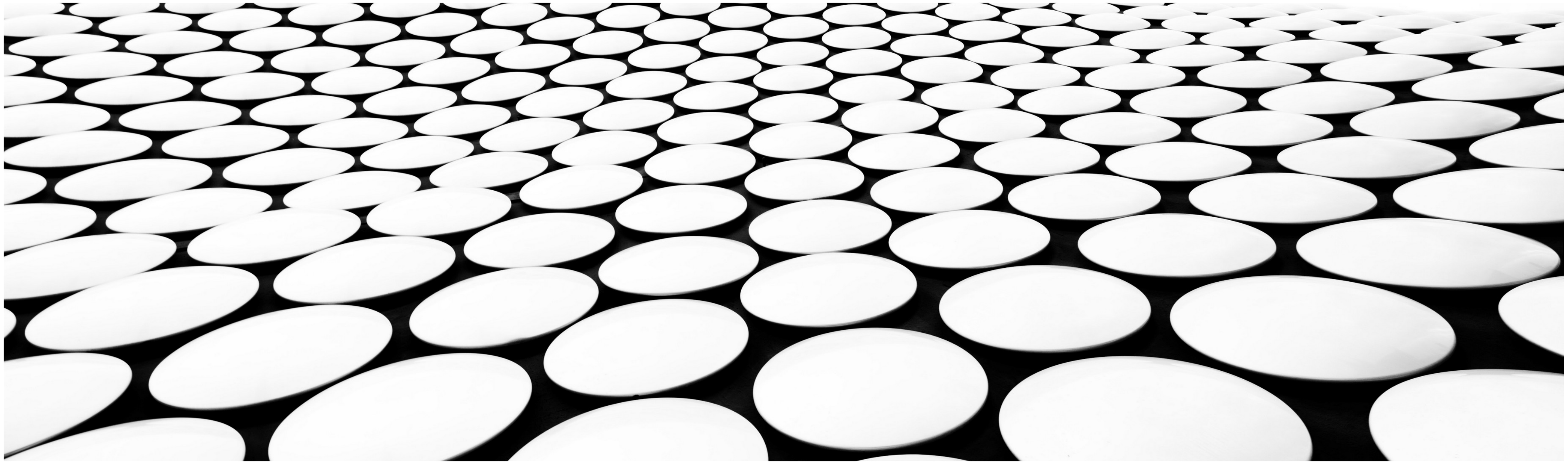
# DSA 456 – WEEK 12

AUGMENTED DATA STRUCTURES



## AUGMENTED DATA STRUCTURES

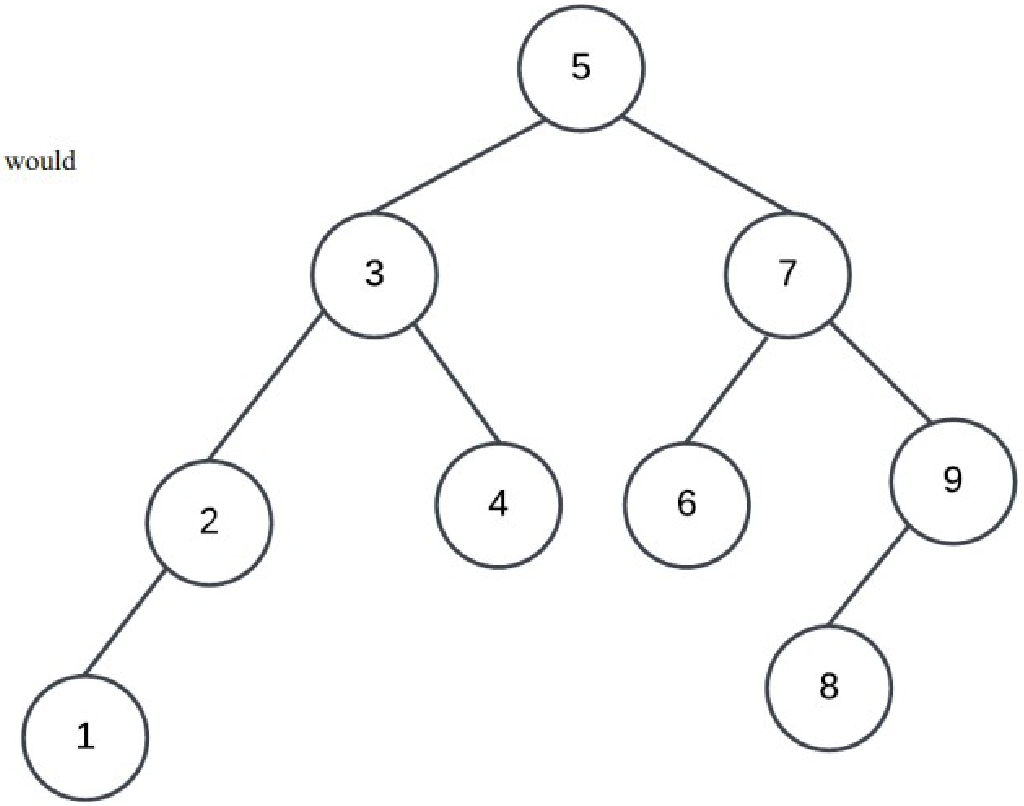
An augmented data structure is a data structure where we add an extra piece of information so that we can acheive better performance. From a user's perspective the augmented data structure works no differently than the non-augmented version.

We will be looking at two augmented data structures in this section and they both help to solve a problem with the basic version of the BST.

Consider the following set of numbers:

5, 3, 7, 2, 9, 4, 6, 1, 8

If we were to insert each of these values in order into a BST using the algorithm we currently have studied, our tree would look like this:

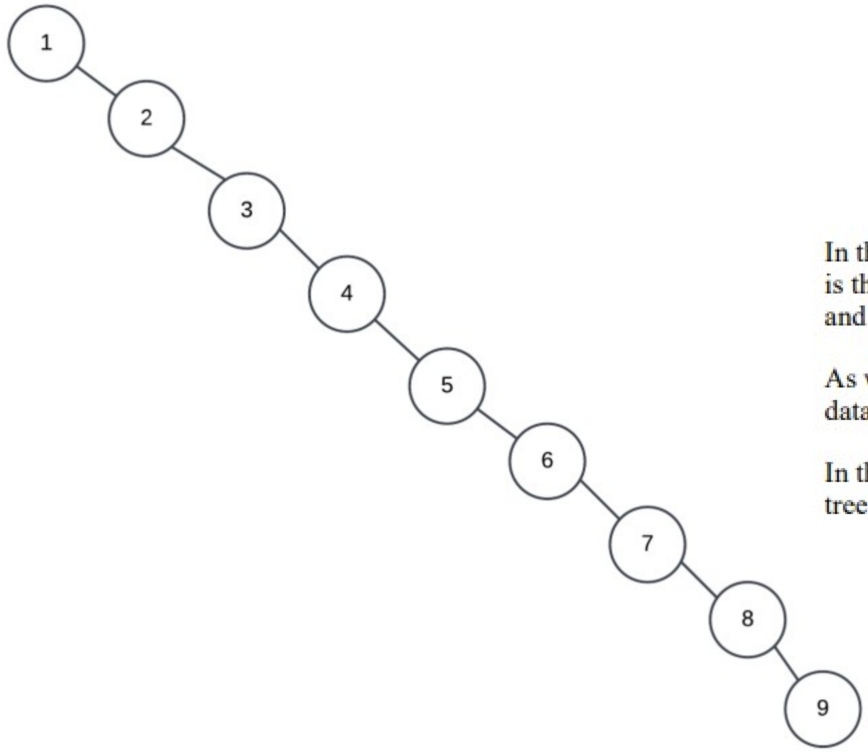


However, suppose we were to created a BST with exactly the same numbers but in the following order instead:

1, 2, 3, 4, 5, 6, 7, 8, 9

## AUGMENTED DATA STRUCTURES

The tree created by the insertion algorithm would look like this:

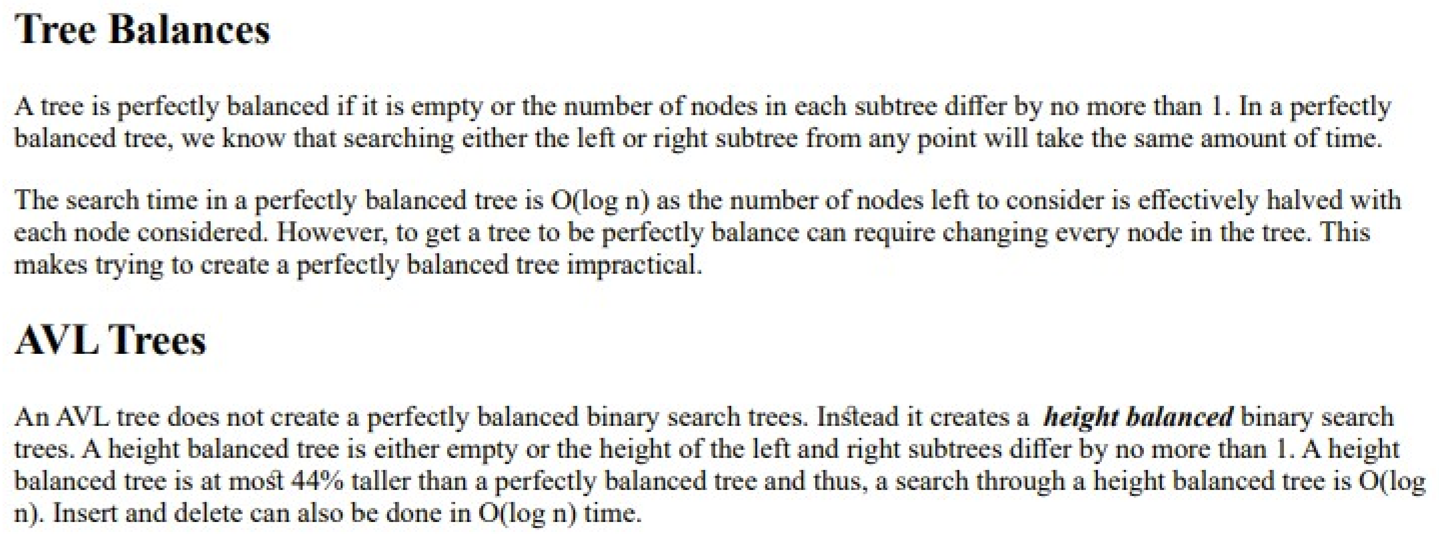


In the first case, the tree looks like tree. In the second case it looks like a stick. The problem of a tree that looks like a stick is that its run time for searching is not log n. its basically a linked list and its performance will match that of a linked list and thus search will perform in O(n) time.

As we cannot know how our data structure will be used, or prevent someone from using our data structure with sorted data, it would be good if we can ensure that no matter how the data comes in, the tree will be tree shaped.

In this chapter we will look at how we can add a little bit of extra information to our tree and use that to help ensure our tree does not turn into a stick no matter how we receive our data.

## AUGMENTED DATA STRUCTURES



### Tree Balances

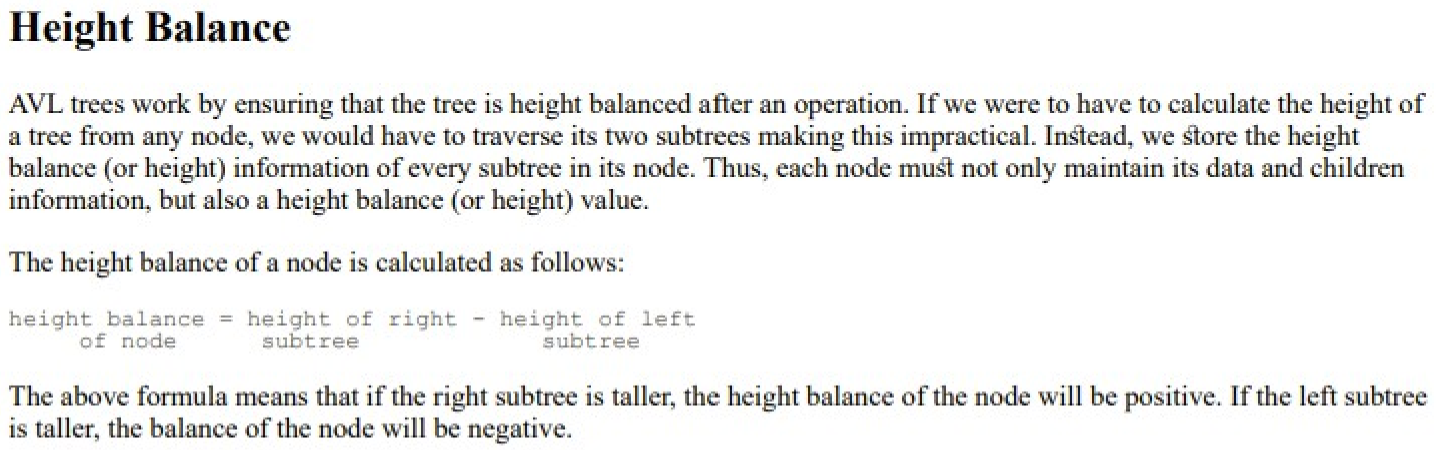
A tree is perfectly balanced if it is empty or the number of nodes in each subtree differ by no more than 1. In a perfectly balanced tree, we know that searching either the left or right subtree from any point will take the same amount of time.

The search time in a perfectly balanced tree is O(log n) as the number of nodes left to consider is effectively halved with each node considered. However, to get a tree to be perfectly balance can require changing every node in the tree. This makes trying to create a perfectly balanced tree impractical.

### AVL Trees

An AVL tree does not create a perfectly balanced binary search trees. Instead it creates a ***height balanced*** binary search trees. A height balanced tree is either empty or the height of the left and right subtrees differ by no more than 1. A height balanced tree is at most 44% taller than a perfectly balanced tree and thus, a search through a height balanced tree is O(log n). Insert and delete can also be done in O(log n) time.

## AUGMENTED DATA STRUCTURES



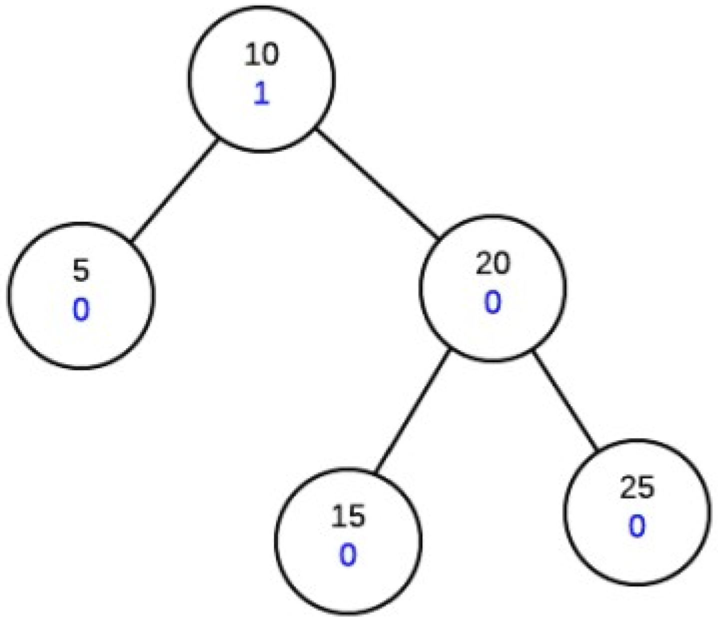
### Height Balance

AVL trees work by ensuring that the tree is height balanced after an operation. If we were to have to calculate the height of a tree from any node, we would have to traverse its two subtrees making this impractical. Instead, we store the height balance (or height) information of every subtree in its node. Thus, each node must not only maintain its data and children information, but also a height balance (or height) value.

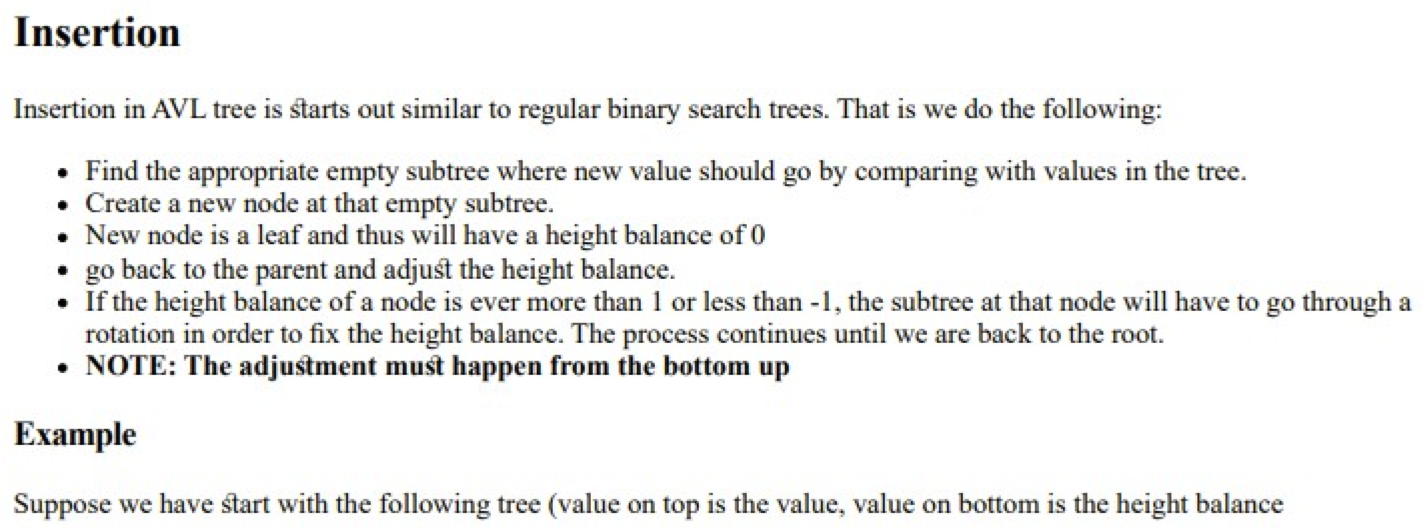
The height balance of a node is calculated as follows:

```
height balance = height of right - height of left
     of node       subtree             subtree
```

The above formula means that if the right subtree is taller, the height balance of the node will be positive. If the left subtree is taller, the balance of the node will be negative.



# AUGMENTED DATA STRUCTURES



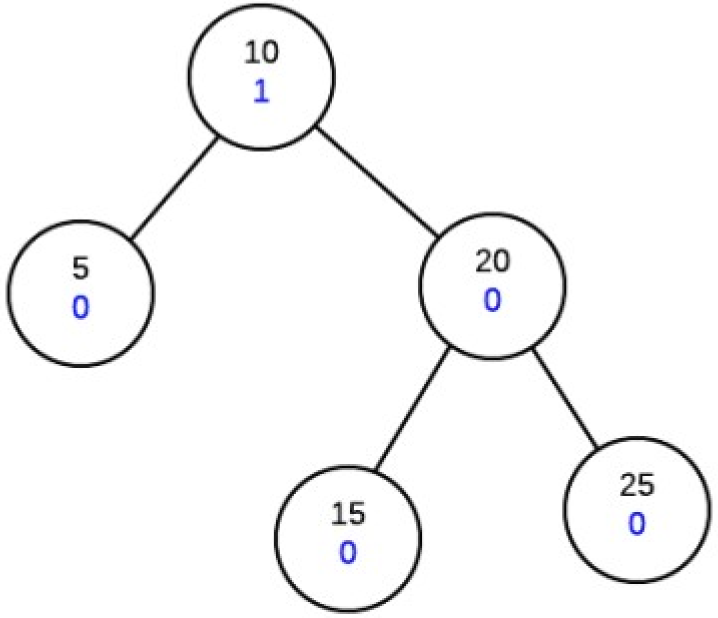
## Insertion

Insertion in AVL tree is starts out similar to regular binary search trees. That is we do the following:

- Find the appropriate empty subtree where new value should go by comparing with values in the tree.
- Create a new node at that empty subtree.
- New node is a leaf and thus will have a height balance of 0
- go back to the parent and adjust the height balance.
- If the height balance of a node is ever more than 1 or less than -1, the subtree at that node will have to go through a rotation in order to fix the height balance. The process continues until we are back to the root.
- **NOTE: The adjustment must happen from the bottom up**

## Example

Suppose we have start with the following tree (value on top is the value, value on bottom is the height balance

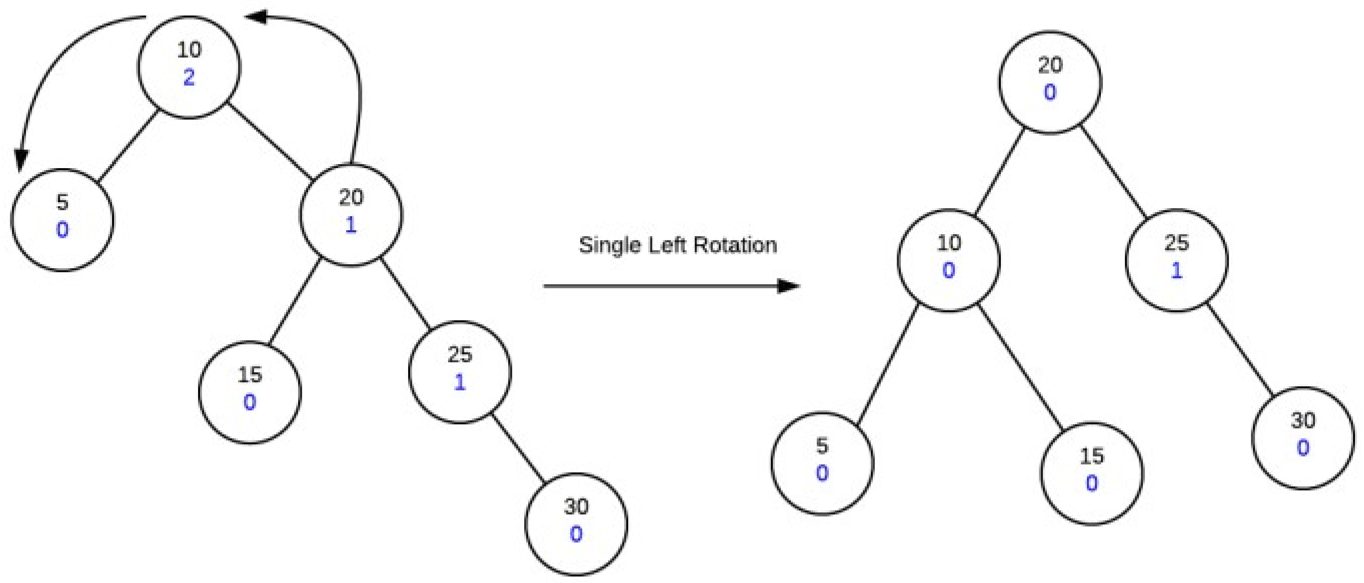


## AUGMENTED DATA STRUCTURES

**Insert 30**

At this point, we have adjusted all the height balances along the insertion path and we note that the root node has a height balance of 2 which means the tree is not height balanced at the root.

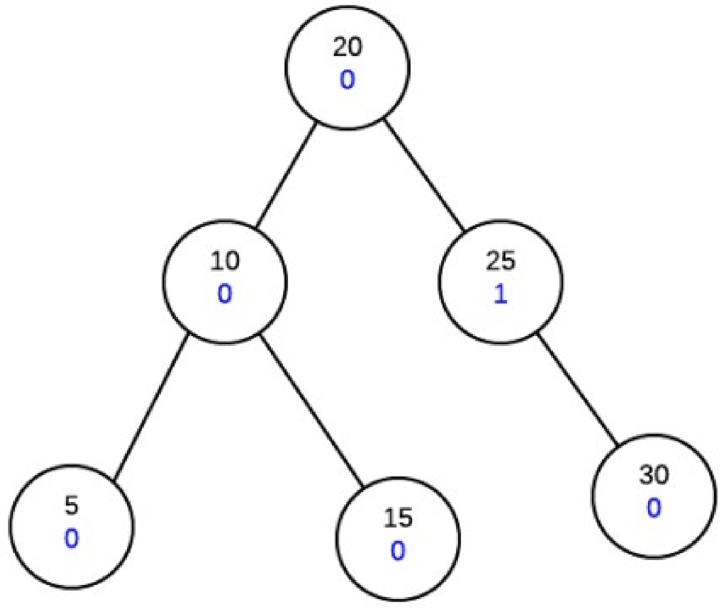
In order to fix our tree, we will need to perform a rotation



# AUGMENTED DATA STRUCTURES

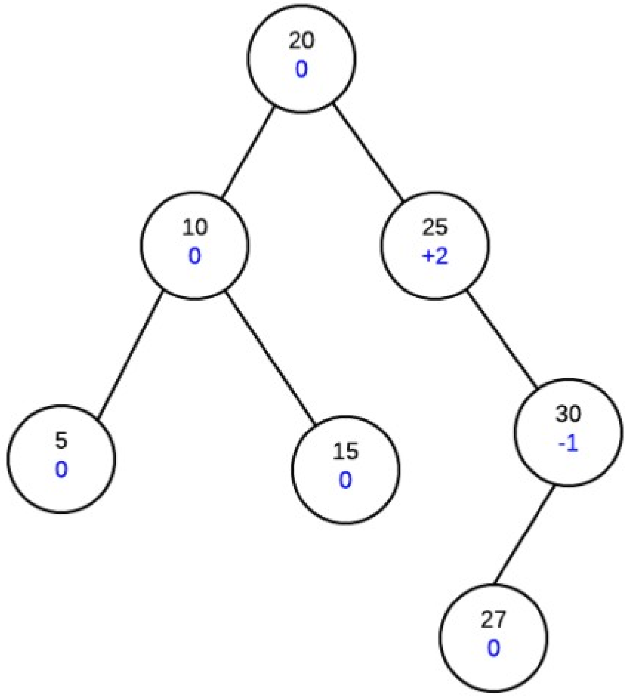
**Insert 27**

We start with our tree:



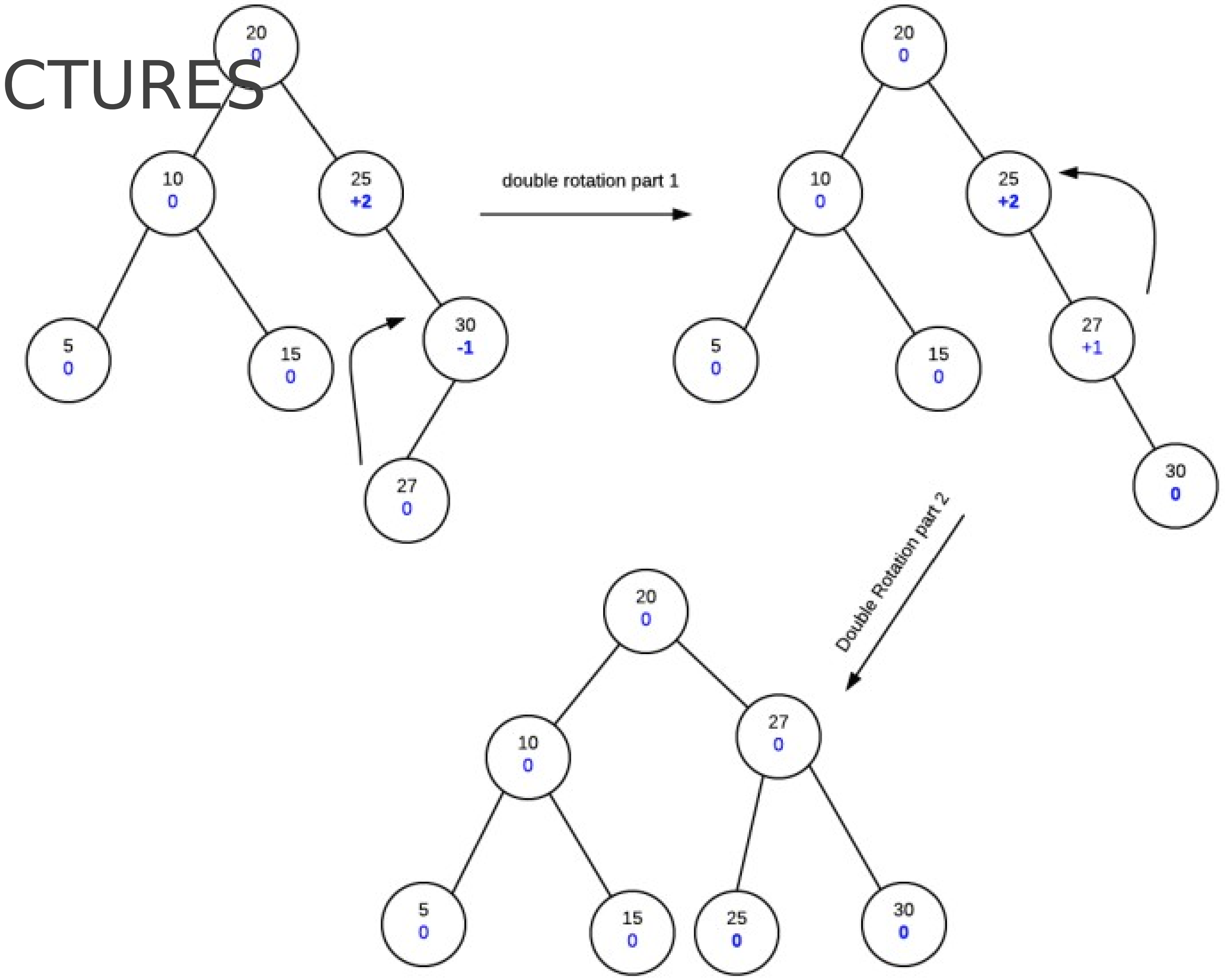
Now we find the correct place in the tree and insert the new node and fix the height balances

## AUGMENTED DATA STRUCTURES



Now, as we reach node 25 and see that it the height balance is +2. If we then look at it's child's height balance we find that it is -1. As the signs are different, it indicates that we need a double rotation. Different signs indicate that the unbalance is in different directions so we need to do a rotation to make it the same direction then another to fix the unbalance.

## AUGMENTED DATA STRUCTURES



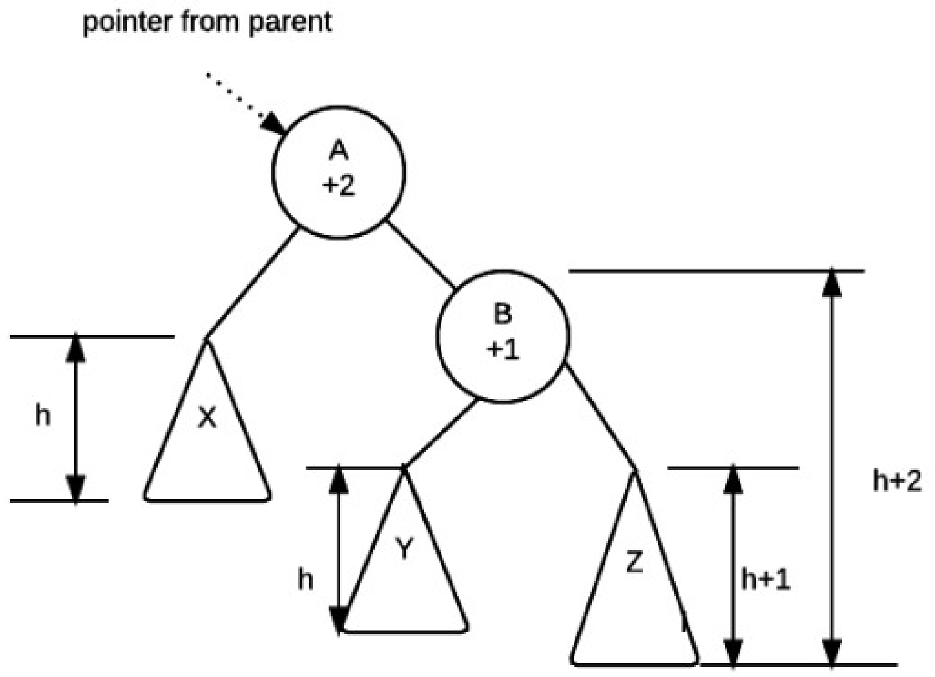
# AUGMENTED DATA STRUCTURES

## Why does this work?

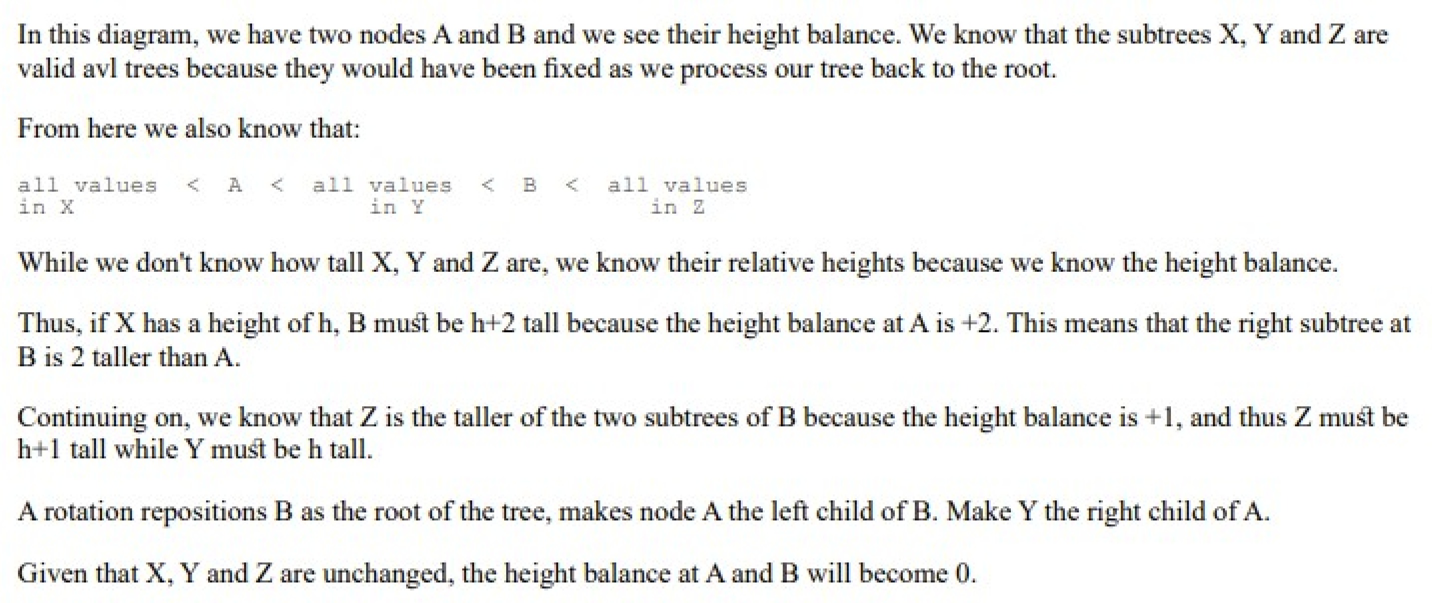
### Single Rotation

We always fix nodes starting from the insertion point back to the root. Thus, any node in the insertion path further towards leaf nodes must already be fixed.

Consider the following idea of what an avl tree looks like:



## AUGMENTED DATA STRUCTURES



In this diagram, we have two nodes A and B and we see their height balance. We know that the subtrees X, Y and Z are valid avl trees because they would have been fixed as we process our tree back to the root.

From here we also know that:

```
all values  <  A  <  all values  <  B  <  all values
in X                 in Y                  in Z
```

While we don't know how tall X, Y and Z are, we know their relative heights because we know the height balance.

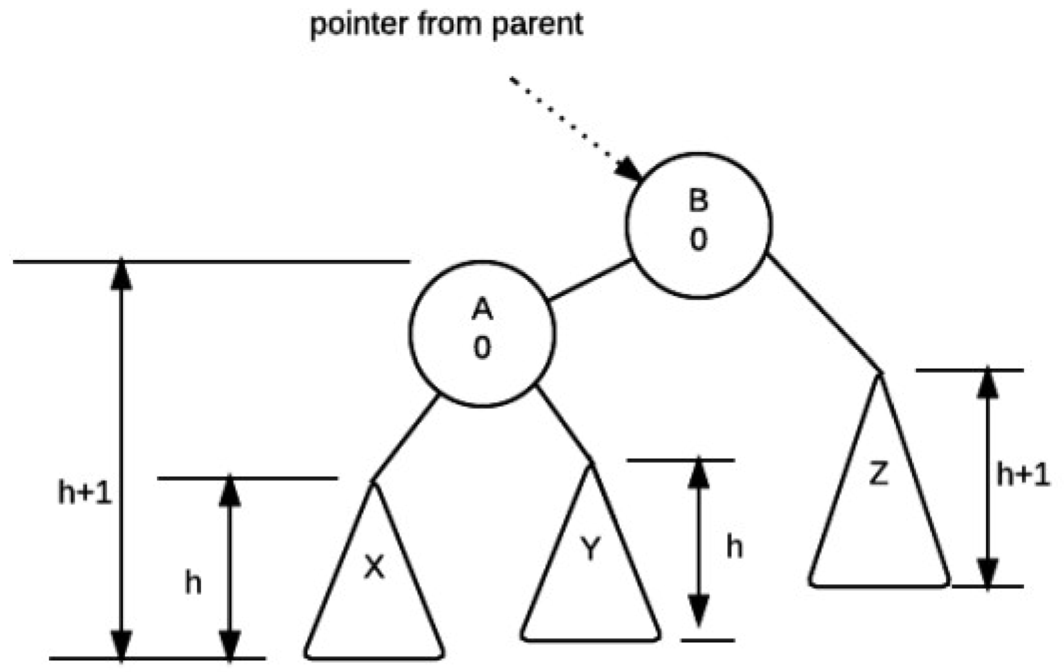
Thus, if X has a height of h, B must be h+2 tall because the height balance at A is +2. This means that the right subtree at B is 2 taller than A.

Continuing on, we know that Z is the taller of the two subtrees of B because the height balance is +1, and thus Z must be h+1 tall while Y must be h tall.

A rotation repositions B as the root of the tree, makes node A the left child of B. Make Y the right child of A.

Given that X, Y and Z are unchanged, the height balance at A and B will become 0.

## AUGMENTED DATA STRUCTURES

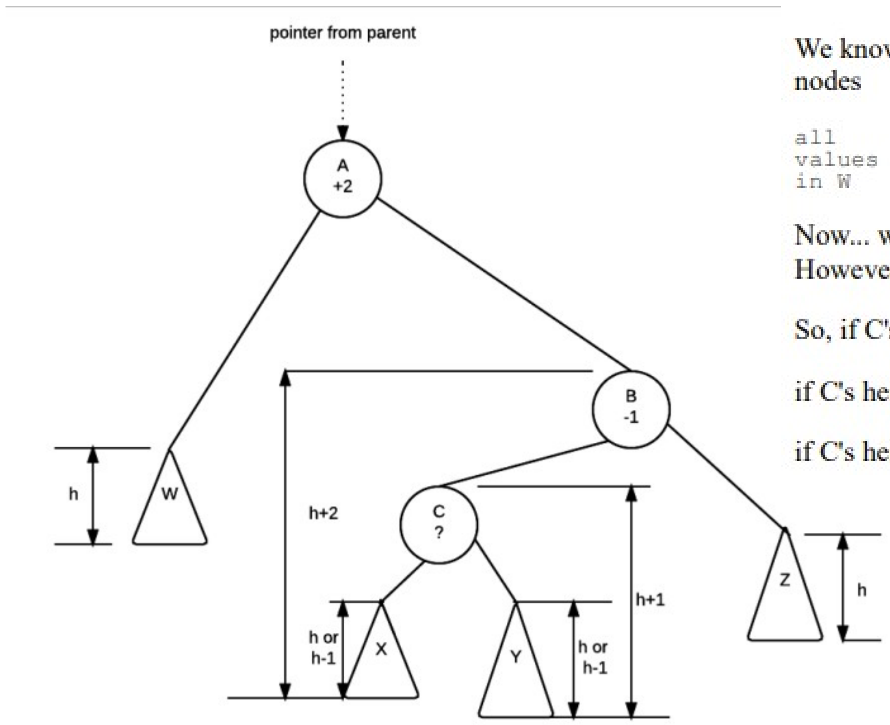


The mirror of the above is true for single left rotation

# AUGMENTED DATA STRUCTURES

**Double Rotations**

A similar explanation of why double rotations work can be reasoned out by looking at the following tree:



We know that W, X, Y and Z are all valid avl trees. We also know the following about the values in each of the trees and nodes

```
all                all                  all           all
values  <  A  <  values  <  C  <  values < B <  values
in W                in X                 in Y           in Z
```

Now... we know the height balance of A and B (off balance node and child) we do not know the exact height balance of C. However, we do know that it is a valid avl tree, so C's height balance must be either -1, 0 or +1.

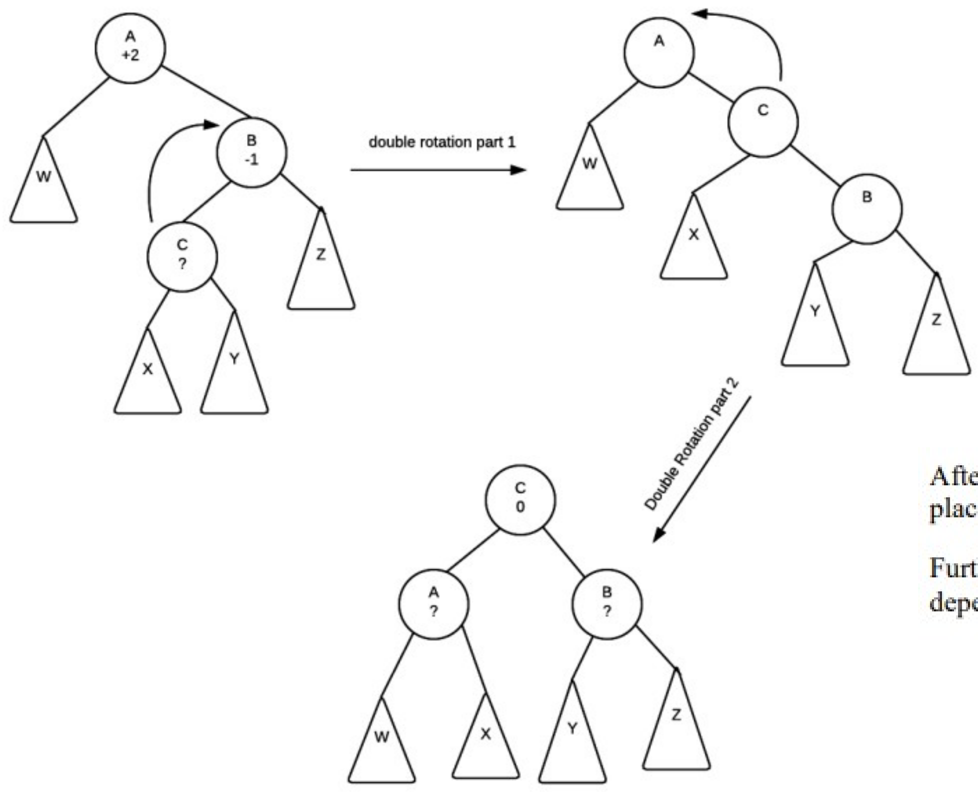
So, if C's height balance is 0, then both x and y will have height of h.

if C's height balance is +1 then y will be h and x would be h-1

if C's height balance is -1 then x would be h and y would h-1

## AUGMENTED DATA STRUCTURES

Perform a double rotation:

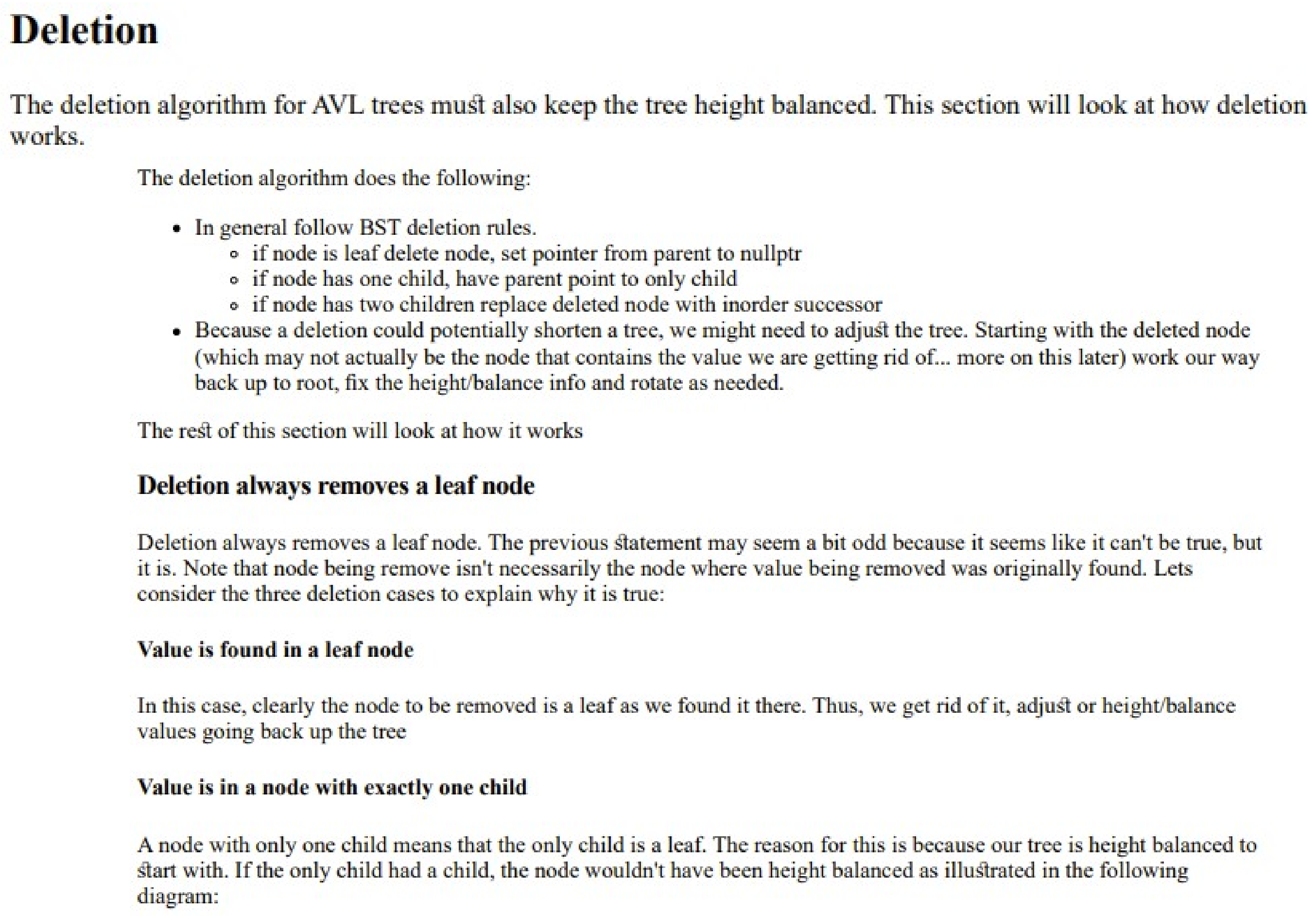


After the rotation is completed, notice that the position of the subtrees W,X, Y and Z along with the nodes A, B and C are placed in a way where their ordering is properly maintained.

Furthermore, The height balance of C becomes 0 regardless of what it was initially. The final height balance of A and B depends on what the original height balance of C was:

| original height balance of C | height of X | height of Y | final height balance of A | final height balance of B |
|---|---|---|---|---|
| -1 | h | h-1 | 0 | +1 |
| 0 | h | h | 0 | 0 |
| +1 | h-1 | h | -1 | 0 |

## AUGMENTED DATA STRUCTURES



### Deletion

The deletion algorithm for AVL trees must also keep the tree height balanced. This section will look at how deletion works.

The deletion algorithm does the following:

- In general follow BST deletion rules.
  - if node is leaf delete node, set pointer from parent to nullptr
  - if node has one child, have parent point to only child
  - if node has two children replace deleted node with inorder successor
- Because a deletion could potentially shorten a tree, we might need to adjust the tree. Starting with the deleted node (which may not actually be the node that contains the value we are getting rid of... more on this later) work our way back up to root, fix the height/balance info and rotate as needed.

The rest of this section will look at how it works

#### Deletion always removes a leaf node

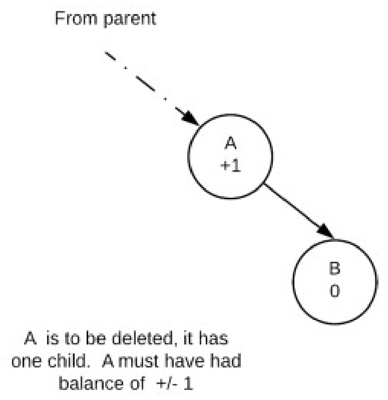
Deletion always removes a leaf node. The previous statement may seem a bit odd because it seems like it can't be true, but it is. Note that node being remove isn't necessarily the node where value being removed was originally found. Lets consider the three deletion cases to explain why it is true:

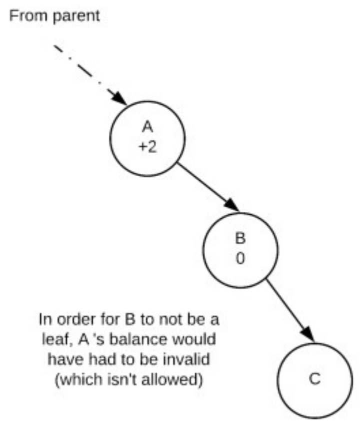
**Value is found in a leaf node**

In this case, clearly the node to be removed is a leaf as we found it there. Thus, we get rid of it, adjust or height/balance values going back up the tree

**Value is in a node with exactly one child**

A node with only one child means that the only child is a leaf. The reason for this is because our tree is height balanced to start with. If the only child had a child, the node wouldn't have been height balanced as illustrated in the following diagram:

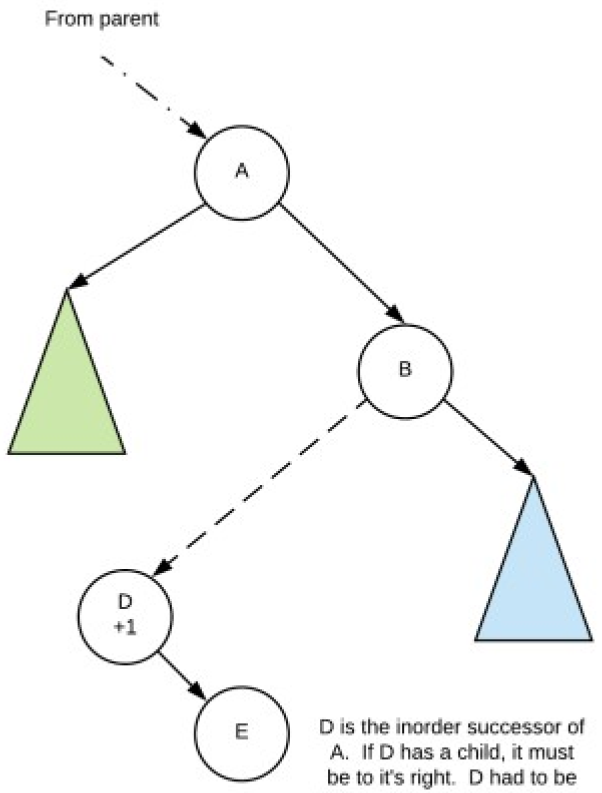

From parent
A
+1
B
0
A is to be deleted, it has one child. A must have had balance of +/- 1


From parent
A
+2
B
0
C
In order for B to not be a leaf, A 's balance would have had to be invalid (which isn't allowed)

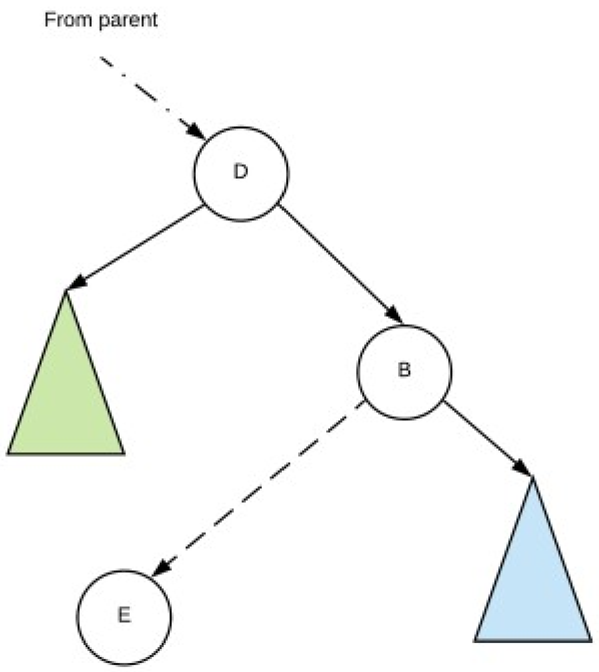
Thus, what effectively happens is that we end up with a tree that is shaped like we were removing B

**Value is in a node with exactly two children**

The algorithm to remove a node with two children is to replace the node with its inorder successor node. This node is found by going one node to the right then going left until you can't. The inorder successor itself can only have a right child (if it had a left child it wouldn't be the inorder successor). As with the case of nodes with only one child, the inorder sucessor's right child must be a leaf because the inorder successor must have been height balanced before the deletion operation.



D is the inorder successor of A. If D has a child, it must be to it's right. D had to be height balance to begin with thus E must be leaf (same reason why nodes with one child had to be a leaf



Deletion of A, involves promoting D to take the place of A and having E take D's old spot Effectively the node being removed is the one where E was

## Fixing the tree

When will we need to fix the tree?

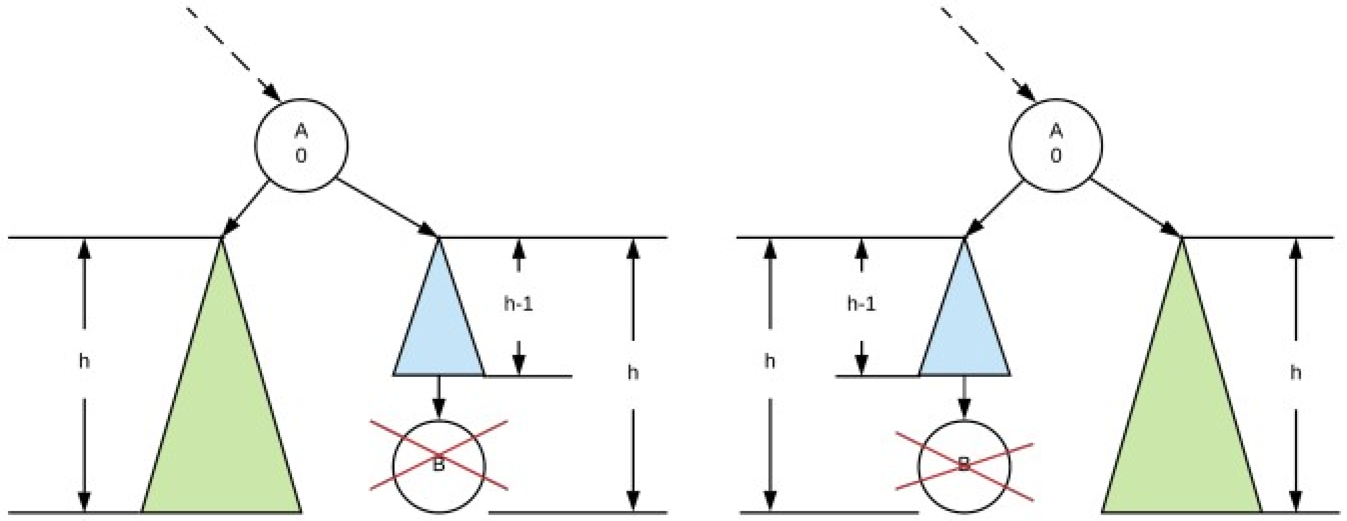
Clearly if the deletion doesn't shorten its subtree, there is nothing that needs to be done.

Thus, we only need to consider what happens if deleting the node causes the tree to shorten. In the examples below we only consider those situations.

Let us consider the following. In each case, B is the node being removed, A is some node along the path to B

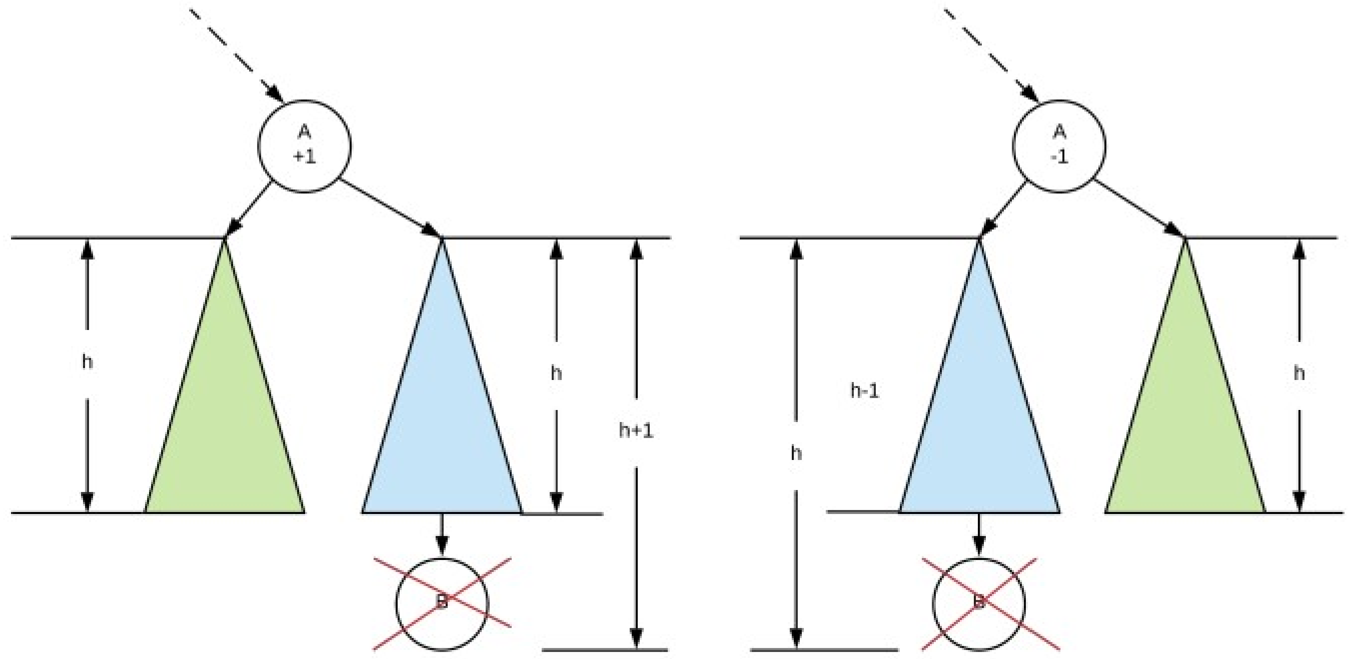
**Balance of A was initially 0**

Suppose that A initially had a balance of 0. Removing B, shortened the blue subtree. However, as the balance was 0 at A, A's balance will simply go either to +/- 1. No rotation needed. Also not only was no rotation needed, the height of the tree at A is actually exactly the same as it was before the deletion and thus, we can stop fixing the tree as nodes further up will not be affected



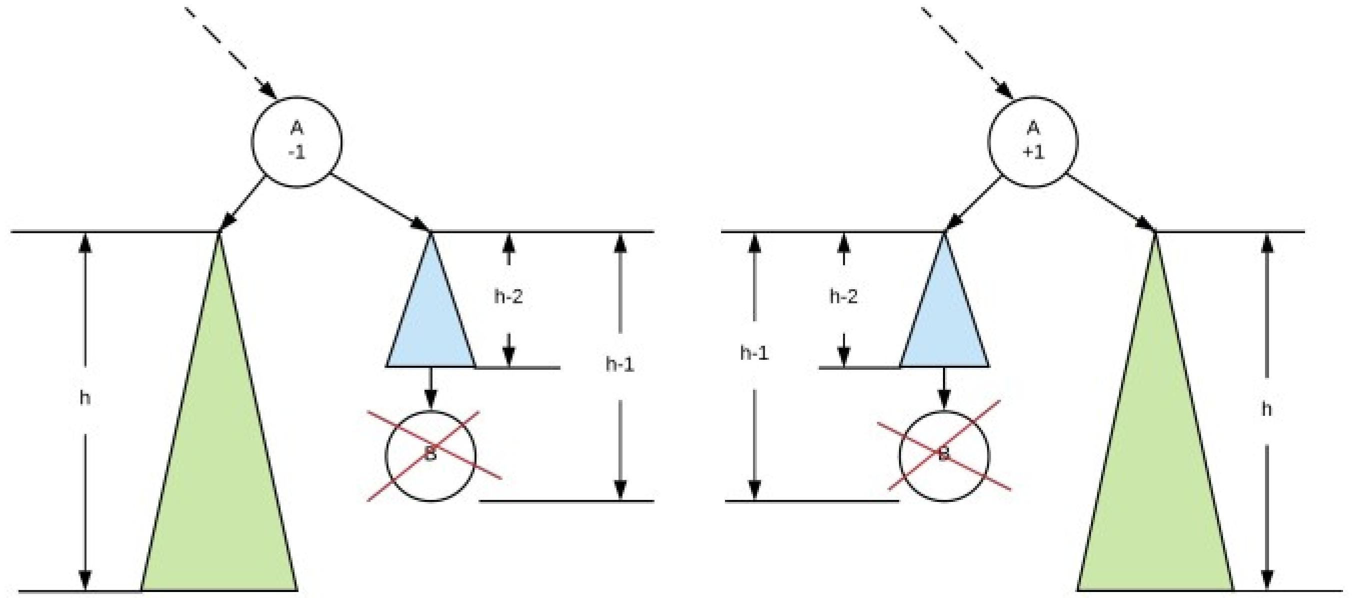
**Balance of A was initially +/-1**

Two things can occur. In the first case our node comes from the taller of the subtrees tree and it causes the tree to shorten.



If that is the case then we do not have to do any rotations at A because A's balance will become 0. However, the subtree with root A did get shorter, so we must continue going up the tree as other nodes further up may require a rotation

The second case is when we take out a node from the shorter subtree:

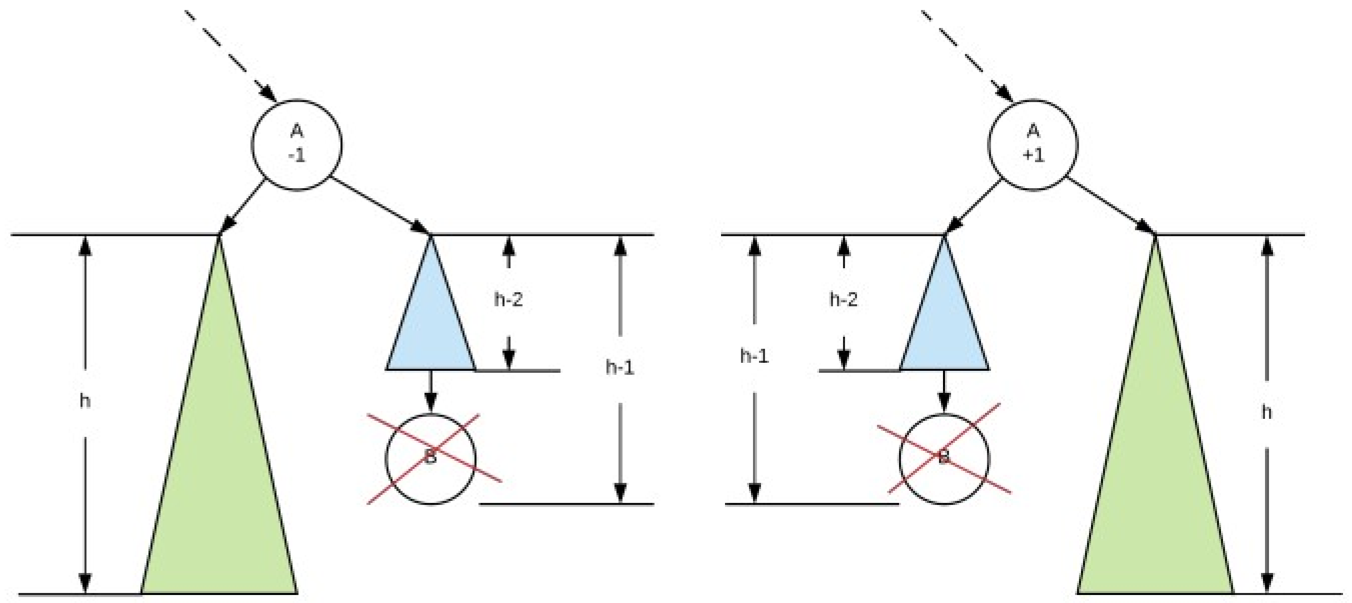


If we did this then our balance will go to +/- 2. This would require rebalancing. Like an insertion, rebalancing is simply a matter of applying a single or double rotation. We simply need to follow the same rules and perform the rotation

After a rebalancing a node the subtree might have become shorter than it was because of the rotation, and thus we may need to go further up the tree to fix it.

If that is the case then we do not have to do any rotations at A because A's balance will become 0. However, the subtree with root A did get shorter, so we must continue going up the tree as other nodes further up may require a rotation

The second case is when we take out a node from the shorter subtree:



If we did this then our balance will go to +/- 2. This would require rebalancing. Like an insertion, rebalancing is simply a matter of applying a single or double rotation. We simply need to follow the same rules and perform the rotation

After a rebalancing a node the subtree might have become shorter than it was because of the rotation, and thus we may need to go further up the tree to fix it.

## Red Black Trees

Red-Black Trees are binary search trees that are named after the way the nodes are coloured.

Each node in a red-black tree is coloured either red or black. The height of a red black tree is at most 2 * log(n+1).

A red black tree must maintain the following colouring rules:

1. every node must have a colour either red or black.
2. The root node must be black
3. If a node is red, its children must be black (note that this also implies that red nodes cannot have red parents)
4. Every path from root to a ***null node*** must have exactly the same number of black nodes.

INFO

***null nodes*** are also sometimes called ***null leafs***. these are not really nodes.. they are the "nodes" that null pointers point to... so we when we think about counting black nodes we think about how many black nodes there are to every nullptr in the tree

## Insertion

We will begin our look at Red-Black trees with the bottom up insertion algorithm. This insertion algorithm is similar to that of the insertion algorithm we looked at for AVL trees/Binary search trees.

Insert the new node according to regular binary search tree insertion rules. Of the 4 colouring rules, the one rule we don't want to break is rule number 4. Everything we do is to avoid breaking rule 4 (every path from root to every null leaf has same number of black nodes). Thus, New nodes are added as red nodes. We then "fix" the tree if any of the rules are broken.

Note that because we inserted a red node to a proper red-black tree, the only 2 rules that might be broken are:

1. rule 2: root must be black
2. rule 3: red node must have black children

Rule 1 is pretty much not broken as we coloured it red. We also won't break rule 4 because we added a red node so the number of black nodes has not increased.

### General Insertion Algorithm

To insert into a red-black tree:

1. find the correct empty tree (like bst) and insert new node as a red node.
2. working way up the tree back to parent fix the tree so that the red-black tree rules are maintained.

## Fixing nodes:

- If root becomes red, change it to black.
  - This won't break any rules because you are just adding 1 black node to every branch of the tree, the number of black nodes increase by 1 everywhere. This can only happen as the root as it is the only node that is part of every path from root to nullleaf
- If there are two red nodes in a row:
  - Identify the following nodes:
    - upper red node as the Parent (**P**)
    - the lower red node as the Child (**C**)
    - parent of parent is Grandparent (**G**)
    - sibling of Parent as Parent's sibling (**PS**)
  - if the **PS** is black
    - perform a rotation (look at G->P->C, if they form a straight line do a zig-zig(single) rotation, if there is a bend, do a zig-zag (double rotation)
    - after rotation exchange G's colour with the node that took over G's spot. In otherwords
      - make which ever node (depends if it was zigzig or zigzag rotation... it will either be **P** or **C**) that took over G's node black
      - make **G** red
  - if the **PS** is red
    - exchange the colour of the grand parent with its two children. In otherwords
      - **G** becomes red
      - **P** and **PS** becomes black

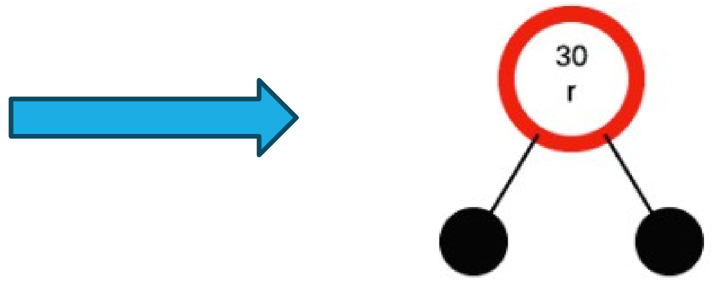
## Example

Starting with an empty tree let us take a look at how red-black tree insertions work.

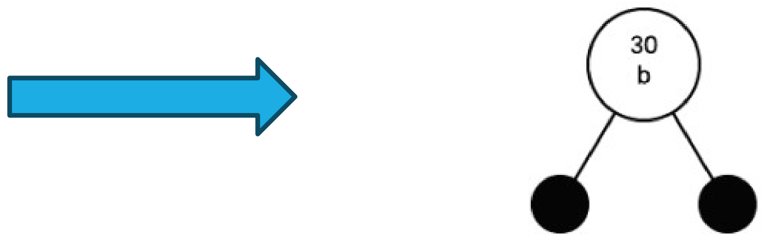
In the pictures below, null nodes (empty subtrees) are denoted as black circles

**Insert 30**

All nodes are inserted as red nodes:

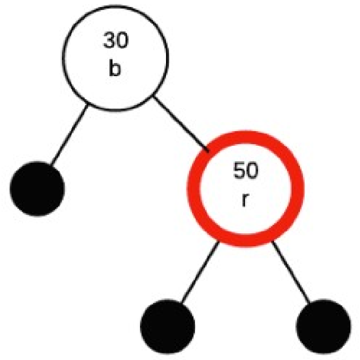


If the root is red, make it black:



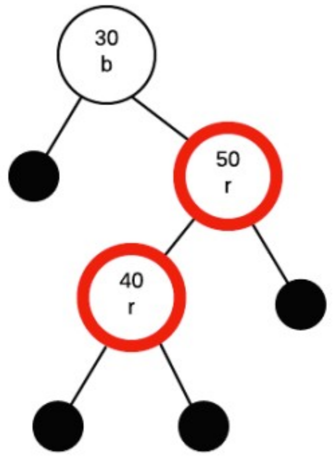
**Insert 50**

Insert 50 as a red node, parent is black so we don't have to change anything



**Insert 40**

Inserting 40 as a red no

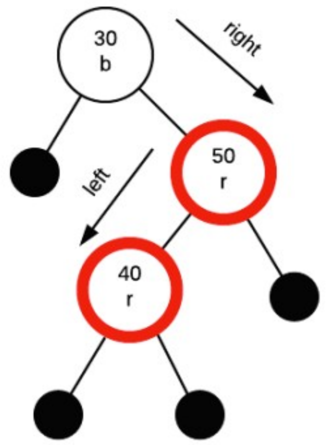


Two red nodes in a row. Identify G, P, PS and C

- **P** - parent (upper red) - 50
- **C** - child (lower red) - 40
- **G** - grandparent 30
- **PS** - parent's sibling - null node to left of 30

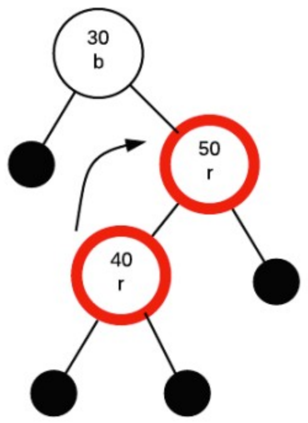
What we do depends on colour of **PS.** In this case PS is black, so we will be fixing this with a rotation

The type of rotation depends on the configuration of G, P and C. If the path is from G to C is straight (both left or both right) do a zigzig (single) rotation. If it is angled (left then right or right then left) we need to do a zigzag (double) rotation.

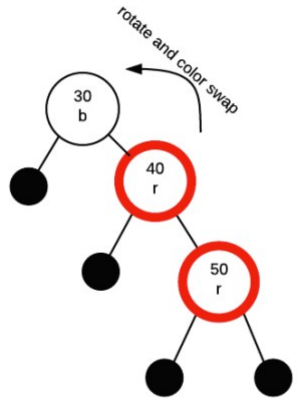


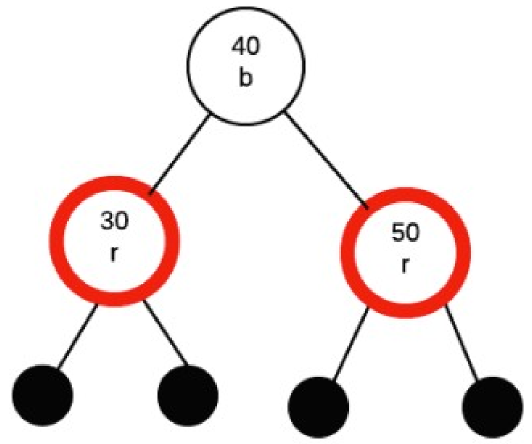
In this case, we need to do a zig zag (double) rotation.

Rotate first 40 and 50



then rotate again with 30 and 40, this time doing a colour swap. A zigzag rotation is just an extra step that is needed to make the insertion path go in the same direction.

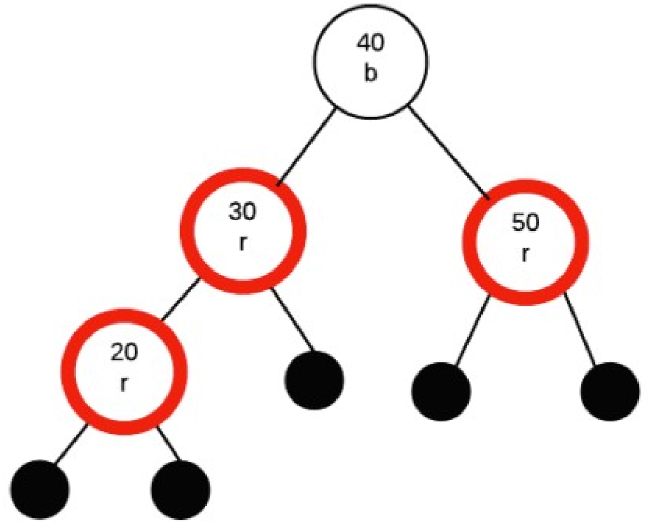




After rotations are complete, we exchange the colour between the node that took over G's spot (40 in this case) and G. Thus, 40 becomes black and 30 becomes red.

**Insert 20**

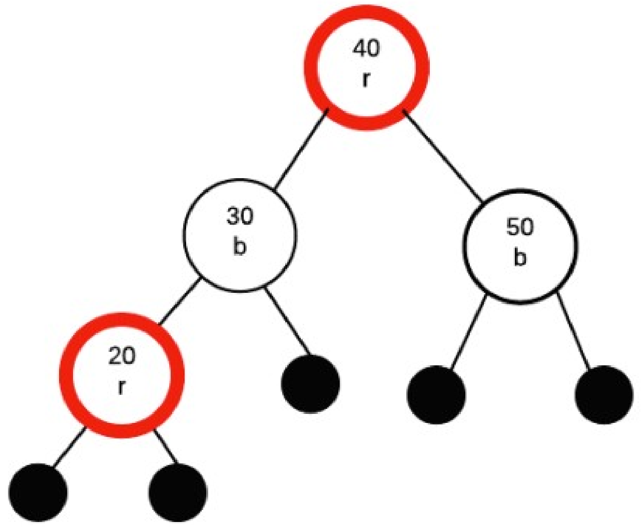
inserting 20 as a red node.



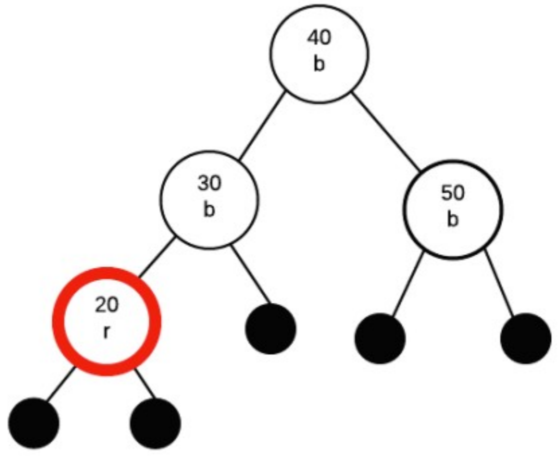
Two red nodes in a row. Identify G, P, PS and C

- **P** - parent (upper red) - 30
- **C** - child (lower red) - 20
- **G** - grandparent - 40
- **PS** - parent's sibling - 50

What we do depends on colour of **PS.** In this case PS is red. Thus we exchange colours between G and its two children:

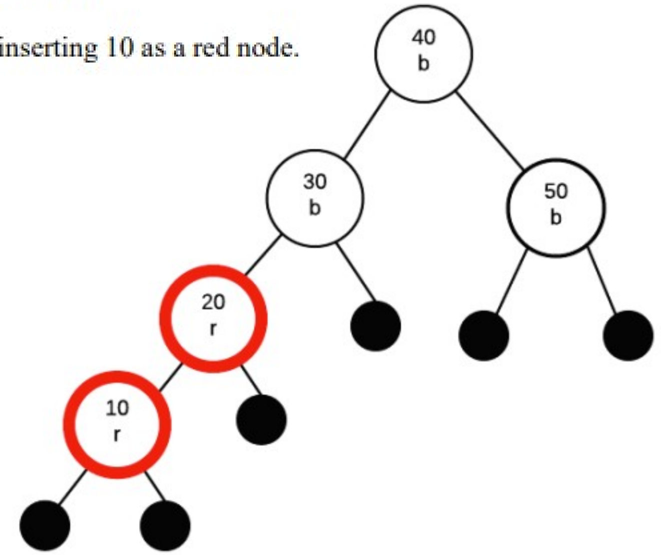


Doing so breaks rule 2: roots must be black. Thus, we need to fix that. As it is the root, we can just change it to black without causing other problems.



**Insert 10**

inserting 10 as a red node.

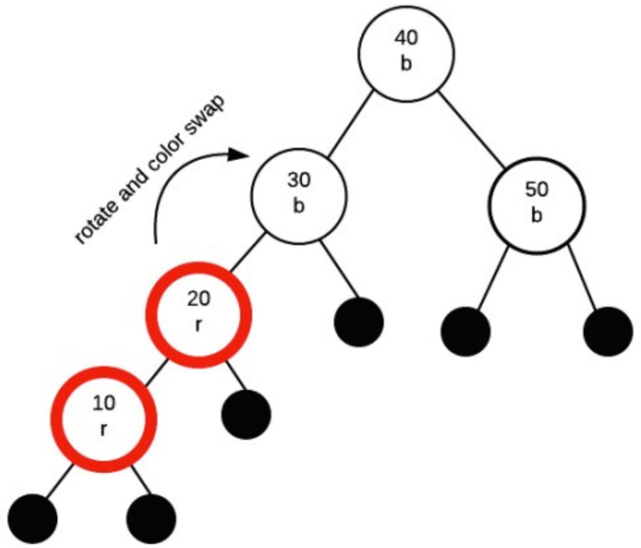


Two red nodes in a row. Identify G, P, PS and C

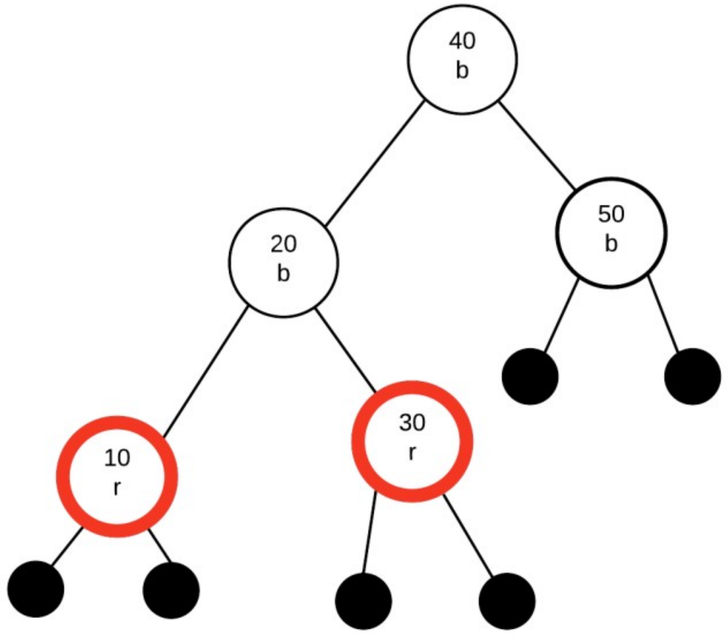
- **P** - parent (upper red) -20
- **C** - child (lower red) - 10
- **G** - grandparent 30
- **PS** - parent's sibling - null node to right of 30

What we do depends on colour of **PS.** In this case, the parent's sibling is black (null nodes are black). Thus, we will fix this with a rotation. Rotations are always done with G as the root of the rotation (the A in the rotation diagram)

This time, the path from G to P to C is "left" then "left". Thus, we only need to perform a single rotation, followed by swapping the colours of G and the node that took G's spot.



Finally we get:



## 2-3 Trees

A 2-3 Tree is a specific form of a B tree. A 2-3 tree is a search tree. However, it is very different from a binary search tree.

Here are the properties of a 2-3 tree:

1. each node has either one value or two value
2. a node with one value is either a leaf node or has exactly two children (non-null). Values in left subtree < value in node < values in right subtree
3. a node with two values is either a leaf node or has exactly three children (non-null). Values in left subtree < first value in node < values in middle subtree < second value in node < value in right subtree.
4. all leaf nodes are at the same level of the tree

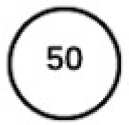
## Insertion

The insertion algorithm into a two-three tree is quite different from the insertion algorithm into a binary search tree. In a two-three tree, the algorithm will be as follows:

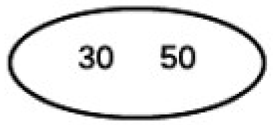
1. If the tree is empty, create a node and put value into the node
2. Otherwise find the leaf node where the value belongs.
3. If the leaf node has only one value, put the new value into the node
4. If the leaf node has more than two values, split the node and promote the median of the three values to parent.
5. If the parent then has three values, continue to split and promote, forming a new root node if necessary

**Example:**

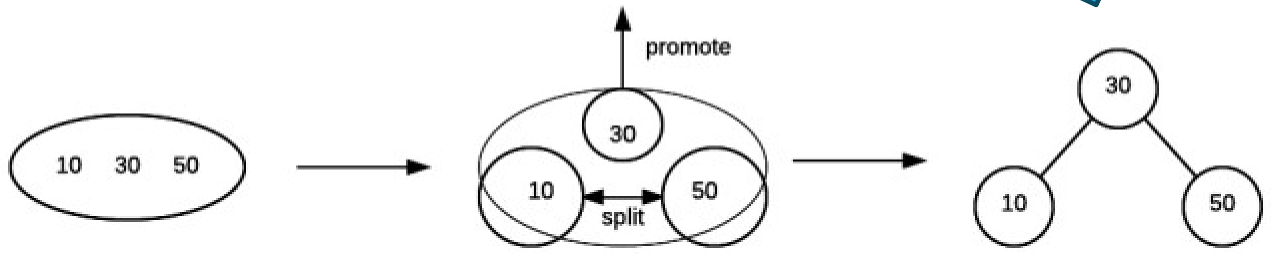
Insert 50



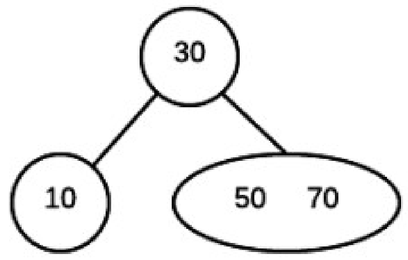
Insert 30



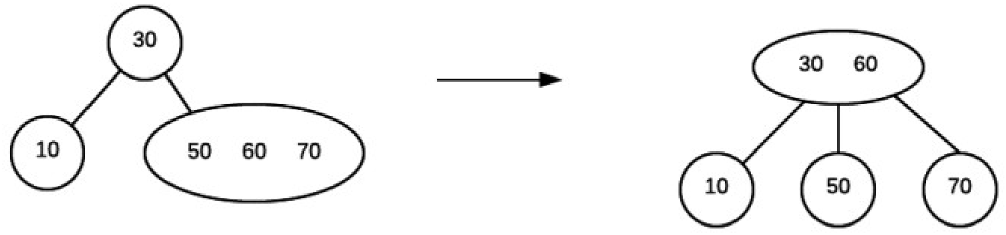
Insert 10



Insert 70



Insert 60



## END OF WEEK 12

- Please complete your assignment 3