# AVL Trees

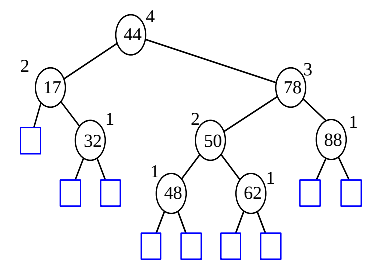
Height of an AVL Tree
Insertion and restructuring
Removal and restructuring
Costs

AVL tree, named after the initials of its inventors: Adel'son-Vel'skii and Landis



## 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.*



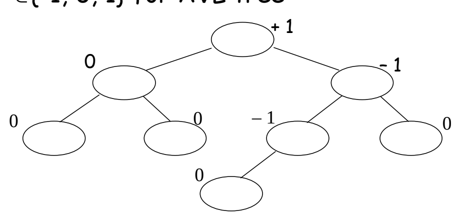

An example of an AVL tree where the heights are shown next to the nodes:



## Balancing Factor

=
height(right s.a.) – height(left s.a.)

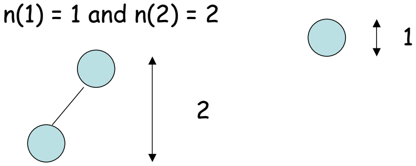
$\in$ {-1, 0, 1} for AVL tree





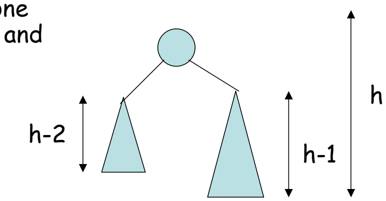

## Height of an AVL Tree

- **Proposition**: The ***height*** of an AVL tree T storing n keys is O(log n).
- **Justification**: The easiest way to approach this problem is to find **n(h): the *minimum number of internal nodes* of an AVL tree of height h.**
- We see that n(1) = 1 and n(2) = 2



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

For $n \geq 3$, 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.

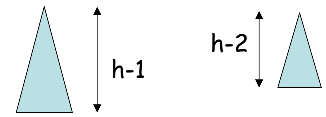



i.e. $n(h) = 1 + n(h-1) + n(h-2)$



## Height of an AVL Tree

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




But: $n(h-1) > n(h-2)$,

that is: $n(h) > 1 + n(h-2) + n(h-2)$
which is $> 2n(h-2)$

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



## Height of an AVL Tree

So, now we know: $n(h) > 2n(h-2)$

but then also: $n(h-2) > 2n(h-4)$

$\Big\}$ $n(h) > 2n(h-4) = 2\ 2n(h-4)$

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

but then also: $n(h-4) > 2n(h-6)$

$n(h) > 8n(h-4)$

We can continue:

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

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

...

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



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

with

$n(1) = 1$

$n(2) = 2$

$h-2i = 1$

for $i = h/2 - 1$

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

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

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

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

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

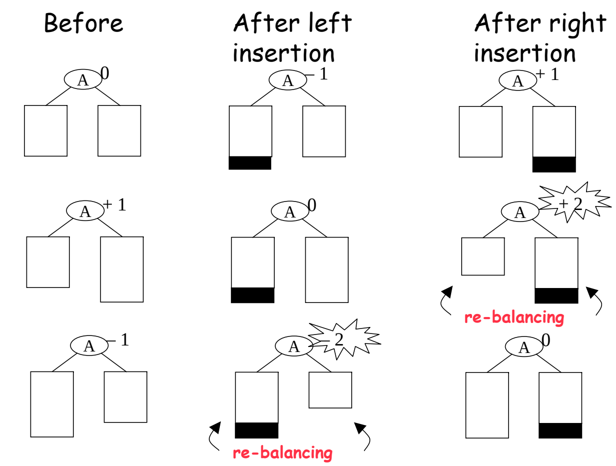
which means that h is O(log n)

## 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...

9

## Insertion




10

# Rebalancing after insertion

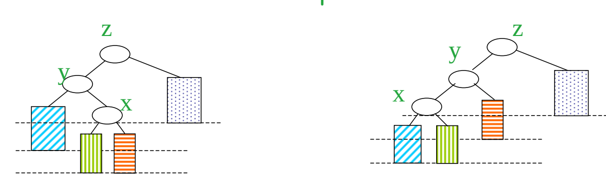
We are going to identify 3 nodes which form a grandparent, parent, child triplet and the 4 subtrees attached to them. We will rearrange these elements to create a new balanced tree.

11

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 .....

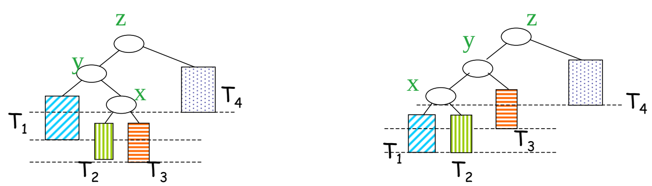



12

## 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 .....

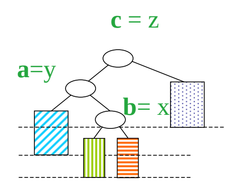



13

## Rebalancing

Step 3: Rename x, y, z to a, b, c according to their inorder traversal i.e. if y< x < z then label y 'a', x 'b' and z 'c'.

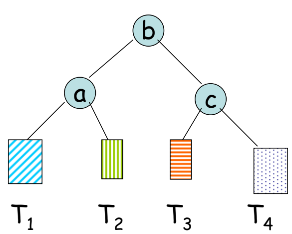
Example




14

## Rebalancing

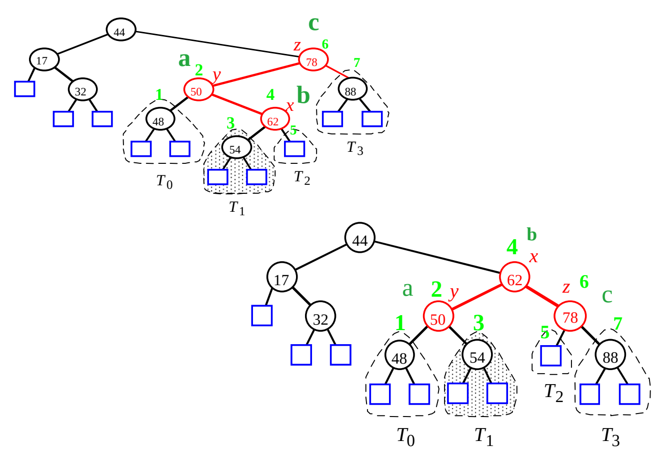
Step 4: Replace the tree rooted at z with the following tree:




Rebalance done!

15

## Example




16

# Does this really work?

We need to see that the new tree is :

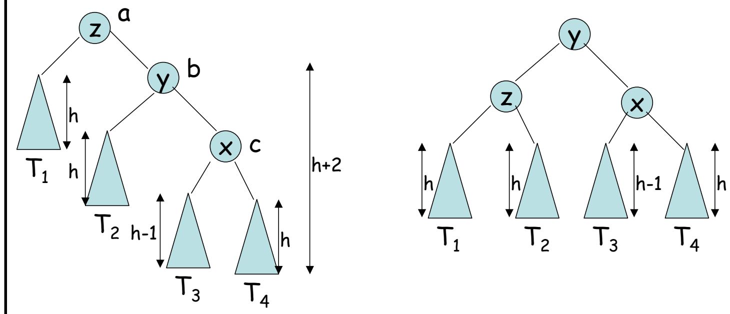
a) A Binary search tree - the inorder traversal of our new tree should be the same as that of the old tree

b) Balanced: have we fixed the problem?

We consider 2 types of examples



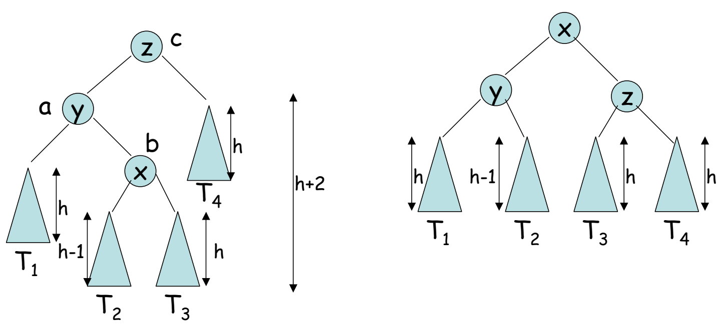
# Example 1




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 as 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
```



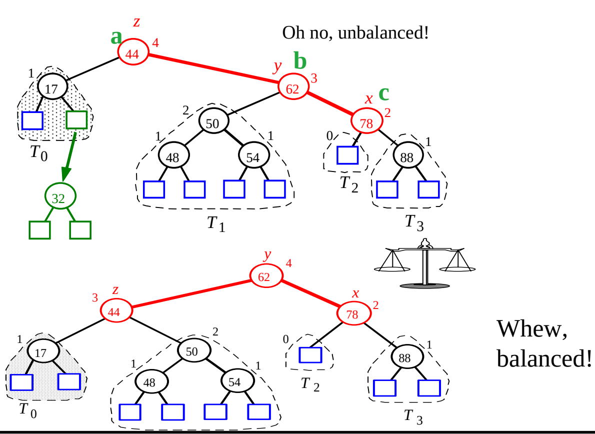
## Removal

- 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.)

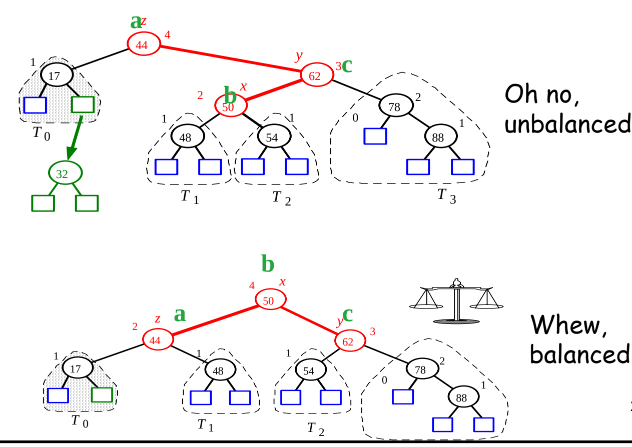
the choice of x is not unique !!!






## Removal (contd.)

- we could choose a different x:



## COMPLEXITY

Searching: findElement(k):

Inserting: insertItem(k, o):

Removing: removeElement(k):

O(log n)