We will now consider data structures which are used to store and lookup data in an efficient manner.

We may want these data structures to be dynamic, that is, we should be able to insert and delete elements as needed.

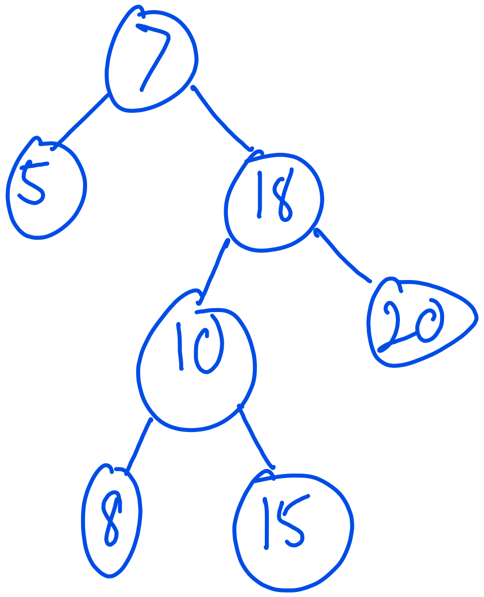
We want to be able to perform various operations on the data structure. For example:

- Search
- Max
- Min
- Insert
- Delete

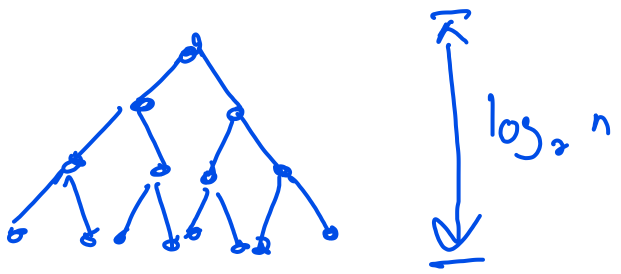
We want these operations to be computed as quickly as possible.

A **binary search tree** is a special type of binary tree in which each node stores a value such that the following properties hold for a node x:

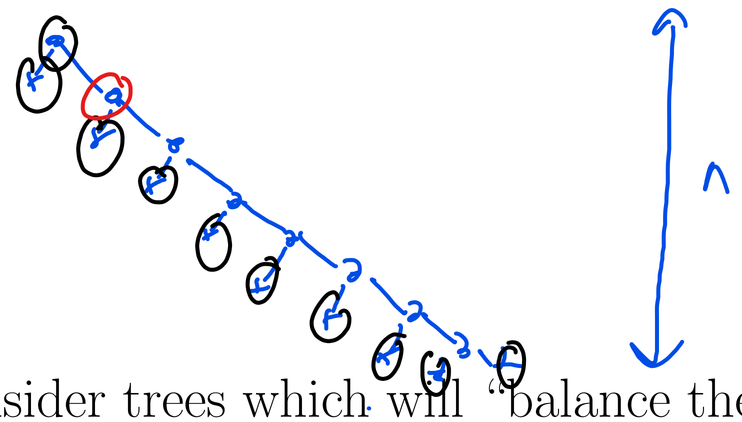
1. For each node y in the left subtree of x, we have $y \leq x$.
2. For each node y in the right subtree of x, we have $x < y$.



If a binary tree with n nodes has height $O(\log n)$, then we can find any node in the tree (or determine that it isn't there) in time $O(\log n)$.



Since we want the tree to be dynamic, a "bad case" sequence of inserts can leave us with a tree of height n.



We will consider trees which will "balance themselves" in the event that we see a bad sequence of inserts (or deletions).

Red-Black Tree

Satisfies these properties.

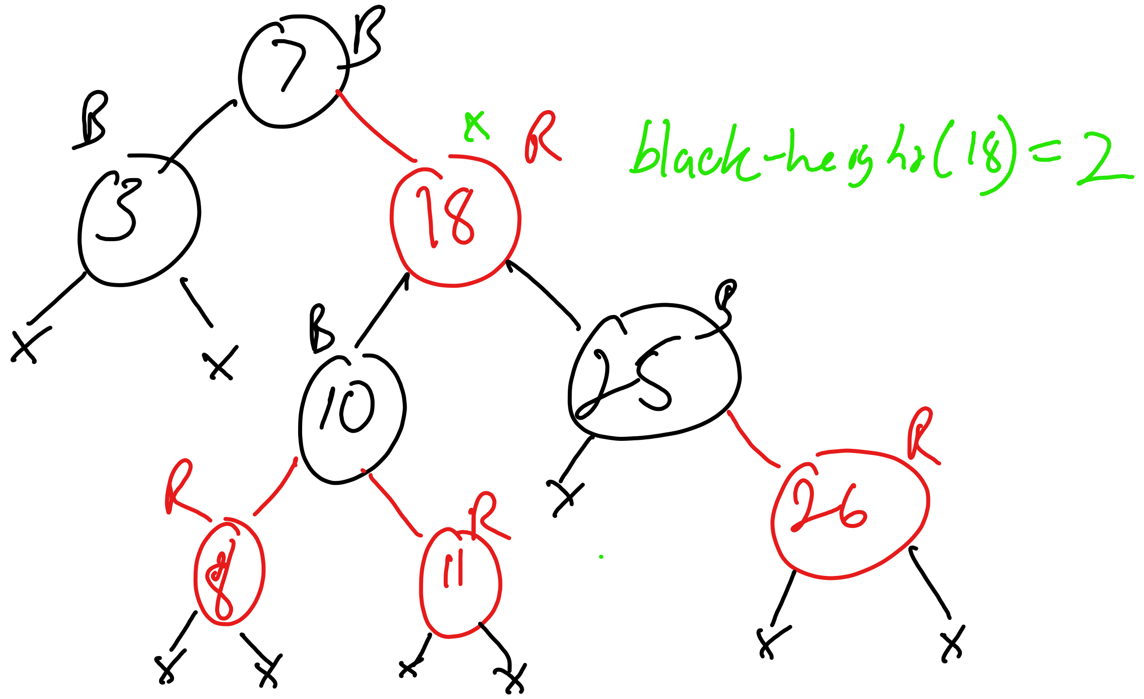
1) Every node is red or black.

2) Root is black.

3) Leaves (null) are black.

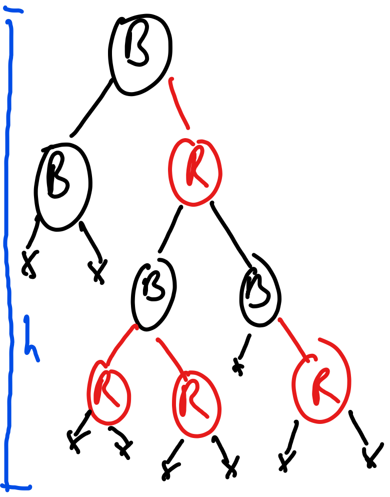
4) If a node is red, both children are black.

5) All simple paths from any node x, excluding x, to any descendant leaf has the same # of black nodes on it.
black-height(x)



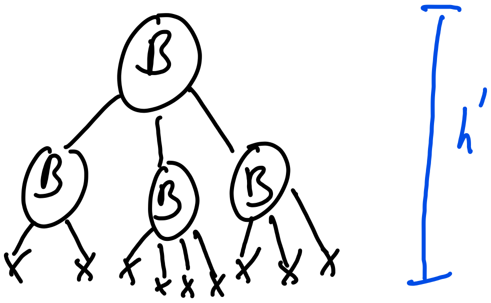
Thm

A RB-tree with n keys has height $h \le 2\log(n+1)$



Merge red nodes into black parents

$\Longrightarrow$



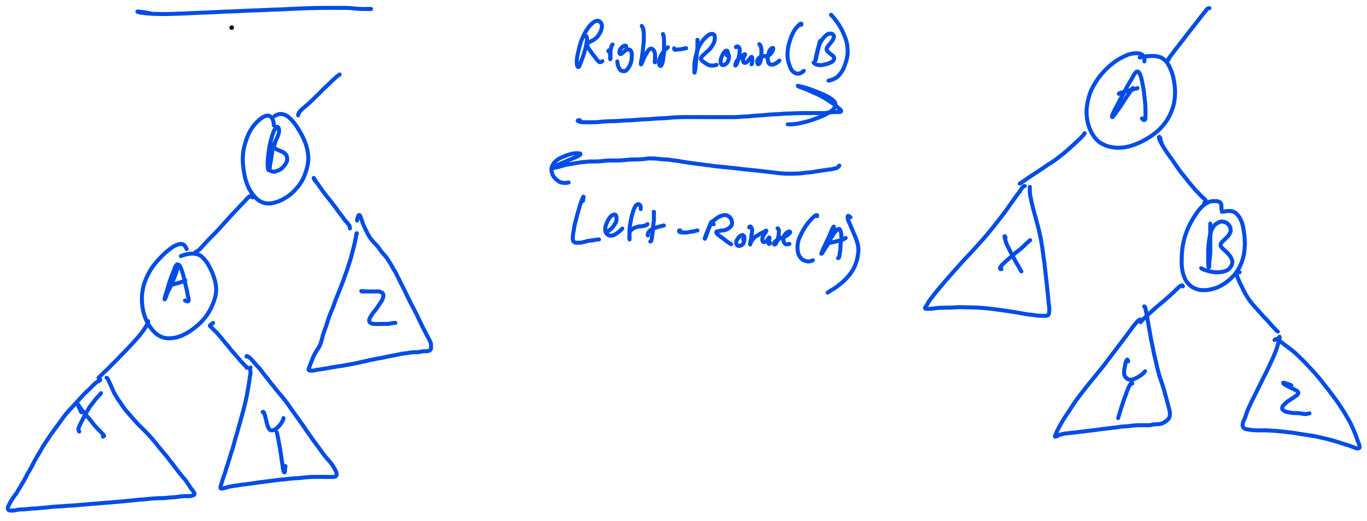
For each path of length h, there are $\le h/2$ red nodes (property 4).
So $h' \ge h/2$

Right tree has all null leaves at the same height (property 5), so it is perfectly balanced.

The number of null leaves is $\le n+1$.

$n+1 \ge 2^{h'} \Rightarrow \log_2(n+1) \ge h' \ge h/2 \quad \Rightarrow \boxed{h \le 2 \cdot \log_2(n+1)}$

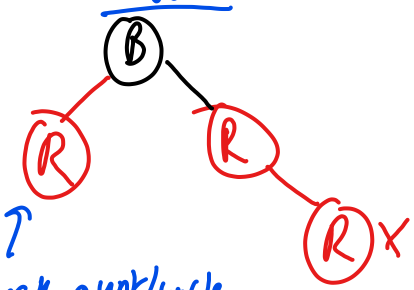
Rotations



$X < A < Y < B < Z$

Insert(X) - Insert X into tree as a leaf and color it red. Only property 4 could be violated. Assume it is.

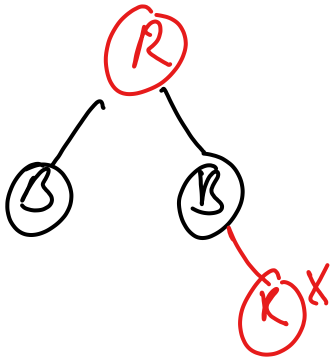
Case 1



Check aunt/uncle. If red we are in case 1.

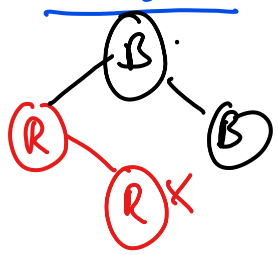
Swap colors of parent, grandparent, & aunt

$\Longrightarrow$



Repeat with grandparent as new x.

Case 2



aunt is black and the path from x to its grandparent is a zig-zag

Left-Rotate(parent)