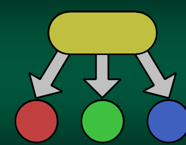
Balanced Trees

Section 3.3

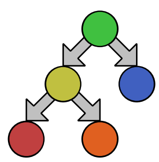


2-3 Trees

Balance using really big nodes

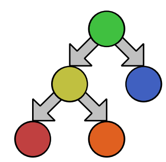
Binary Search Tree Issues

- Binary Search Trees have the ability to find data in O(log n)
- This is incredibly more efficient than a linear search of O(n)
- However, internal nodes never change and have a huge impact on the tree

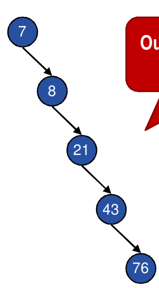


Binary Search Tree Issues

- There are cases where the tree is unbalanced – one particular path contains all the data
- In this case, the time complexity slowly deteriorates to O(n)



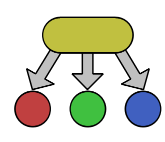
Very Unbalanced Tree



Our tree is no better than a linked list!

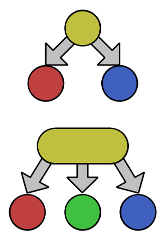
2-3 Trees

- The *2-3 Tree* is a special type of BST invented by *John Hopcroft* in 1970
- It automatically maintains balance as it grows!
- It does this by using a clever variation of the node that can contain multiple values



2-3 Trees

- *2-Nodes* contain 1 values and two children: left and right
- *3-Nodes* contains 2 values and three children: left, middle and right
- Both are easily to code and traversal logic is straight forward



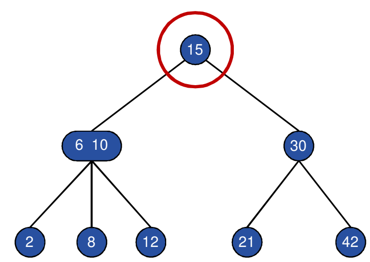
Searching a 2-3 tree

- Searching a 2-3 Tree is very similar to a Binary Search Tree, but with a minor difference
- 2-nodes are treated the same as they are in BSTs:
  - if less than, go left
  - if greater than, go right

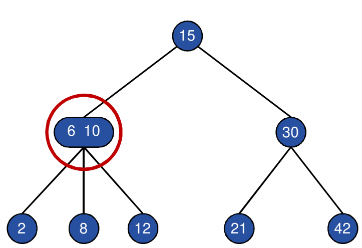
Searching a 2-3 tree

- 3-nodes are a bit different
- Since they have 2 values, *a* and *b*, we do the following:
  - if less than *a*, go left
  - if between *a* and *b*, go middle
  - if greater than *b*, go right

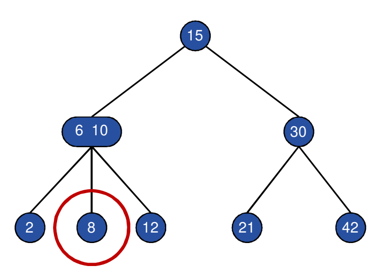
Search for 8: Go Left



Search for 8: Go Middle

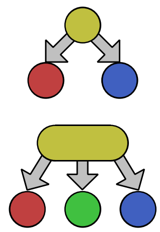


Search for 8: Found



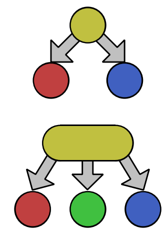
Adding to a 2-3 tree

- For BSTs, when a value is added, it will search and then create a new left or right leaf
- 2-3 Trees, however, will merge the value into the bottom node (rather than creating a new node)

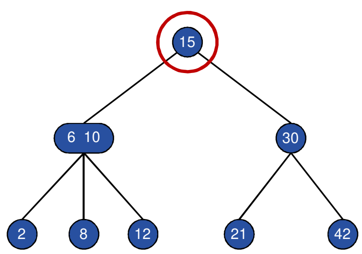


Adding to a 2-3 tree

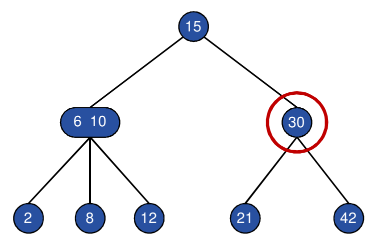
- This will convert a 2-Node into a 3-Node (it now has two values and three links)
- A 3-Node will convert into a temporary structure called a 4-Node… but we will get to that later



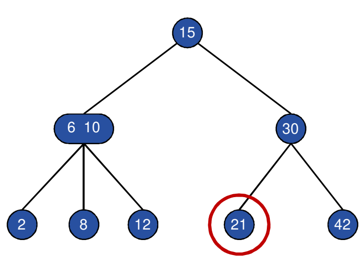
Add 25: Go Right



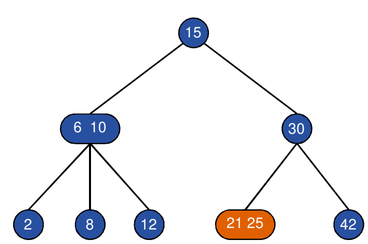
Add 25: Go Left



Add 25: Can't go further



Add 25: Convert 2-Node to 3-Node

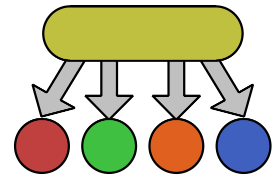


Adding to a 2-3 tree

- Notice, when the value was added to the 2-3 Tree, that the height of the tree *did not change*
- Binary Search Tree would have added another child node and the height *would have changed*

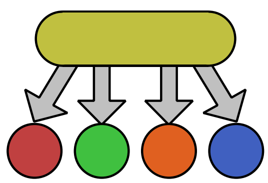
The 4-Node

- So, what happens when we add a value to a 3-node?
- It becomes a *4-Node*, which has 3 values and 4 children
- **This is temporary**, it will be converted

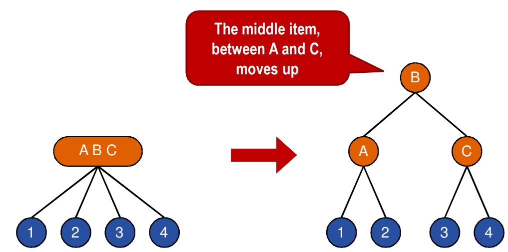


The 4-Node

- When a 4-Node is created, the 2-3 Tree algorithm will *split* it into other nodes
- Given that 4 is a nice even number, we can split equally
- … and balanced!

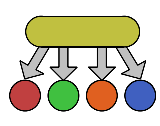


One Way to Split a 4-Node



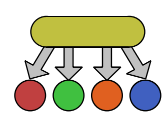
The Types of Splits

- There are a total of six different splits that can occur in a 2-3 tree
- In each split, the **middle** value **ascends** up to the parent node



The Types of Splits

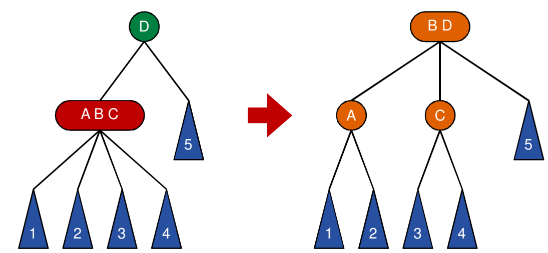
- This will change a parent from a 2-Node to 3-Node
- … or from 3-Node to 4-Node
  - then, the parent will split
  - it continues to bubble up – possibly all the way to the root
  - this is O(log n)



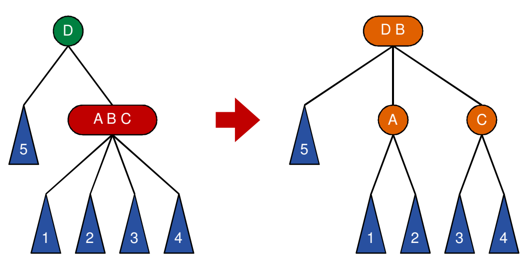
The Six Splits

- Parent is 2-Node:
  - node is the left child of the parent (1)
  - node is the right child of the parent (2)
- Parent is 3-Node:
  - node is the left child of the parent (3)
  - node is the middle child of the parent (4)
  - node is the right child of the parent (5)
- Node is the root (6)

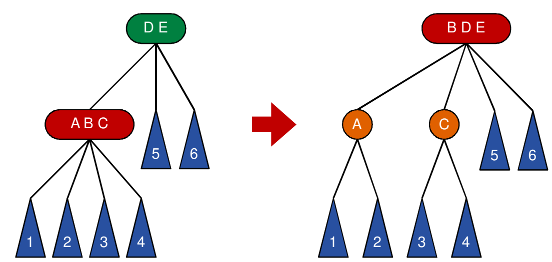
2-Node Parent, Left Child



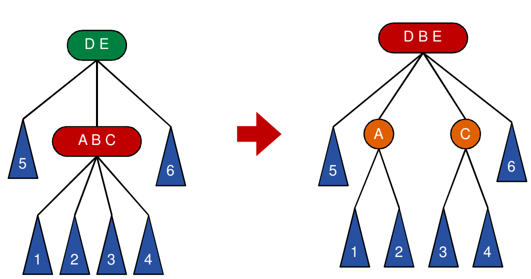
2-Node Parent, Right Child



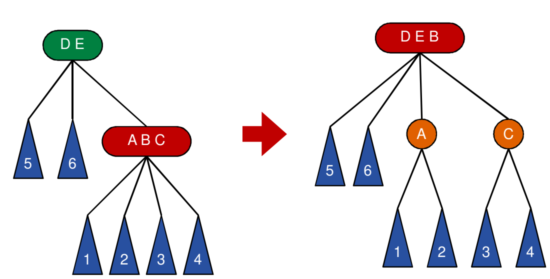
3-Node Parent, Left Child



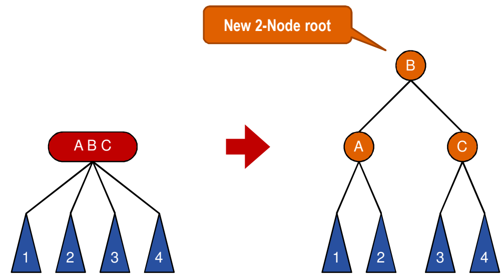
3-Node Parent, Middle Child



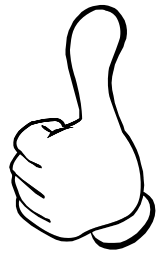
3-Node Parent, Right Child



Node is a the root



Why Does This Work?

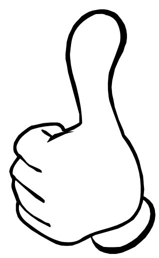


- Notice that, of the six splits, only one created a new node and changed the height
- So, *a 2-3 tree grows in depth only when the root is split*
- … and it splits balanced on the left and right side!

Why Does This Work?

- 2-3 Trees grow from the top rather than from the bottom like Binary Search Trees
- And, the tree auto-balances do to the very nature of how the nodes split
- They are always O(log n)

Why Does This Work?

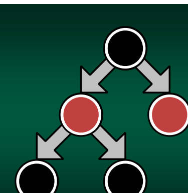


- Additional, 2-3 Trees are *incredibly* easy to write
- When a recursive call completes, in the case of a split, you can return the middle value
- So, as recursion bubbles up, you can handle all splits

Let's Try Some…

- Let's try two sets of numbers inserted into a Binary Search Tree and 2-3 Tree
- This will allow us to contrast the difference

```
1. Numbers: 6, 2, 1, 3, 5, 4, 7
2. Numbers: 1, 2, 3, 4, 5, 6, 7
```

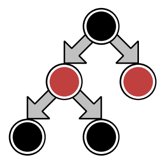



Red-Black Trees

Bringing Balance with Ease

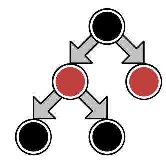
Red-Black Trees

- 2-3 Trees are re-mark-a-ble!
- However, the nodes are a tad complex
- Can we implement the same concept by using the Binary Search Tree's basic 2-Node?
- The answer is: ***yes!***

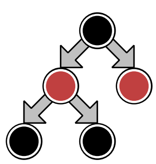


Red-Black Trees

- The *Red-Black Tree* is ADT that implements a 2-3 tree using strictly 2-nodes
- However, this does add some complexity to our balancing logic… but we will get the same results

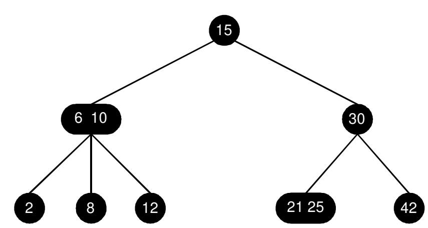


Red-Black Trees

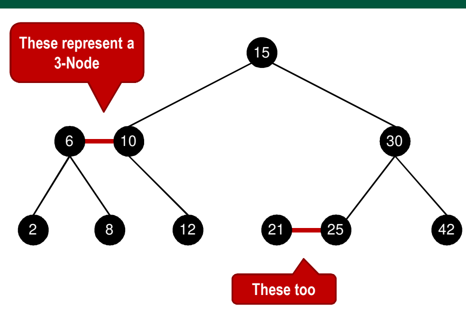


- So, let's look a 2-3 tree and make some modifications
- First, we will convert all of our 3-nodes into a chain of two 2-Nodes
- So we know that they belong together, let's make the branch as **red**

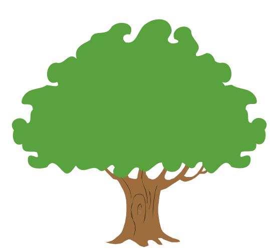
Basic 2-3 Tree



Represented with only 2-Nodes

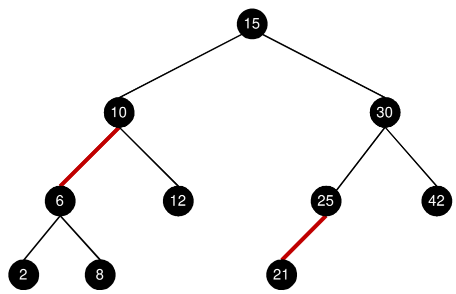


Red-Black Trees

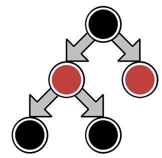


- Of course, we don't typically represent trees using horizontal links
- So, let's rearrange the nodes into a typical tree structure

Same tree – normal layout

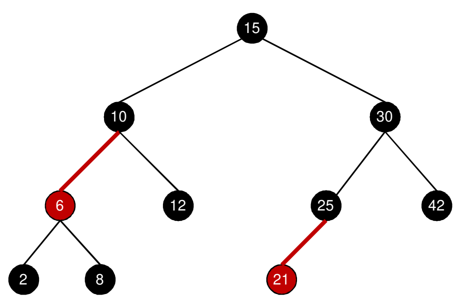


Coloring the node



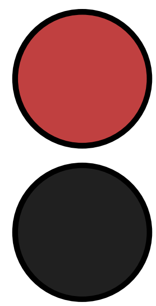
- Naturally, we can't color branches (which are just links) in Java
- … or any major language
- We can color the nodes, that are children of the red-branch, as **red**

Coloring the Nodes Red



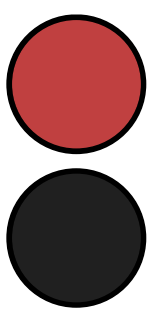
Red-Black Tree Definition

- In a *Red-Black Tree*, every node is marked as either **Red** or **Black**
- Colors were arbitrarily chosen
  - there is no metaphor
  - they looked best on laser printers at the time



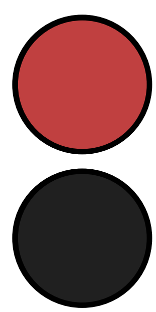
Red-Black Tree Definition

- If a node is **Red** then both children are **Black**
- That makes sense, or it would be representing a 4-Node (or something even larger)



Red-Black Tree Definition

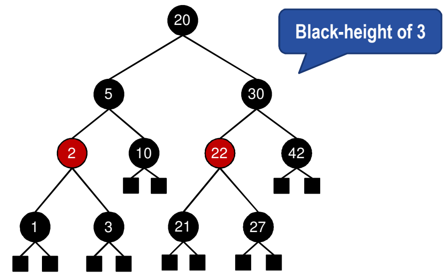
- The root considered **Black**
- Null pointers are considered **Black** nodes
  - even though they are not
  - the algorithm uses this logic



The Black-Height

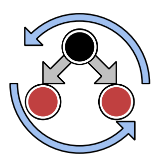
- *Black-height* of a node is the number of **Black** nodes on any path to a null
- We don't count red nodes since they are represent part of a 3-Node
- We also don't count the root
- Every path from any node to a null contains the same number of **Black** nodes

Black-heights



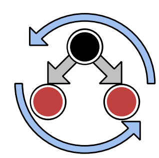
Balancing the tree

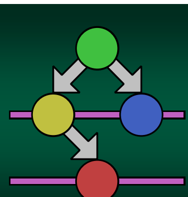
- Balancing the Red-Black tree is done in the same manner as a 2-3 tree
- However, because we use 2-Nodes, we use a series of *rotations* to get the same effect as splits



Balancing the tree

- They are tad more complicated
- Unfortunately, we don't have time to cover them this Summer, but they are cool



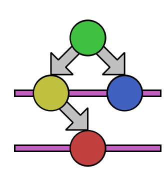


AVL Trees

Bringing balance… aggressively

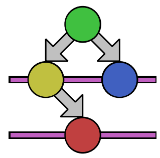
AVL Trees

- *AVL Tree* is a height-balanced binary search tree invented by Adelson-Velskii and Landis
- The ADT keeps track of the height of each subtree and reorders the data as needed



AVL Trees

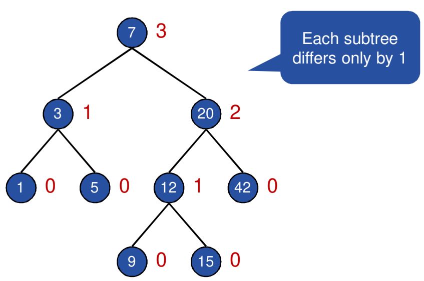
- AVL Trees aggressively balance the nodes – which ensures the O(log n) search
- So, searching is optimized
- However, these steps require considerable work and hurts efficiency



AVL Trees

- Each subtree has a "height" property
  - it is the maximum between the height of the left and right subtree + 1
  - leafs have a height of zero
- As long as the right and left branches only differ by **1**, the AVL Tree is sufficiently balanced
- If not, they are balanced by "rotating"

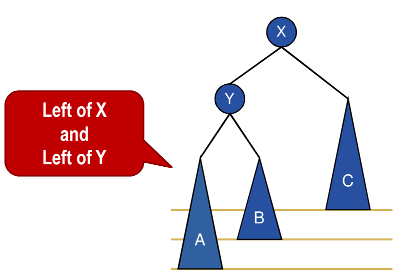
Subtree Heights



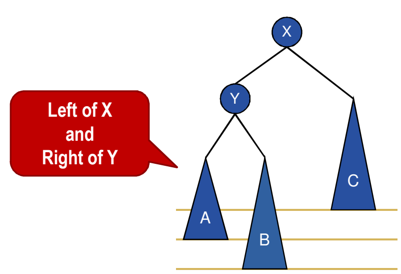
Inserting Nodes

- Unless values are inserted in a very specific order, the tree will, naturally, become unbalanced
- Imbalance falls into two distinct categories
  1. Left-Left (or Right-Right) imbalance
  2. Left-Right (or Right-Left) imbalance

Left-Left Imbalance



Left-Right Imbalance

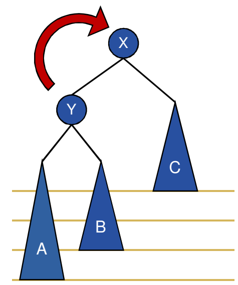


Insert and Rotate

- Only nodes on the path from insertion point to root node have possibly changed in height
- So after the Insert…
  - start balancing starting at the lowest node
  - recurse back up to the root rotating *as needed*

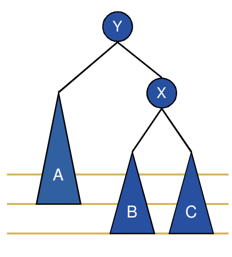
Left-Left Imbalance

- B and C have the same height
- A's height is 1 larger than B and C
- Rotate right…
  - make Y the new root
  - X its right child of Y
  - B and C subtrees of X

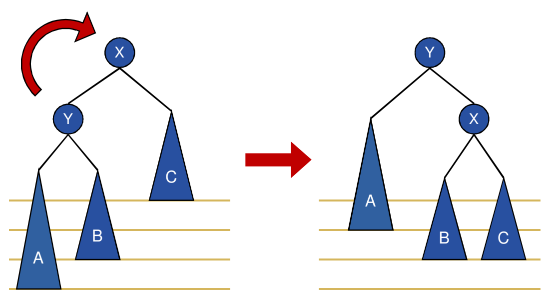


Left-Left Imbalance

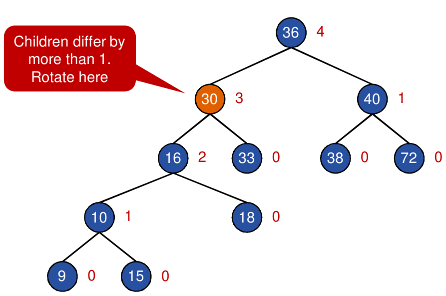
- B and C have the same height
- A's height is 1 larger than B and C
- Rotate right…
  - make Y the new root
  - X its right child of Y
  - B and C subtrees of X



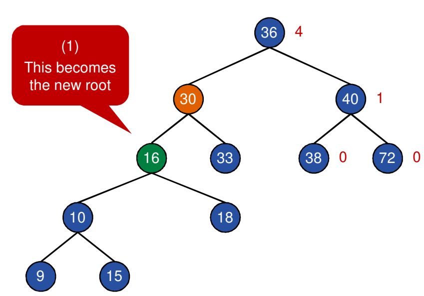
Left-Left Rebalance



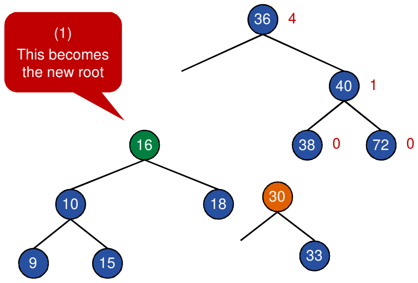
Example: Left-Left Unbalance



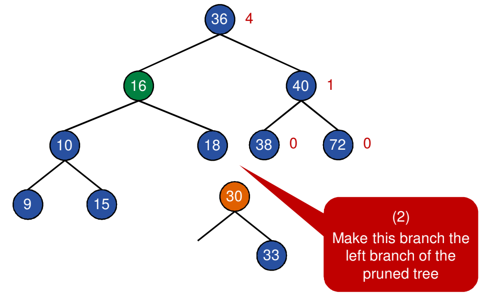
Example: Left-Left Unbalance



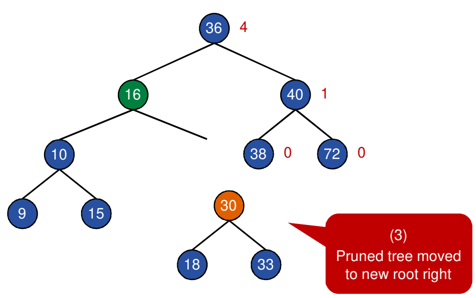
Example: Left-Left Unbalance



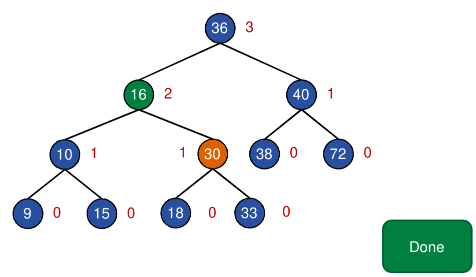
Example: Left-Left Unbalance



Example: Left-Left Unbalance

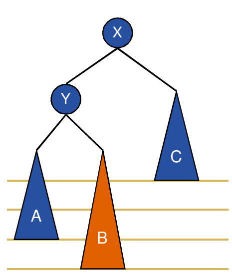


Example: Left-Left Unbalance



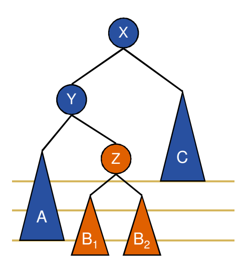
Left-Right Imbalance

- Can't use the Left-Left balance trick - because now it's the *middle subtree*, i.e. B, that's too deep.
- Instead consider what's inside B...



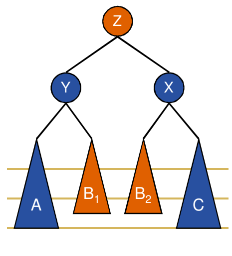
Left-Right Imbalance

- B will have two subtrees containing at least one item (just added
- We do not know which is too deep - set them both to 0.5 levels below subtree A



Left-Right Imbalance

- Neither X nor Y worked as root node so make Z the root
- Rearrange the subtrees in the correct order
- No matter how deep B1 or B_2 (+/- 0.5 levels) we get a legal AVL tree again



Left-Right Rebalance