# Theory Lecture 6

Algorithms for Trees

## Learning Objectives

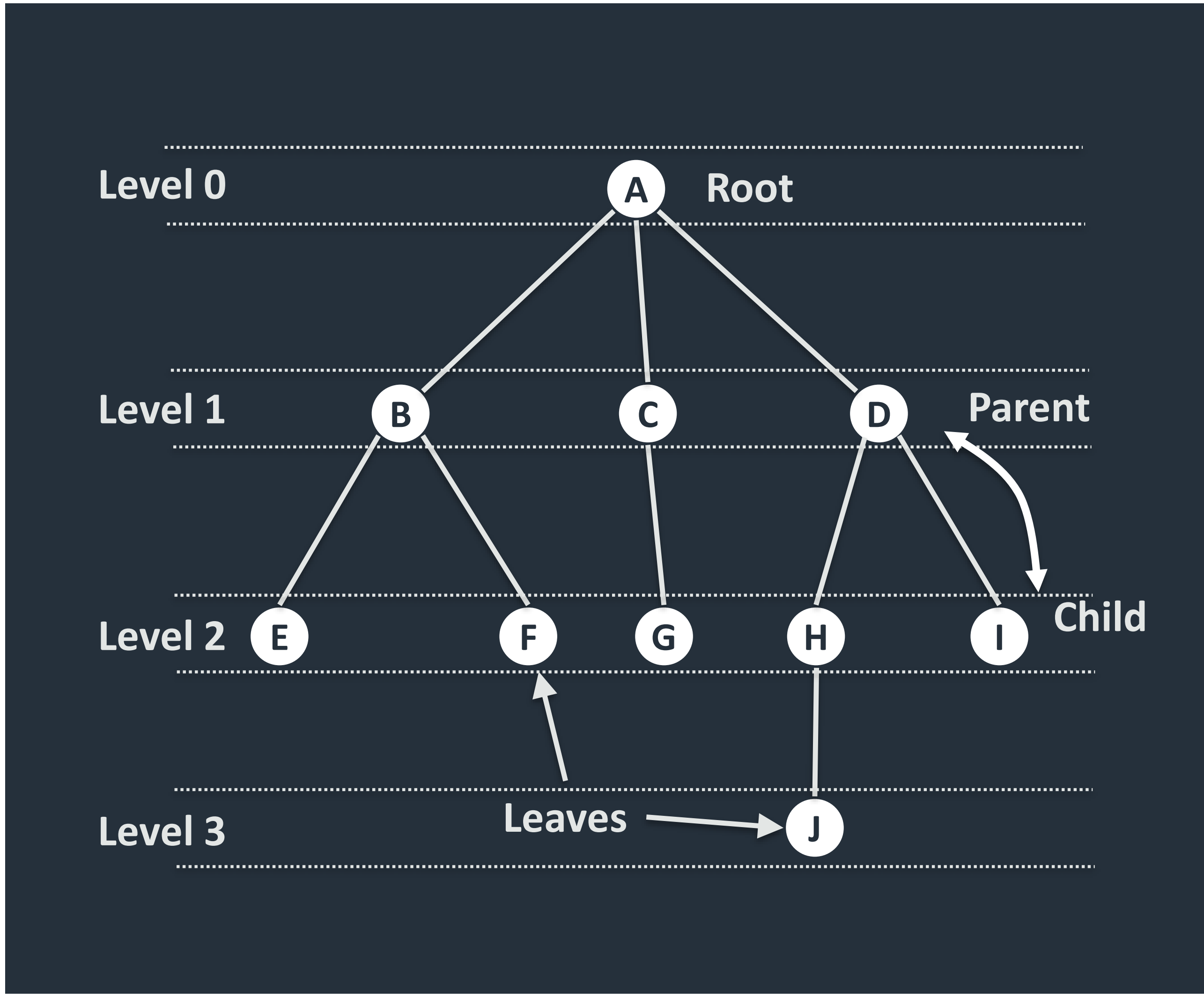
- Look at some algorithms for trees in depth
- Discuss the different tree traversal orders
- Understand binary search trees
- Study the formation of balanced trees and AVL trees

## Tree traversal

## Tree Refresher

We looked at the basic terminology for trees in theory lecture 4.

Trees are partially ordered – the levels allow us to order, but vertices on the same level are not ordered.



## Tree Traversal

*Traversing* a tree means moving through the tree, visiting every node in some order.

There are a number of reasons why we might want to do this

- Serialisation – writing out the data as a linear sequence
- Search – visit every vertex to find something
- Tree properties – calculating overall properties of the tree

# Tree Traversal - DFS

You looked at graph search in SOF1 week 8

with depth-first search (DFS) and breadth-first search (BFS).

The ideas are the same, but we have some ordering in the tree and can be more consistent about the search order. The search always starts at the root.

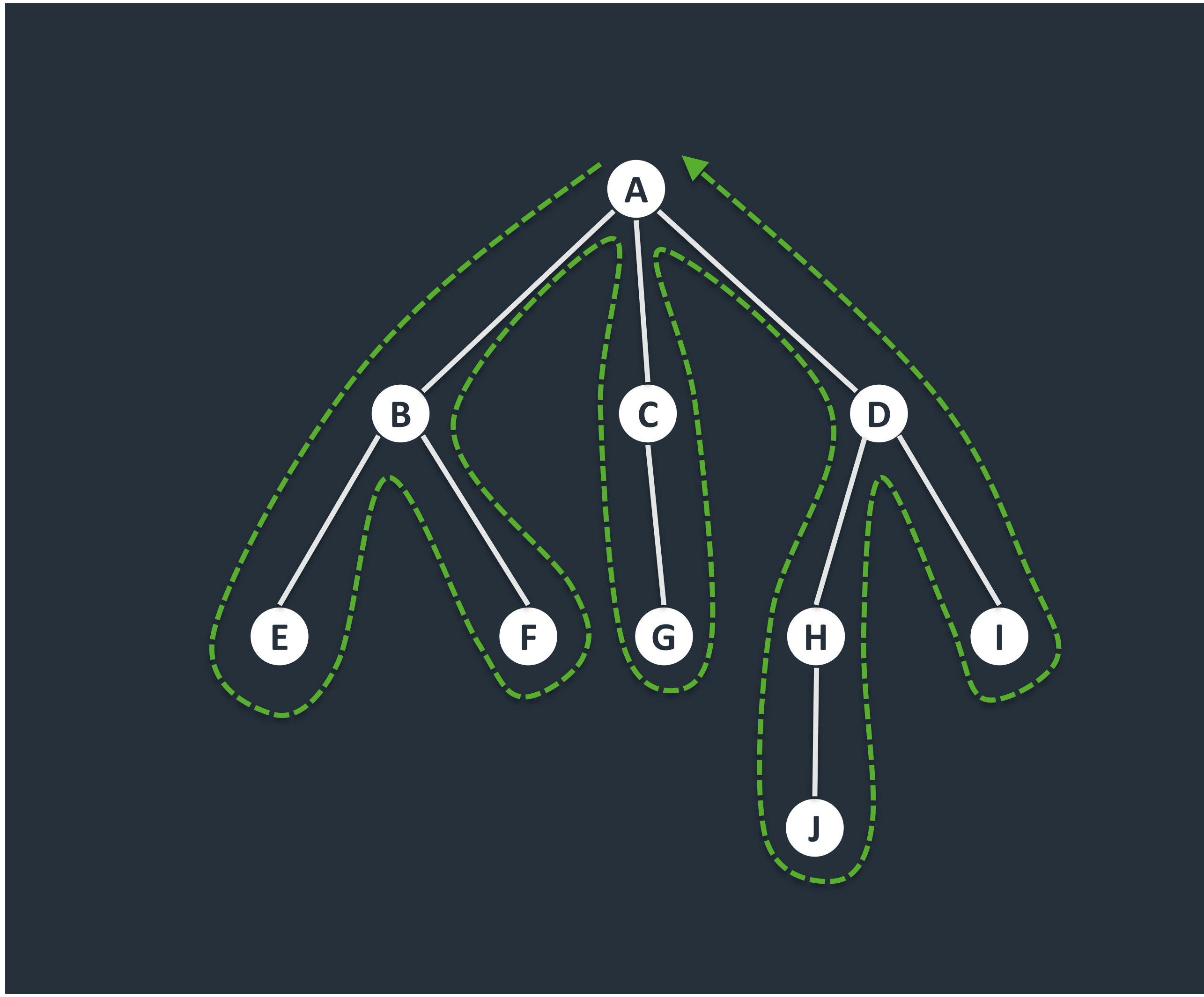
**DFS** – visit the children of a vertex before processing the siblings

**Pre-order**: The vertex is visited before the next step

ABEFCGDHJI

**Post-order**: The vertex is visited after the children are processed

EFBGCJHIDA

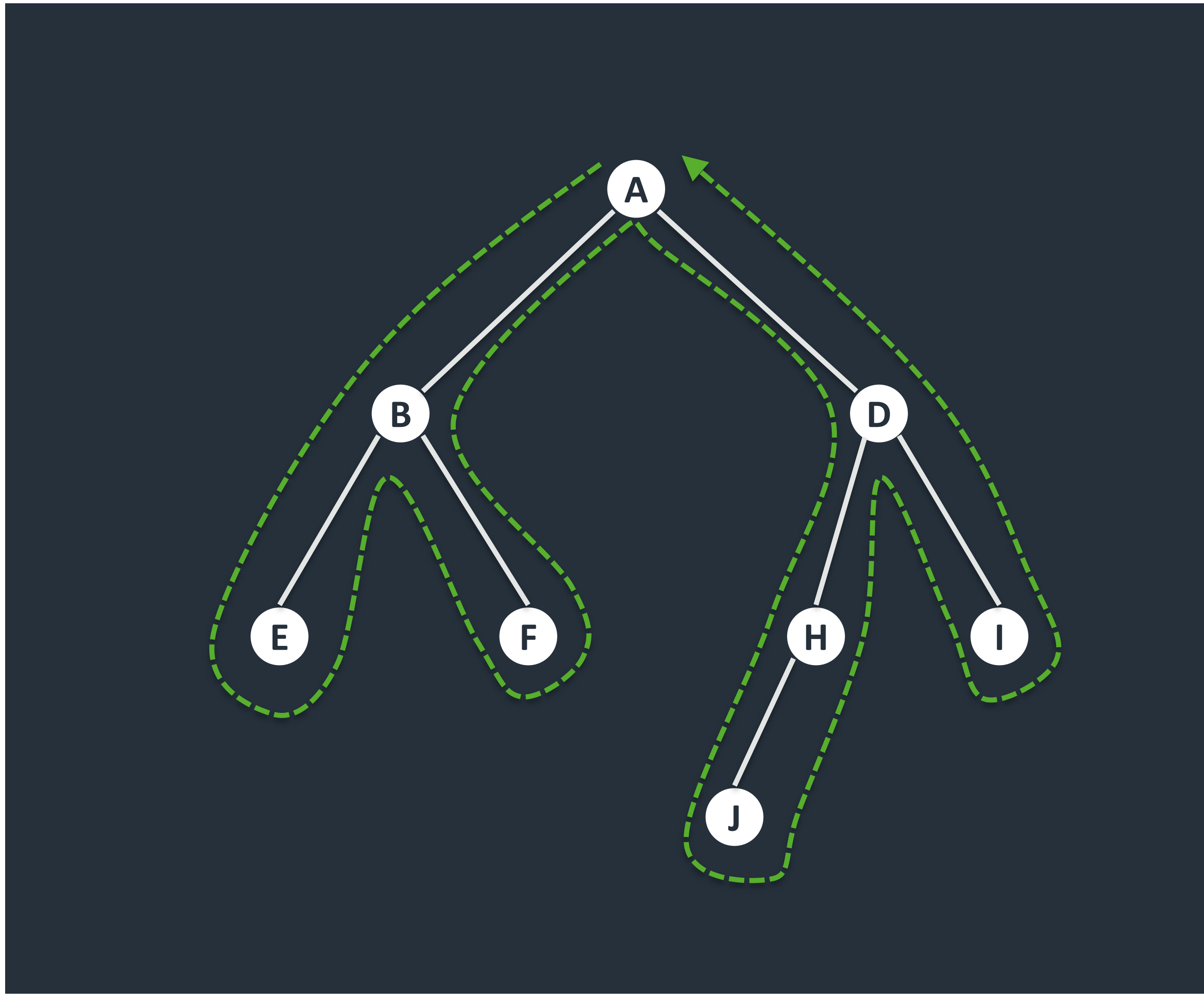


# DFS – Binary Tree

If the tree is a binary tree, it has a no more than two children, designated left and right.

Then **in-order** is possible. The vertex is visited after the left child and before the right.

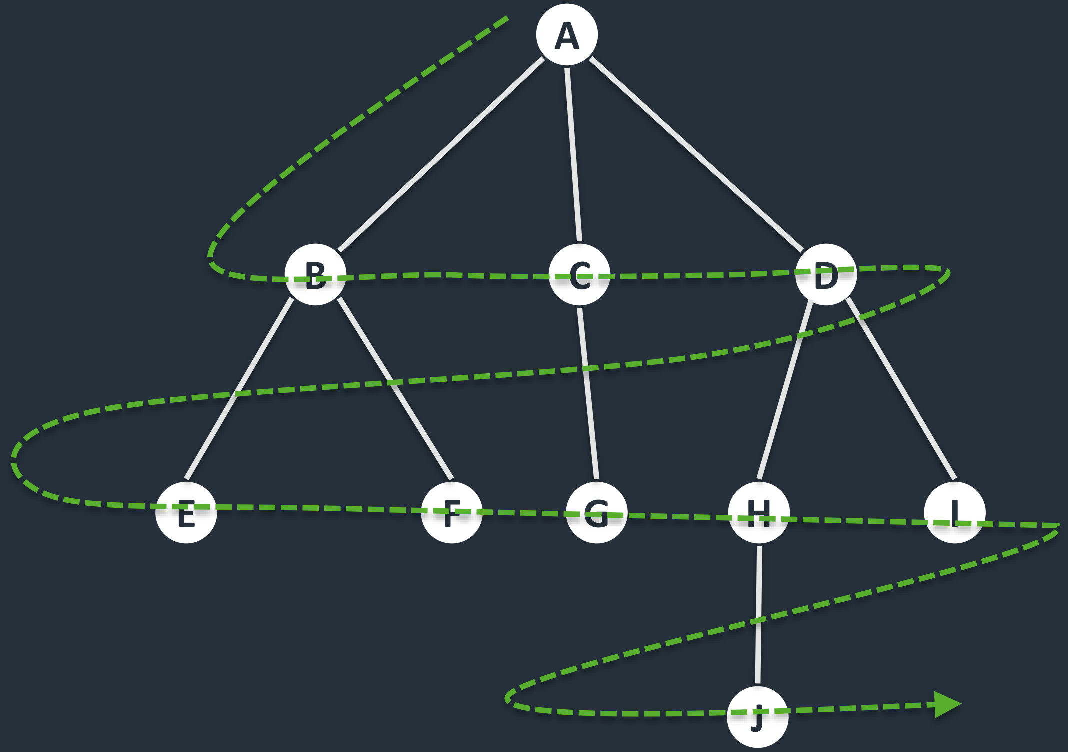
EBFAJHDI



# Tree Traversal - BFS

**BFS** – visit all the siblings before processing any of their children

ABCDEFGHIJ



## Binary search tree

## Binary Search Tree

We met binary trees in lecture 4

- Each vertex has at most 2 children
- Compact representation as array
- Useful for representing heap efficiently

A Binary Search Tree (BST) is a more organised version of the heap, represented by a binary tree. It can be used for

- Sets
- Dictionaries
- Priority queues

Basic operations take time proportional to the tree height

# Operations and Structure

BST Data Structure

**Organization**

Binary Tree

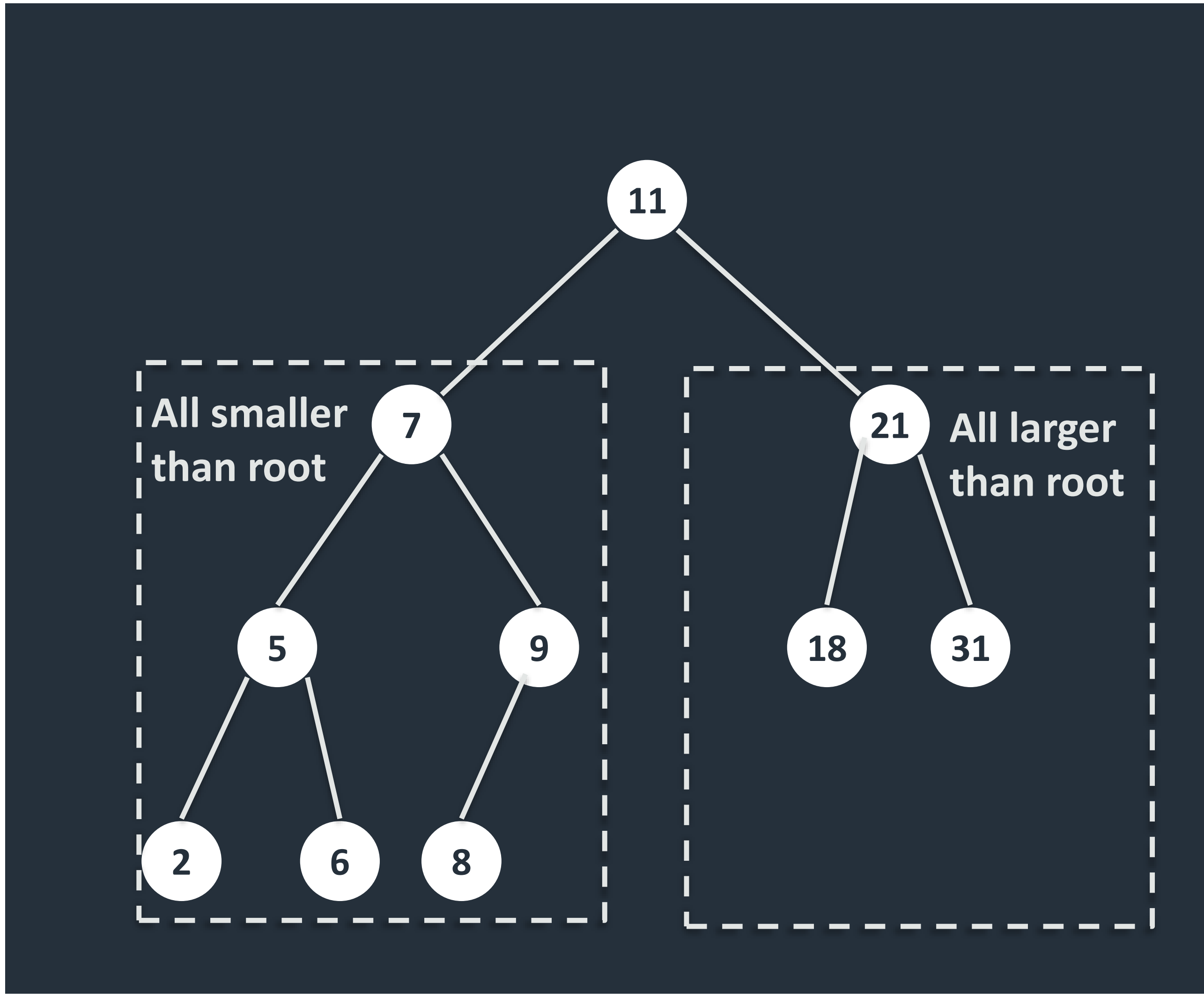
**Common operations**

| | |
|---|---|
| `Search(v)` | Find element v |
| `Insert(v)` | Insert element v |
| `Delete(v)` | Delete element v |
| `Minimum()` | Find the smallest element |

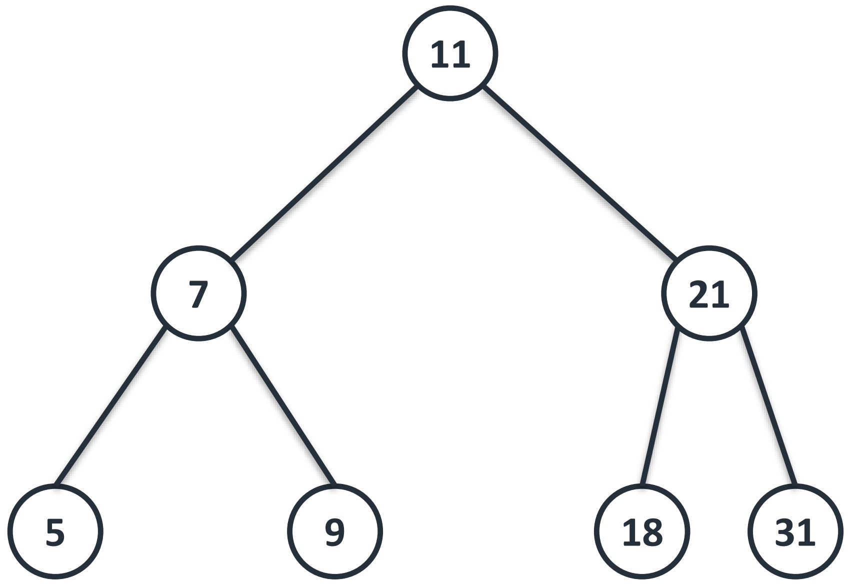
The tree is divided around the root

- All smaller keys in left subtree
- All larger keys in right subtree

Each subtree is also a BST



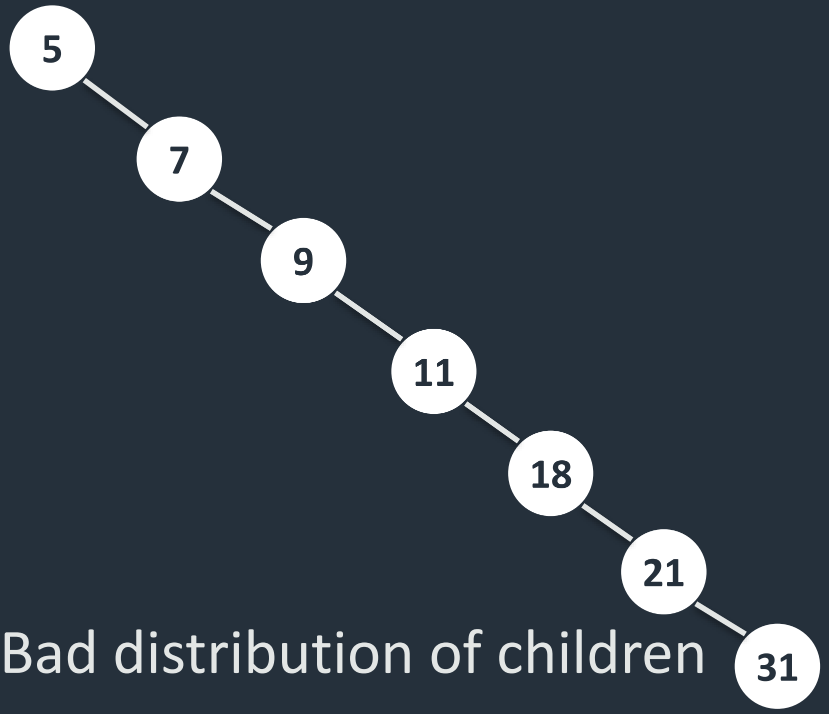
## Operations and Structure



Balanced Tree

Equal children on both left and right

Height $h \simeq \log_2 n$ so operations are $O(h) = O(\log n)$



Bad distribution of children

Height $h \simeq n$ so operations are $O(h) = O(n)$

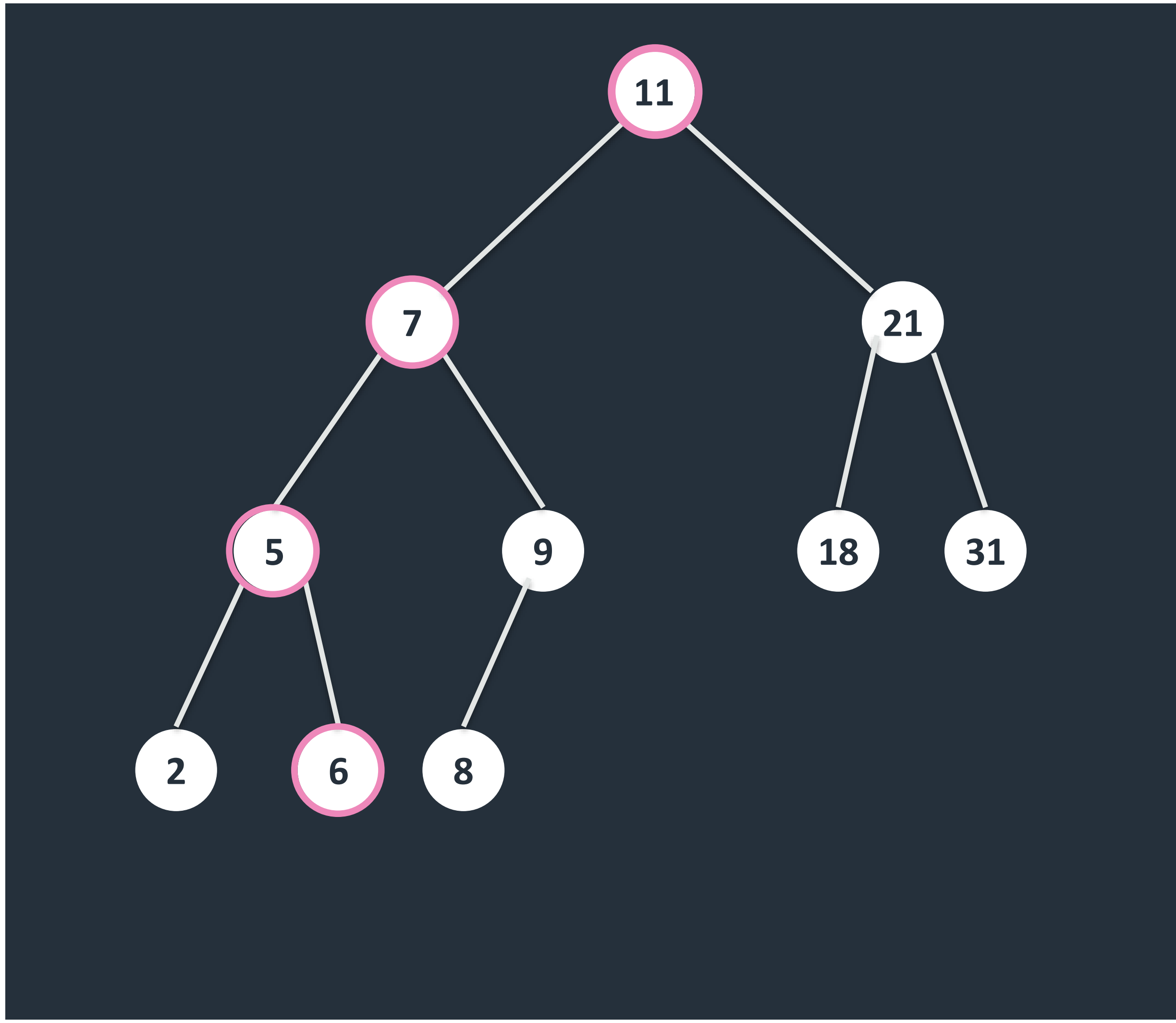
Balanced trees are better

# Search

Search BST

```
BSTSearch(t : BST, x)
if t is empty then return None
if x==t.value then
   return t
if x<t.value then
   return BSTSearch(t.left,x)
return BSTSearch(t.right,x)
```

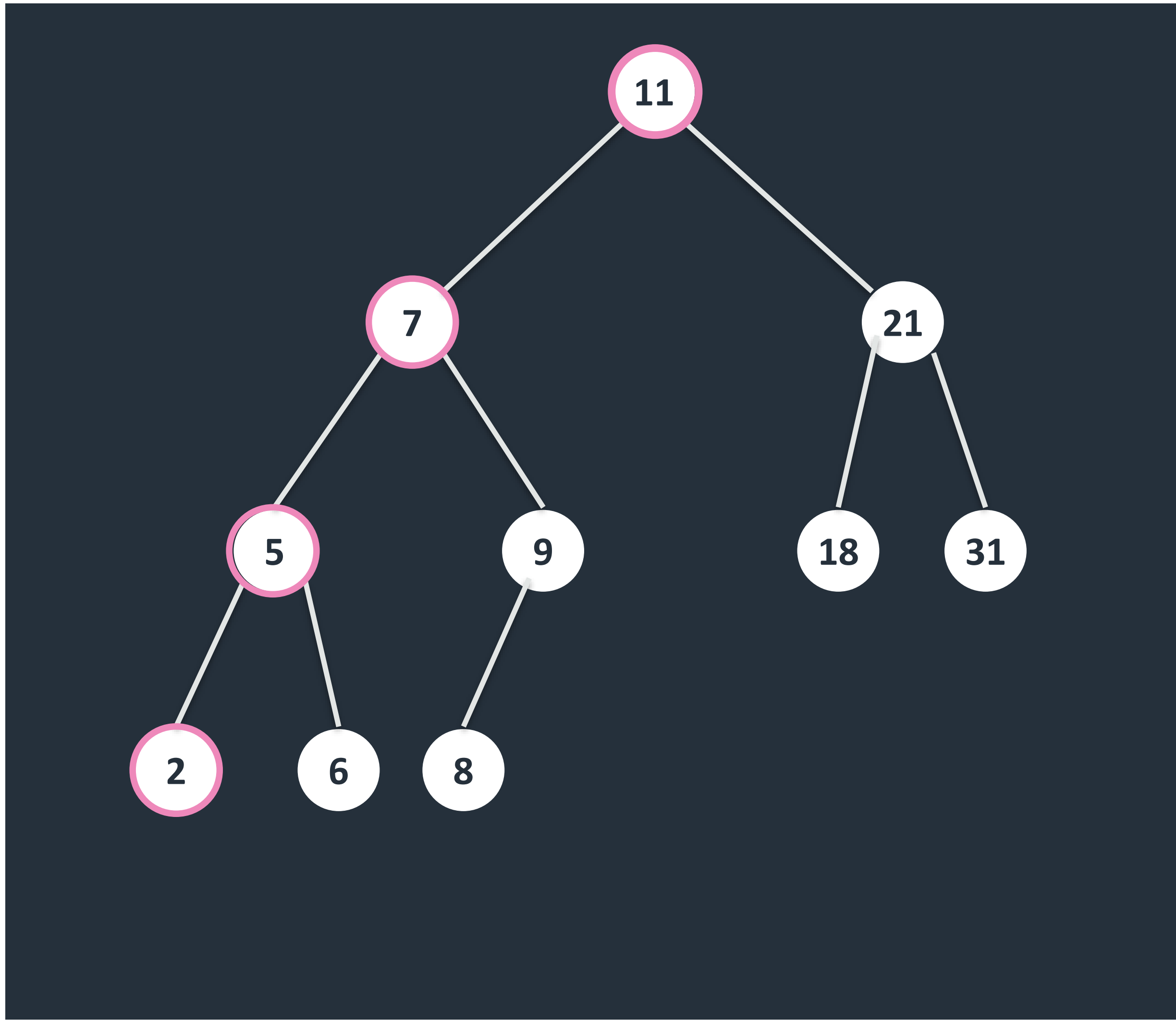
Example: search for 6



# Min/Max

The minimum is always on the far left

The maximum is on the far right

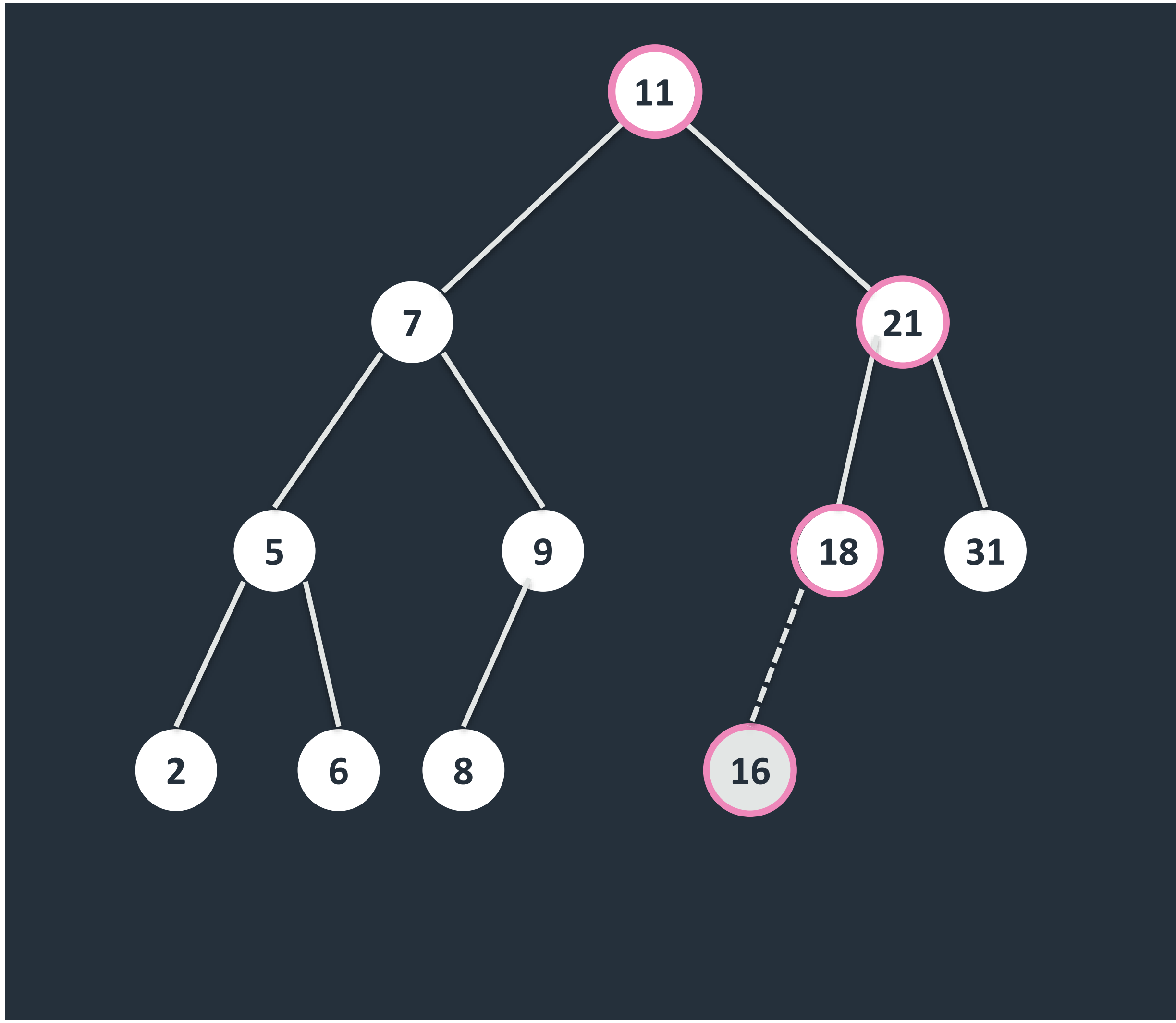


# Insert

BST Insert

```
BSTInsert(t : BST, x)
if x<=t.value then
   if t.left!=None then
      BSTInsert(t.left,x)
   else
      add vertex(x) as left child
else
   if t.right!=None then
      BSTInsert(t.right,x)
   else
      add vertex(x) as right child
```

Example: insert 16

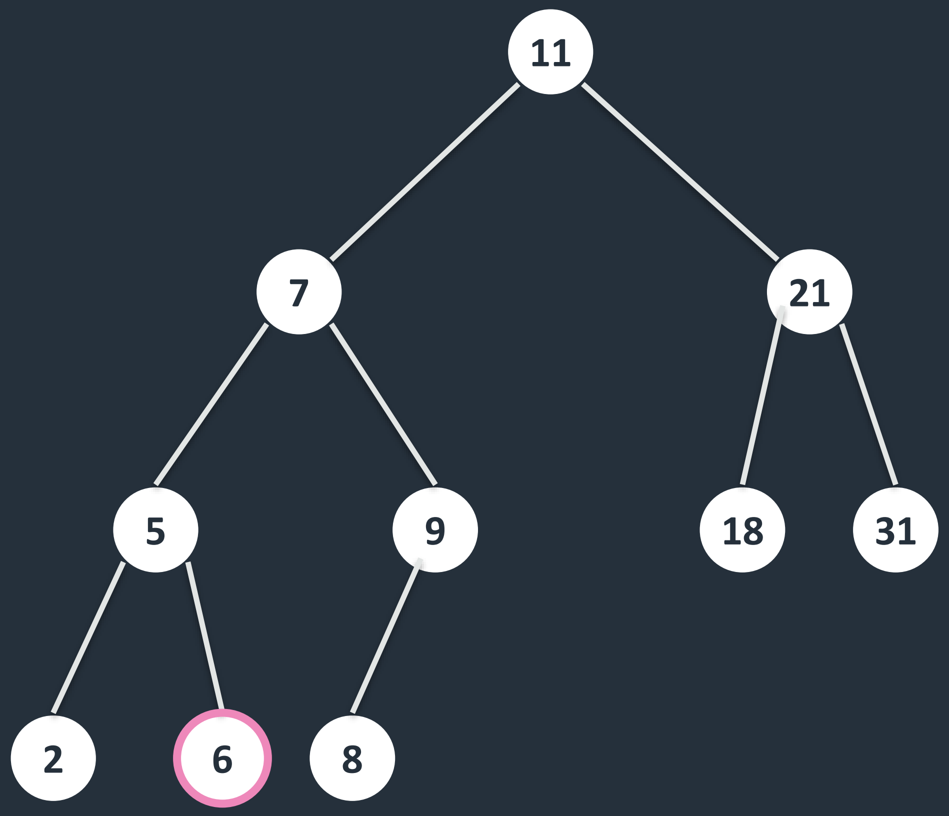


## Delete – Childless Vertex

Deleting a vertex with no children is simple.

Delete the vertex and the left or right link from the parent.

Example: delete 6

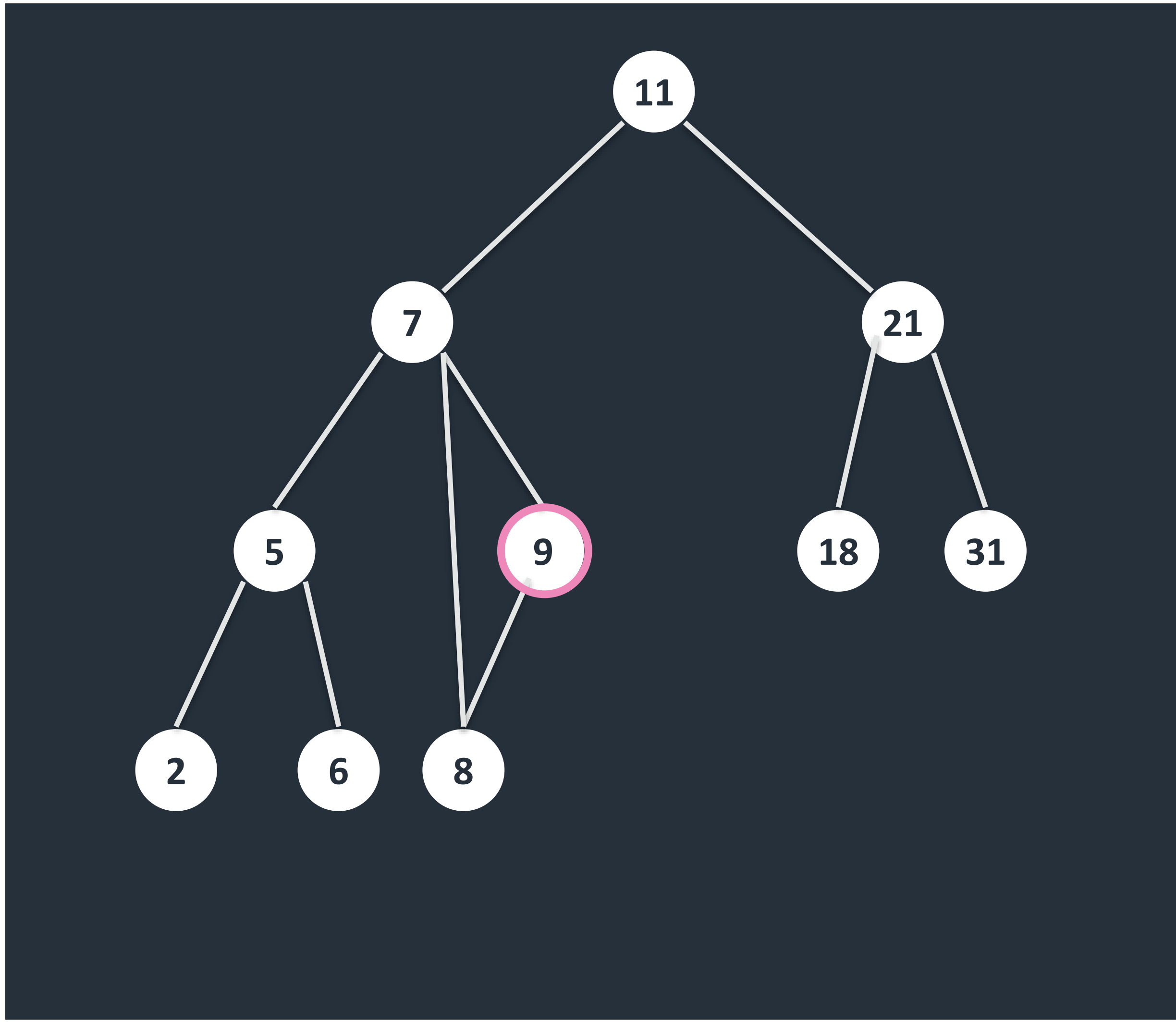


# Delete – Vertex with one child

Deleting a vertex with one child involves relinking.

Delete the vertex and reconnect the child to the left or right link from the parent.

Example: delete 9



# Delete – Vertex with two children

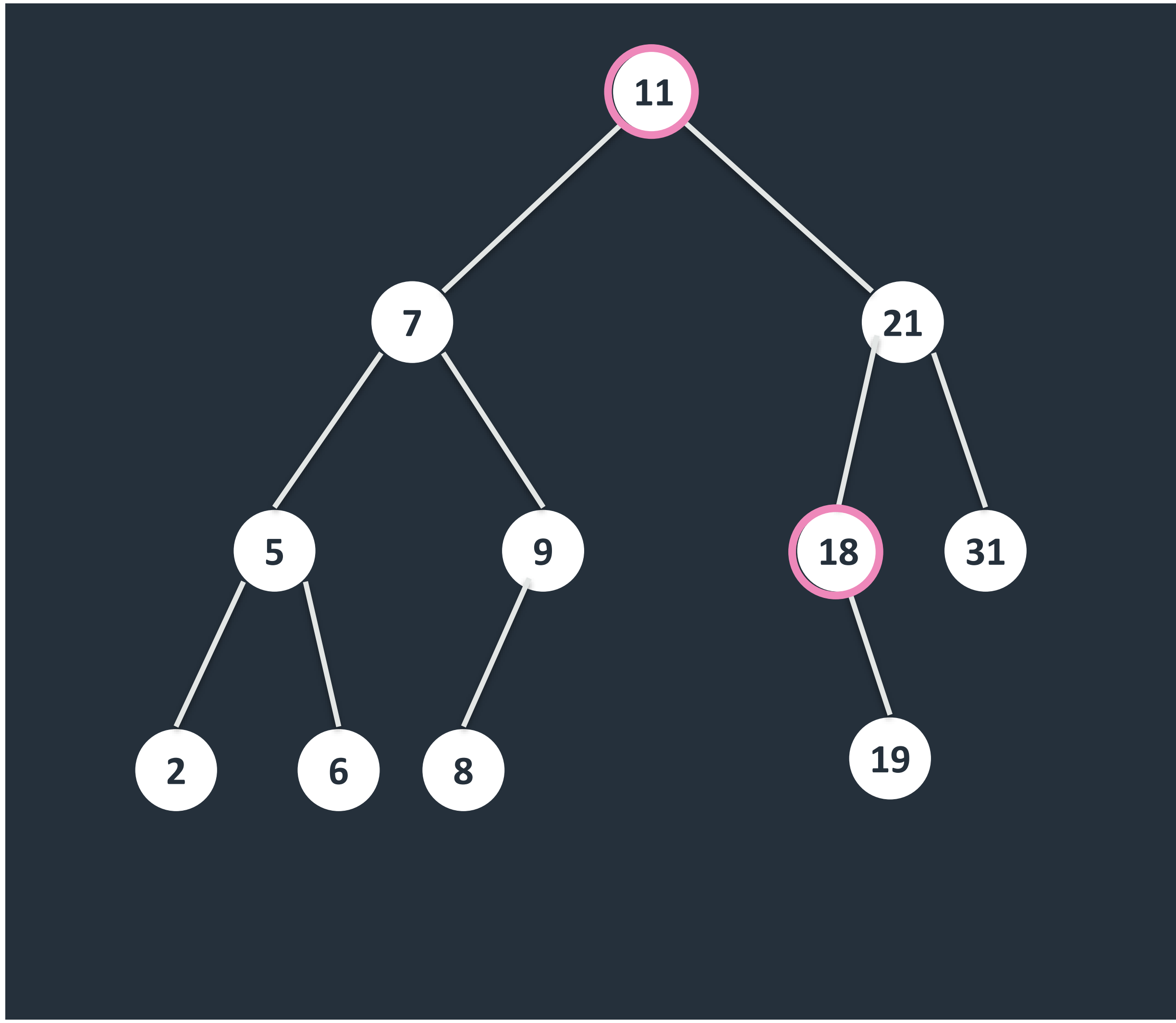
Deleting a vertex with two children involves reorganisation of the tree.

Find the next in order from the tree. This will be the leftmost from the right subtree.

Replace the vertex to be deleted with the found one.

Delete the found vertex. This vertex will have at most one child.

Example: delete 11



# Delete – Vertex with two children

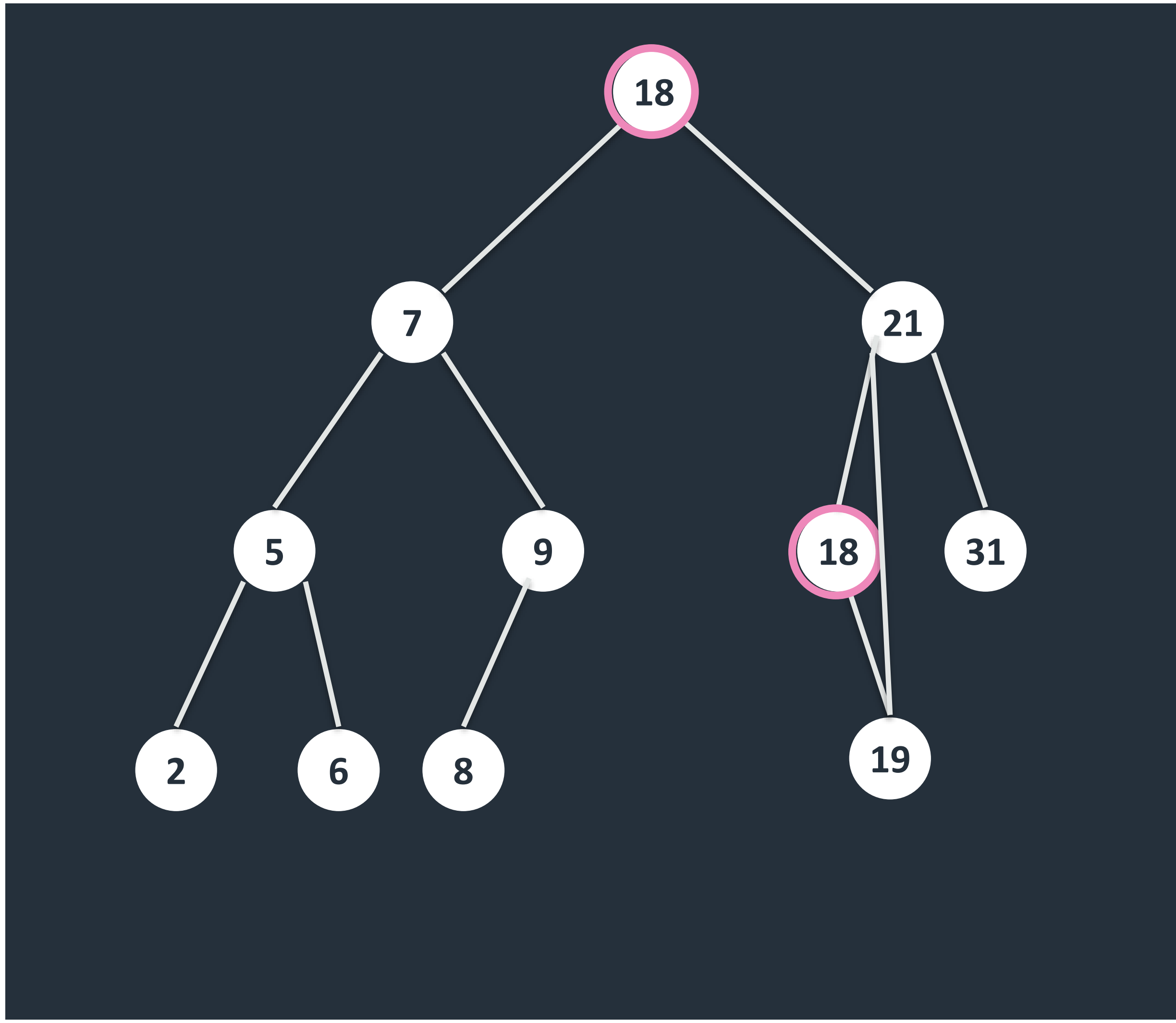
Deleting a vertex with two children involves reorganisation of the tree.

Find the next in order from the tree. This will be the leftmost from the right subtree.

Replace the vertex to be deleted with the found one.

Delete the found vertex. This vertex will have at most one child.

Example: delete 11



## Balancing

# Tree rotation

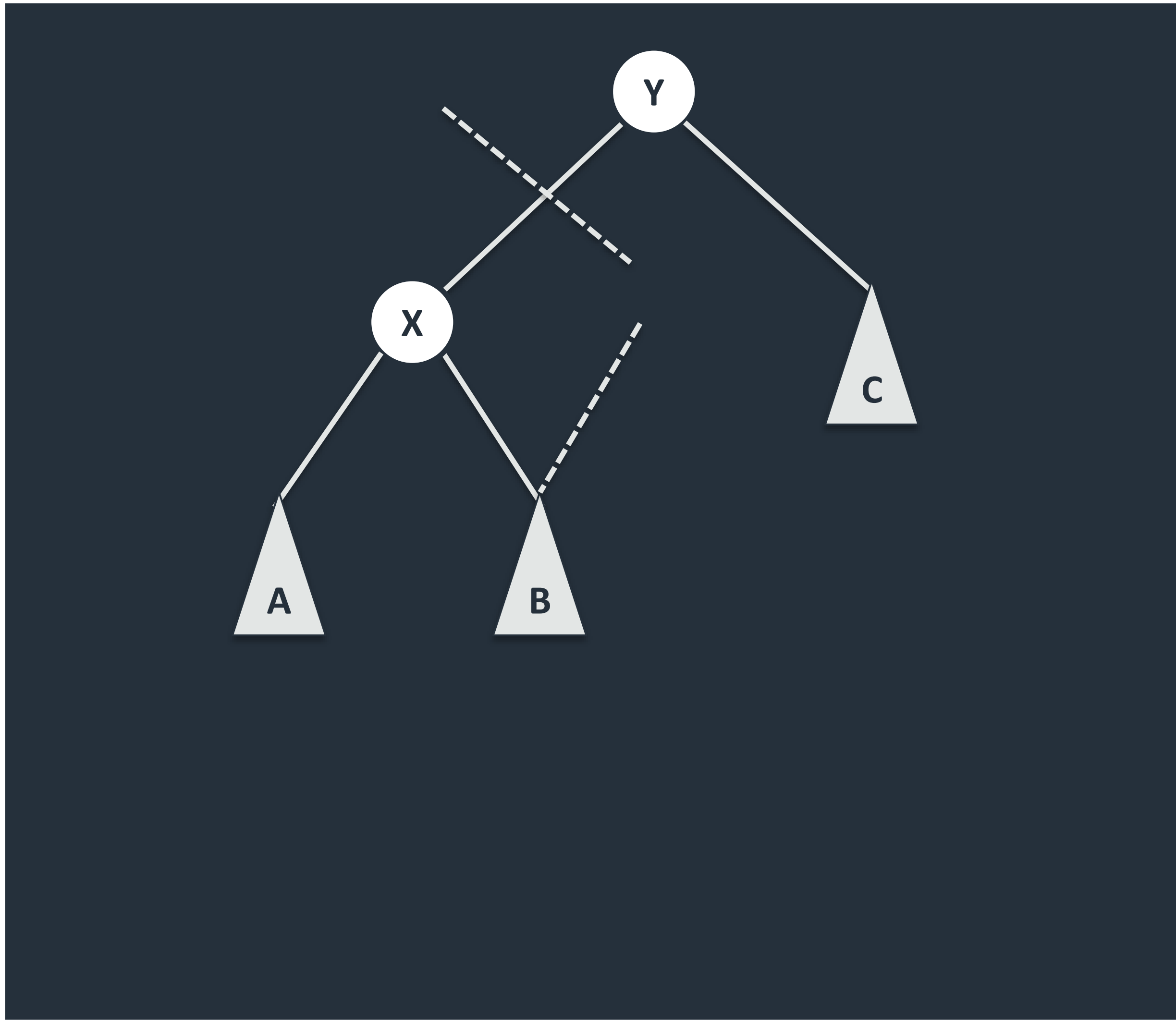
Consider the binary search tree on the left.

X and Y are vertices, Y is the root.

A, B, and C are binary trees (subtrees of the original tree)

A≤X ≤Y, X ≤B ≤Y,  C ≥Y

Rotate *right*



## Tree rotation

X is now the root.

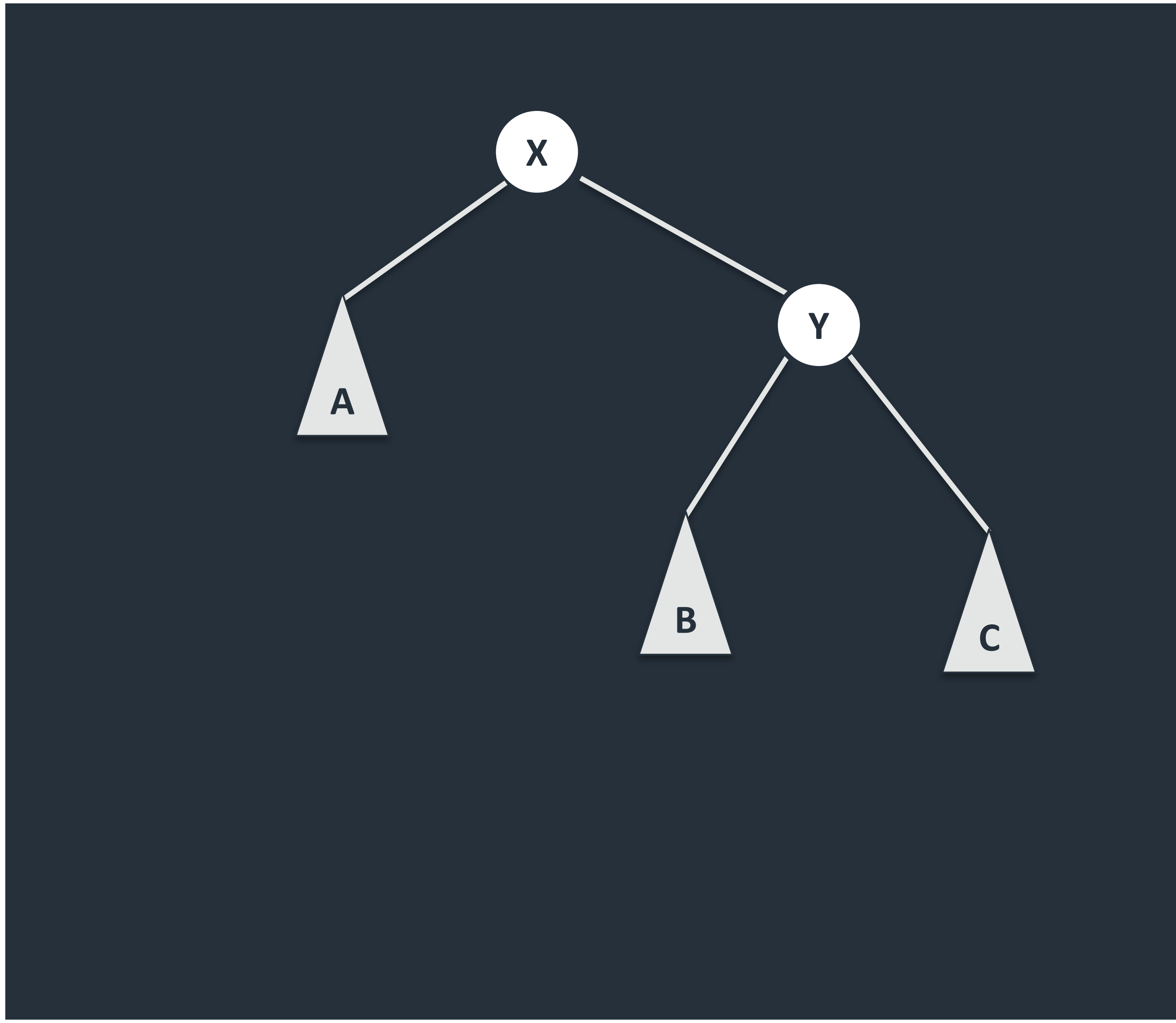
X ≤B ≤Y from the original tree properties, so configuration is still a binary search tree.

The height to the top of A is reduced.

The height to the top of C is increased.

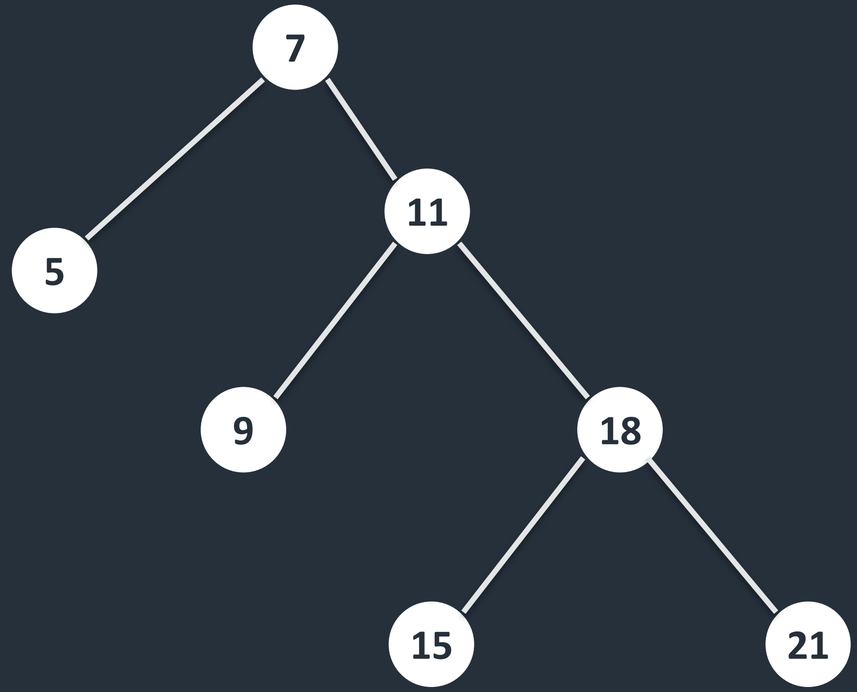
B remains the same.

Rotate *left* is the inverse operation, i.e. from this tree to the previous one.



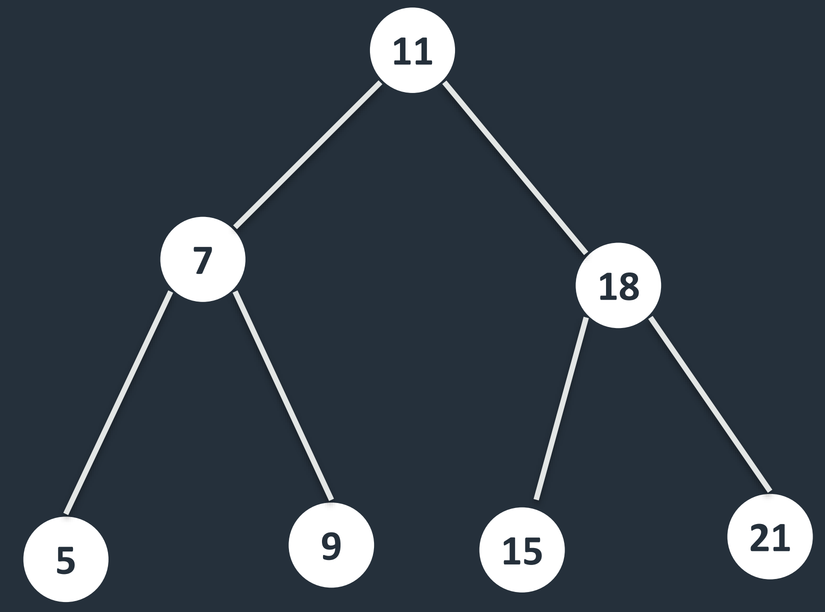
## Example

Rotate left



# Example

Tree is now balanced.



## AVL Tree

**A**delson-**V**elsky and **L**andis

# Balance criteria

In order to balance a tree, we need to measure the amount of balance.

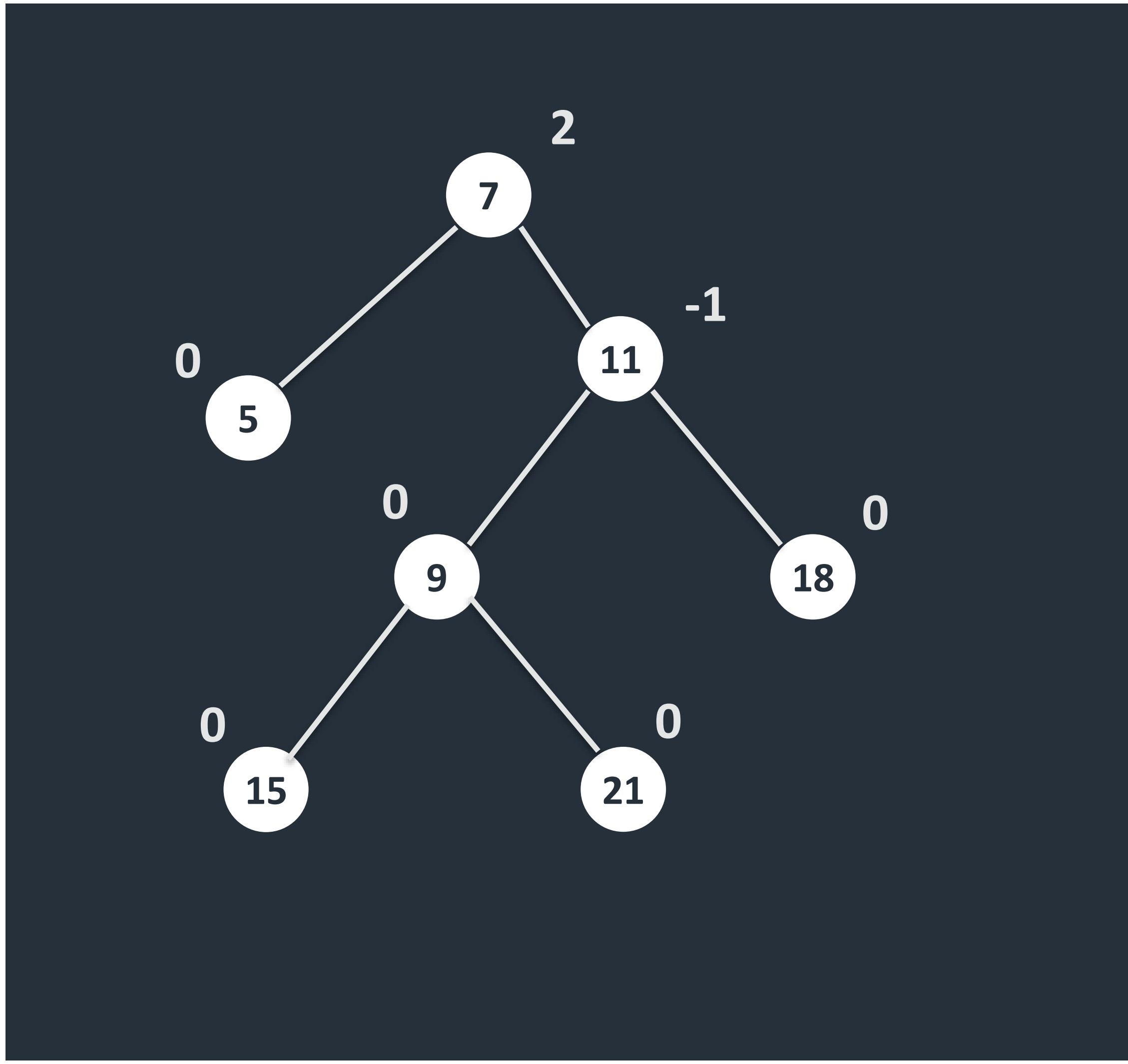
Number of ways to measure

- Red-black tree
- AVL tree
- and others

In an AVL tree, the balance factor of a vertex is

$$BF = \text{Height(RightSubtree)} - \text{Height(LeftSubtree)}$$

The tree is AVL balanced if $BF \in \{-1,0,1\}$ for all vertices



## Rebalancing

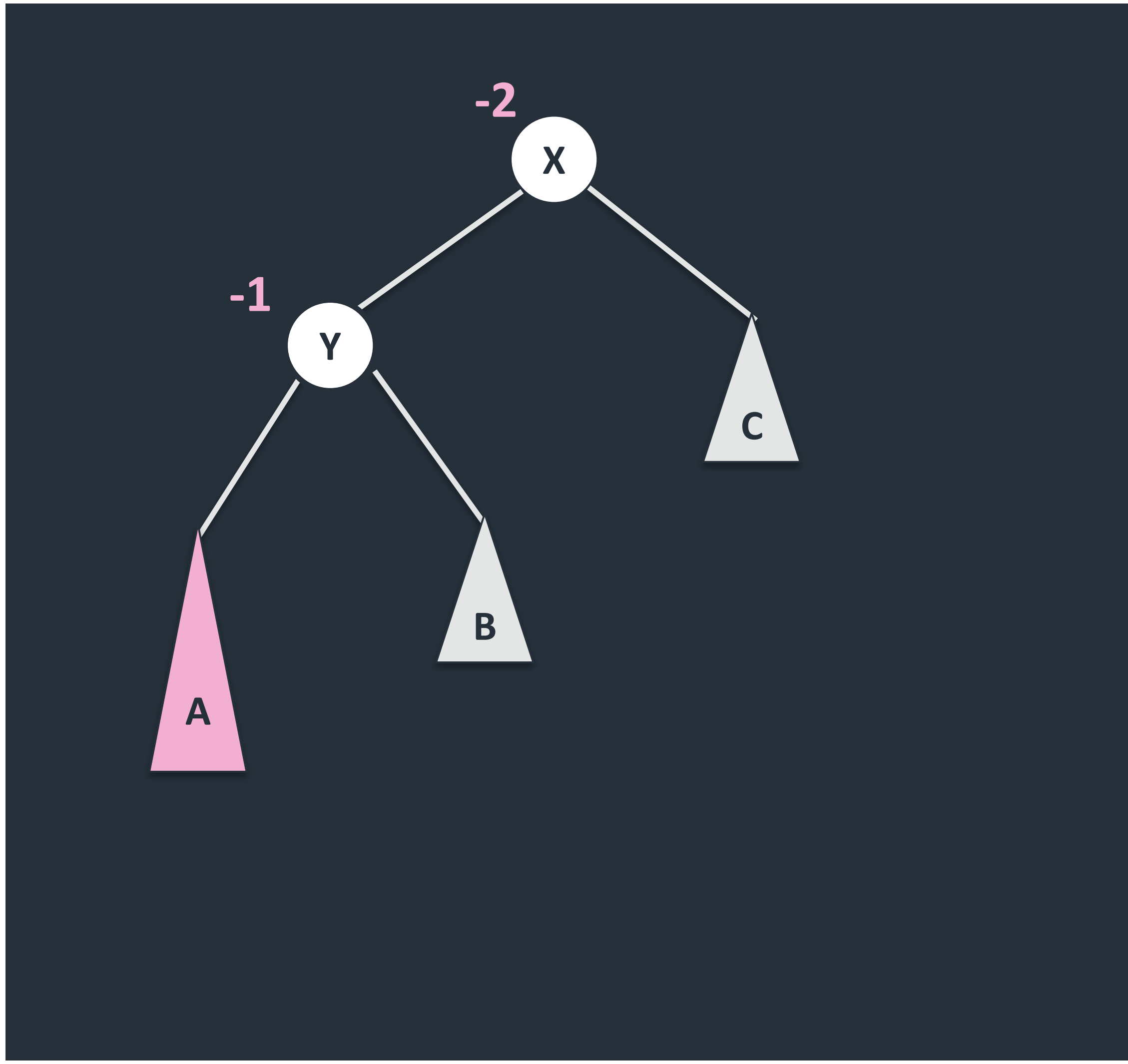
An AVL may become unbalanced when adding or deleting a vertex.

A balance factor becomes $\pm 2$

We can then rebalance the tree using rotations

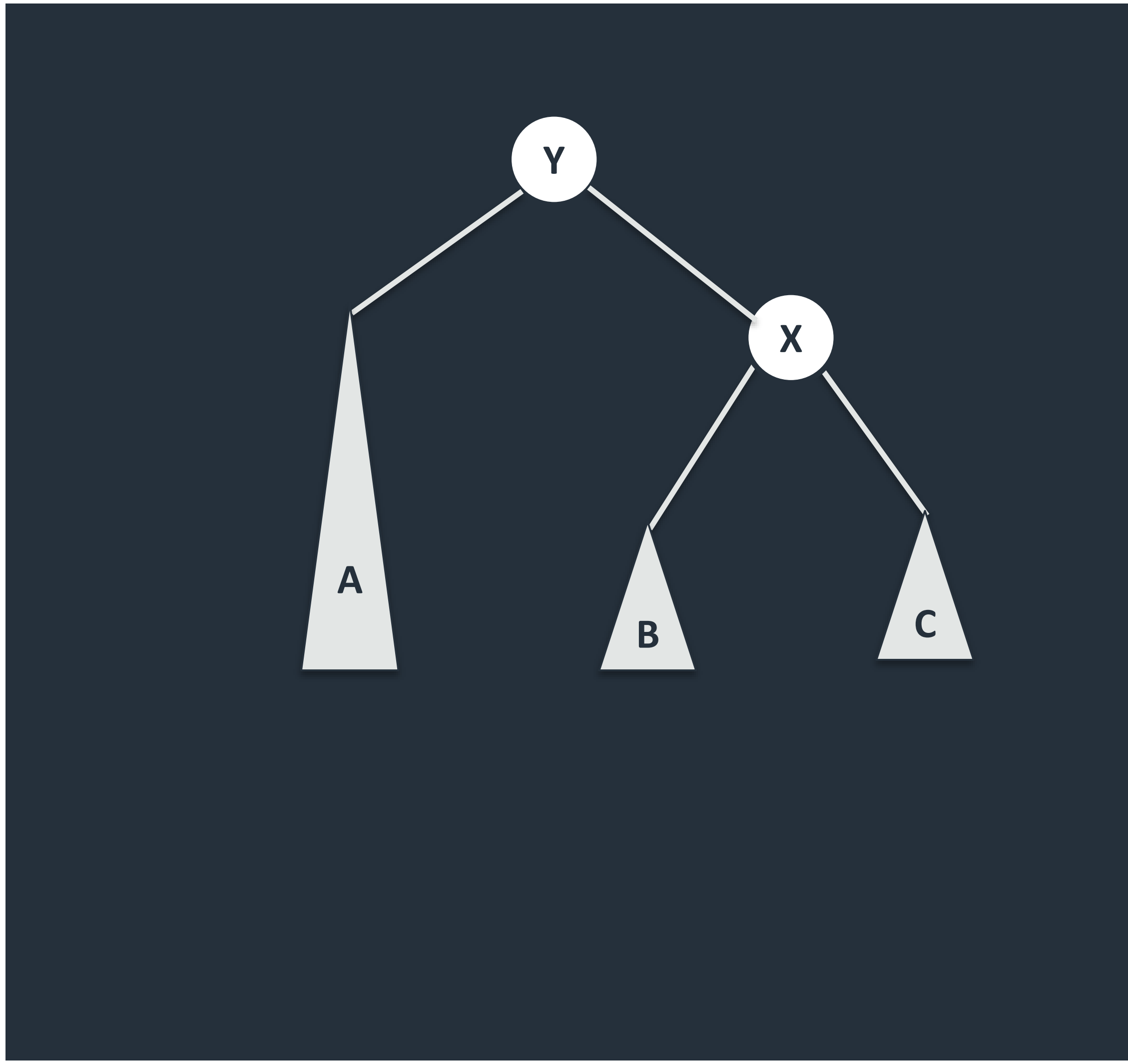
Example left: Y is a left child of X, and Y is imbalanced to the left (A is too tall)

This is a left-left imbalance and simply fixed by a right rotation of X and Y



# Rebalancing left-left

A is moved 1 closer to the root, rebalancing the tree.

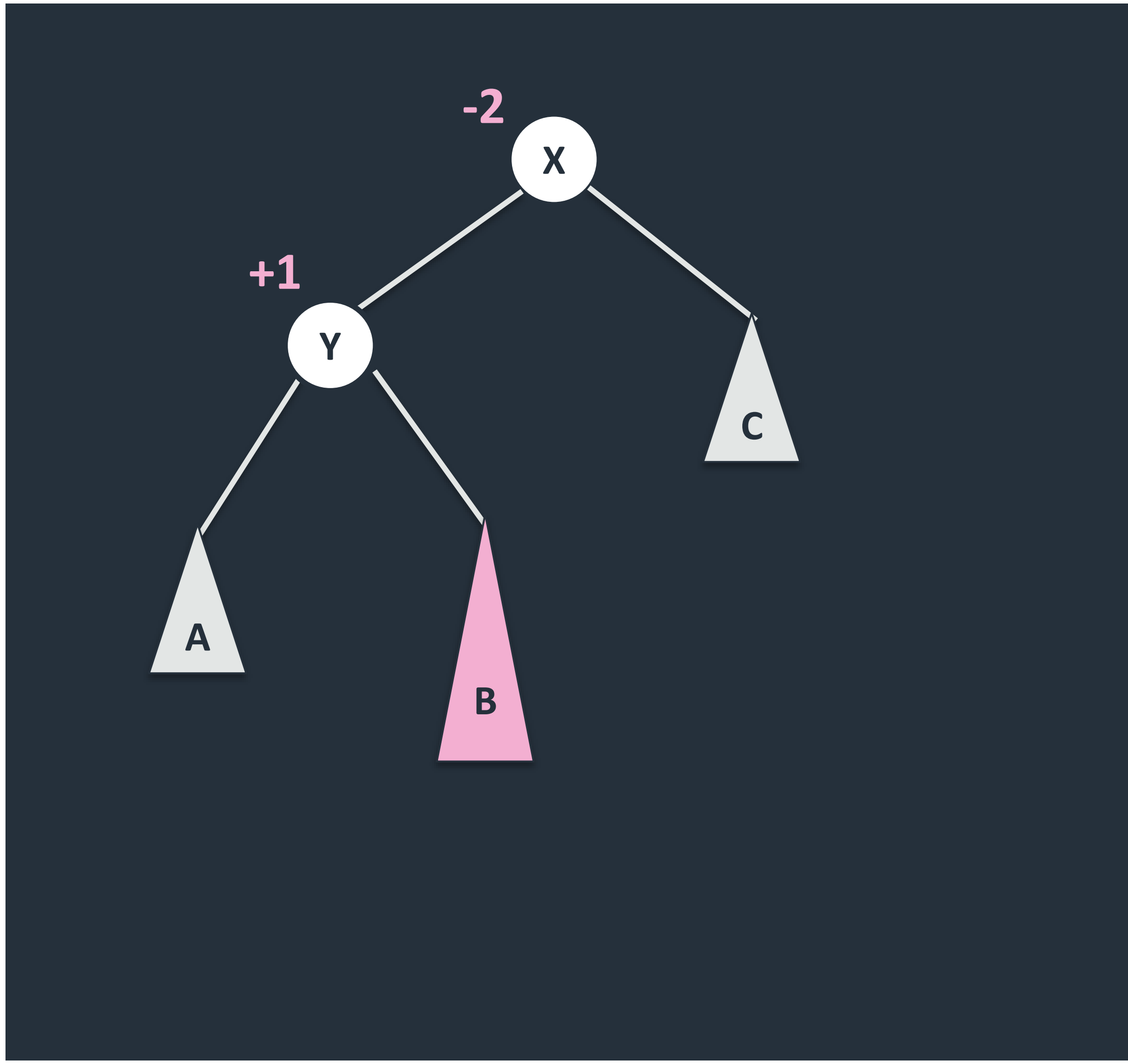


# Rebalancing left-right

In this example Y is a left child of X, and Y is imbalanced to the right (B is too tall)

This is called a left-right imbalance.

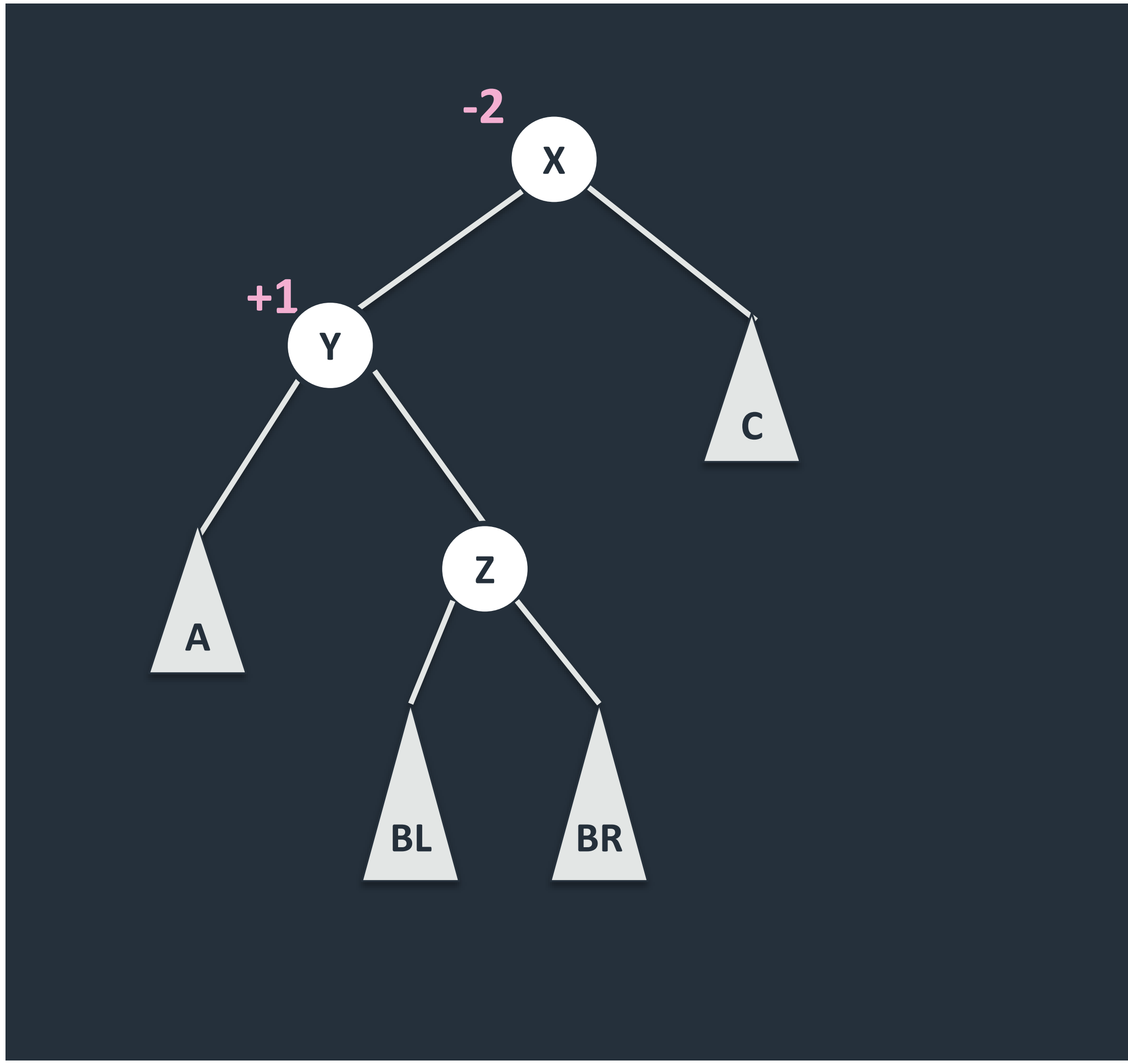
This is a more complex situation. Rotating right will not change the height of B.



# Rebalancing left-right

Consider the root of subtree B as a separate vertex.

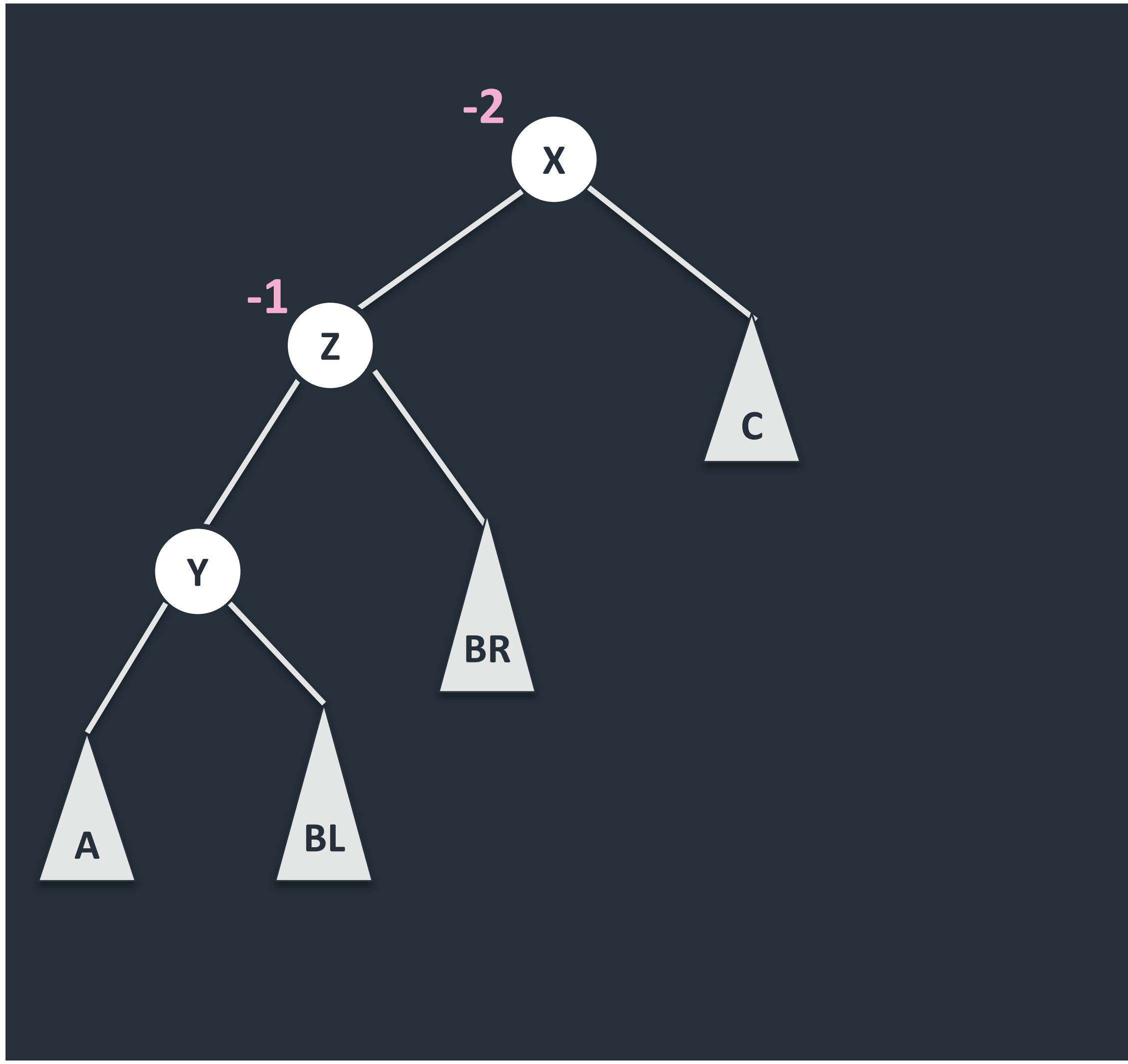
Now rotate left with Y and Z.



# Rebalancing left-right

This creates a tree with a left-left imbalance.

Cure by a right rotation with X and Z, as before.

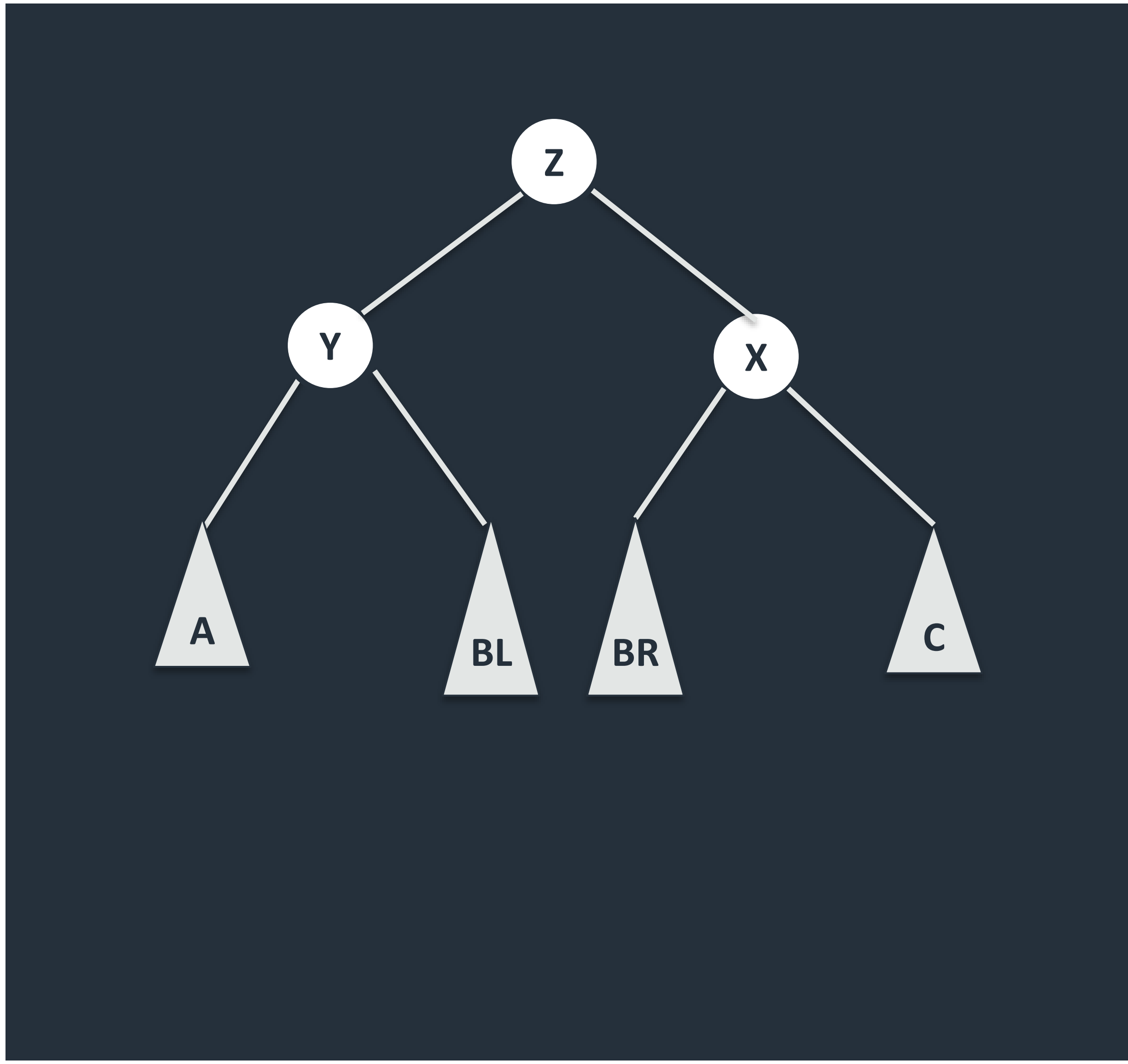


# Rebalancing left-right

This creates a tree with a left-left imbalance.

Cure by a right rotation with X and Z, as before.

BL and BR are shortened by 1, rebalancing the tree.

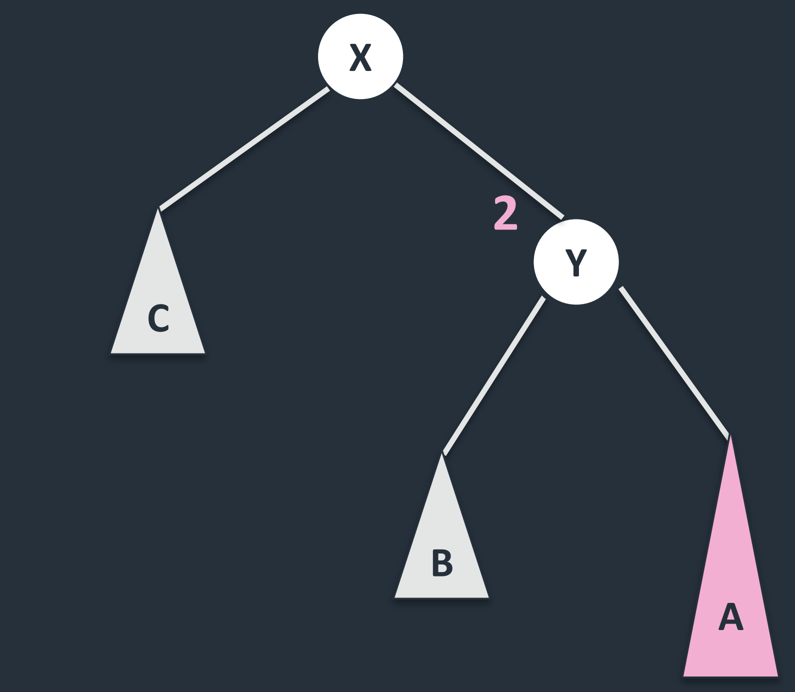


# Right imbalance

The right imbalances, right-right and right-left can be handled in the same way, but are mirror images.

Right-right: Rotate left

Right-left: Rotate right + rotate left



## Summary

Understand:

- DFS and BFS search orders
- The structure of a binary search tree
- Algorithms and complexity of basic operations om BSTs
- Rebalancing and AVL trees

Read

- Skiena, Sections 3.4,
- Attempt exercises 4-2, 4-20, 4-21

Next

- Algorithms for graphs