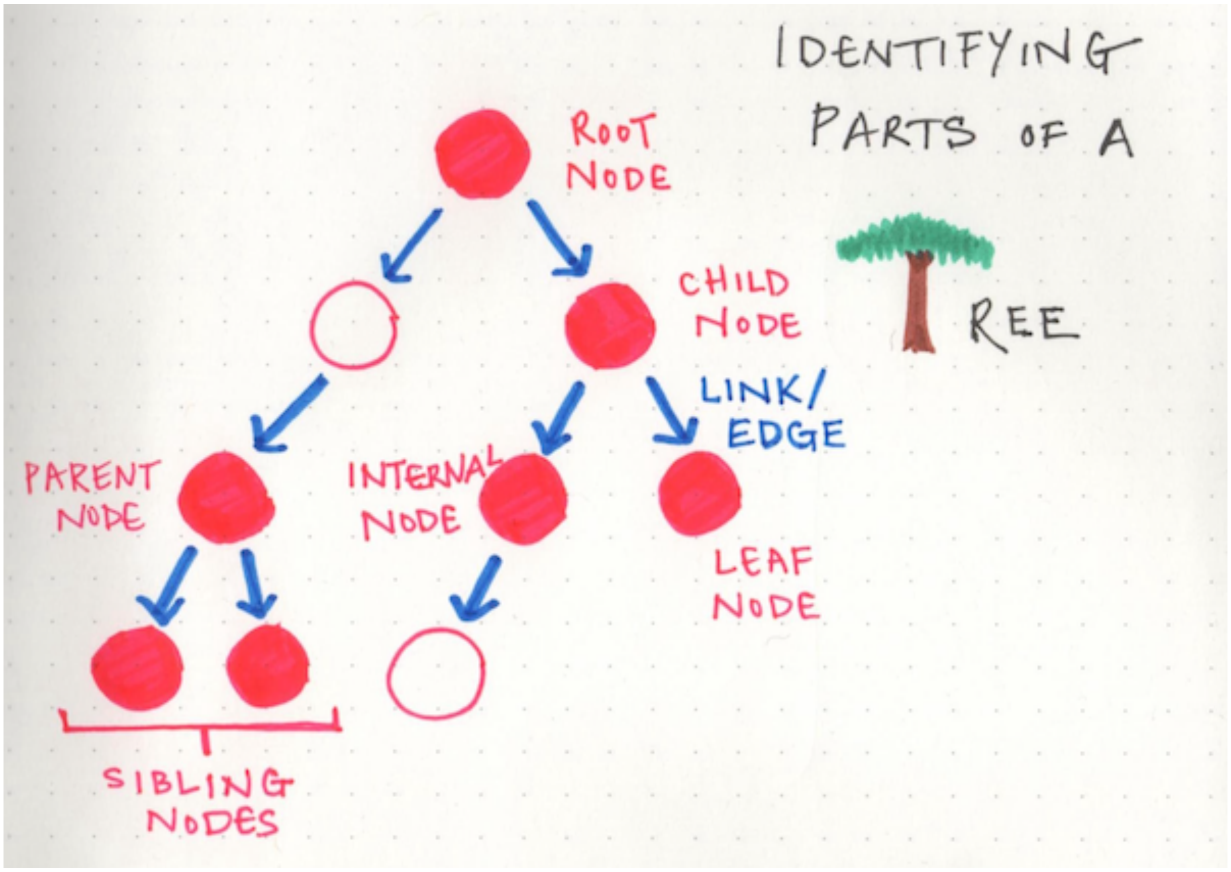
Binary Tree

It's a tree data structure where each node has at most 2 children (left/right) children. Binary trees are a type of non-linear data structure (which means we don't care much about the order but how each element relates to others).
Trees are recursive data structures, which means that a single tree is made up of many others.

Parts of a Tree

Before we dive into the tree parts, it's valid to notice that the parts of a tree are the same as a linked list (Made of nodes and connection between nodes)

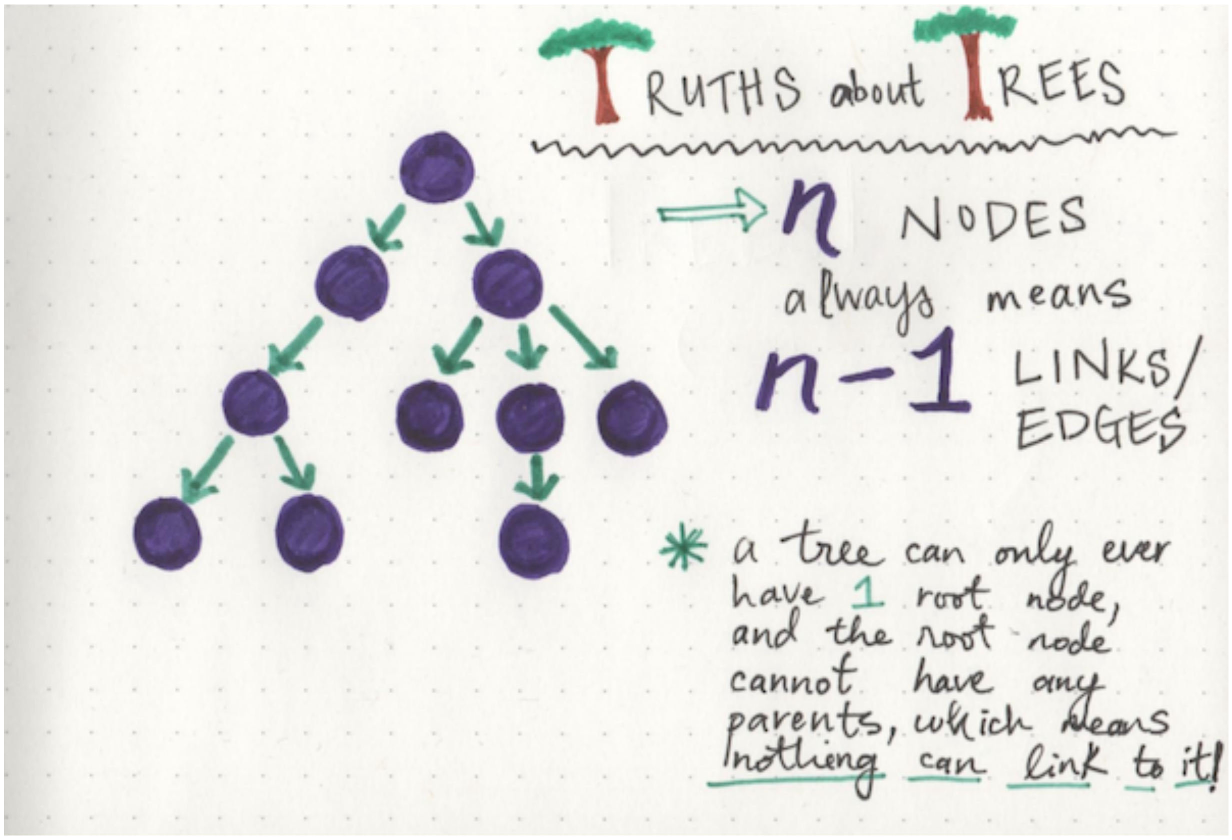
- Root: Topmost node of the Tree
- Child: Any node that has some parent
- Link/Edge: It's a reference that a parent node has to it's child nodes
- Sibling: Any group of nodes that are children of the same parent node
- Parent/Internal: Any node that has children
- Leaf: Any node that doesn't have child nodes.

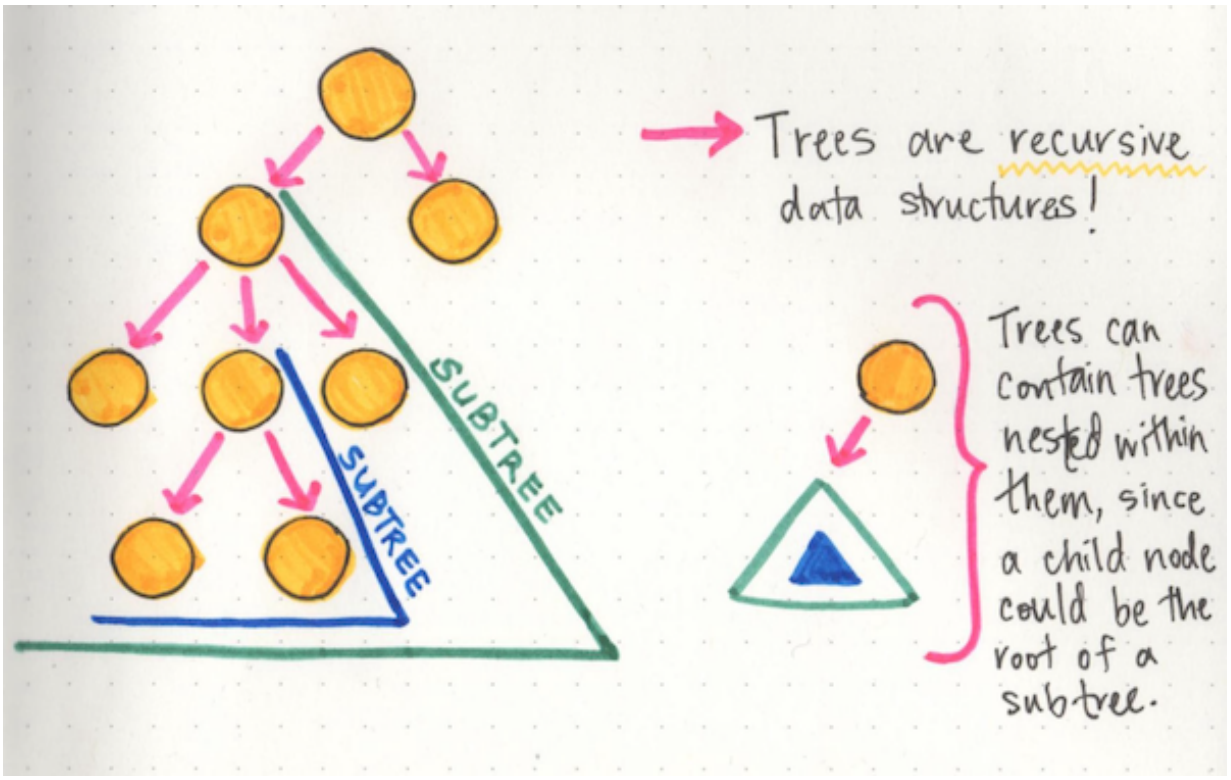


Some Properties of Trees

Those properties are valid for any type of Trees:

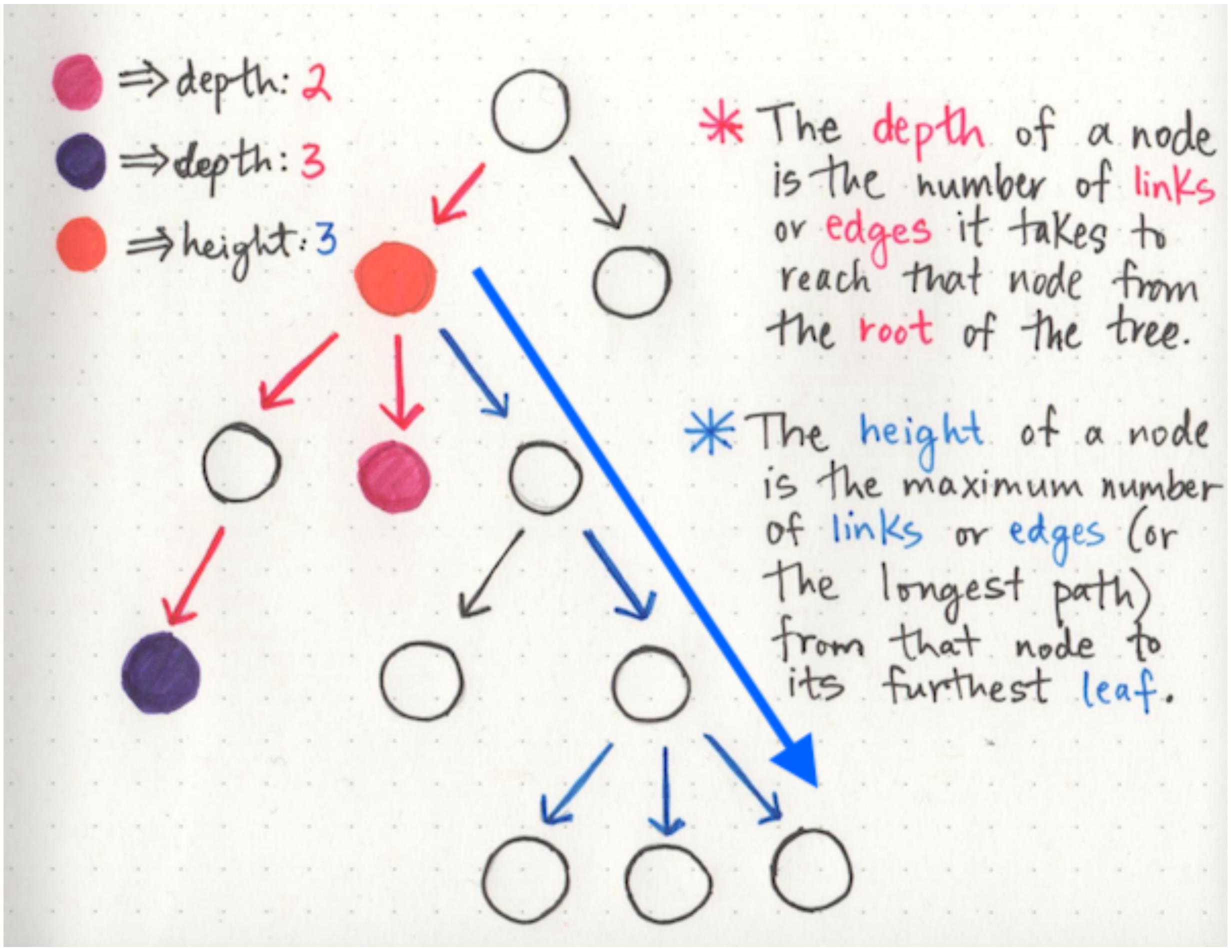
- If a tree has n nodes, it will always have one less number of edges (n-1).
- Trees Are recursive you can have a sub-tree that starts with a child node


TRUTHS about TREES
n NODES always means n-1 LINKS/EDGES
a tree can only ever have 1 root node, and the root node cannot have any parents, which means nothing can link to it!

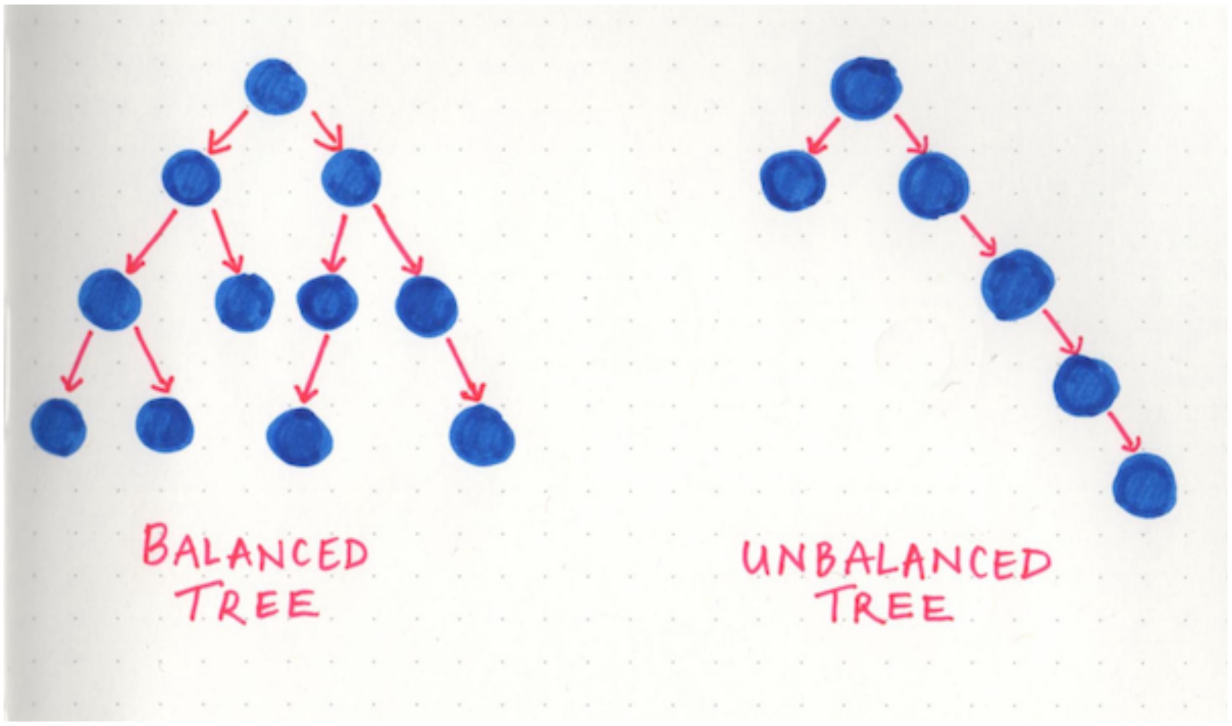

Trees are recursive data structures!
SUBTREE
SUBTREE
Trees can contain trees nested within them, since a child node could be the root of a subtree.

Depth and Height of Nodes

The depth of the node is the number if links that are needed to reach the root node.
The height of the node is the number of links from that node until the most deep leaf


⇒ depth: 2
⇒ depth: 3
⇒ height: 3
* The depth of a node is the number of links or edges it takes to reach that node from the root of the tree.
* The height of a node is the maximum number of links or edges (or the longest path) from that node to its furthest leaf.

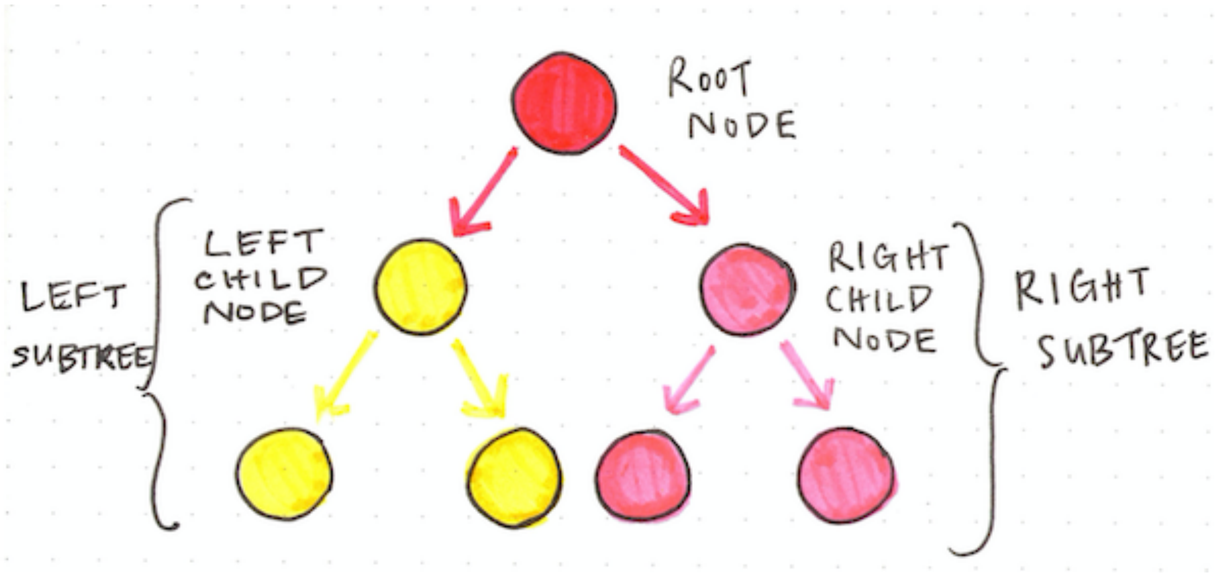
The reason that depth and height are so important is because they tell us a lot about what a tree looks like, so for example we can detect if the tree is balanced or unbalanced, which is important because of the tree is unbalanced it’s operations (insert, delete, traverse) will be slower O(n).



A tree is considered to be balanced if any two sibling subtrees do not differ in height by more than one level. However, if two sibling subtrees differ significantly in height (and have more than one level of depth of difference), the tree is unbalanced.

Recursive Characteristics of Trees

Every binary tree will contain two subtrees within it: a left subtree and a right subtree. And this rule keeps applying as we go down the tree: both the left subtree and the right subtree are binary trees in and of themselves because they are recursively part of the larger tree.



The recursive aspect of a binary search tree is part of what makes it so powerful.

The very fact that we know that one single BST can be split up and evenly divided into mini-trees within it will come in handy later when we start traversing down the tree!

Traversal

It's the process of visiting every node on the tree, we always start the traversal from the root, and we should always remember that every node may represent a subtree itself.
There are 3 types of binary tree traversal:

- In-order Traversal (left subtree is visited first, then the root and later the right sub-tree.)
- Pre-order Traversal (root node is visited first, then the left subtree and finally the right subtree)
- Post-order Traversal (the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.)

```
# Inorder traversal
# Left -> Root -> Right
    def inorderTraversal(self, root):
        res = []
        if root:
            res = self.inorderTraversal(root.left)
            res.append(root.data)
            res = res + self.inorderTraversal(root.right)
        return res
```

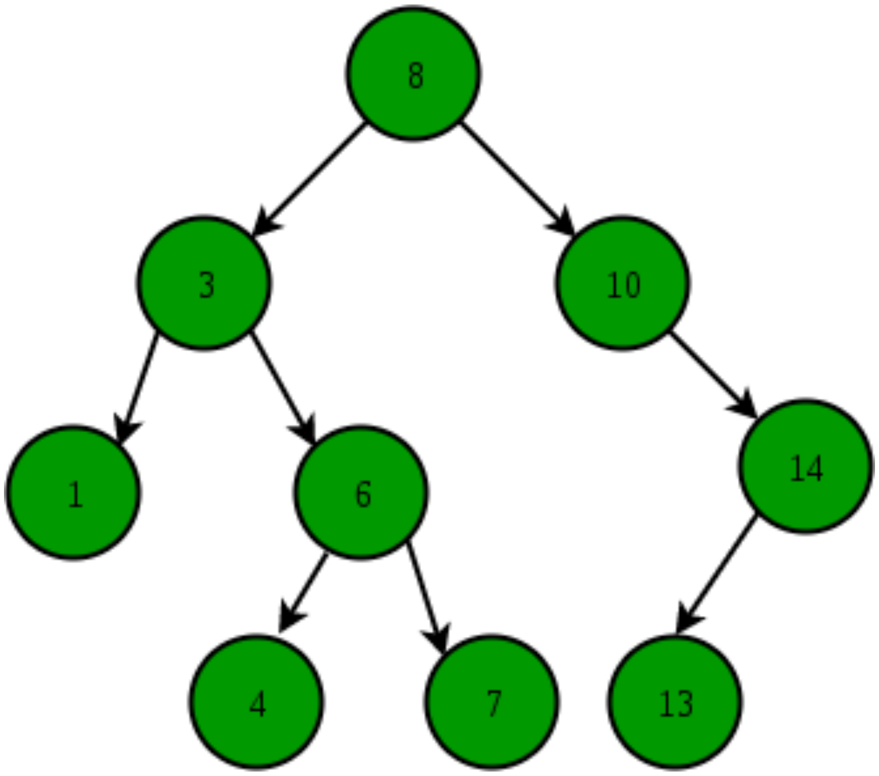
```
# Preorder traversal
# Root -> Left ->Right
    def PreorderTraversal(self, root):
        res = []
        if root:
            res.append(root.data)
            res = res + self.PreorderTraversal(root.left)
            res = res + self.PreorderTraversal(root.right)
        return res
```

It's valid to notice that all those methods are kind of variants of the depth first search.

Search on Binary Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

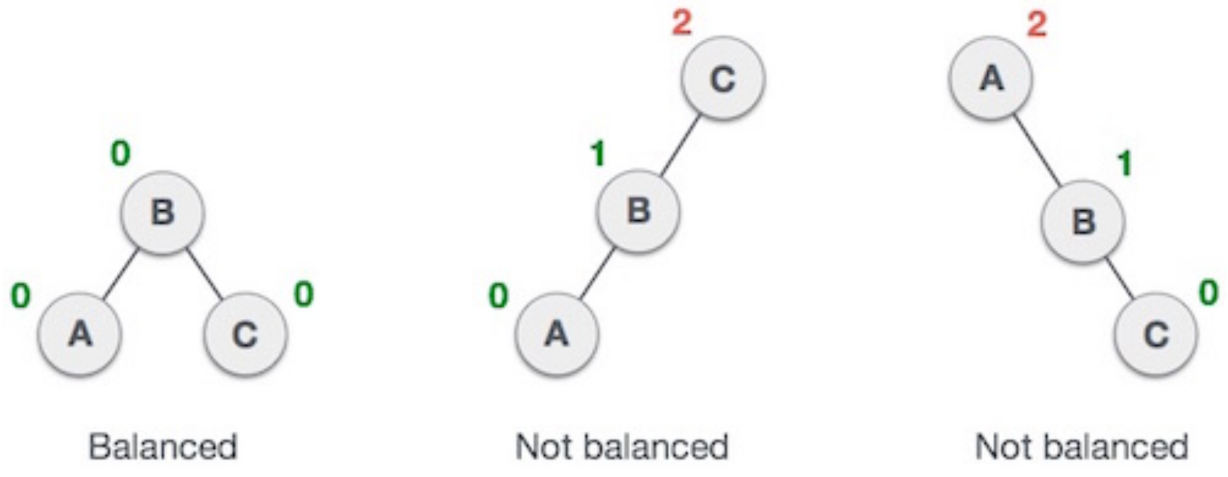
- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.



```python
# A utility function to search a given key in BST
def search(root,key):

    # Base Cases: root is null or key is present at root
    if root is None or root.val == key:
        return root

    # Key is greater than root's key
    if root.val < key:
        return search(root.right,key)

    # Key is smaller than root's key
    return search(root.left,key)
```

Balanced Search Tree

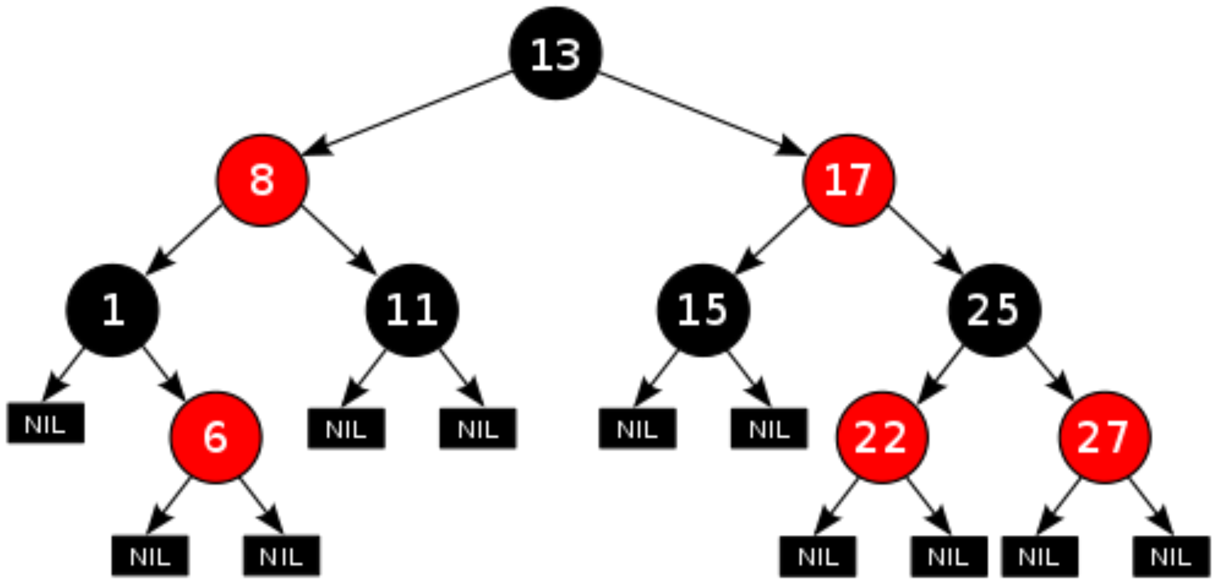
The advantage of storing the data as trees is that its operations will be time complexity O(log(n)) if the tree is balanced, otherwise the tree might become a linked list and the time complexity will be O(n).



Red Black Tree

It's a way to organize the tree so it will always be balanced, it's criteria:

- Nodes are Red or black
- The Root and (NIL) are always black
- If a node is red all it's children will be black
- All paths from a node to it's leaves contain the same number of black nodes
- Our node class will need a color bit, but the space complexity is still O(n)
- The shortest path has all black nodes
- The longest path alternates between Red and Black

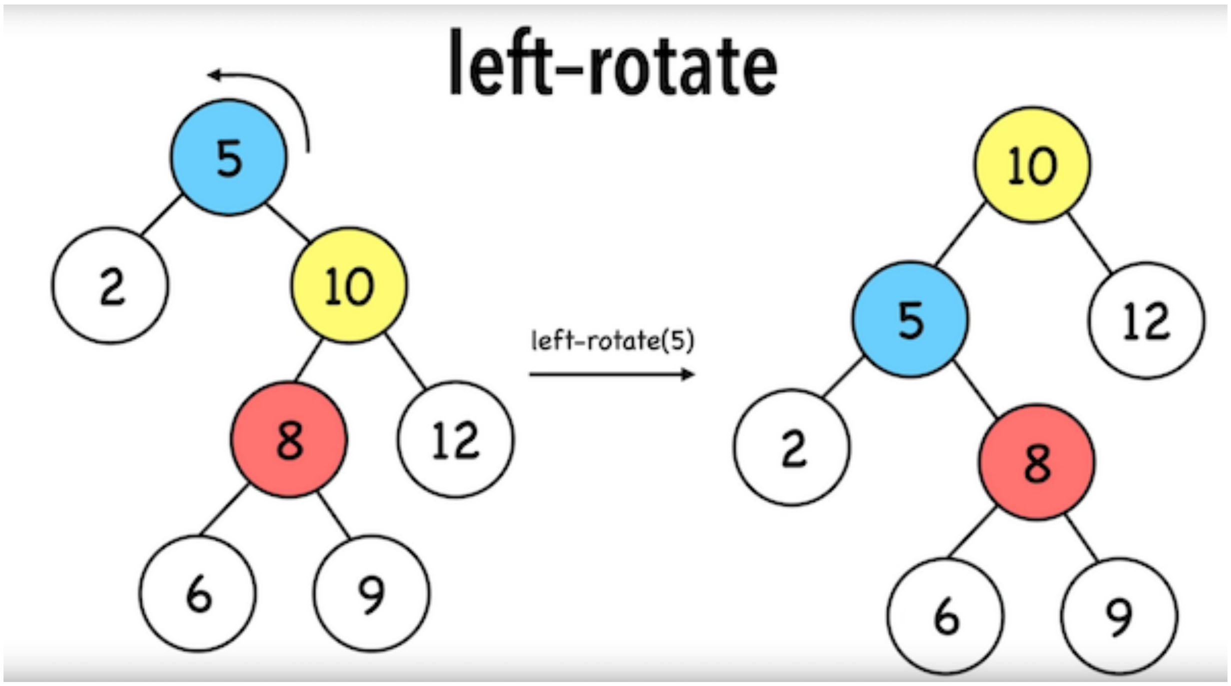


The time complexity of Search/Insert/Remove are all O(log(n)), but Insert/Remove requires rotation to keep the criteria true.

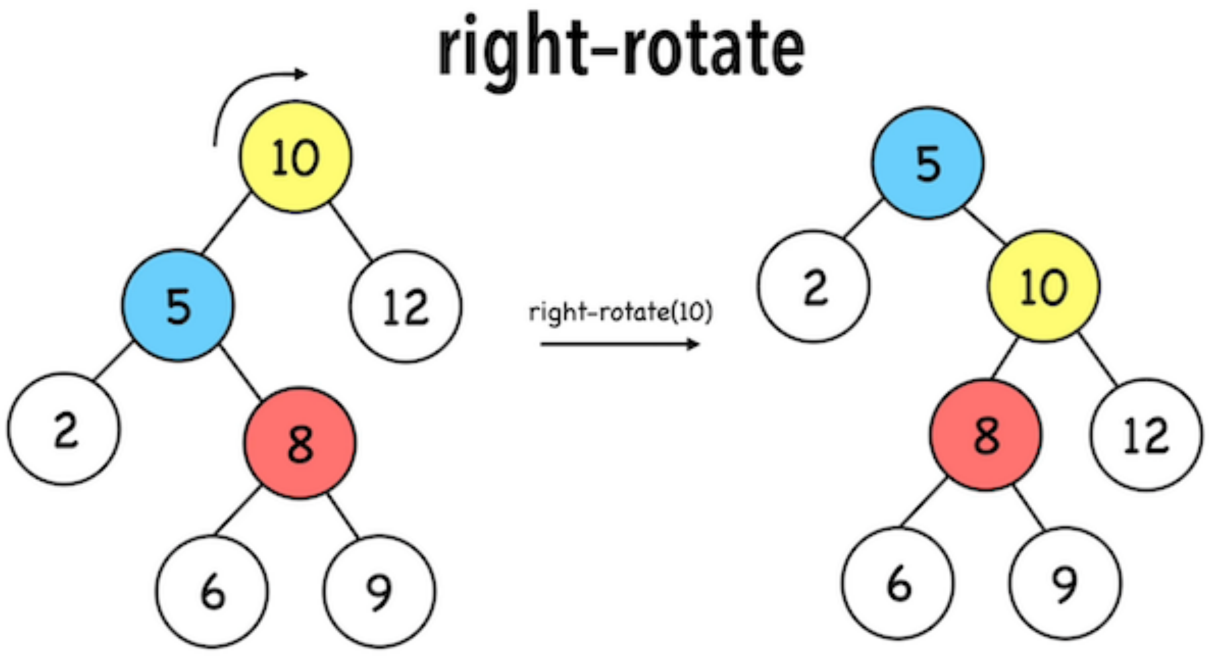
Rotation

There are 2 types of rotations on Red-Black-Trees, they are needed to keep the red/black tree criteria's and need to be applied after insert/remove operations

- Left-Rotate
- Right-Rotate



```
LEFT-ROTATE$(T, x)$
$y = x.right$                    // set $y$
$x.right = y.left$               // turn $y$'s left subtree into $x$'s right subtree
if $y.left \neq T.nil$
    $y.left.p = x$
$y.p = x.p$                      // link $x$'s parent to $y$
if $x.p == T.nil$
    $T.root = y$
elseif $x == x.p.left$
    $x.p.left = y$
else $x.p.right = y$
$y.left = x$                     // put $x$ on $y$'s left
$x.p = y$
```



References

- https://www.youtube.com/watch?v=yC83Kp2xig8
- https://medium.com/basecs/how-to-not-be-stumped-by-trees-5f36208f68a7
- https://www.youtube.com/watch?v=qvZGUFHWChY
- https://en.wikipedia.org/wiki/Binary_tree
- https://www.youtube.com/watch?v=95s3ndZRGbk
- https://github.com/Bibeknam/algorithmtutorprograms/blob/master/data-structures/red-black-trees/red_black_tree.py
- https://www.tutorialspoint.com/python_data_structure/python_tree_traversal_algorithms.htm
- https://medium.com/@codingfreak/binary-tree-interview-questions-and-practice-problems-439df7e5ea1f