Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University

Six ways to make people like you – Dale Carnegie

1. *Become genuinely interested in other people*
2. *Smile*
3. *A person's name is to that person the sweetest and the most important sound in any language*

Recap

- Tree ADT
- BST
- Degenerate Tree
- Randomly Built BST

Objectives

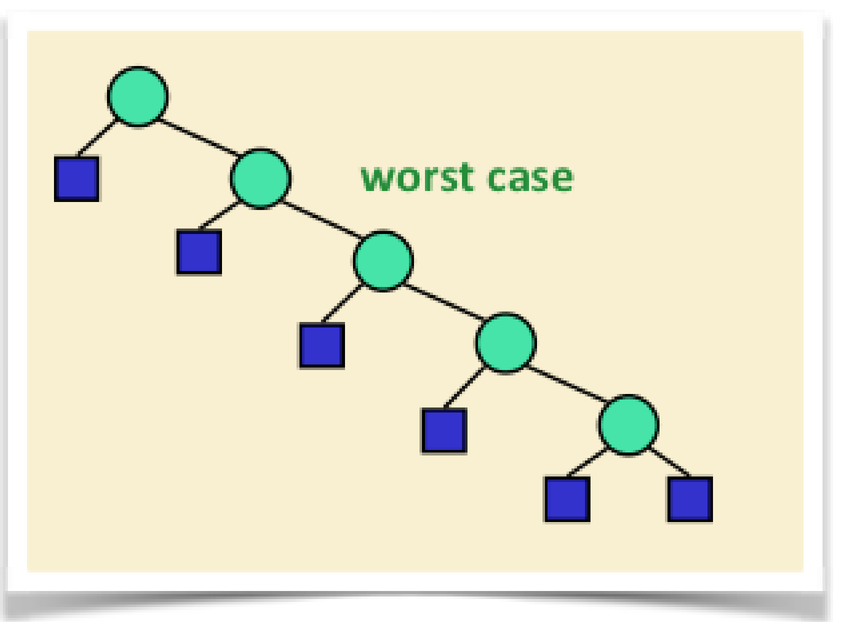
- Balanced Binary Search Trees
- AVL Trees
- Insertions and Deletions in AVL Trees
  - Trinode Restructuring (Rotations)
- Height of AVL Trees

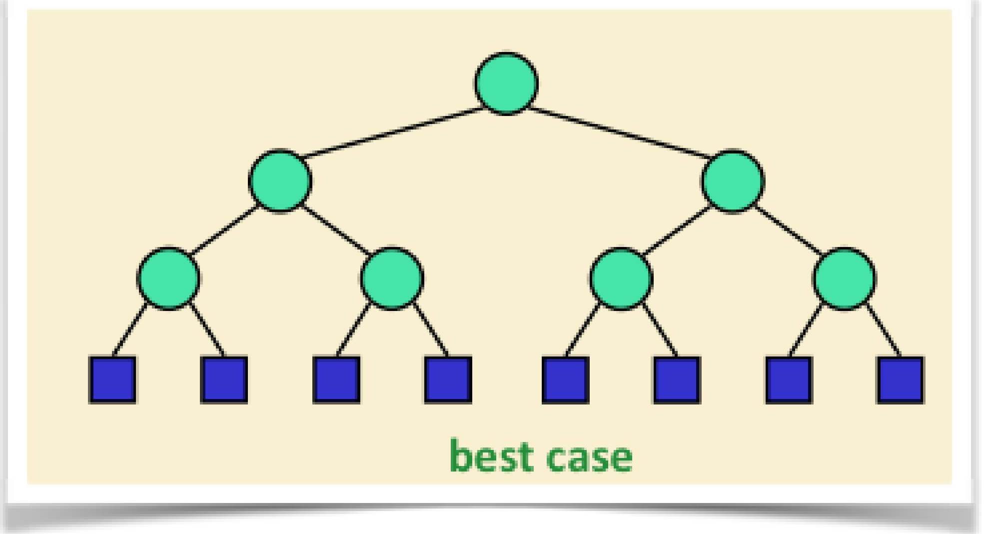
Binary Search Tree

- For a binary search tree with n nodes
  - Search and insertion time is $O(\log n)$
- However, this is only true is the tree is “balanced”
- That is, the “height” of the tree is balanced

Binary Search Tree

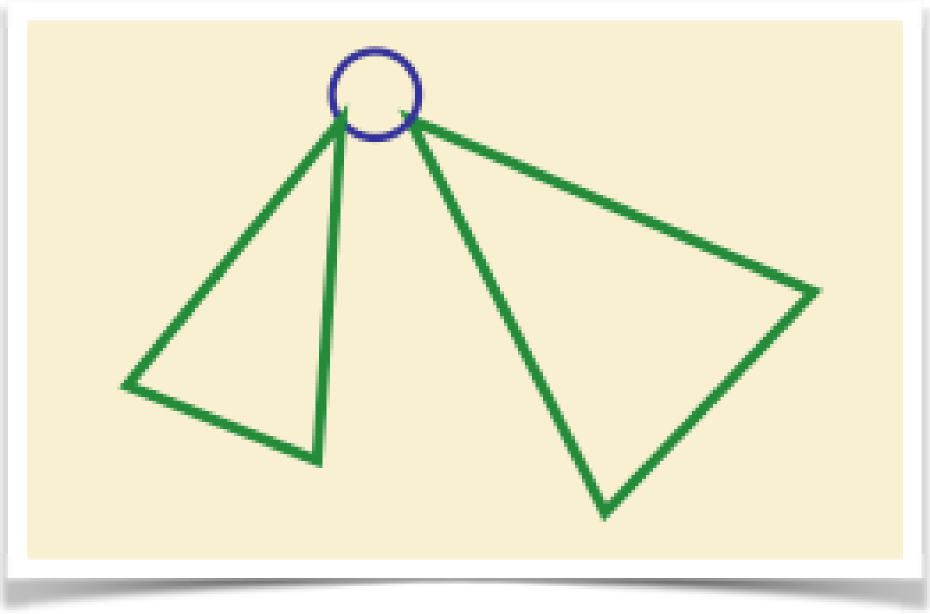
- In the worst case, insertion and searching time becomes $O(n)$
- Because the height is $O(n)$





Binary Search Tree

- In a dynamic tree, nodes are inserted and deleted over time
- So we must find a way to keep the height of a binary search tree always O(log *n*)
- To achieve this, the tree must always be balanced



"for any node, its left subtree should not be much higher then its right subtree, and vice-versa"

AVL Trees

AVL Trees

- Adelson-Velskii and Landis in 1962 introduced a binary tree structure that is balanced with respect to the heights of its subtree

- Insertions and deletions are made such that the tree always remain height-balanced

AVL Trees

- Definition
- An empty tree is height-balanced
- If T is non-empty binary tree with left and right sub-trees T_1 and T_2

T is balanced if and only if

  - T_1 and T_2 are balanced, and
  - $|height(T_1) - height(T_2)| \leq 1$

Recall: Binary Tree Terminology

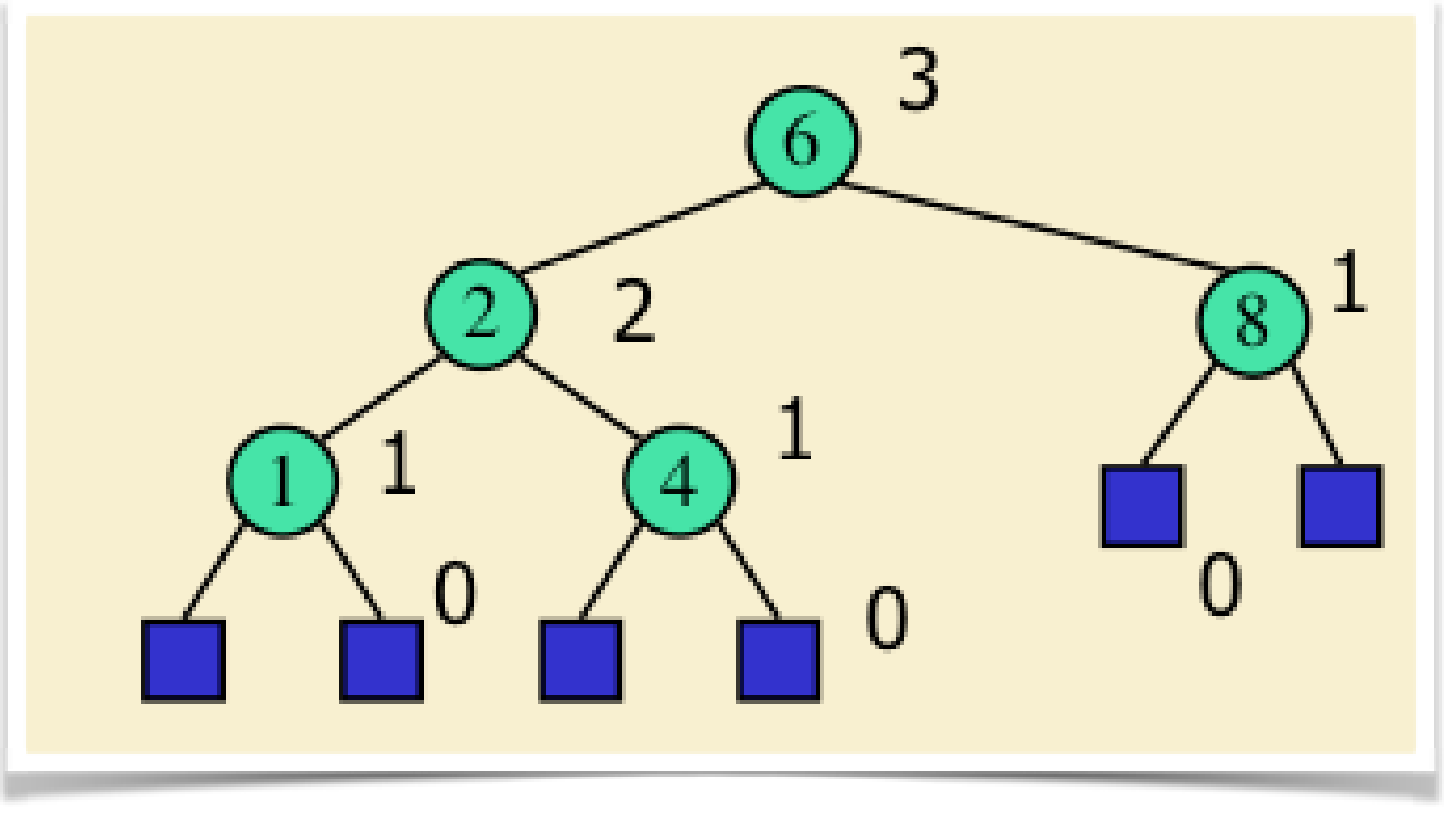
- Height of a tree T
- $$h(T) = \begin{cases} 0, & T\ is\ empty \\ 1 + \max(height(T_1), height(T_2)), & otherwise \end{cases}$$

where $\boldsymbol{T_1}$ and $\boldsymbol{T_2}$ are the subtrees of the root node

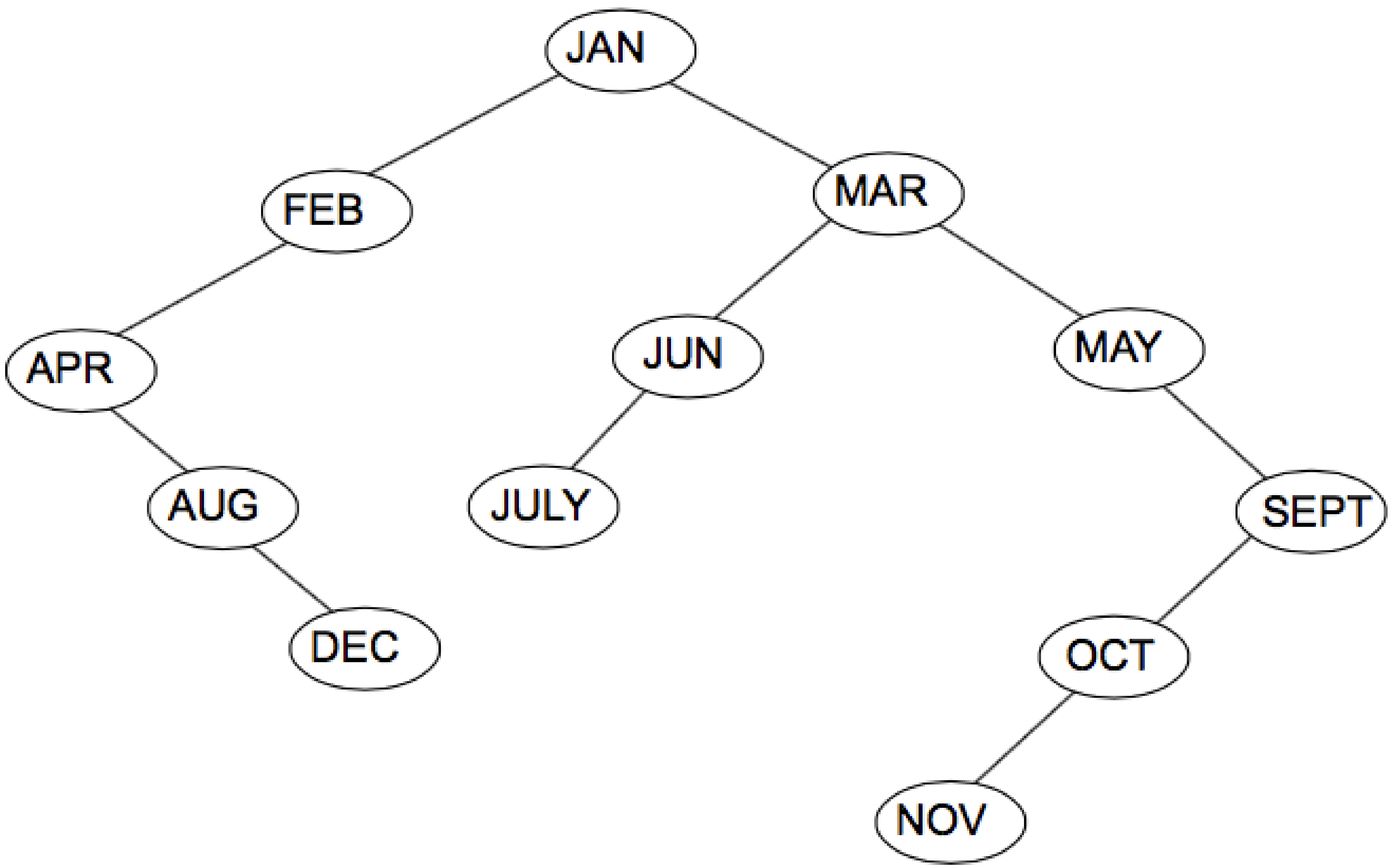
Recall: Binary Tree Terminology

- Height Numbering
  - Number all external (leaf) nodes 0
  - Number each internal node to be one more than the maximum of the heights of its children
  - Then number of the root node is the height of **T**

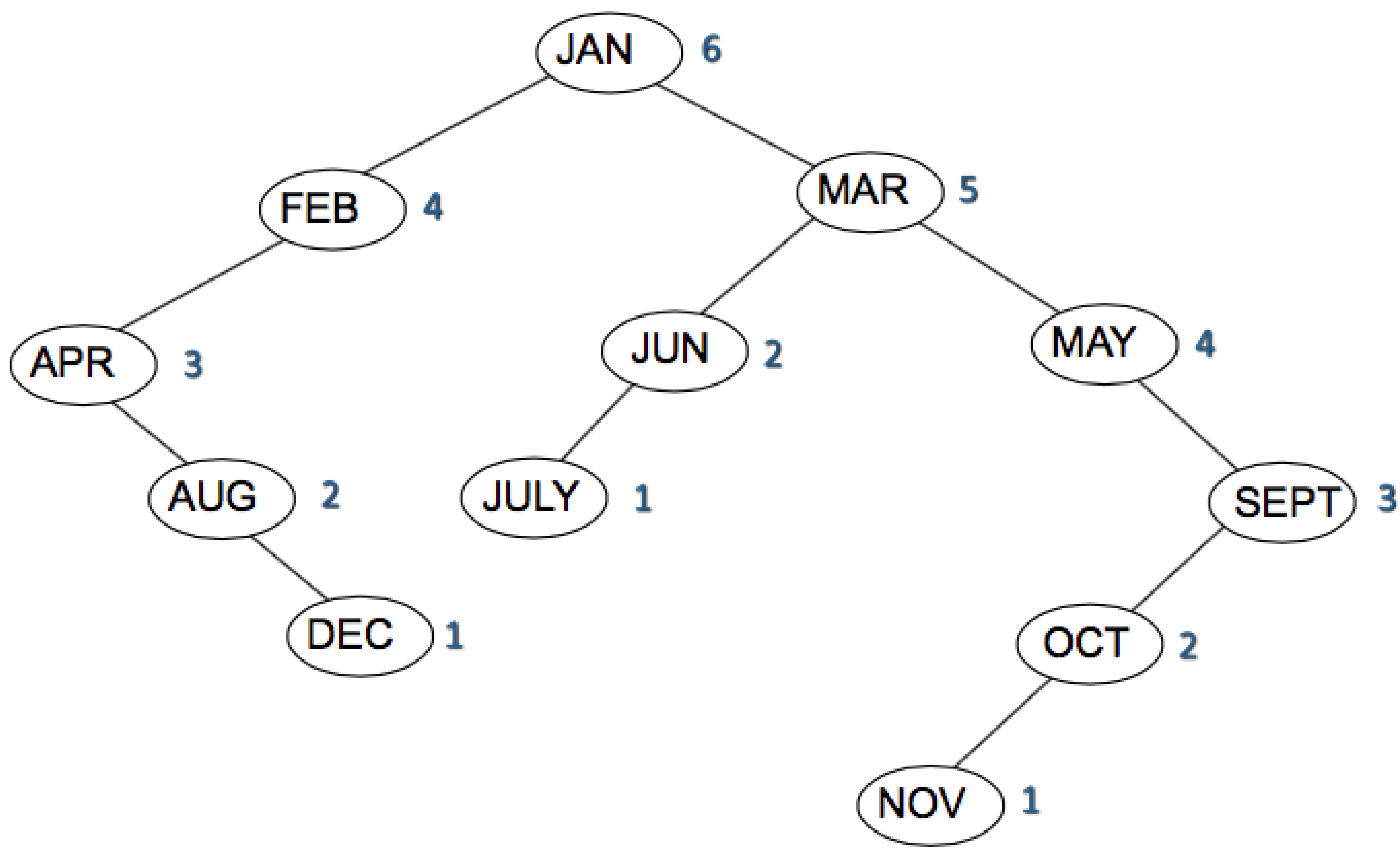
AVL Trees



AVL Trees

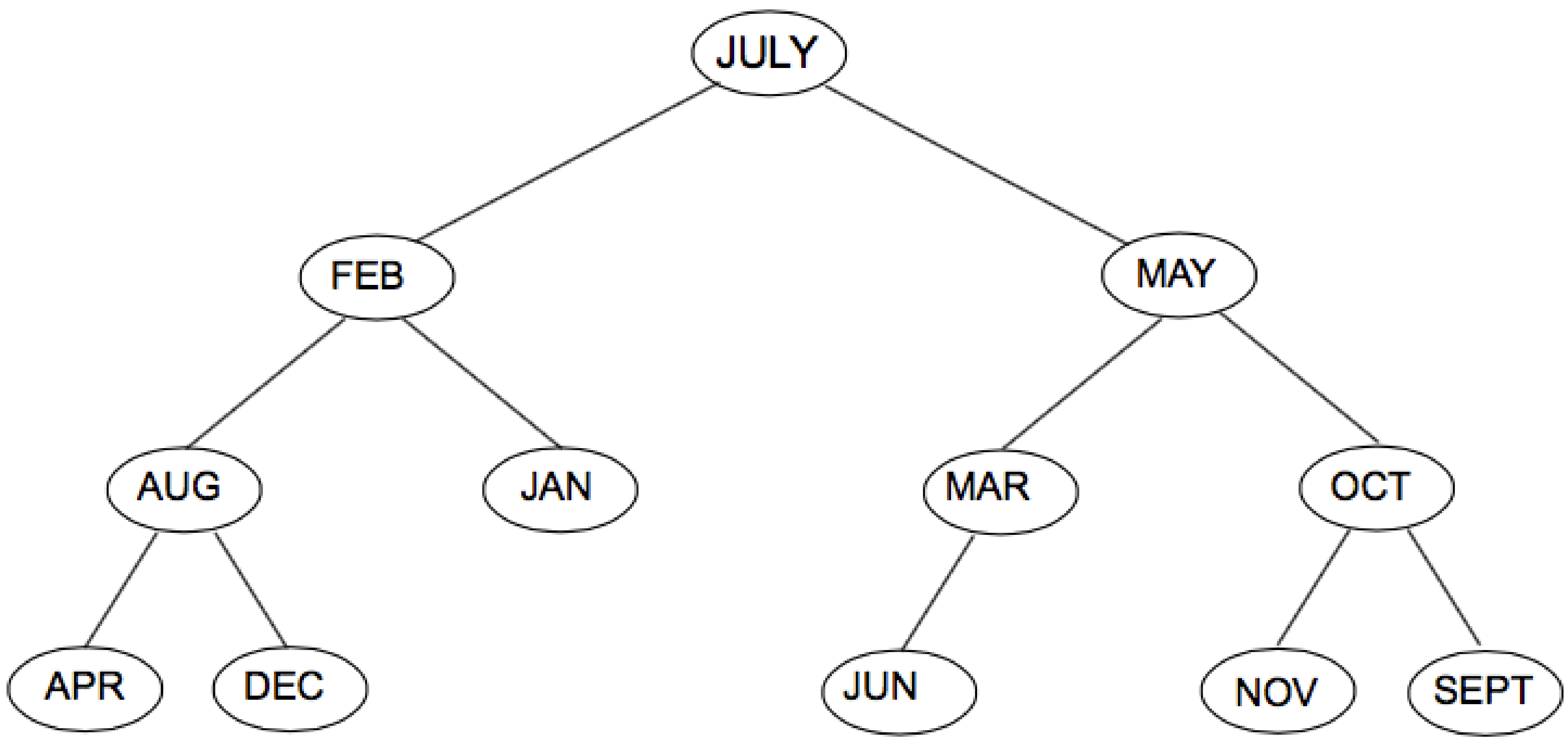


AVL Trees



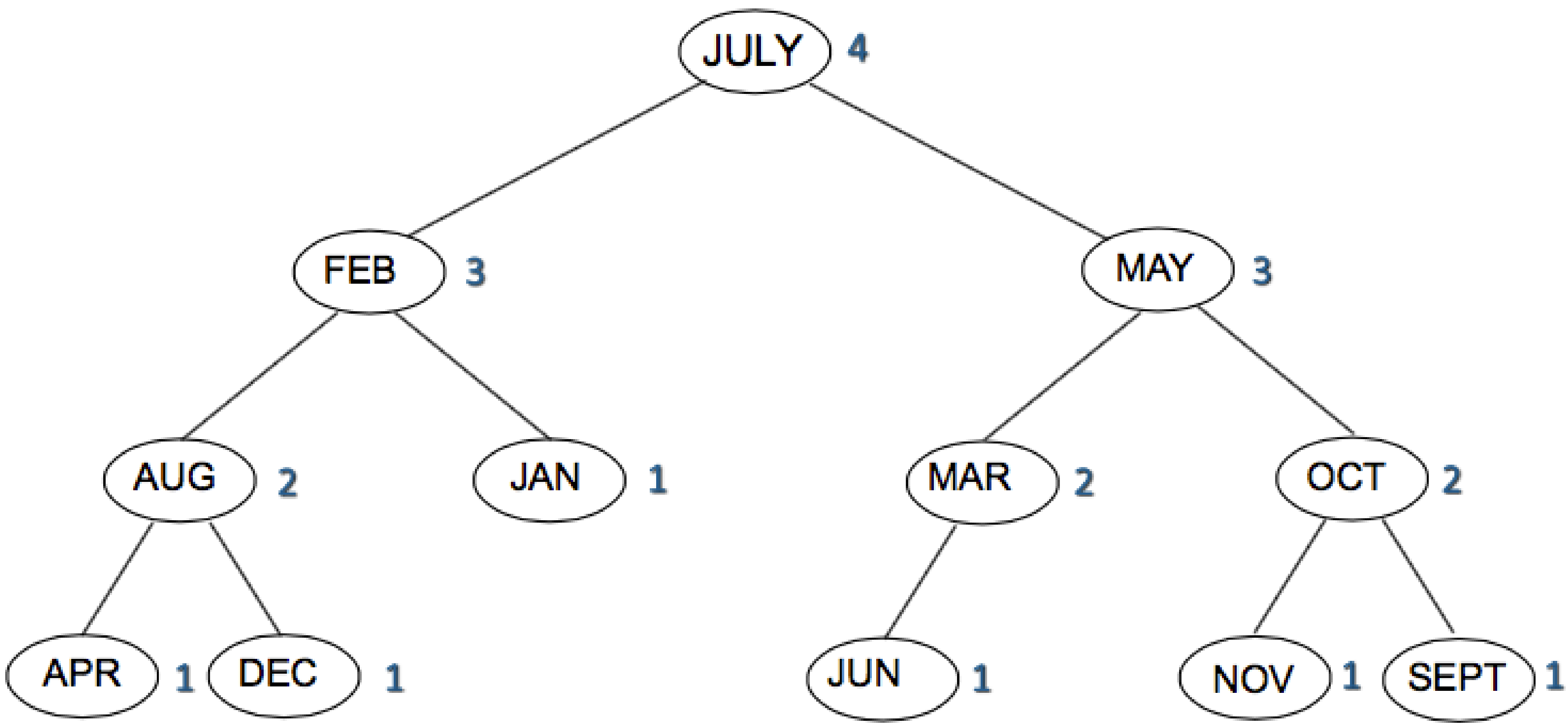
JAN 6
FEB 4
MAR 5
APR 3
JUN 2
MAY 4
AUG 2
JULY 1
SEPT 3
DEC 1
OCT 2
NOV 1

AVL Trees



A Balanced Tree for the Months of the Year

AVL Trees



A Balanced Tree for the Months of the Year

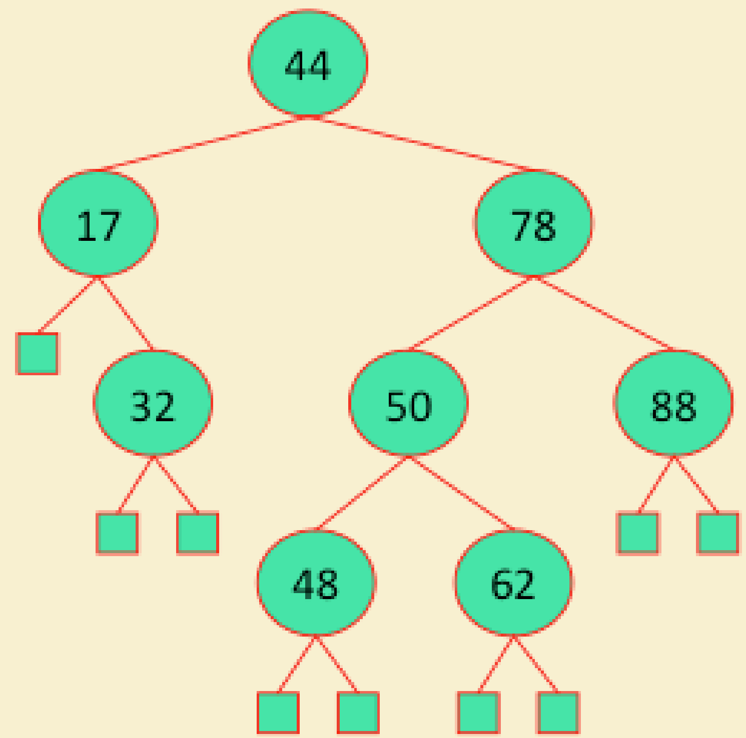
Operations in an AVL Tree

- The height of an AVL tree is $O(\log n)$
- Thus the **search** operation takes $O(\log n)$
  - Performed just like in a binary search tree since AVL tree is a binary search tree
- What we need to show is how to **insert** and **remove** in AVL trees while maintaining
  - the height balanced property
  - the binary search tree order

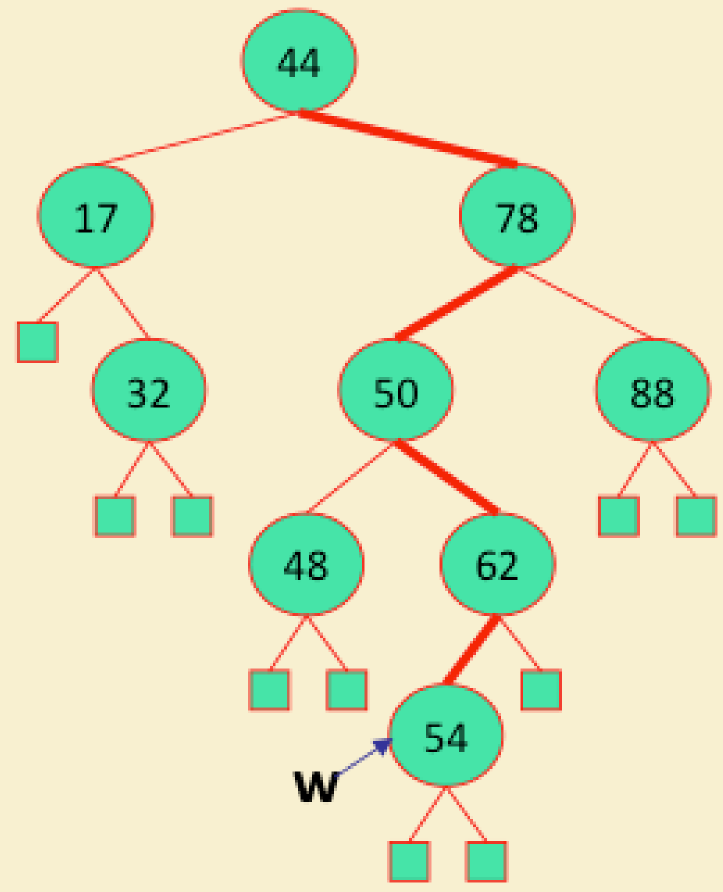
Insertion in an AVL Tree

- Starts as in a binary search tree

- Always done by expanding an external node



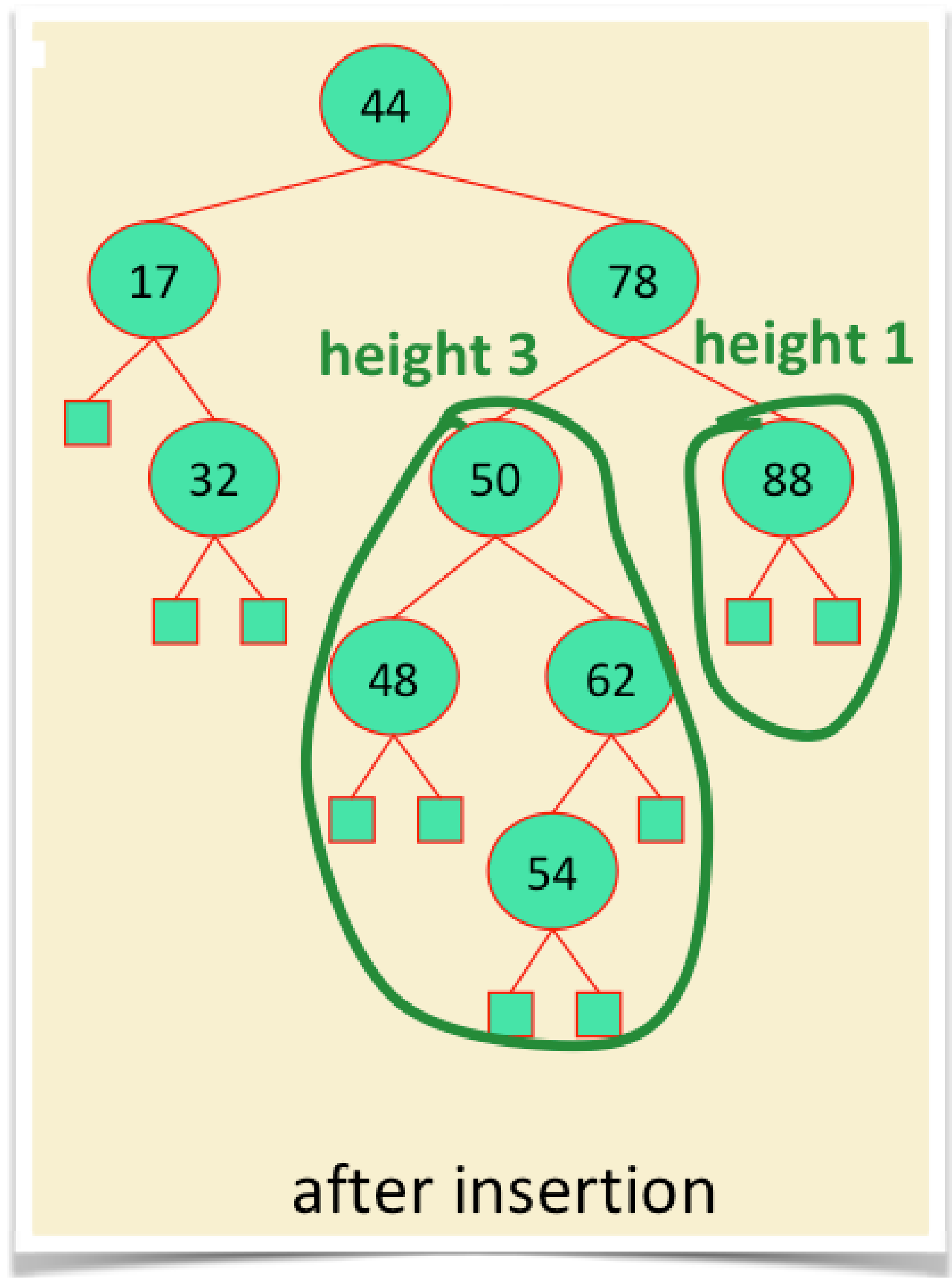
before inserting 54



after insertion

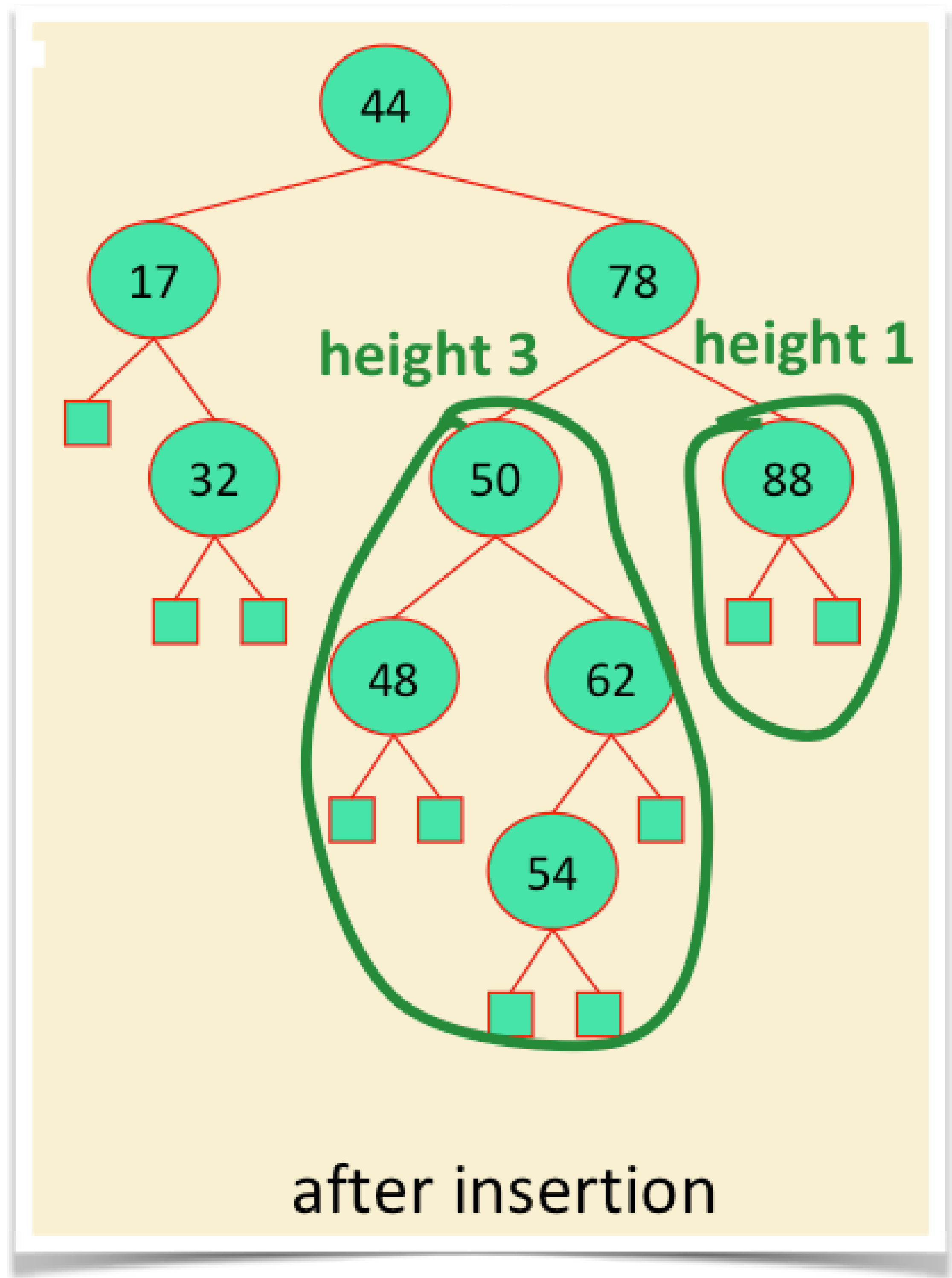
Insertion in an AVL Tree

- After inserting a new node into an AVL tree, the height-balanced property of the AVL tree is very likely lost

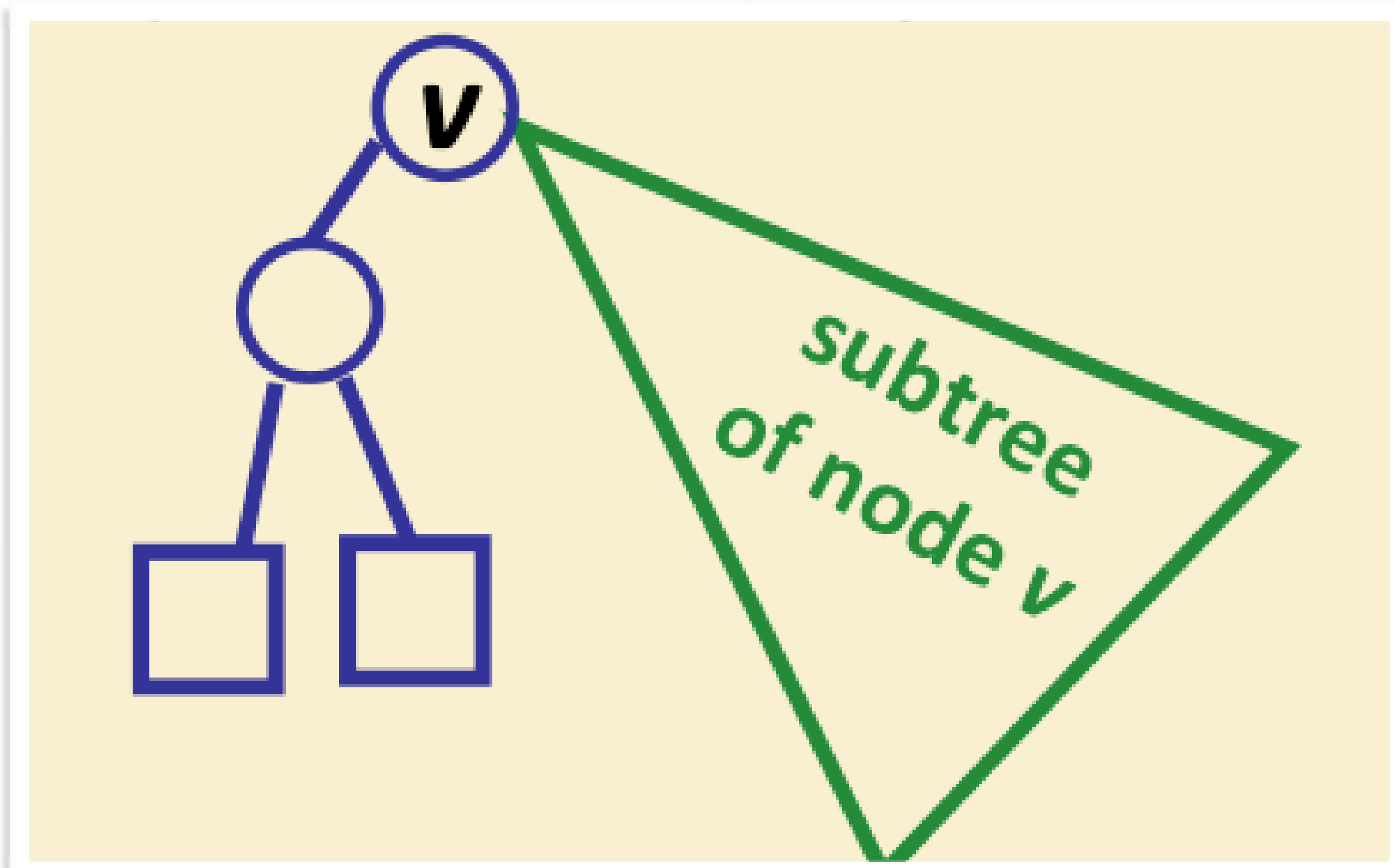


Insertion in an AVL Tree

- Thus, to make it an AVL tree again, we need to **restore the balance** by **restructuring the tree**

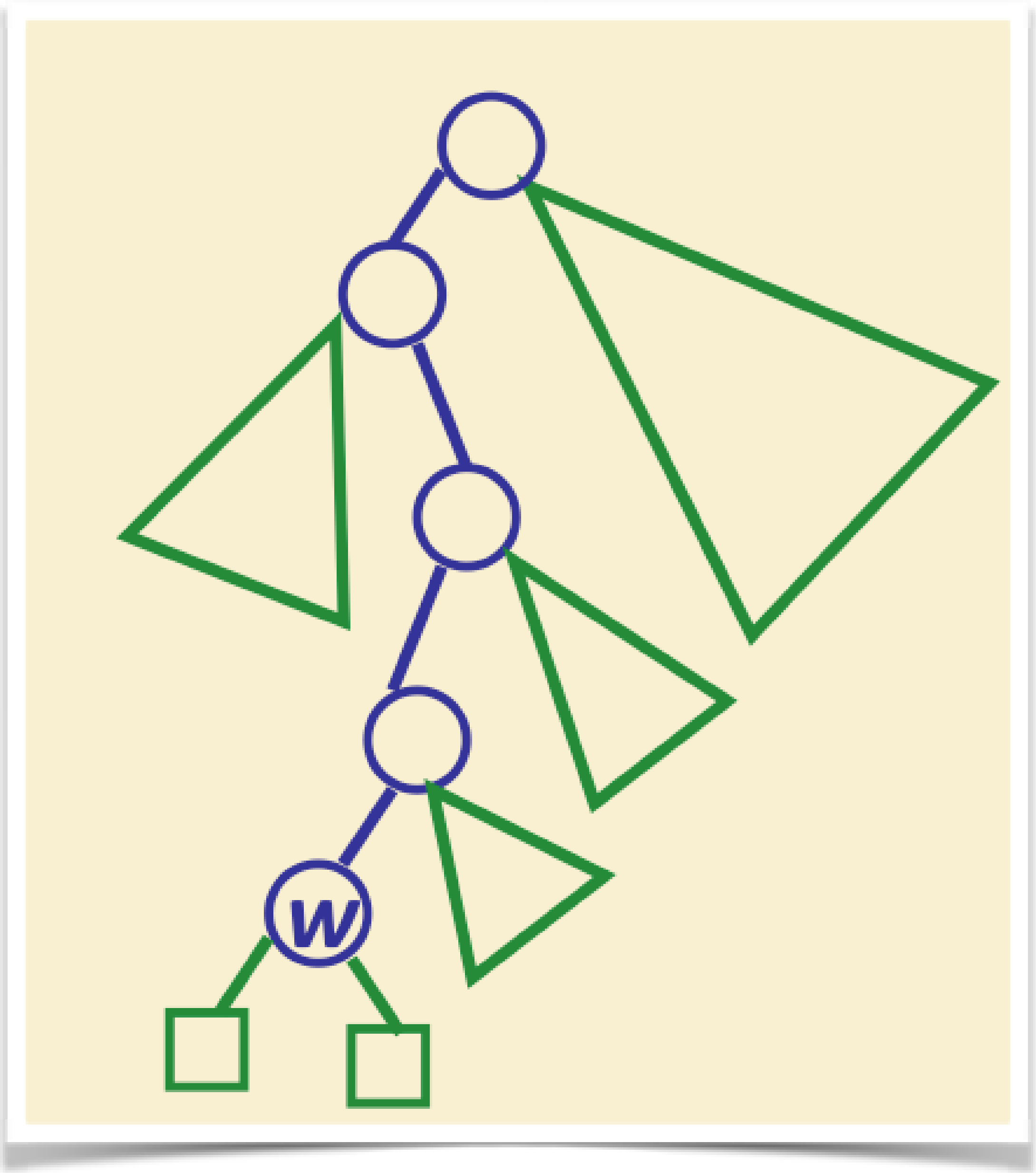


Pictorial Notation



Restructuring

- Let w be the new node, just inserted into an AVL tree



Restructuring

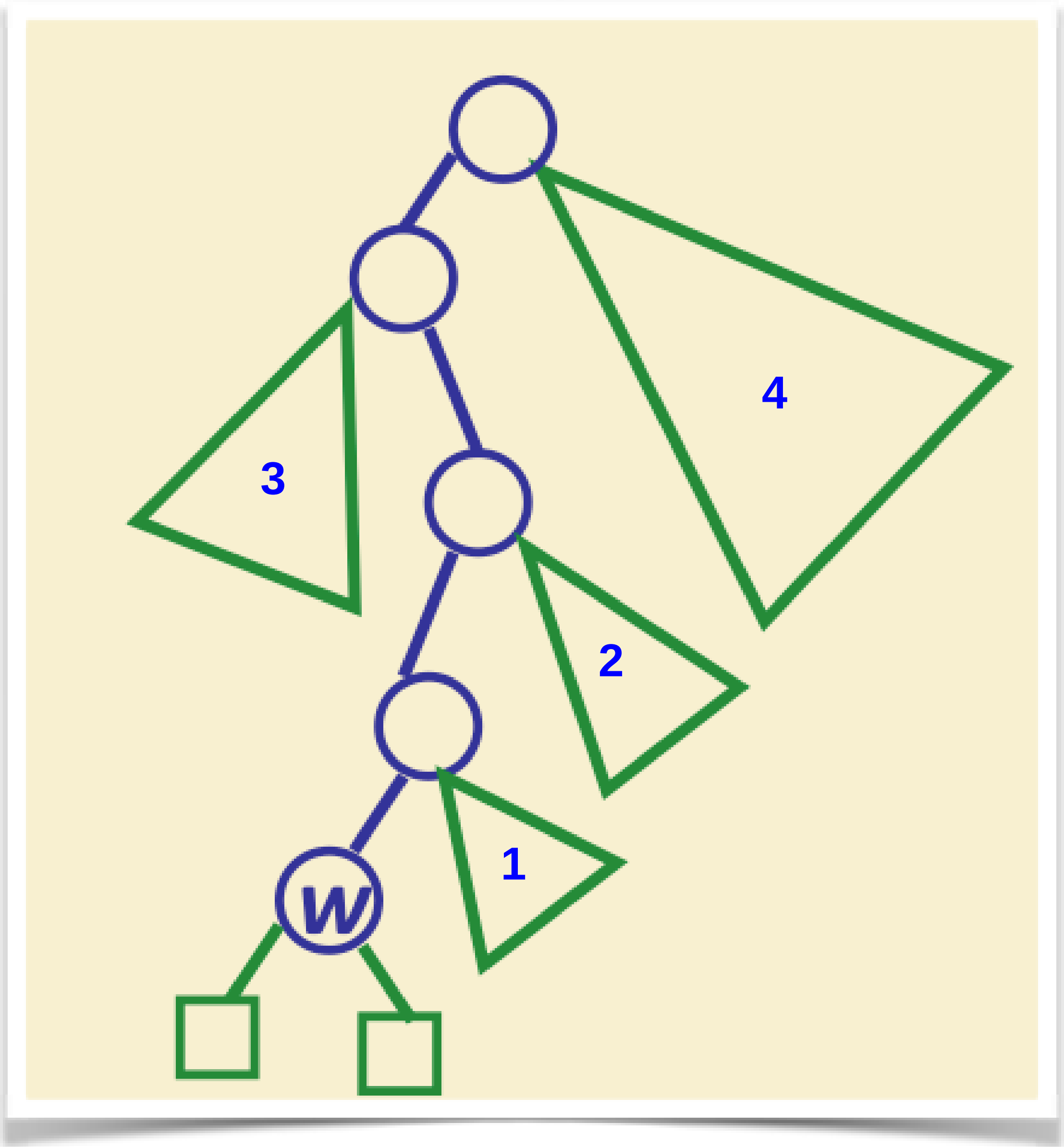
- The next step is to search for the unbalanced node(s)

- Check each node in the tree to see if it is balanced.

- Do you think this approach is efficient?

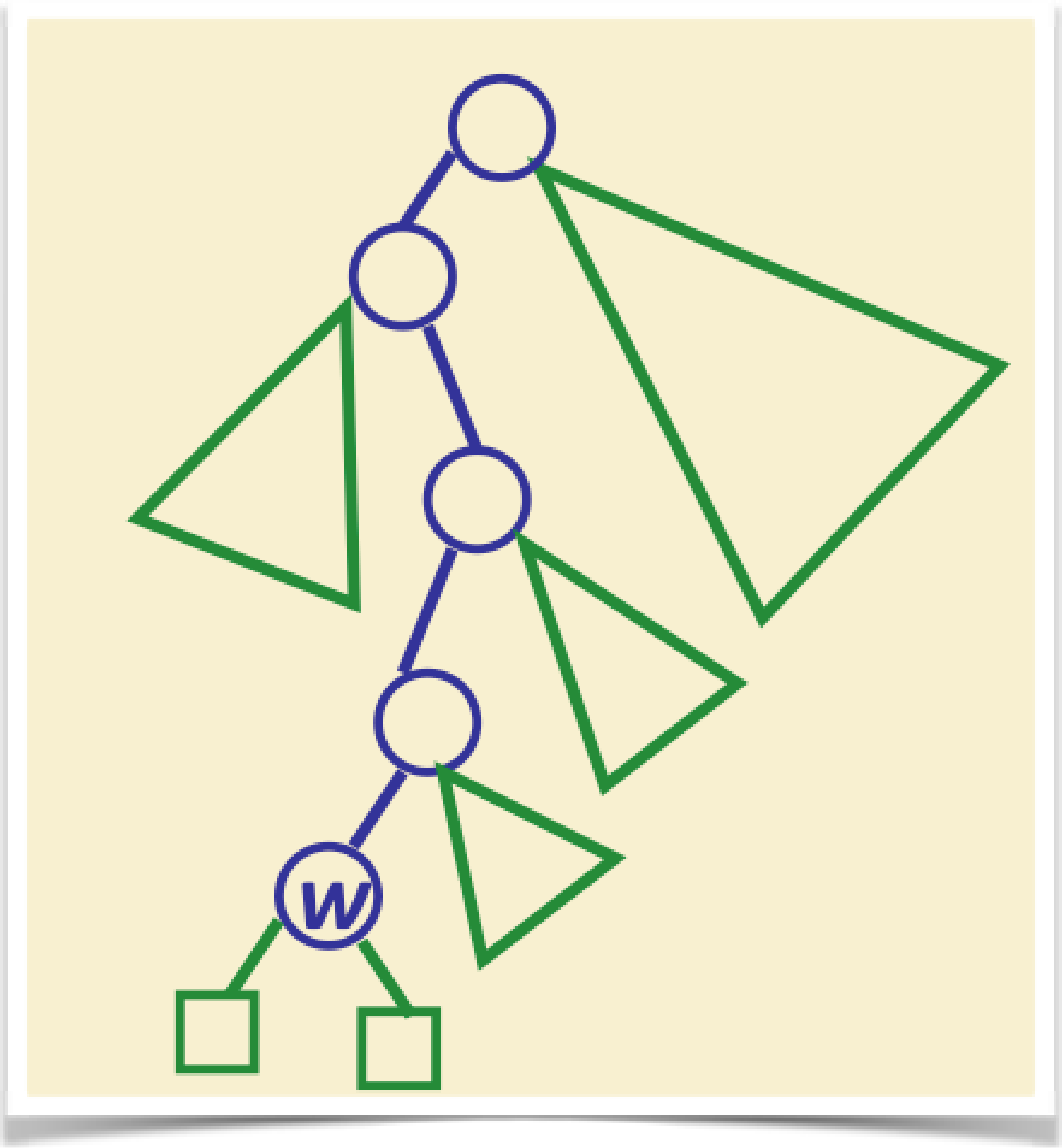
Restructuring

- Tip: after insertion of **w**, heights could change (increase) only for the ancestors of **w**
- Thus only ancestors of w could be unbalanced
- Search up the tree from w checking and correcting any unbalanced node



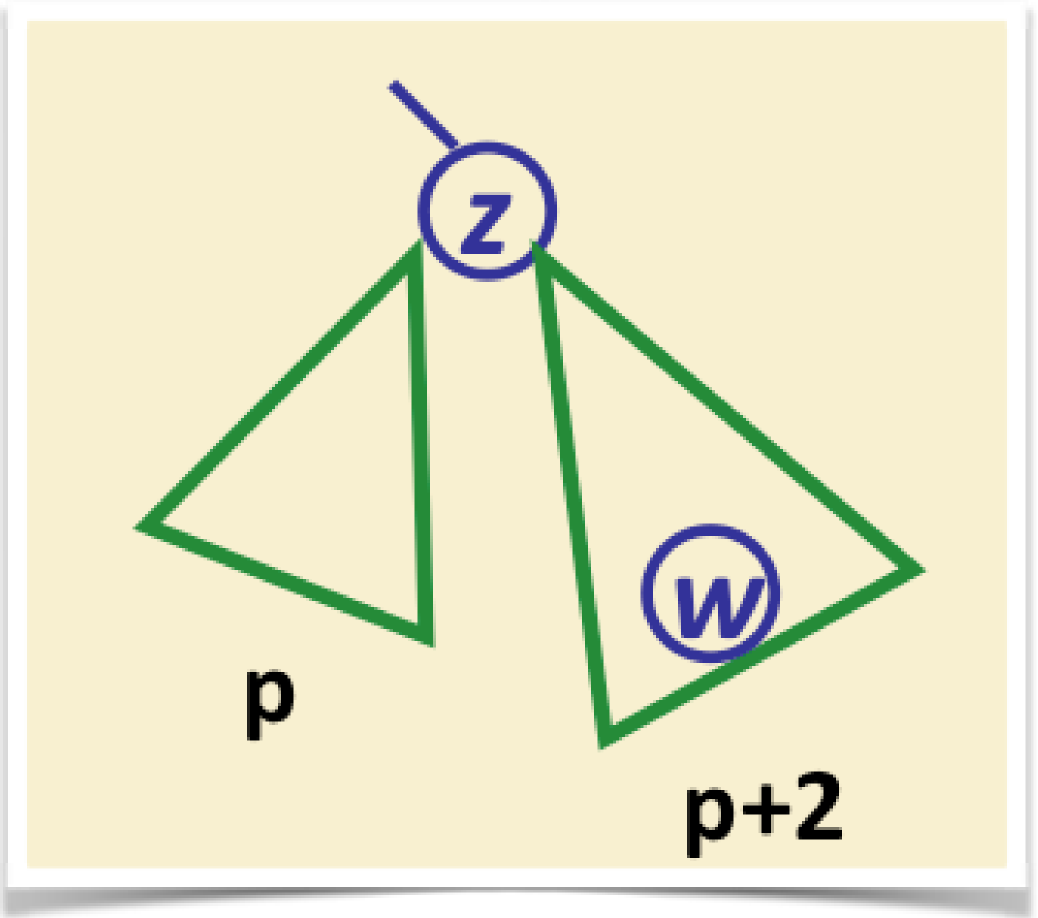
Restructuring

- Follow the path from **w** to the root



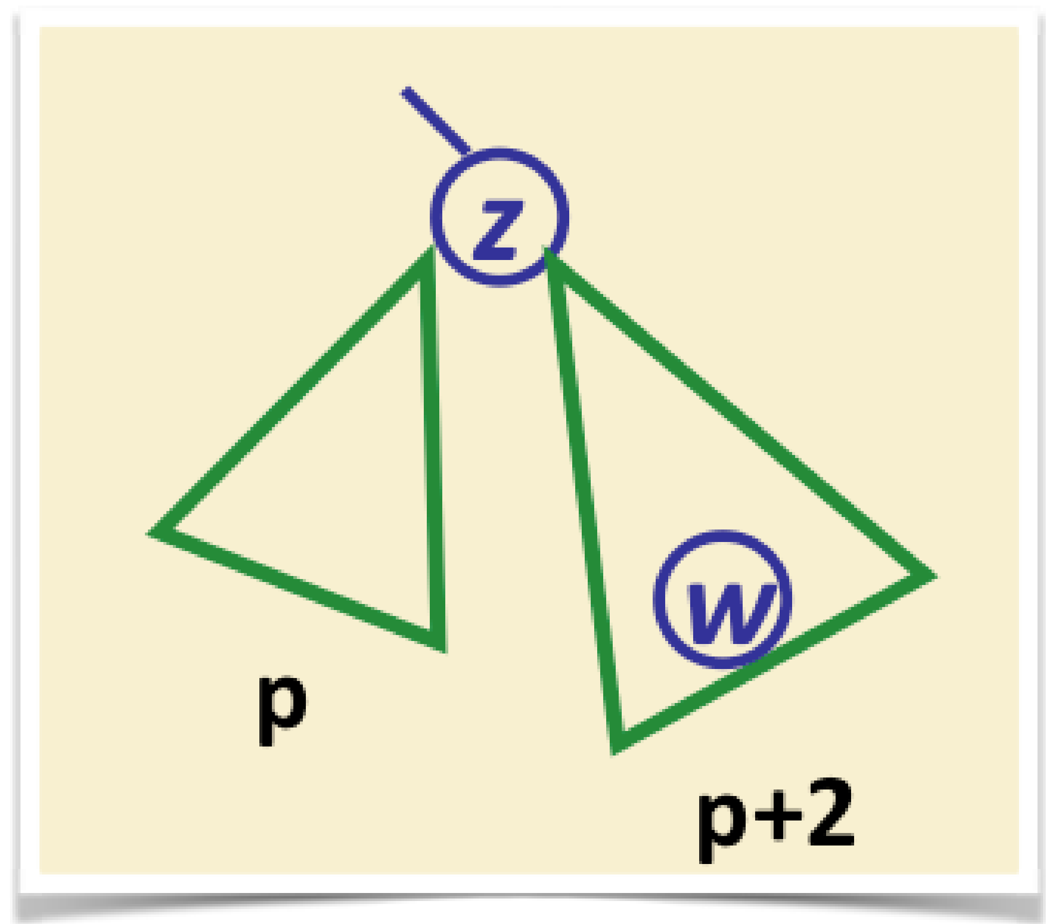
Restructuring

- Suppose the first unbalanced node is at position **z**
- This means that height difference between the left and the right subtree of **z** is more than 1



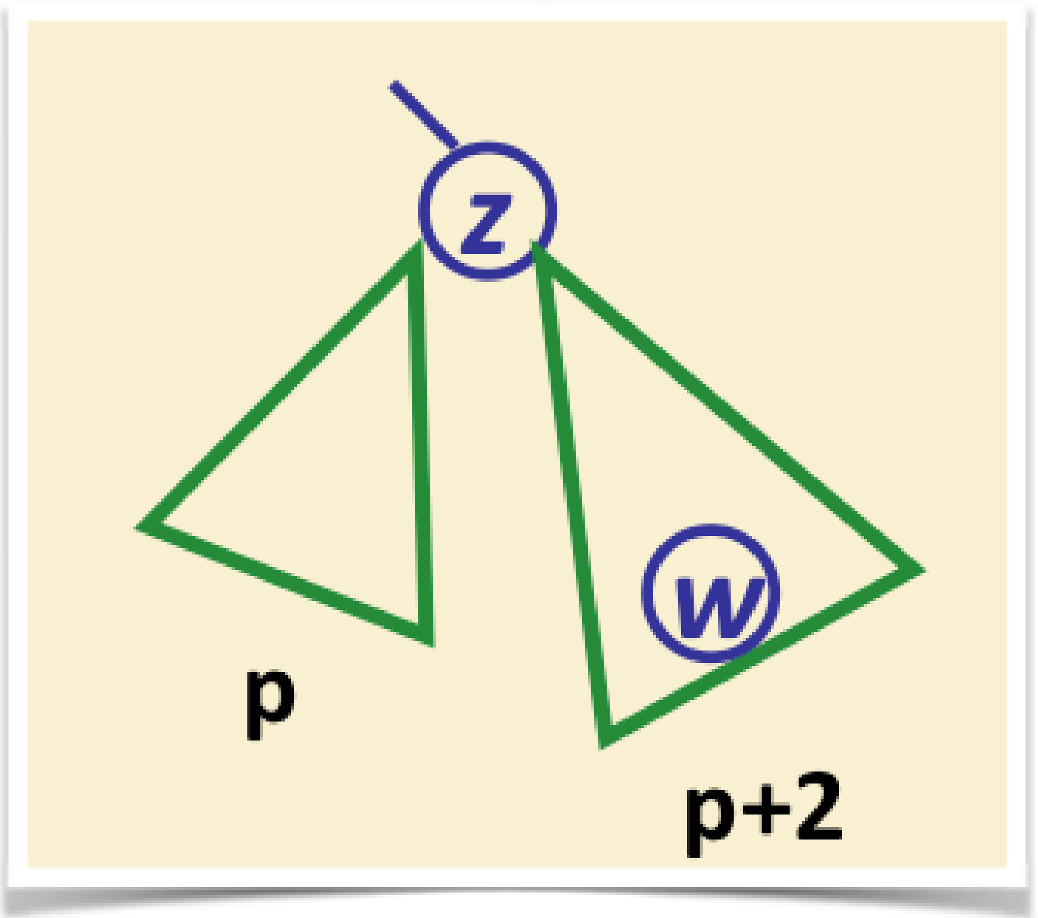
Restructuring

- Suppose the first unbalanced node is at position **z**
- This means that height difference between the left and the right subtree of **z** is more than 1
- In fact, it is **exactly** 2



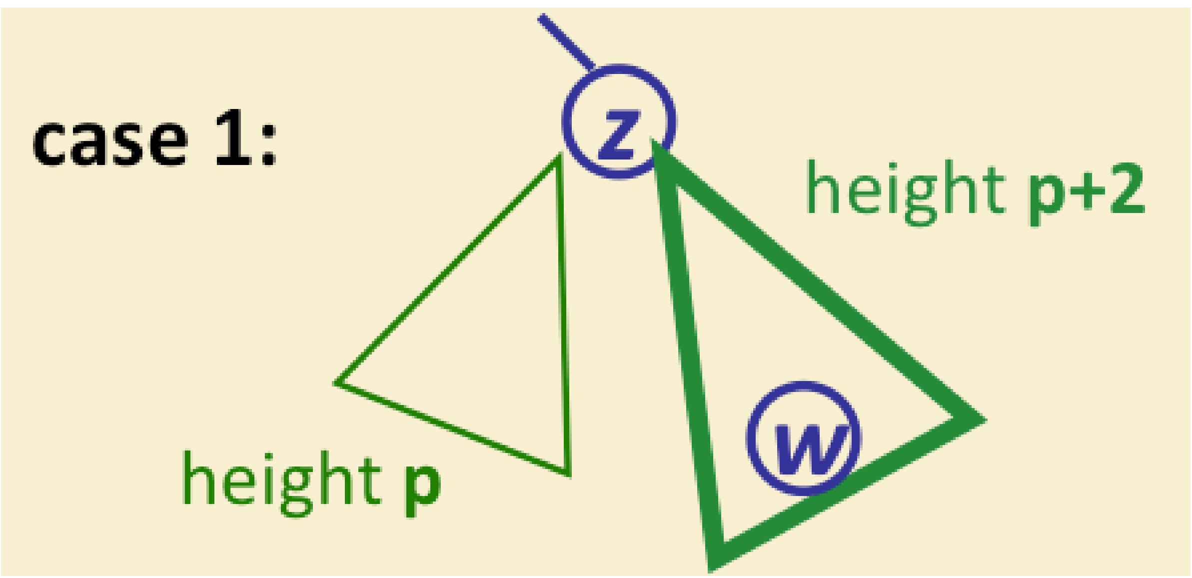
Restructuring

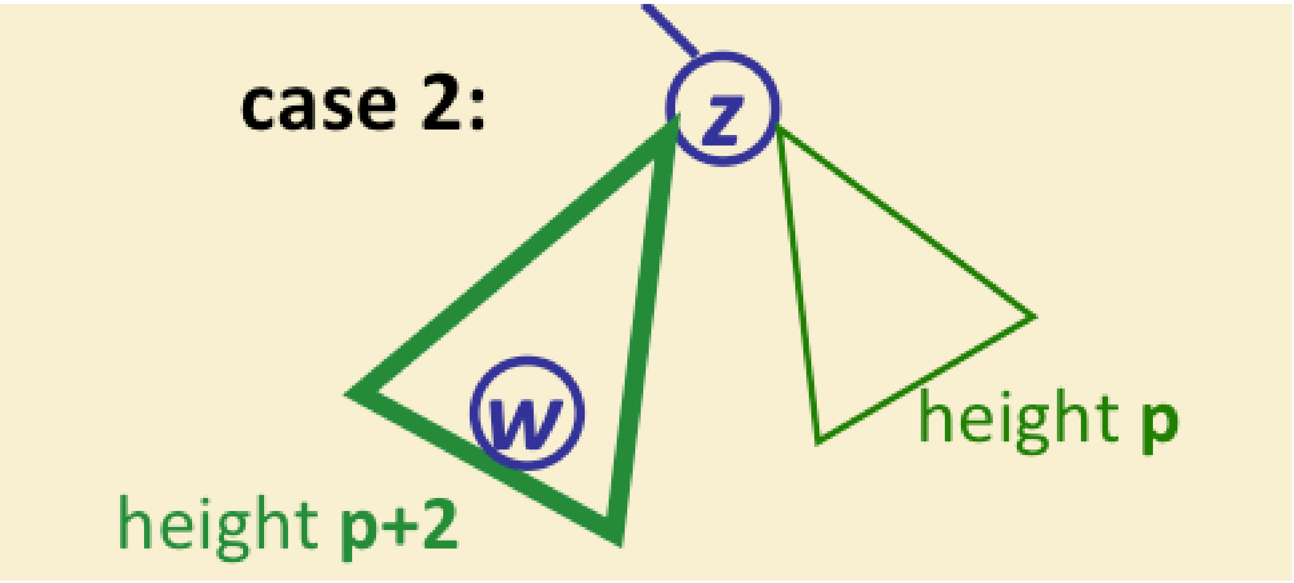
- Suppose the first unbalanced node is at position **z**
- This means that height difference between the left and the right subtree of **z** is more than 1
- In fact, it is **exactly** 2
    - ➢ tree was balanced before insertion
    - ➢ each insertion can change height only by a factor of 1
- **w** is in the higher subtree



Restructuring

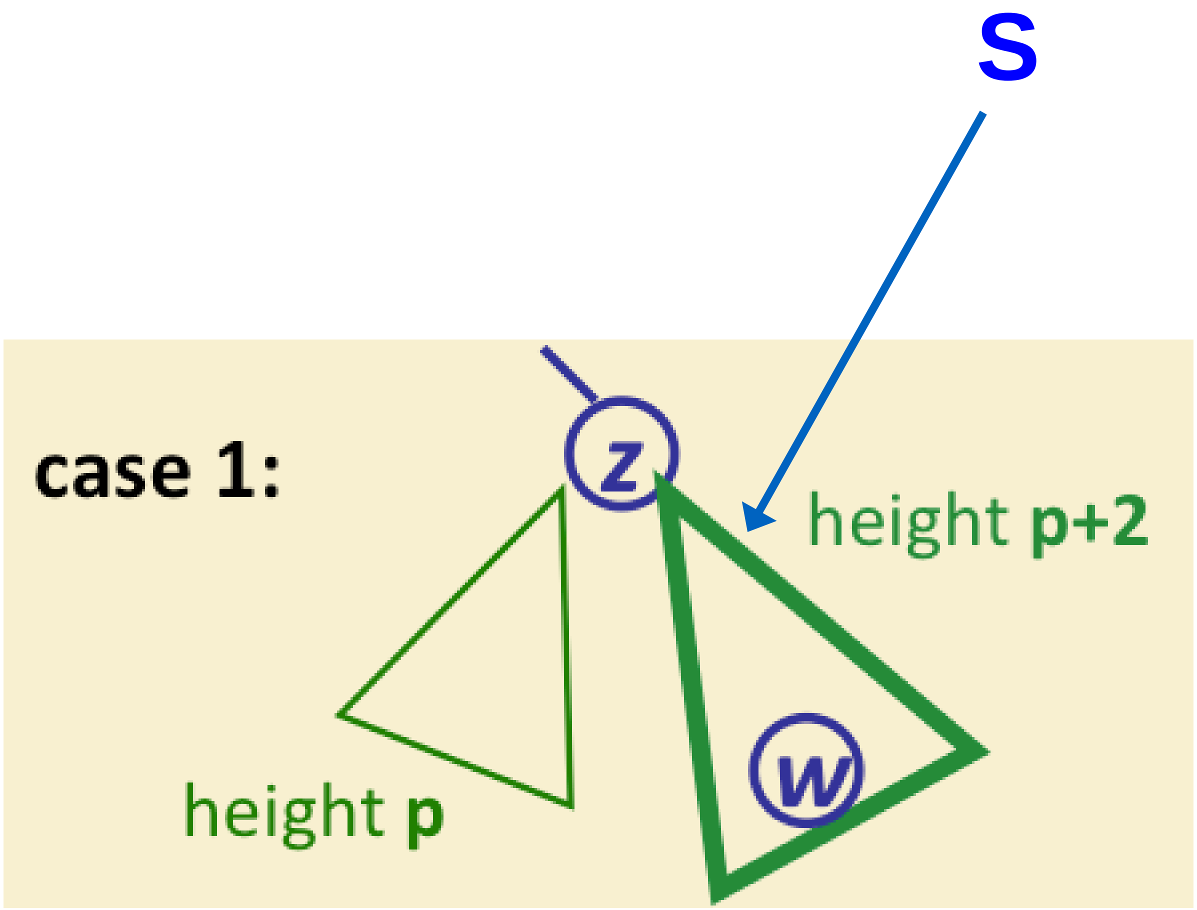
- Two cases:
  - Right subtree is higher
  - Or, left subtree is higher





Restructuring

- Let **S** be the higher subtree, with height p+2
- Let **y** be the root of **S**

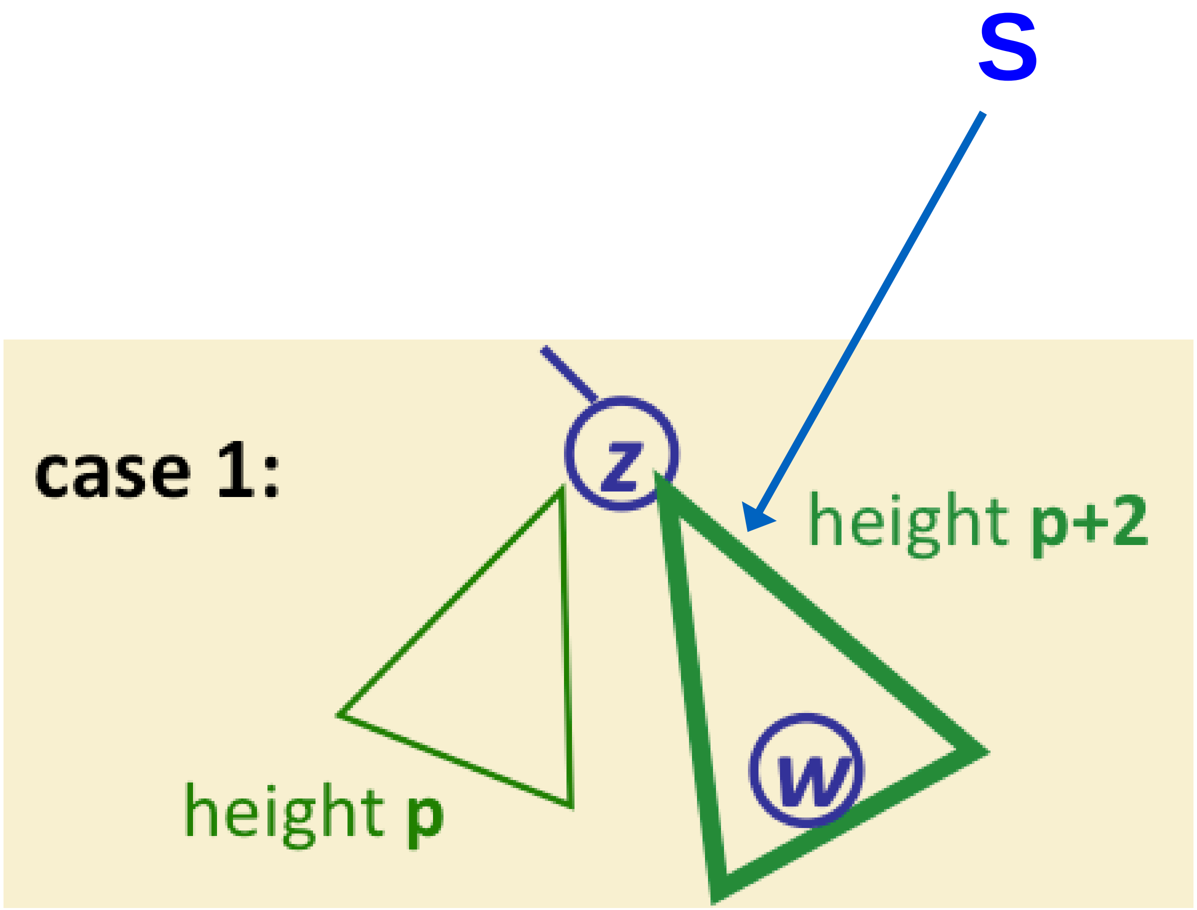


Restructuring

- Let **S** be the higher subtree, with height p+2
- Let **y** be the root of **S**

Refresher

- **w** is the new node
- **z** (ancestor of **w**) is the first unbalanced node
- **S** is the higher subtree of **z**
- **y** is the root of **S**

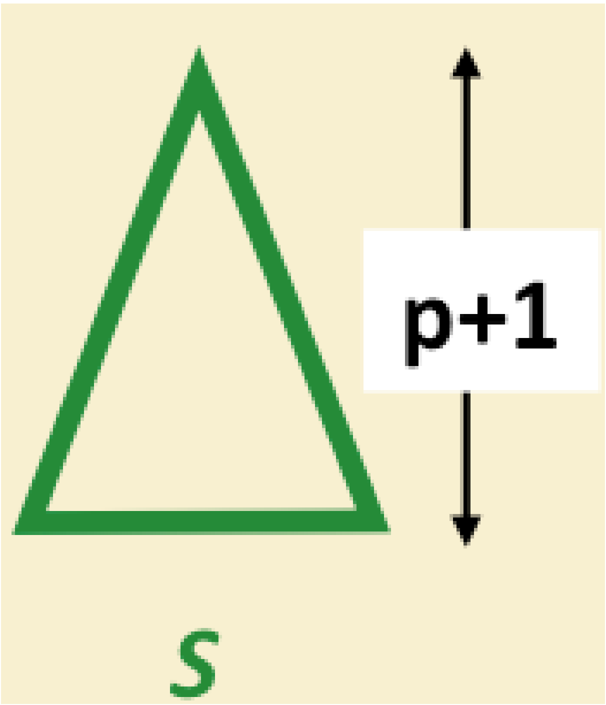


Restructuring

- Just checking!!
- What was the height of **S** before insertion?

Restructuring

- Just checking!!
- What was the **S** before insertion?

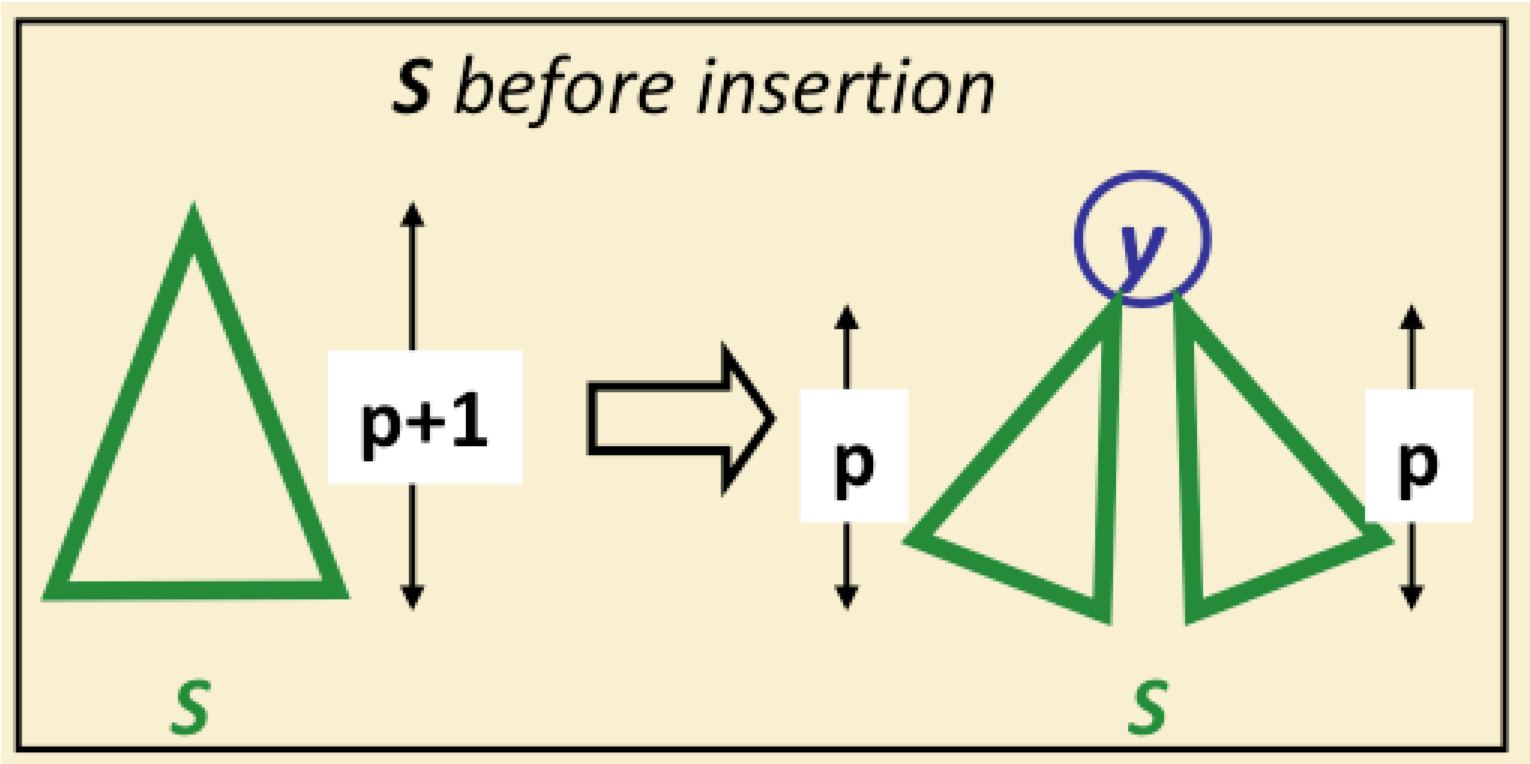


Restructuring

- **What about this one!!**
- What was the height of both subtrees of **S** before insertion?

Restructuring

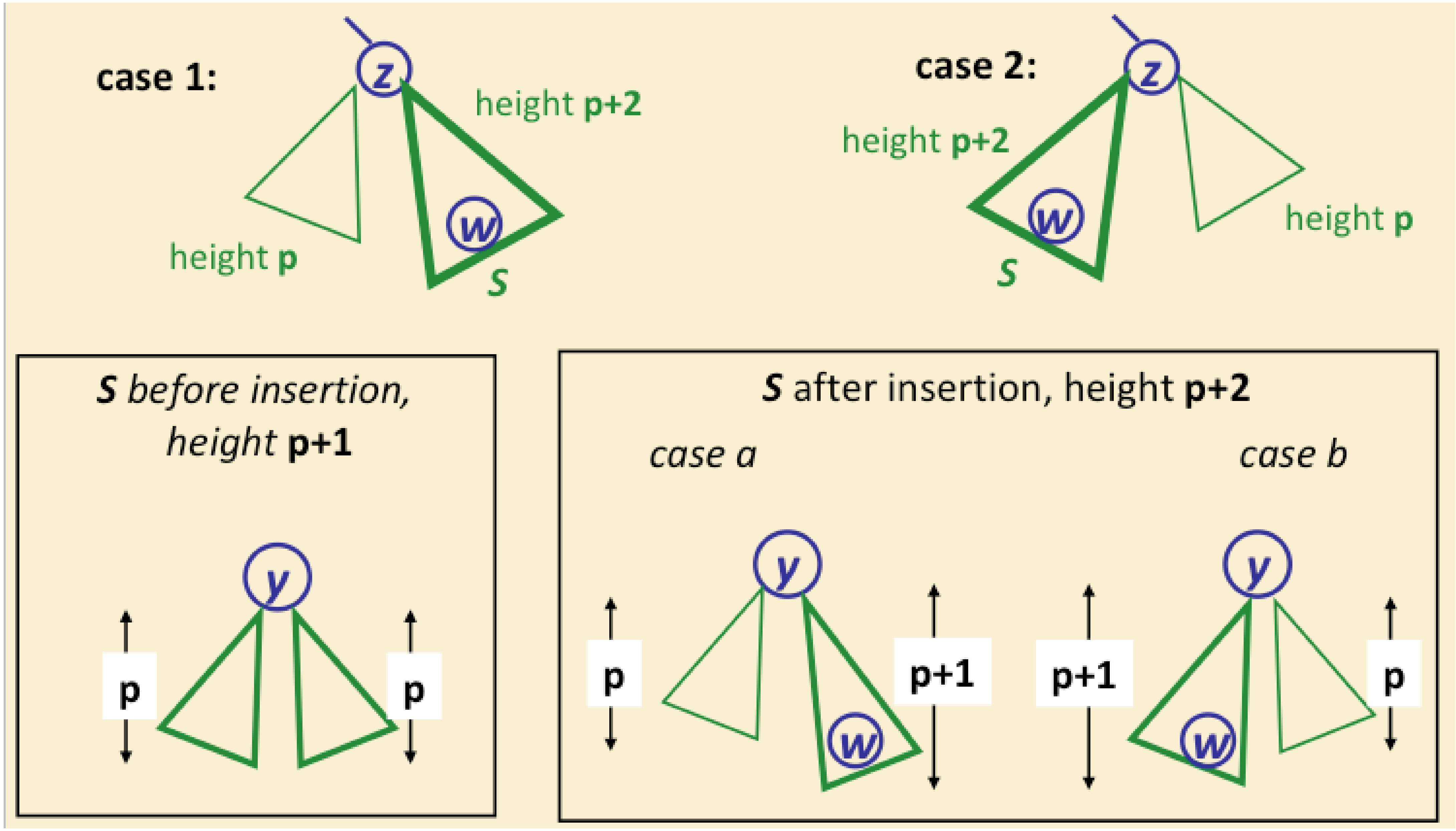
- So, both subtrees of **S** had height exactly **p** before insertion



y is balanced after insertion

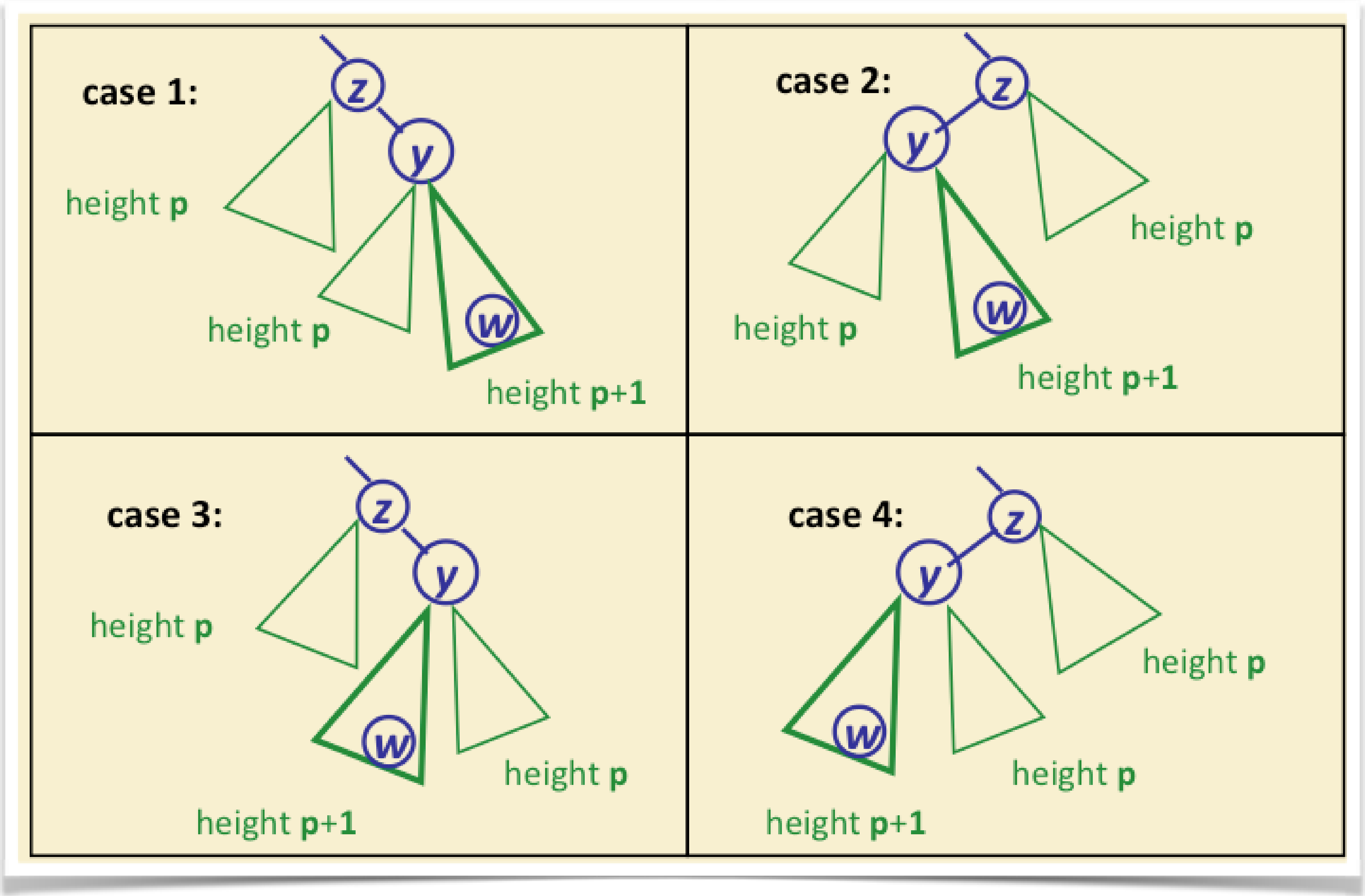
Restructuring

- Complete Picture



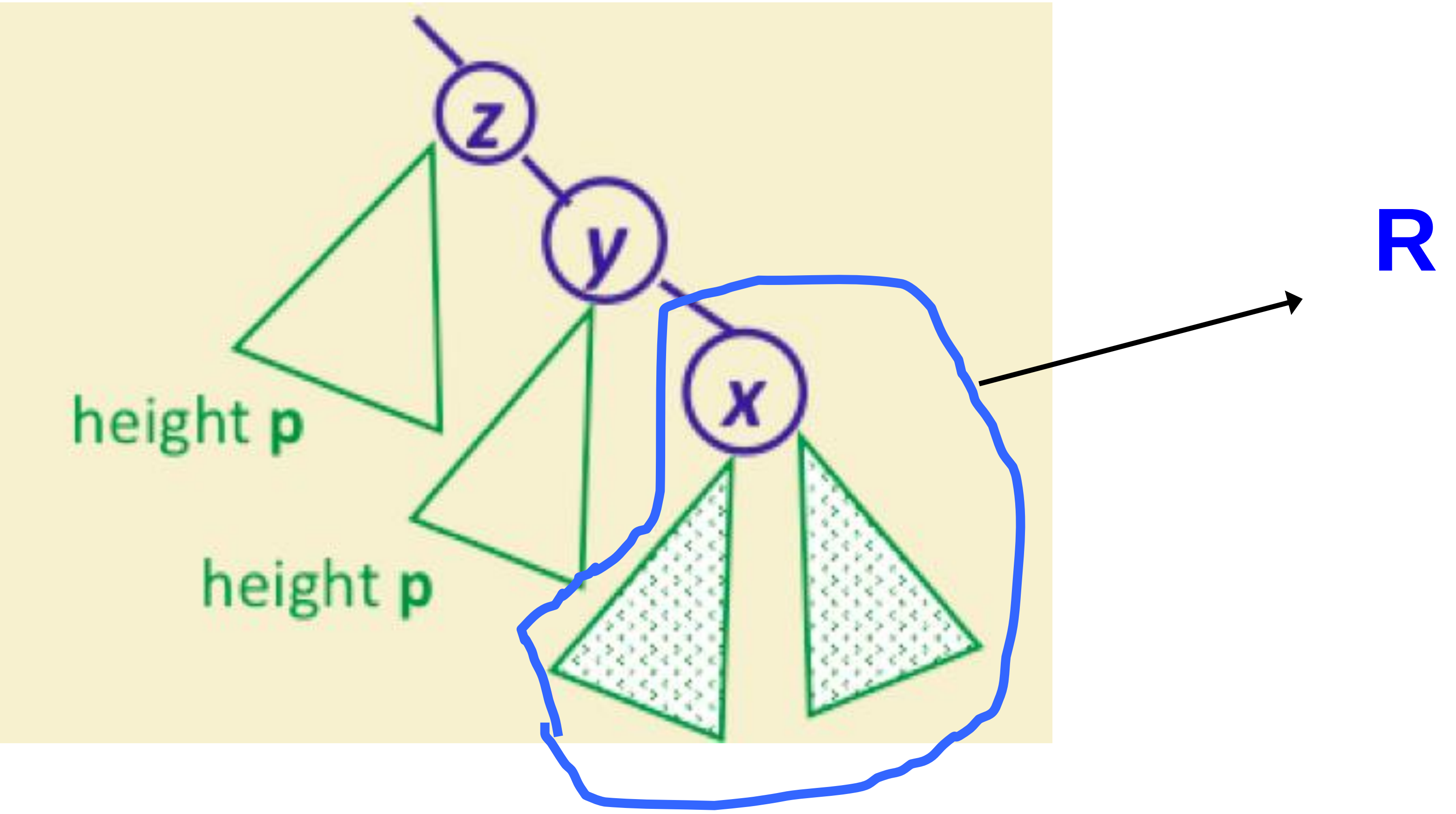
Restructuring

- More Complete Picture :)

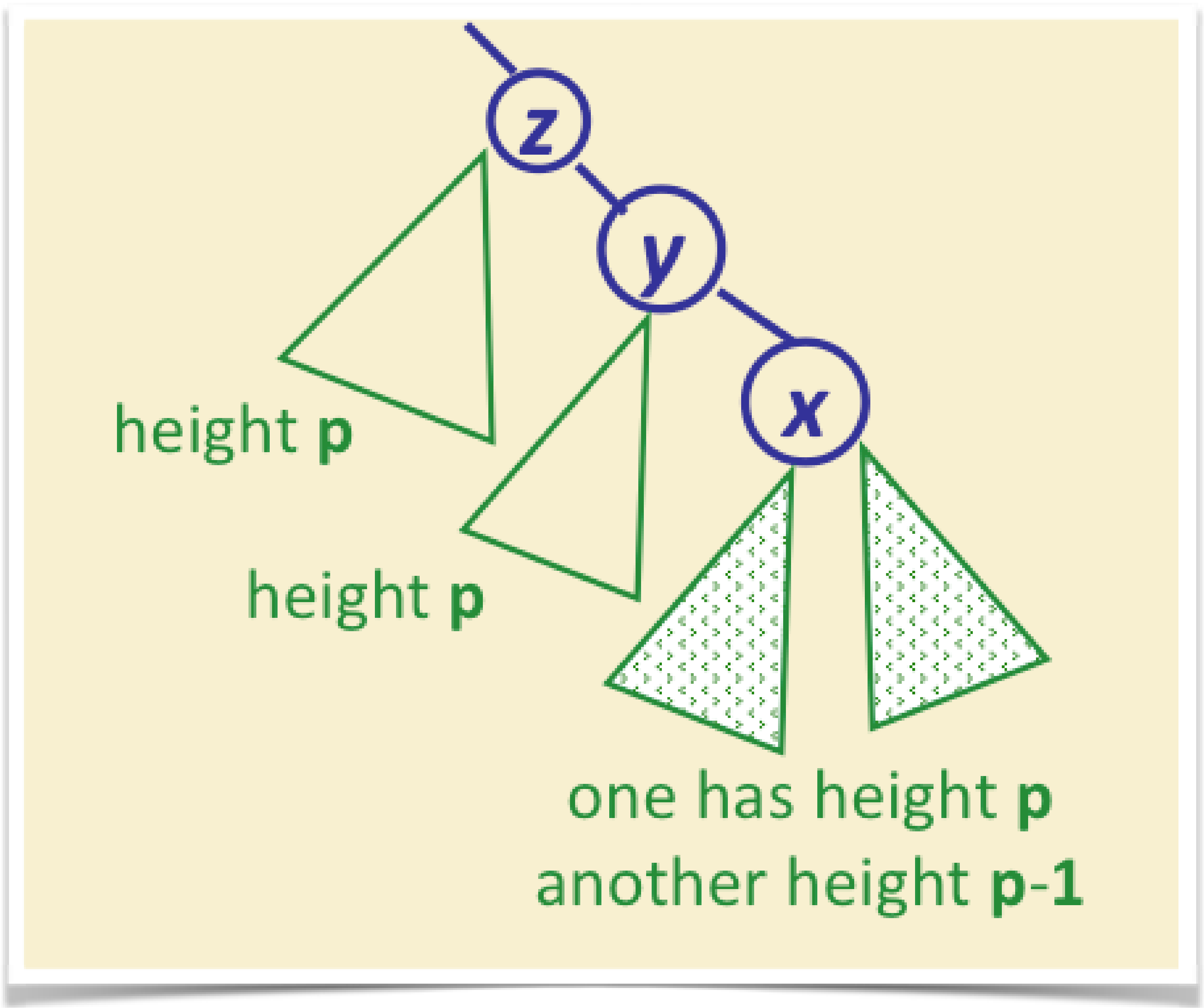


Restructuring

- Let’s consider case 1
- Let R be the right subtree of y
- Let x be the root of R

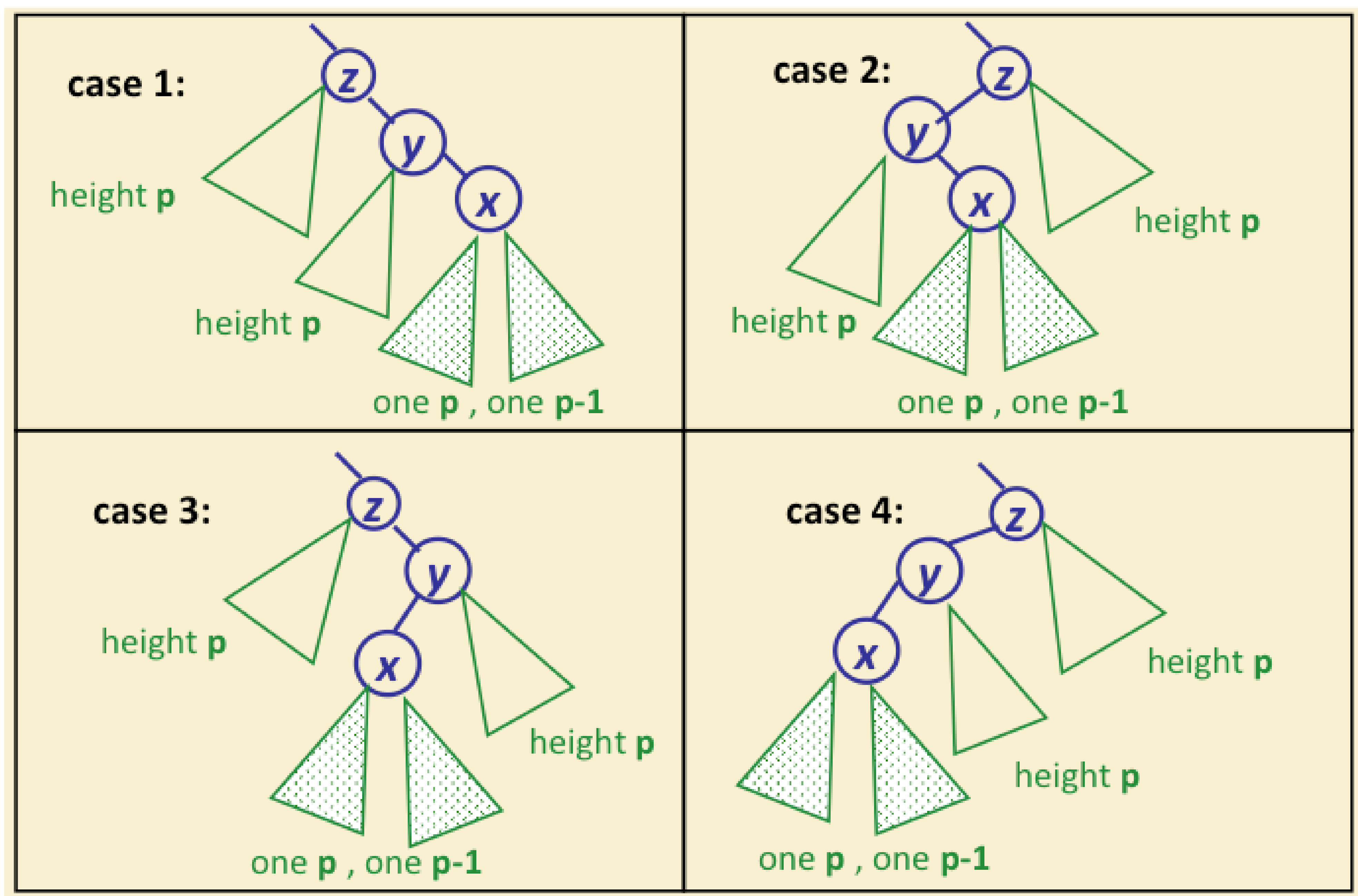


Restructuring



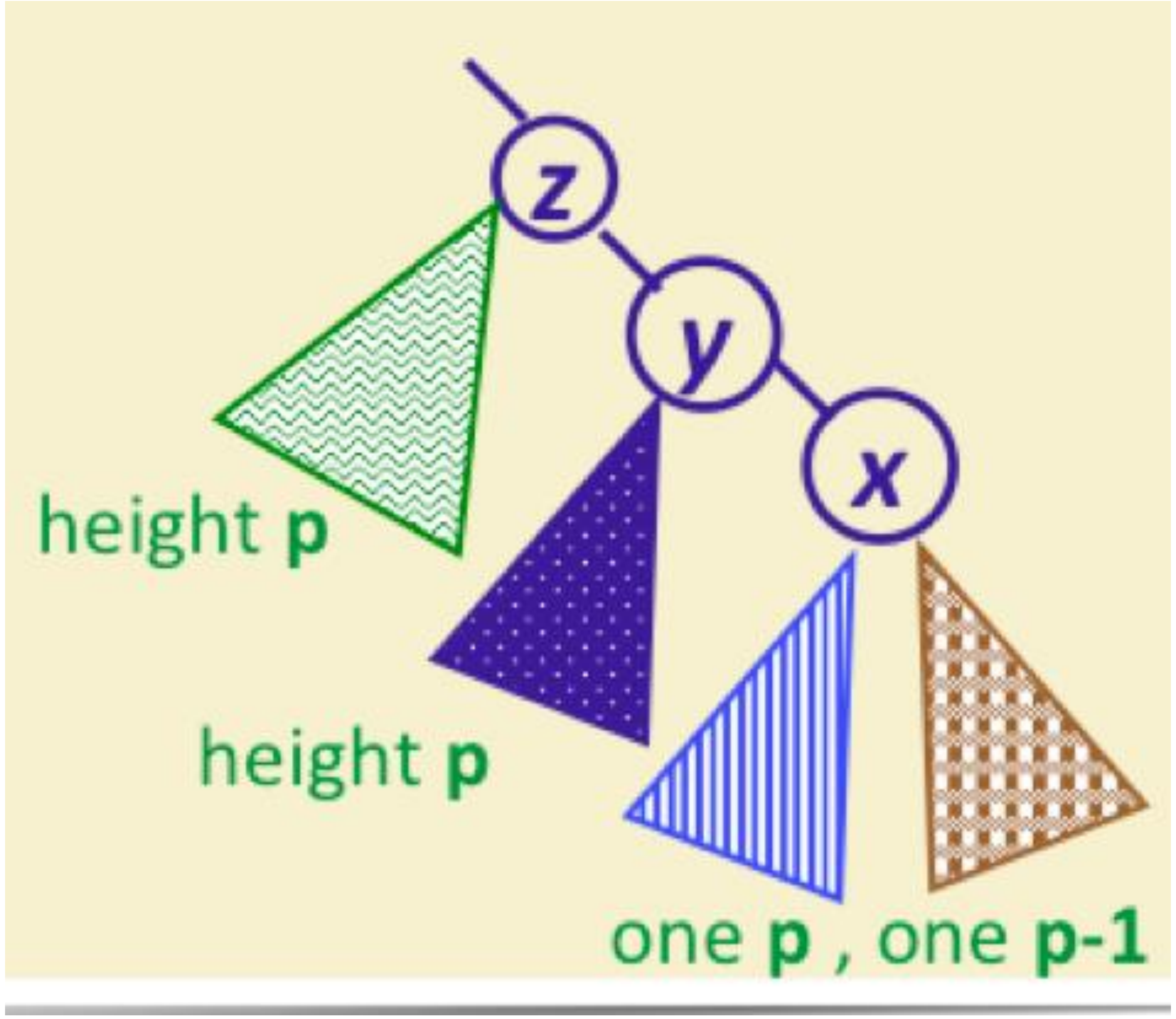
Restructuring

- Even More Complete Picture :)



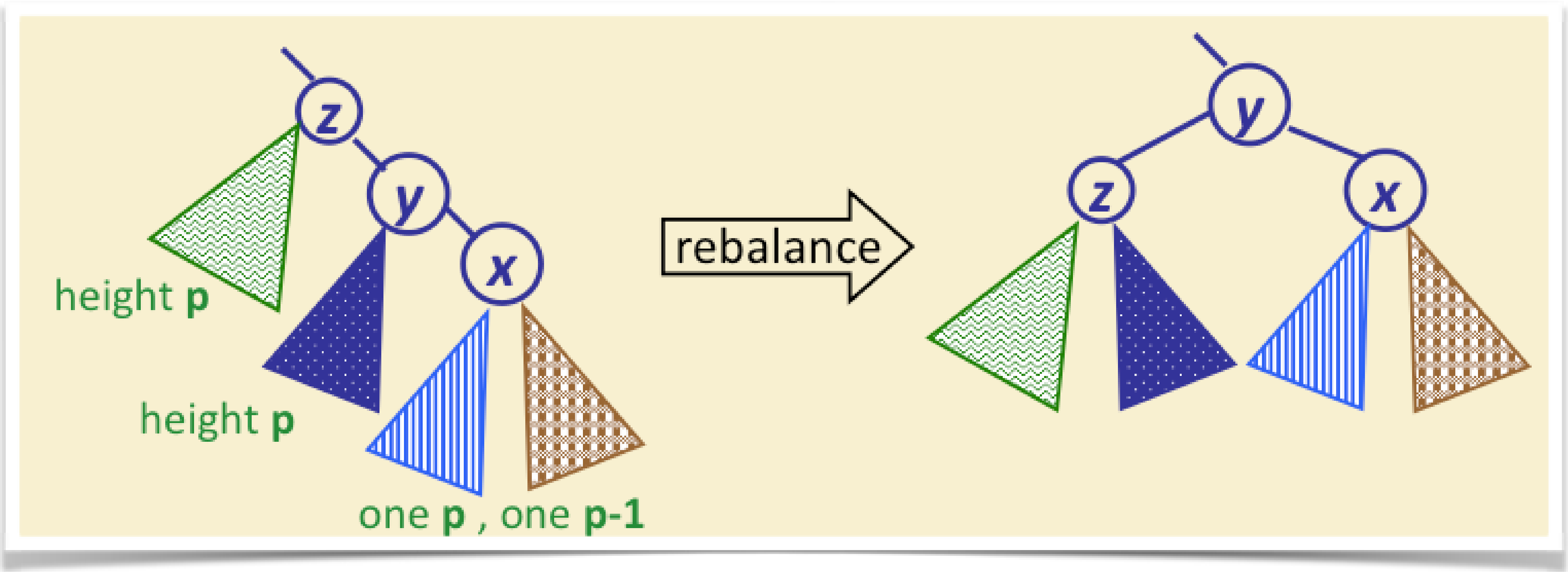
Restructuring

- Finally, let's restructure the tree
- Case 1:



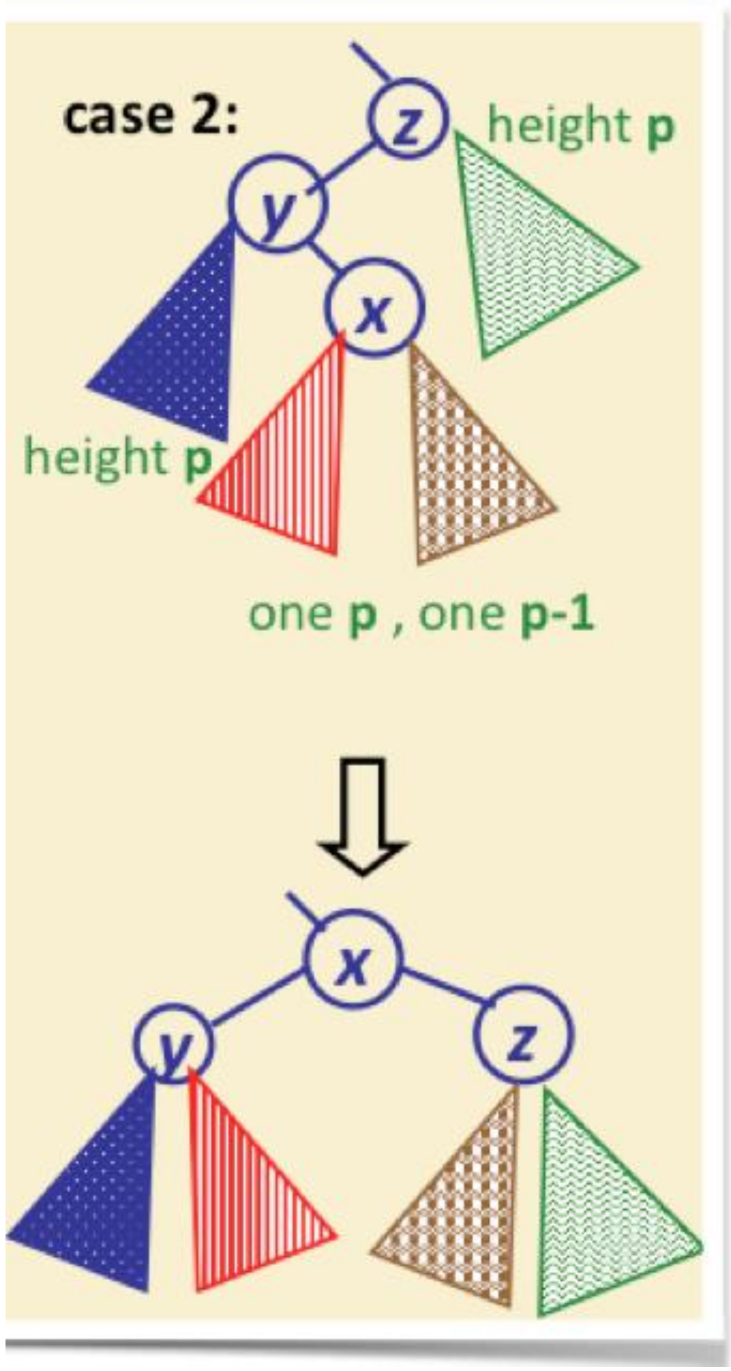
Restructuring

- Finally, let’s restructure the tree
- Case 1:

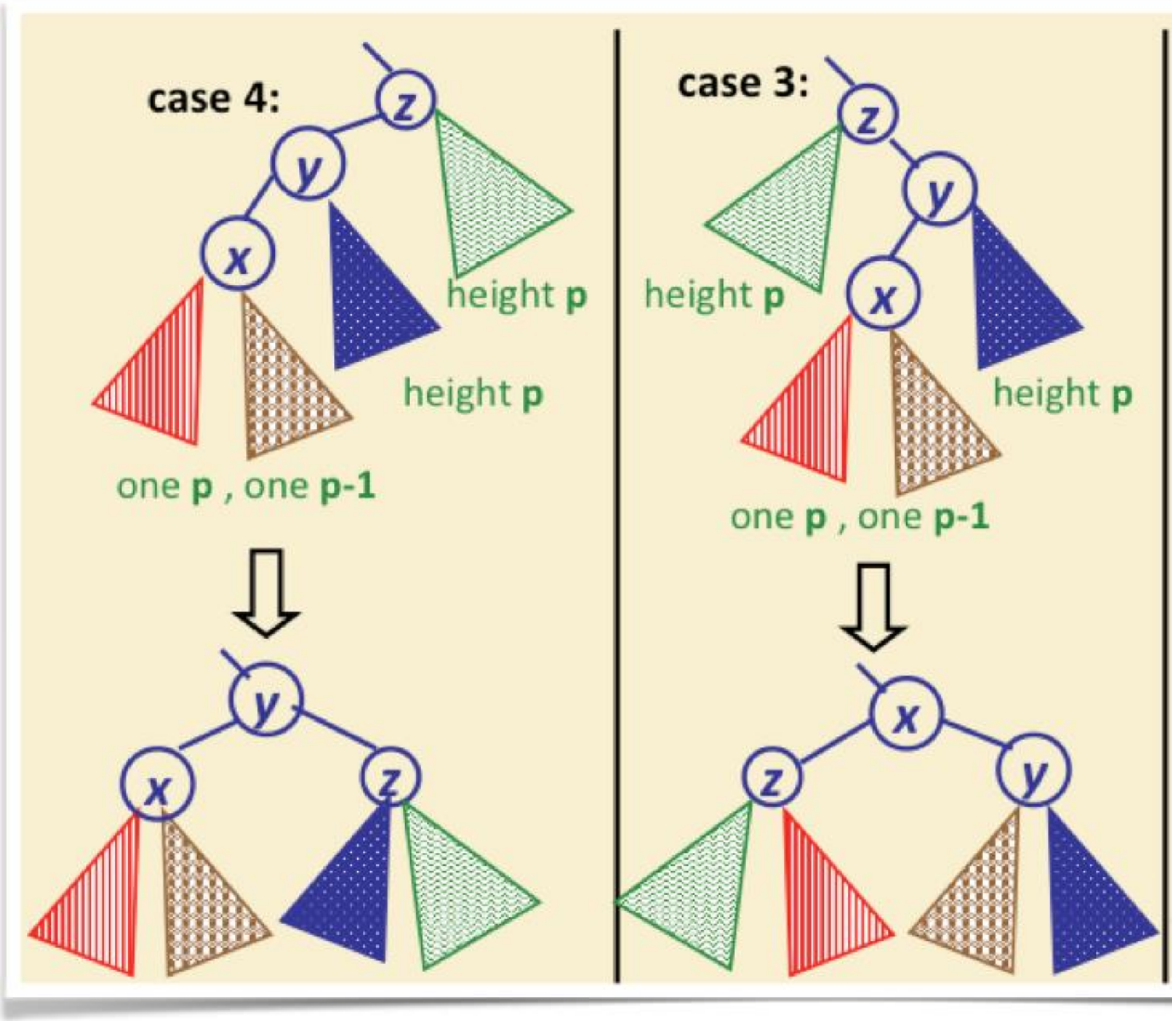


What’s the height differences at nodes **x**, **y** and **z** after restructuring?

Restructuring



Restructuring

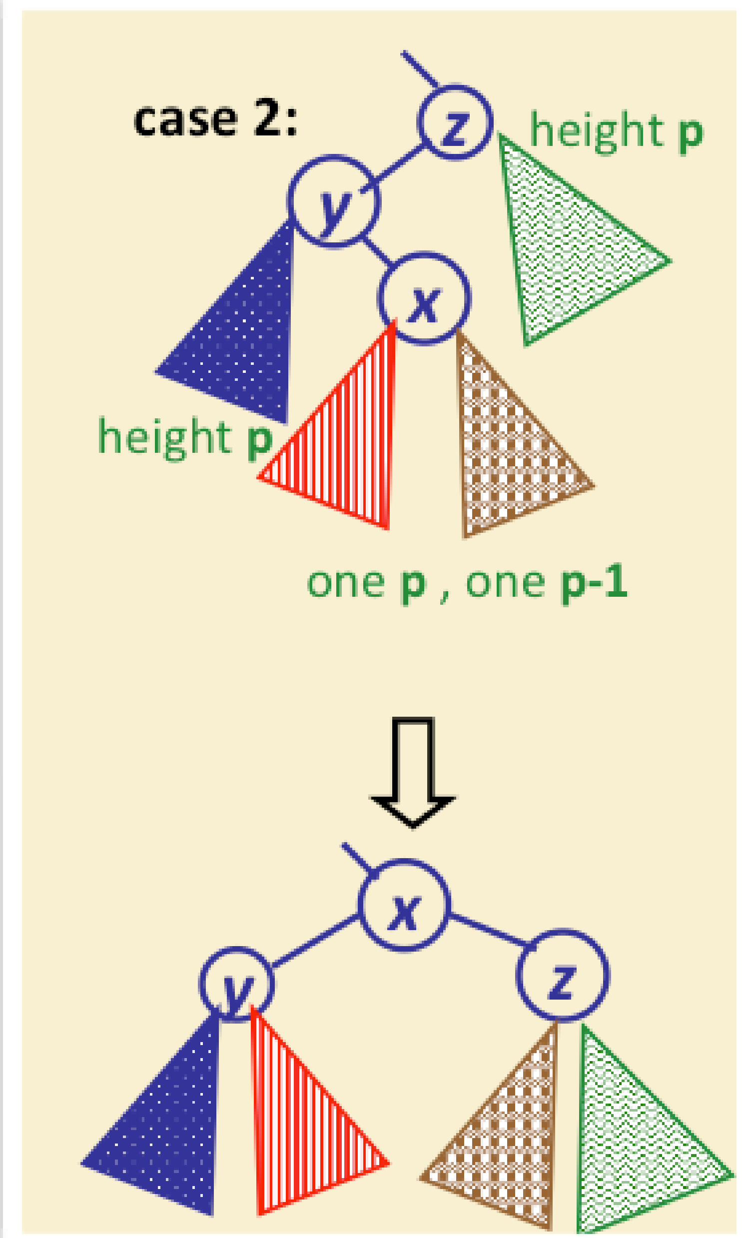


Restructuring

- All four cases can be coded with the same algorithm called: **Trinode Restructuring**
- Trinode because there are three nodes

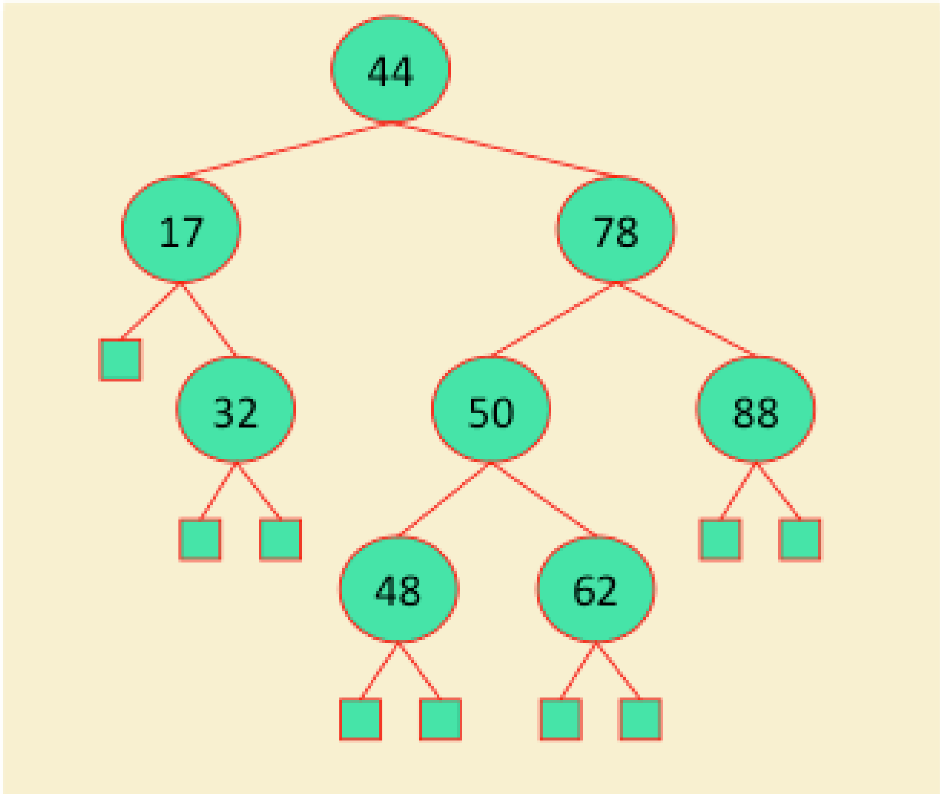
Trinode Restructuring

- In all four cases, out of the three nodes x, y, and z, **make**:
  - node with the middle key the new parent
  - smallest key node its left child
  - largest key node its right child
  - for the new parent, the previous subtree (if present) must be put in appropriate positions
    - Left subtree (if present) goes with the new left child
    - Right subtree (if present) goes with the new right child

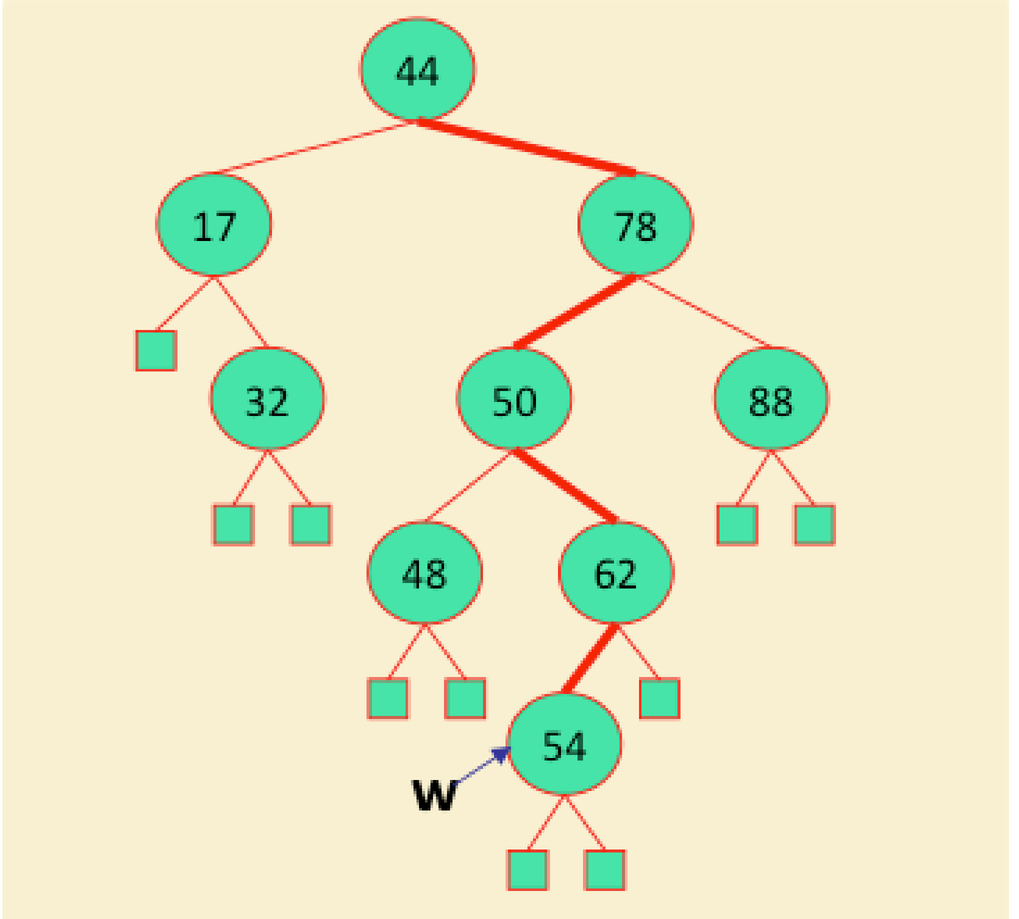


Insertion in an AVL Tree

- So can you fix this now?



before inserting 54



after insertion

Trinode Restructuring

- Takes $O(logn) + O(1)$
- No loops, no recursive calls, constant number of comparisons, and changes in parent-child relationships
- Only 1 trinode restructuring is needed per insertion to restore the height balance property

Deletion

- Deletion from an AVL tress may violate the height-balance property, too
- In this case, procedure for restructuring the tree to restore the balance is the same as in the case of insertion, with some changes
  - how to choose x, y, and z
  - repeated restructuring might be needed, max $O(\log n)$
- For further details, please read section 11.3.1 of your text-books

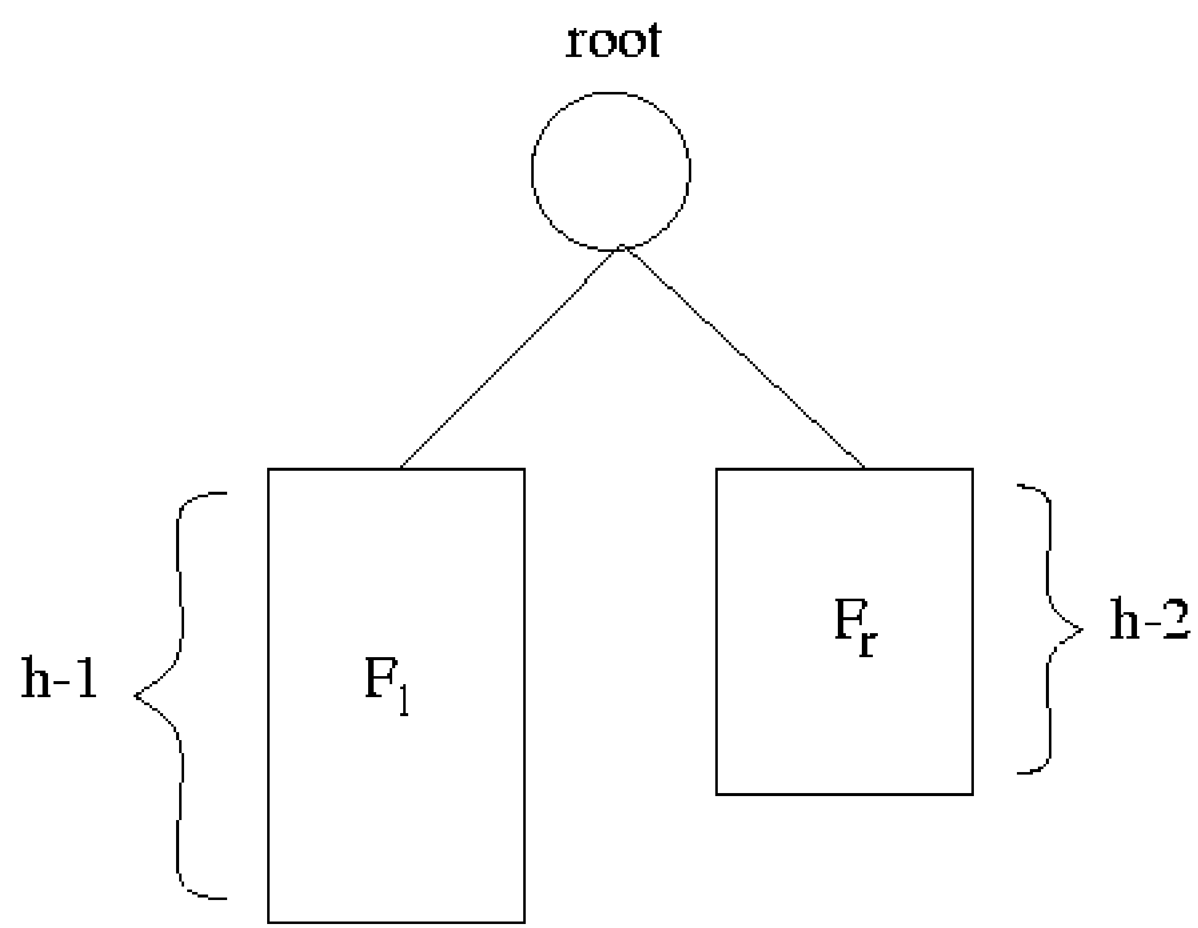
Analysis

- Ok, so we have learned how to keep a binary search tree always balanced (AVL tree) after insertions and deletions
- But why is this important?
- Recall that all we wanted was a way to make and keep the height of a tree with n nodes $O(\log n)$
- Is the height of an AVL tree $O(\log n)$?

Proof

- Proposition: The height of an AVL tree storing $\boldsymbol{n}$ entries is $O(\log n)$

Let T be an AVL tree of height h. T can be visualized as



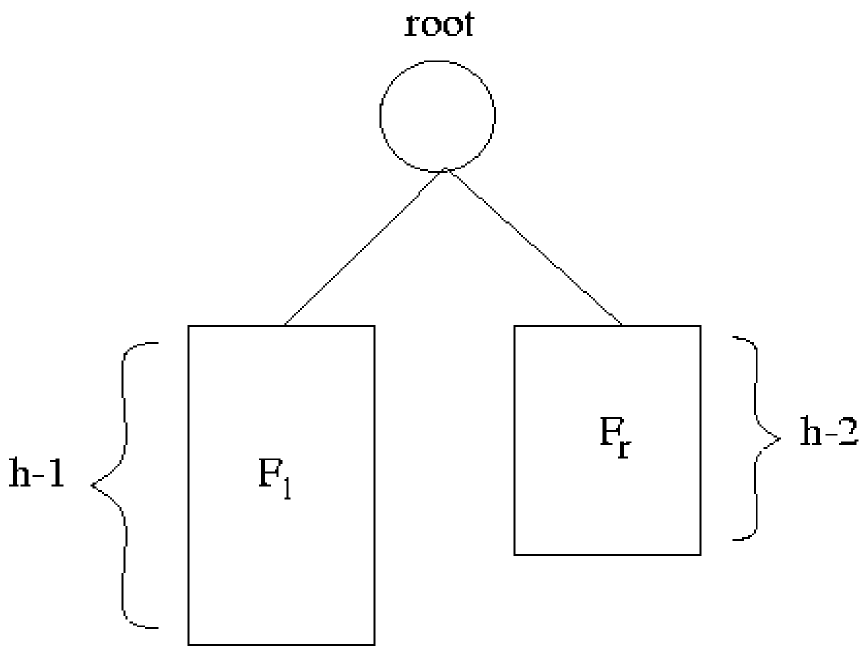
Proof

Let $n(h)$ be the minimum number of **internal nodes** in an AVL tree of height h

we know, $n(1) = 1 \; and \; n(2) = 2$

For $\boldsymbol{h >= 3}$

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



Proof

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

Now that we know this, the rest is just algebra

According to the properties of Fibonacci progressions

$n(h) > n(h-1)$, so $n(h-1) > n(h-2)$

By replacing $n(h-1)$ with $n(h-2)$ and dropping the 1, we get

$$\boldsymbol{n(h) > 2n(h-2)}$$

Proof

$$\boldsymbol{n(h) > 2n(h-2)}$$

We can stop at this point. We have shown that $\boldsymbol{n(h)}$ at least **doubles** when $\boldsymbol{h}$ goes **up by 2**.

Proof

$$\boldsymbol{n(h) > 2n(h-2)}$$

We can stop at this point. We have shown that $\boldsymbol{n(h)}$ at least **doubles** when $\boldsymbol{h}$ goes **up by 2**. This says that $\boldsymbol{n(h)}$ is **exponential** in $\boldsymbol{h}$, and hence $\boldsymbol{h}$ is **logarithmic** in $\boldsymbol{n}$

Proof

$$\boldsymbol{n(h) > 2n(h-2)}$$

We can stop at this point. We have shown that $\boldsymbol{n(h)}$ at least **doubles** when $\boldsymbol{h}$ goes **up by 2**. This says that $\boldsymbol{n(h)}$ is **exponential** in $\boldsymbol{h}$, and hence $\boldsymbol{h}$ is **logarithmic** in $\boldsymbol{n}$

But let's continue

$$\boldsymbol{n(h)\ >\ 2\,(2n(h-4))\ =\ 2^2 n(h-4)}$$

Proof

Thus,

$$For\ any\ i\ >\ 0, \qquad n(h)\ >\ 2^i\, n(h-2i)$$

Let's **get rid of $\boldsymbol{i}$ by expressing it in terms of $\boldsymbol{h}$**, but choose a value that results in making $h-2i$ either 1 or 2.

Proof

Thus,

$$For\ any\ i\ >\ 0, \qquad n(h)\ >\ 2^i\, n(h-2i)$$

Let's **get rid of *i* by expressing it in terms of *h***, but choose a value that results in making $h-2i$ either 1 or 2.

It is because we know the values for $n(1)$ and $n(2)$

That is, let

$$i\ =\ h/2-1$$

Proof

Thus,

$$For\ any\ i\ >\ 0, n(h)\ >\ 2^{i}\, n(h-2i)$$

Let's **get rid of** ***i*** **by expressing it in terms of** ***h***, but choose a value that results in making $h - 2i$ either 1 or 2.

It is because we know the values for $n(1)$ and $n(2)$

That is, let

$$i\ =\ h/2-1$$

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

Proof

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

By taking logarithmic of both sides

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

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

Or

$$h < \log(n)$$

Did we achieve today's objectives?

- Balanced Binary Search Trees
- AVL Trees
- Insertions and Deletions in AVL Trees
  - Trinode Restructuring (Rotations)
- Height of AVL Trees

Home Work!

- Read about AVL trees in Data Structures & Algo. In Java
  - Understand how “deletions work in AVL trees
- Solve some examples on both insertions and deletions in AVL trees
- Read about Number of different binary trees in Introduction to Algorithms