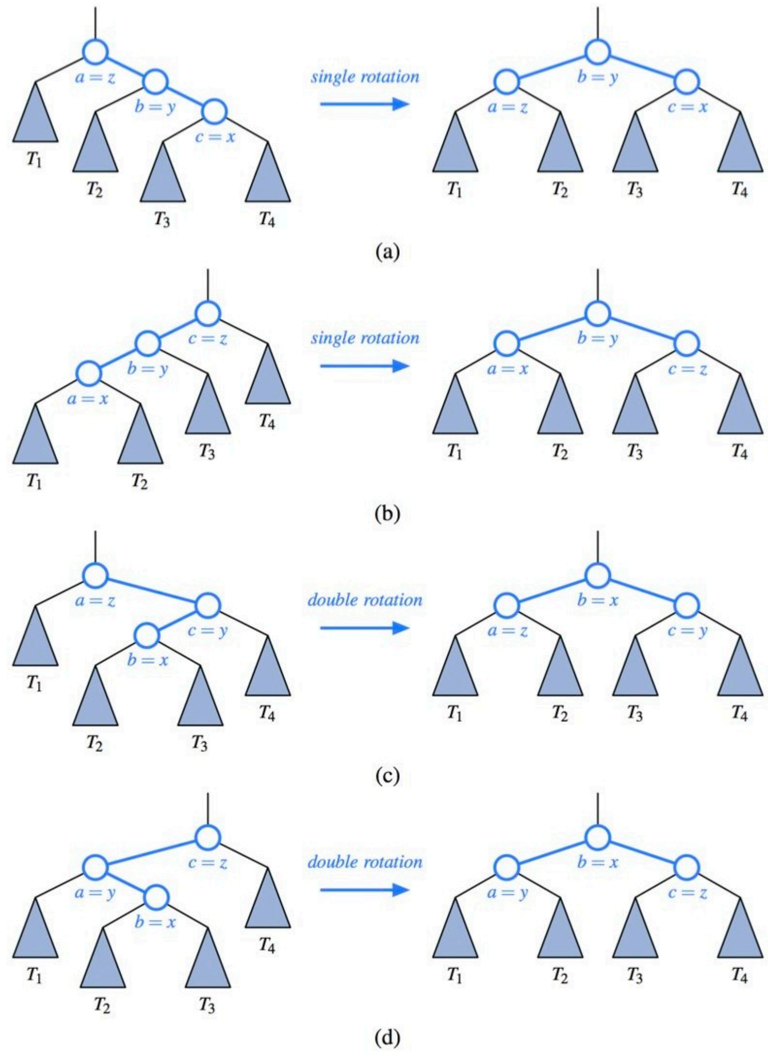


Figure 11.9: Schematic illustration of a trinode restructuring operation: (a and b) require a single rotation; (c and d) require a double rotation.

Rotate:

Restructure:

```
/** Relinks a parent node with its oriented child node. */
private void relink(Node<Entry<K,V>> parent, Node<Entry<K,V>> child,
                    boolean makeLeftChild) {
  child.setParent(parent);
  if (makeLeftChild)
    parent.setLeft(child);
  else
    parent.setRight(child);
}
/** Rotates Position p above its parent. */
public void rotate(Position<Entry<K,V>> p) {
  Node<Entry<K,V>> x = validate(p);
  Node<Entry<K,V>> y = x.getParent();          // we assume this exists
  Node<Entry<K,V>> z = y.getParent();          // grandparent (possibly null)
  if (z == null) {
    root = x;                                  // x becomes root of the tree
    x.setParent(null);
  } else
    relink(z, x, y == z.getLeft());            // x becomes direct child of z
  // now rotate x and y, including transfer of middle subtree
  if (x == y.getLeft()) {
    relink(y, x.getRight(), true);             // x's right child becomes y's left
    relink(x, y, false);                       // y becomes x's right child
  } else {
    relink(y, x.getLeft(), false);             // x's left child becomes y's right
    relink(x, y, true);                        // y becomes left child of x
  }
}
/** Performs a trinode restructuring of Position x with its parent/grandparent. */
public Position<Entry<K,V>> restructure(Position<Entry<K,V>> x) {
  Position<Entry<K,V>> y = parent(x);
  Position<Entry<K,V>> z = parent(y);
  if ((x == right(y)) == (y == right(z))) {    // matching alignments
    rotate(y);                                 // single rotation (of y)
    return y;                                  // y is new subtree root
  } else {                                     // opposite alignments
    rotate(x);                                 // double rotation (of x)
    rotate(x);
    return x;                                  // x is new subtree root
  }
}
}
```

Code Fragment 11.10: The BalanceableBinaryTree class, which is nested within the TreeMap class definition (continued from Code Fragment 11.9).

Rebalancing operation for AVL insertions and deletions:

```
/**
 * Utility used to rebalance after an insert or removal operation. This traverses the
 * path upward from p, performing a trinode restructuring when imbalance is found,
 * continuing until balance is restored.
 */
protected void rebalance(Position<Entry<K,V>> p) {
  int oldHeight, newHeight;
  do {
    oldHeight = height(p);                    // not yet recalculated if internal
    if (!isBalanced(p)) {                     // imbalance detected
      // perform trinode restructuring, setting p to resulting root,
      // and recompute new local heights after the restructuring
      p = restructure(tallerChild(tallerChild(p)));
      recomputeHeight(left(p));
      recomputeHeight(right(p));
    }
    recomputeHeight(p);
    newHeight = height(p);
    p = parent(p);
  } while (oldHeight != newHeight && p != null);
}
/** Overrides the TreeMap rebalancing hook that is called after an insertion. */
protected void rebalanceInsert(Position<Entry<K,V>> p) {
  rebalance(p);
}
/** Overrides the TreeMap rebalancing hook that is called after a deletion. */
protected void rebalanceDelete(Position<Entry<K,V>> p) {
  if (!isRoot(p))
    rebalance(parent(p));
}
}
```

At this point in the class I discuss how AVL trees are implemented in the 6^{th} edition of the textbook by Goodrich, Tamassia and Goldwasser.

Please, refer to pages:

466-470 class TreeMap<K,V>
Note methods: put(K key, V value), remove(K key)

475-478 class BalancedBinaryTree<K,V>
Note: hooks for rebalancing present in TreeMap,
Methods: rotate, restructure

486-487 class AVLTreeMap<K,V>
Note methods: rebalanceInsert, rebalanceDelete, rebalance