Balanced binary search trees

1. Background and motivation
2. Rotations
3. AVL trees
4. Red-black trees

1. Background and motivation

Two numbers associated with any binary tree:

$$n = \text{number of nodes in binary tree}$$
$$h = \text{height of binary tree}$$

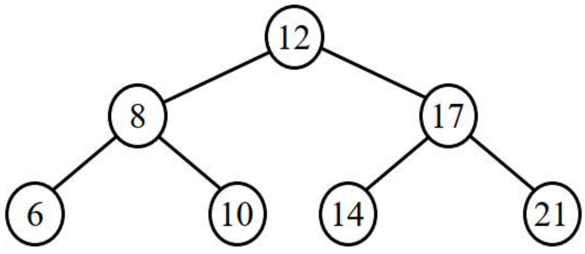
Always true: $2^{h+1} - 1 \geq n \quad \text{or} \quad h \geq \lceil \log_2(n+1) - 1 \rceil \geq \lfloor \log_2(n) \rfloor$

Equality is true for perfect binary tree: all interior nodes have 2 children and all leaves are at same depth.

Example

a perfect binary tree with

$h = 2$ and $n = 7$



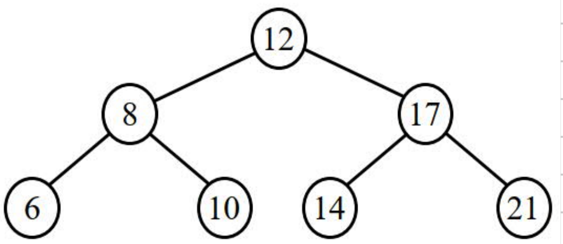
□

Q: What is the runtime efficiency of a **search** operation on a BST?

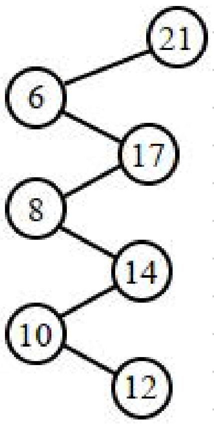
A: $O(h)$

Example

best case: *h* is as small as possible



worst case: *h* is as large as possible



□

Q: For a given n, what is the smallest that h can be?

A: $h \geq \lfloor \log_2(n) \rfloor$

Q: For a given n, what is the largest that h can be?

A: $h \leq (n-1)$

Consequence: **search** is $O(n)$ (same as for unsorted array)

Q: Are we interested only in **search**?

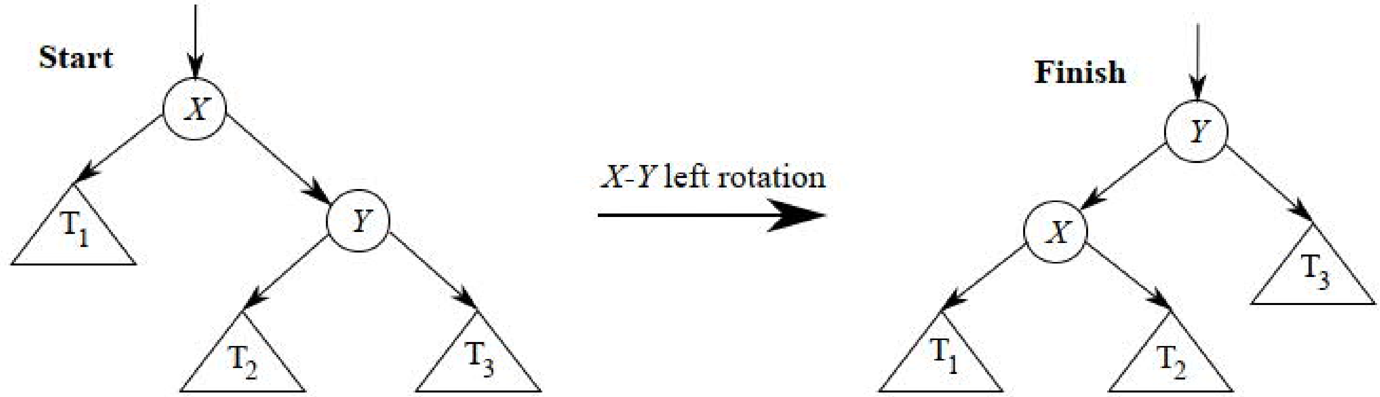
A: No. But operations such as **insert**, **delete**, **max** and **min** are extensions or modifications of **search**.

Conclusions
- to avoid $O(n)$ runtime efficiency, we want BSTs to be balanced

- a mechanism is needed to help balance an unbalanced BST

2. Rotations

Left rotation



BST property preserved:

$$\text{both at } \textbf{start} \text{ and } \textbf{finish} : \quad T_1.key < X.key < T_2.key < Y.key < T_3.key$$

Consider heights

$$h(T_i) = \text{height of subtree } T_i,\ i = 1, 2, 3$$

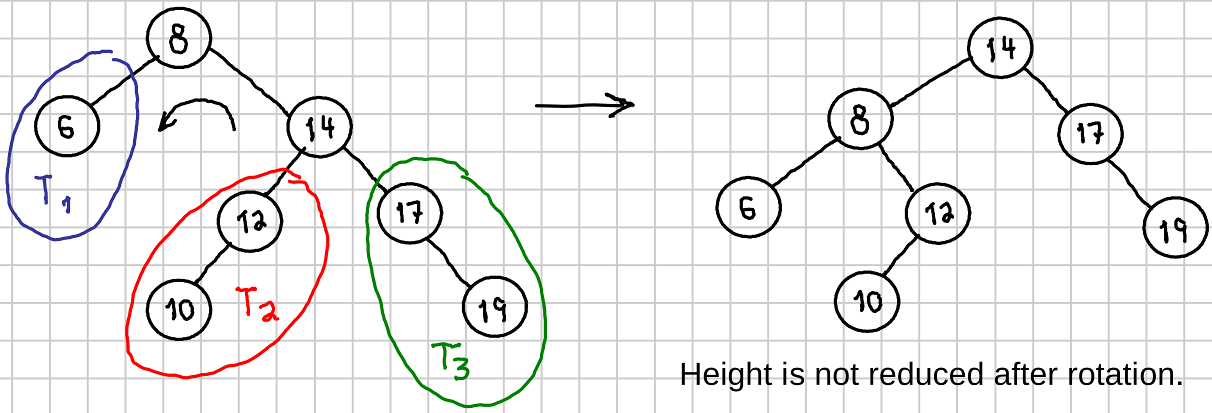
$$h_{start} = \text{height of subtree rooted at } X = 1 + \max(h(T_1),\ 1 + \max(h(T_2), h(T_3)))$$

$$h_{finish} = \text{height of subtree rooted at } Y = 1 + \max(h(T_3),\ 1 + \max(h(T_1), h(T_2)))$$

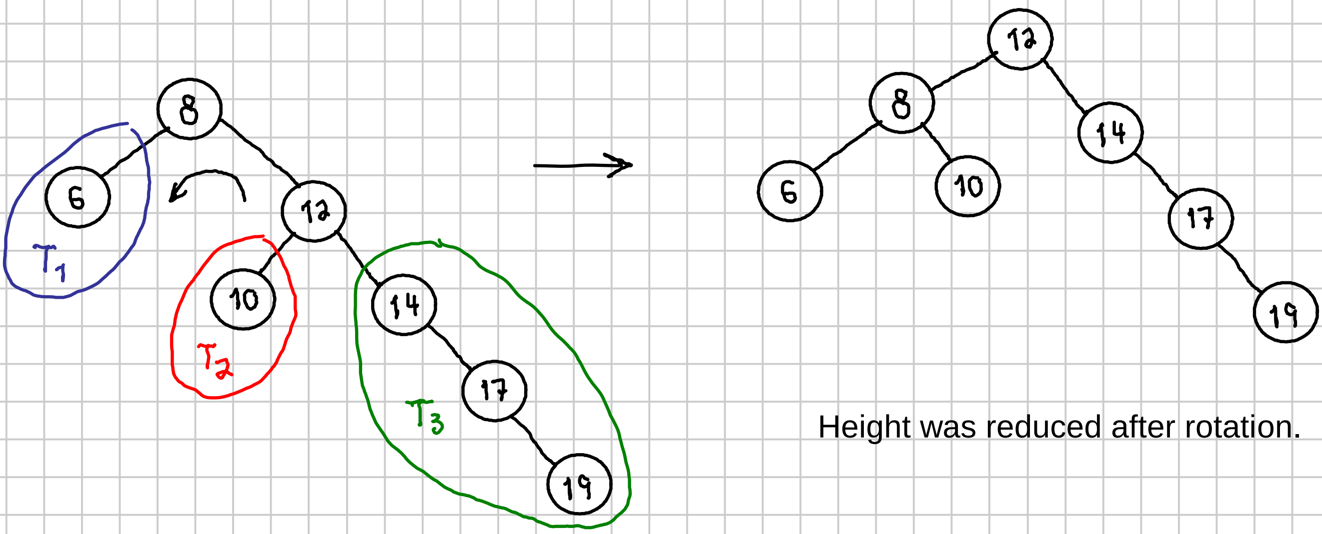
For decrease in total height we need $h_{finish} < h_{start}$:

$$\max(h(T_3),\ 1 + \max(h(T_1), h(T_2))\,) < \max(h(T_1),\ 1 + \max(h(T_2), h(T_3))\,)$$
$$\Rightarrow h(T_3) > h(T_2) \quad \text{and} \quad h(T_3) > h(T_1)$$

Example



Height is not reduced after rotation.



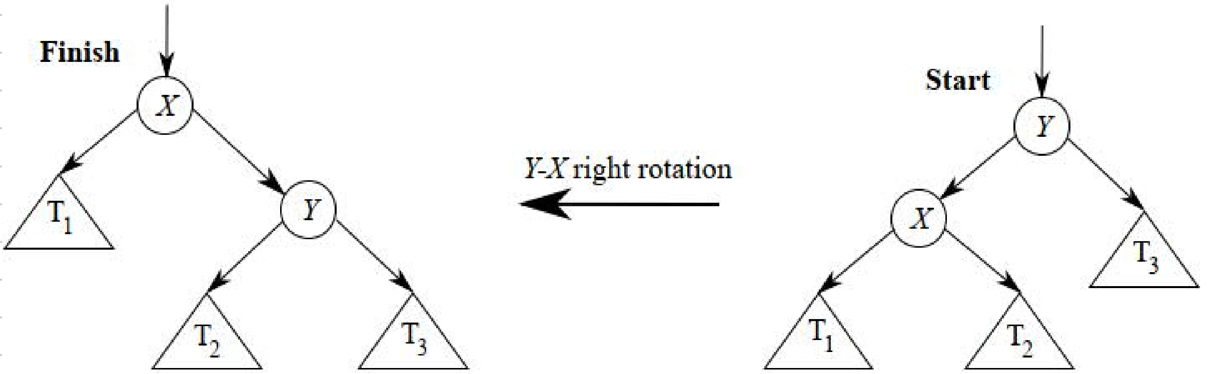
Height was reduced after rotation.

$\square$

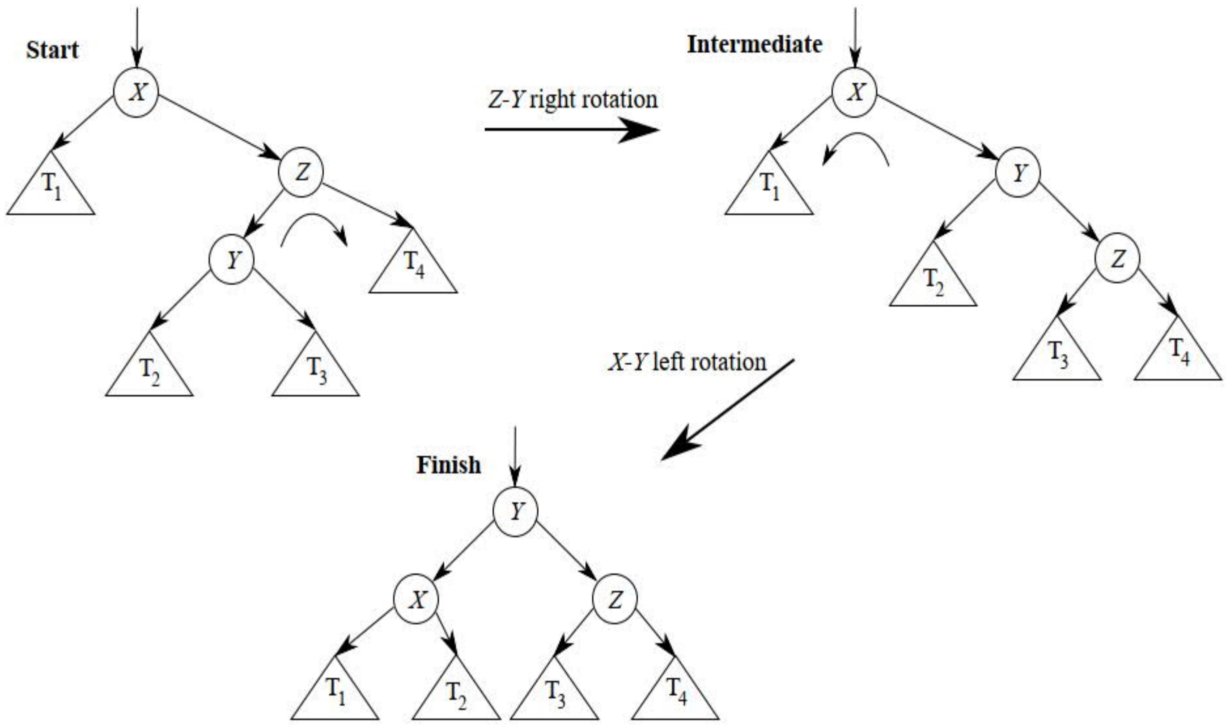
Q: Why would we want to perform rotations?

A: To decrease the height of the BST and hence improve performance.

Right rotation



Right-left double rotation



Left-right double rotation

... left to viewer

3. AVL trees

Q: What is a height-balanced (BST)?
A: In a height-balanced BST, each node x has the following property:

The difference between the heights of the two subtrees of x is at most 1.

An AVL-tree is a BST which is height-balanced. When a BST becomes unbalanced, rotations are performed to restore the height-balance property.

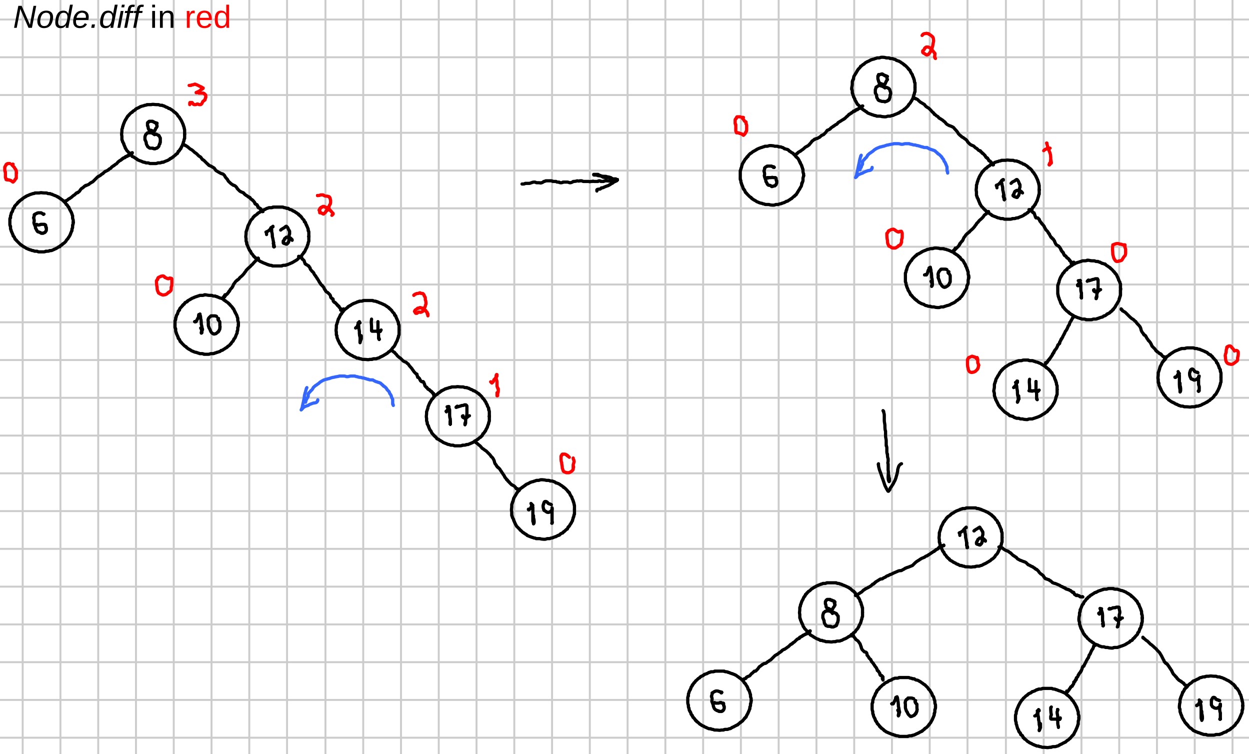
When computing an AVL-tree node attributes:

- $Node.key$ = the key of a node
- $Node.parent$ = pointer to parent of Node or NIL if root
- $Node.left$ = pointer left child for binary tree or NIL if child does not exist
- $Node.right$ = pointer right child for binary tree or NIL if child does not exist
- $Node.hdiff$ = the difference in the heights of the left and right subtrees

$$Node.diff = \begin{cases} < 0 & \text{when the height of the left subtree is greater} \\ 0 & \text{when the two subtrees have equal height} \\ > 0 & \text{when the height of the right subtree is greater} \end{cases}$$

Example

Node.diff in red



□

Typical code when forming AVL trees:

```
if x.hdiff = 2 then
  perform left rotation with x and x.right
else
  if x.hdiff = −2 then
    perform right rotation with x and x.left
  end
end
x.hdiff = 0
```

Detailed example from Wikipedia:

By Bruno Schalch - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=64250599

For BST formed using AVL techniques:

$$h < \log_\phi(n+2), \text{ where } \phi = \frac{1+\sqrt{5}}{2} \approx 1.62$$

Conclusion: **search** (and other operations) in AVL are *O*(log2(*n*))

4. Red-black trees

A red-black BST has the following properites:

property 1 Each node is colored either red or black.

property 2 A node is red does not have a red child.

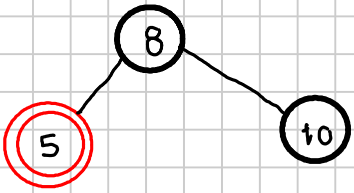
property 3 For each node x, all paths from x to any leaf contain the same number of black nodes.

property 4 The root is colored black.

Example

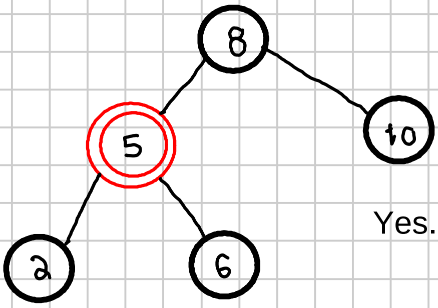
Is the BST a red-black tree?

(i)



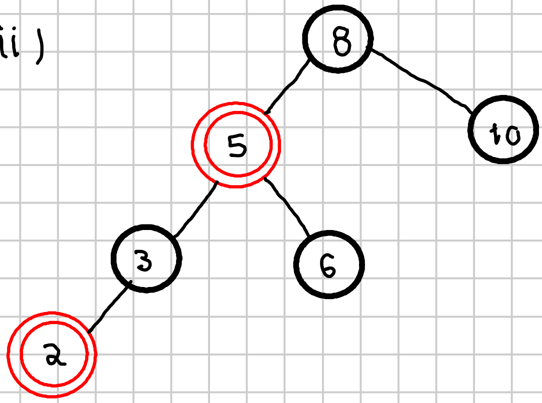
No. It does not have property 3.

(ii)



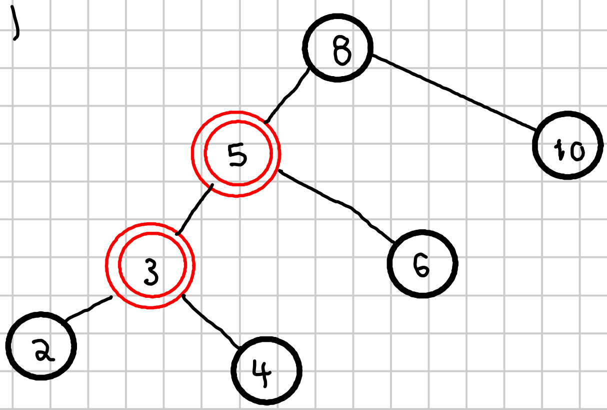
Yes.

(iii)



Yes.

(iv)



No. It does not have property 2.

□

For each node *x*, we get the following from the properties:

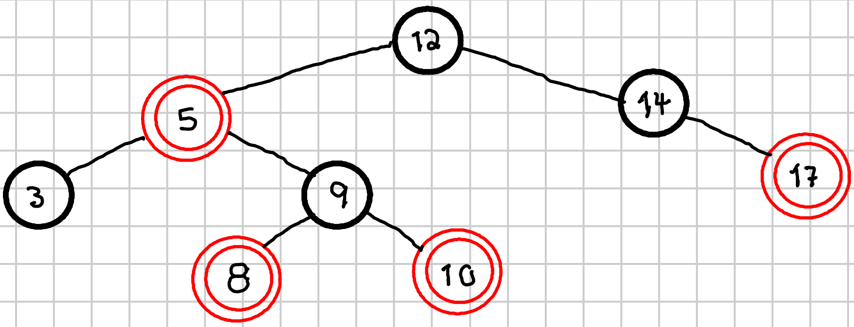
- in each path from x to a leaf, at most every second node is red
- the length of the longest path from x to a leaf is at most twice the length of the shortest path (balancing of some kind)

Red-black trees make use of two mechanisms fro maintaining properties:

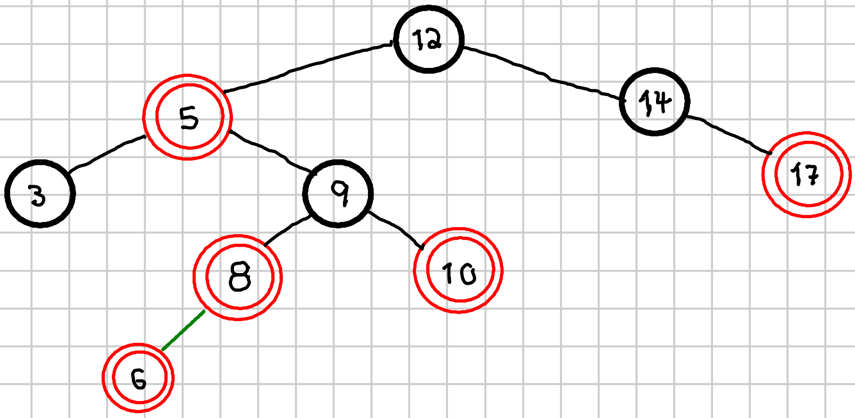
- recoloring
- rotations

Example

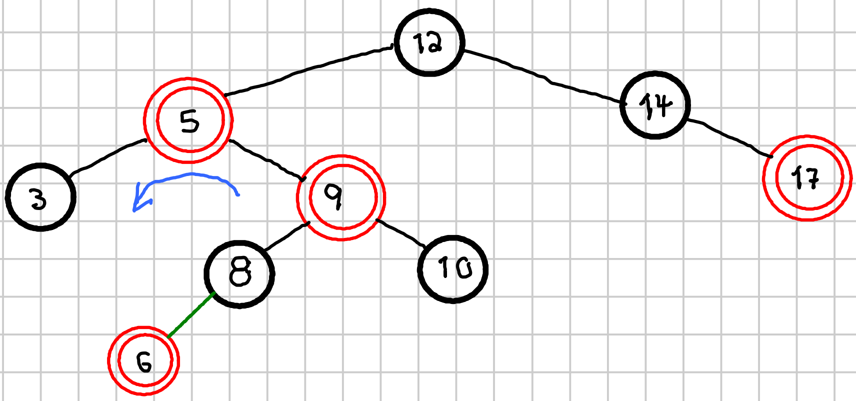
Add a node whose key is 6
to the following red-black tree



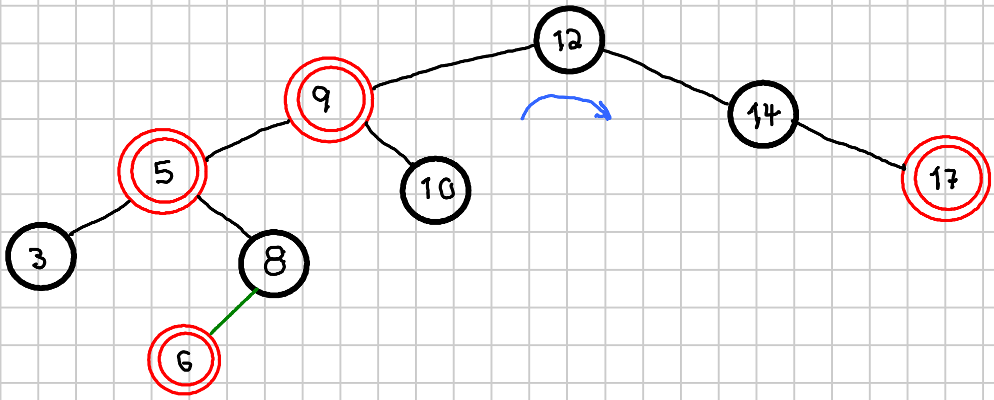
Stage 1
search for correct location and
try to add red node



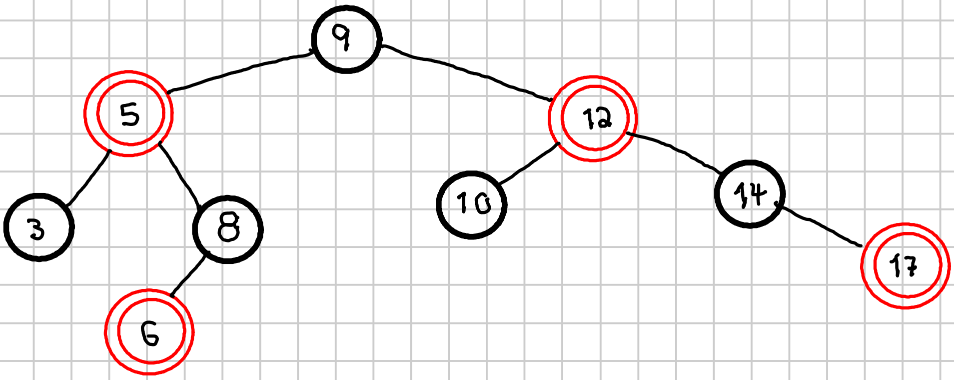
Stage 2: recoloring



Stage 3: rotation



Stage 4: recoloring root



For a red-black BST: $h < 2\log_2(n+2)$

Conclusion: **search** (and other operations) in red-black tree are *O*(log2(*n*))

Visualization of data structures:

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html