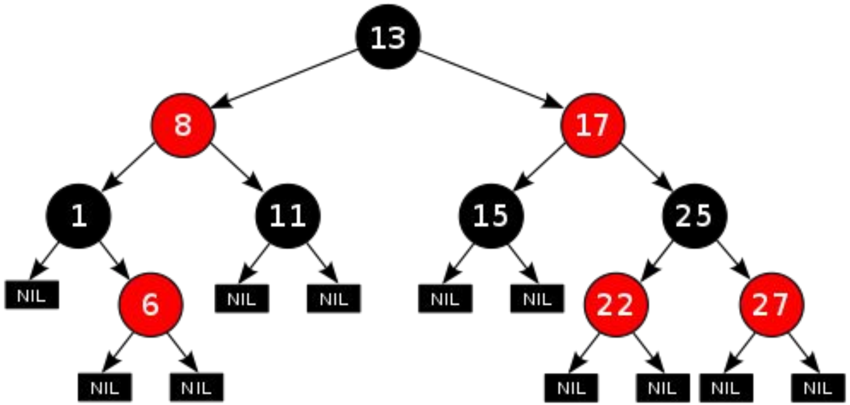
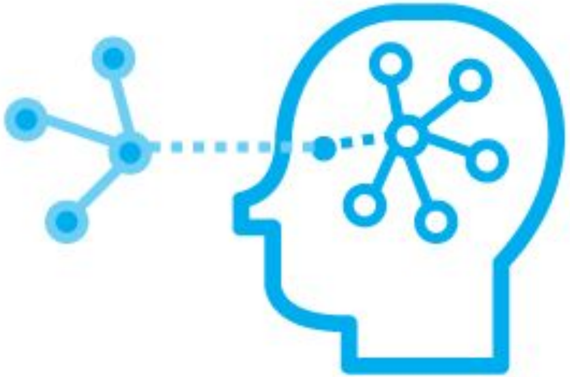


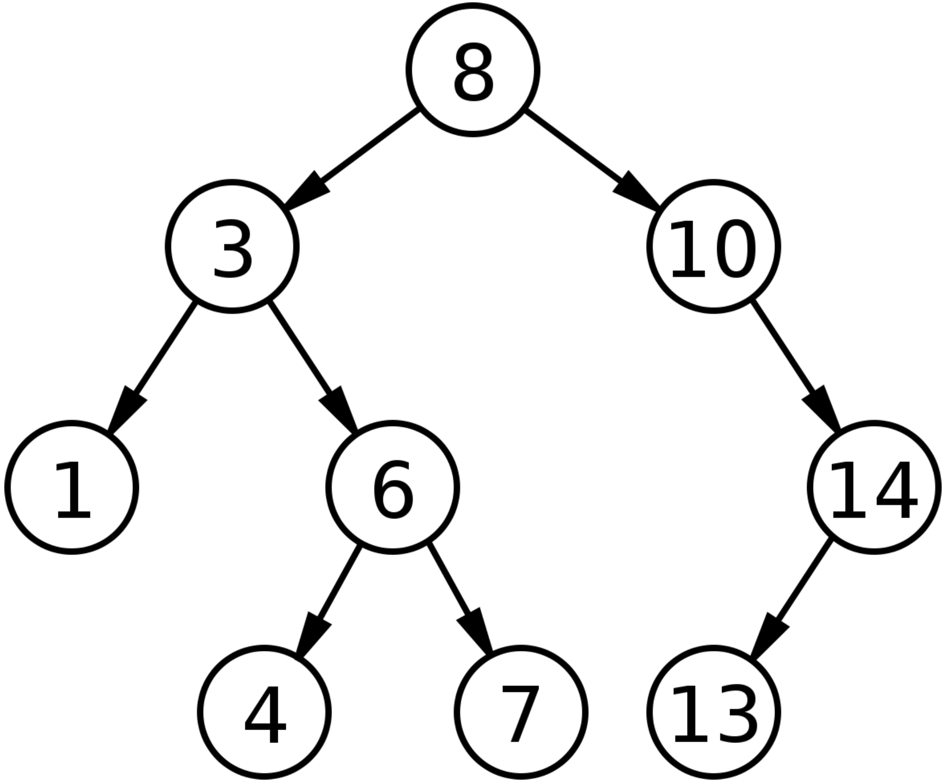
Red Black Tree

David Lazo- Humberto Bernal - Diego Enciso

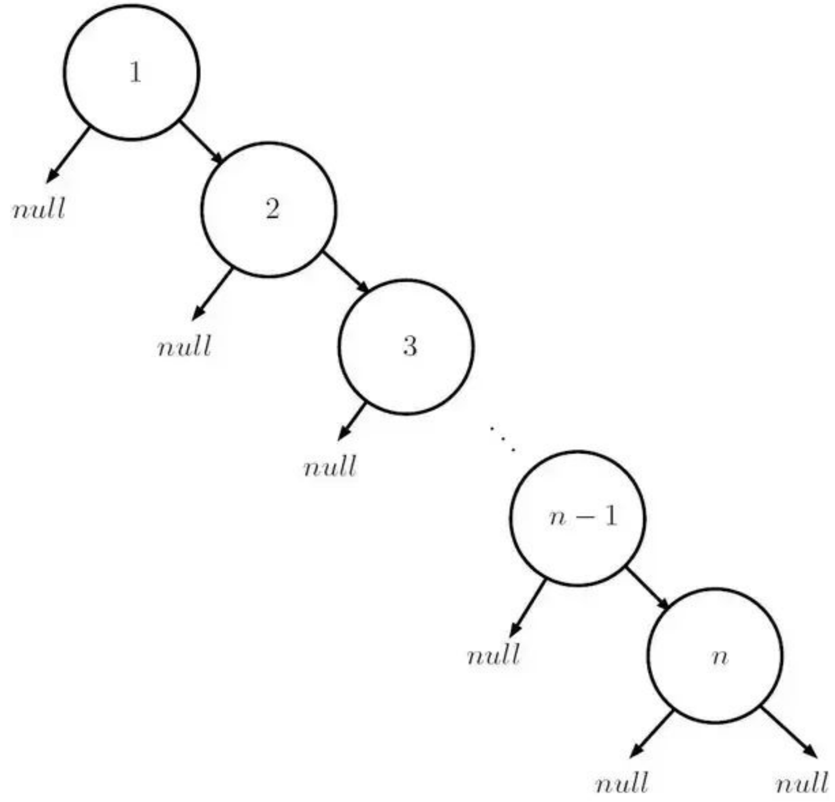
Content

Previous Knowledge

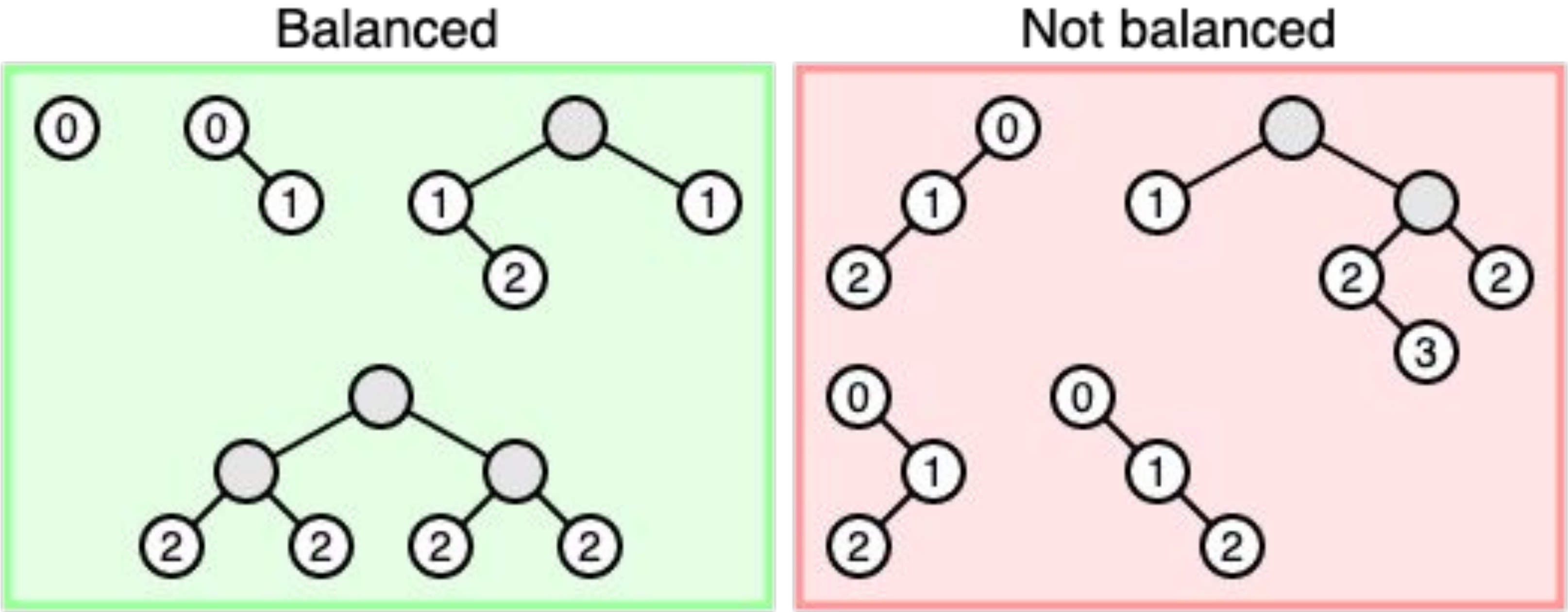




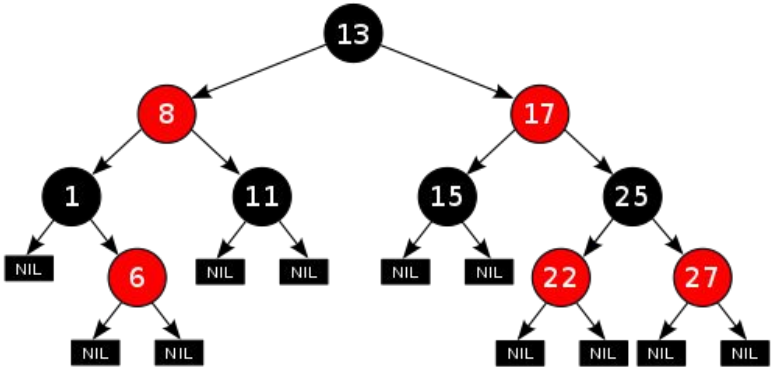
Binary Search Tree



BST worst case: Linked List



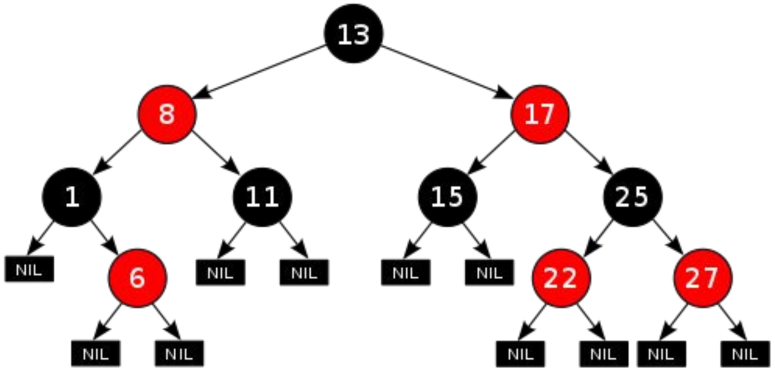
Balanced
Not balanced
0
0
1
1
1
2
2
2
2
2
0
1
2
1
2
2
3
0
1
2
0
1
2



Red Black Tree

Basic Rules

- A Node can be red or black.
- Root and leaves (NIL) are black.
- If a node is red, then his children are black.
- All paths from a node to its NIL descendents contain the same number of black nodes.

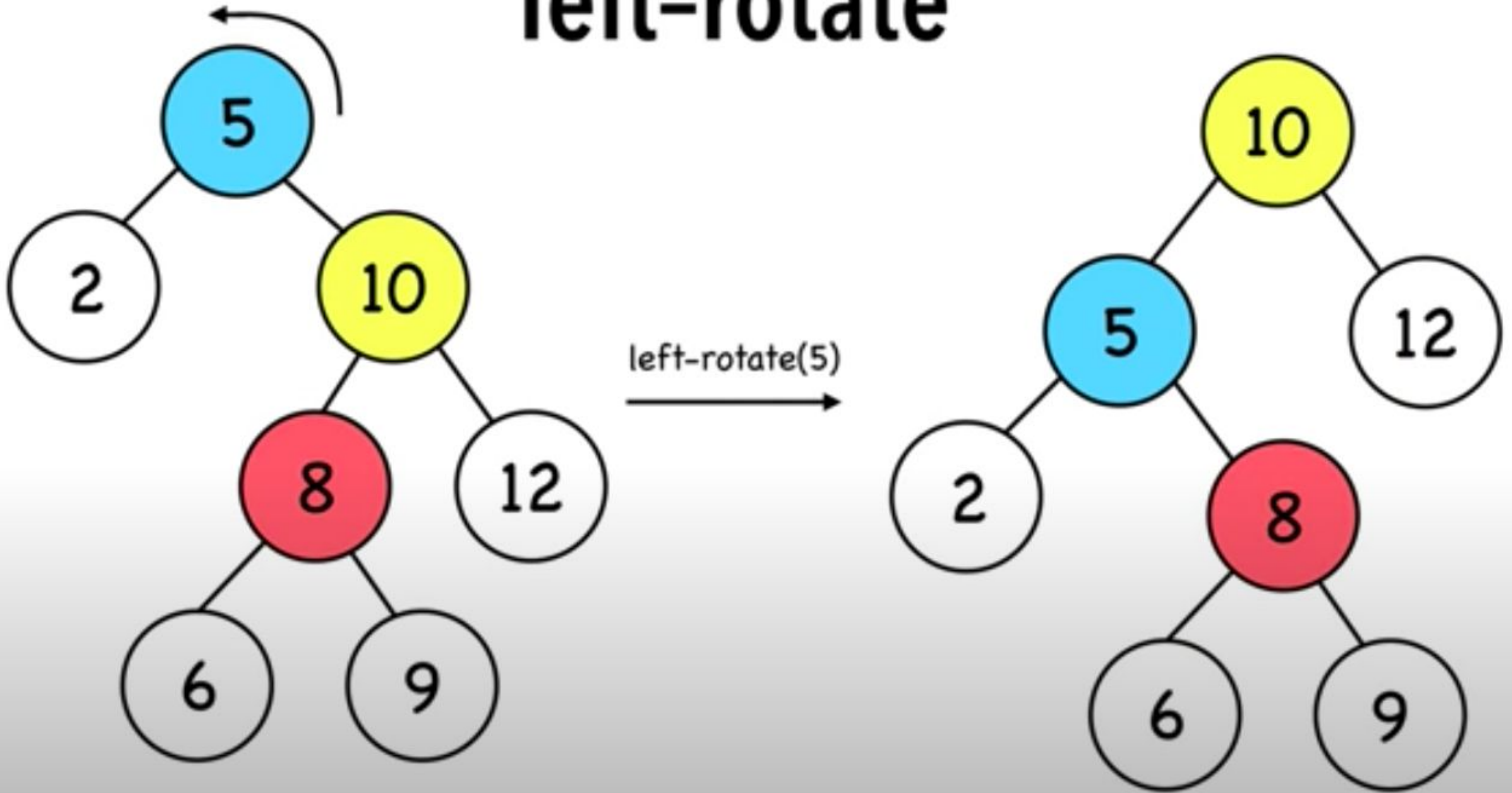


Rotations

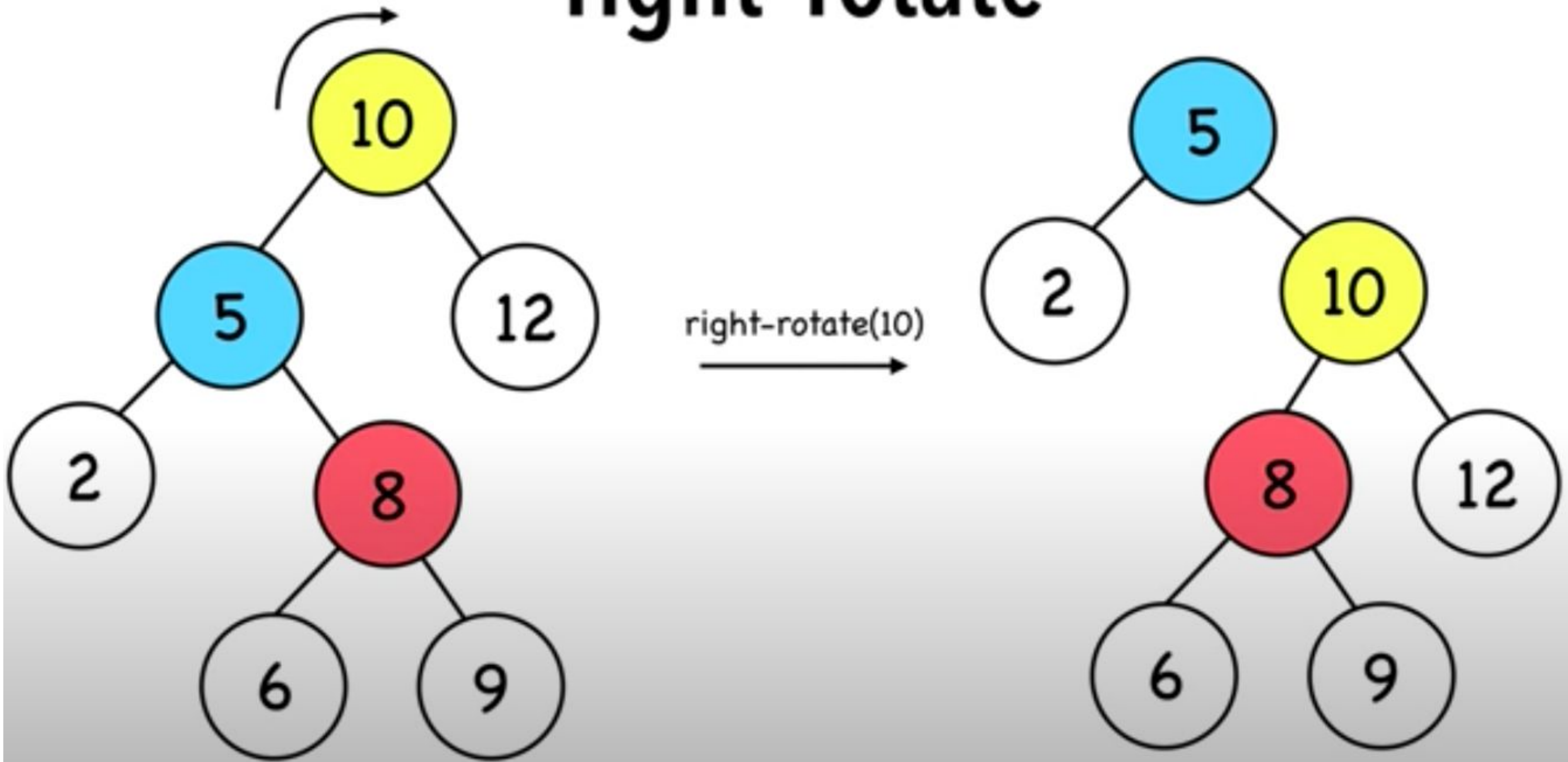
- Rearrange the subtrees
- Goal is to decrease the height of the tree:
  - red black trees: maximum height of O(log n)
  - larger subtrees up, smaller subtrees down
- does not affect the order of elements

O(1)

left-rotate



right-rotate



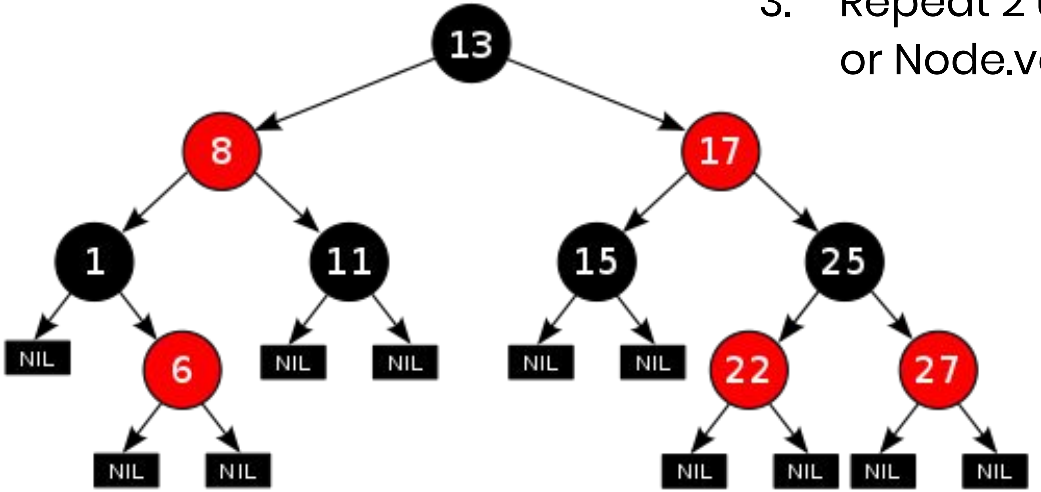
Operations

- Search
- Insert
- Remove

May result in violation of red-black tree properties. To fix this we'll use rotations.

Operations

- **Search**
- Insert
- Remove

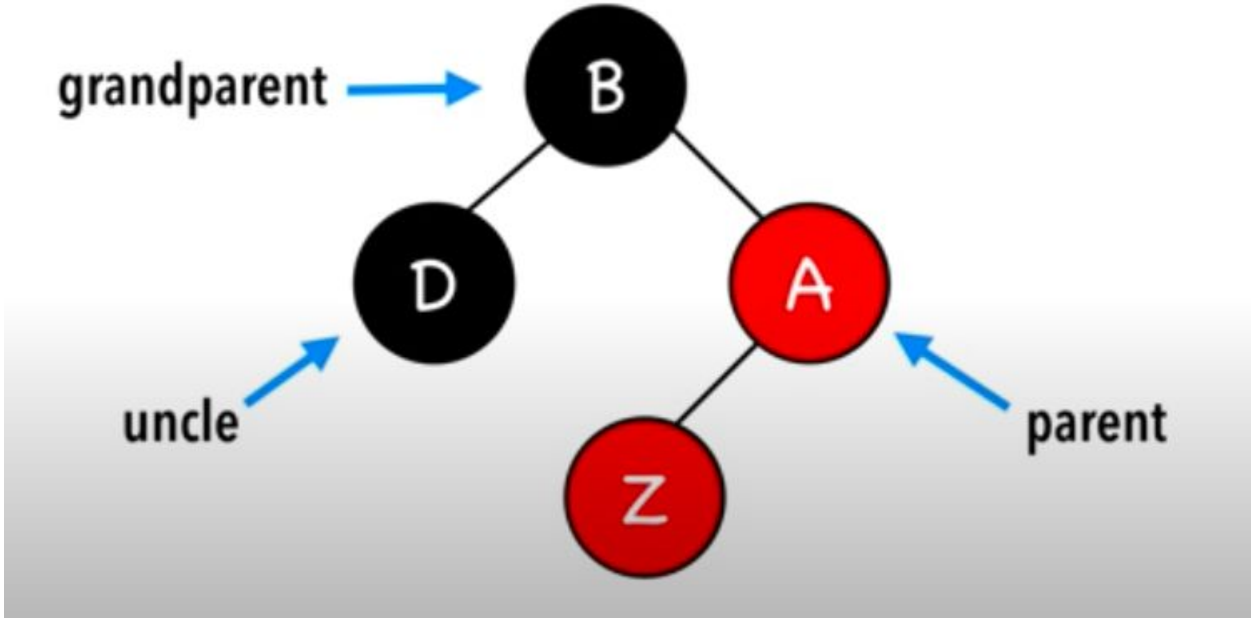


To find the number X:

1. Node starts in root
2. Loop
   a. Node = Node.left if x < Node.value
   b. Node = Node.right if x > Node.value
3. Repeat 2 until Node is NIL
   or Node.value == x is found.

Operations

- Search
- **Insert**
- Remove



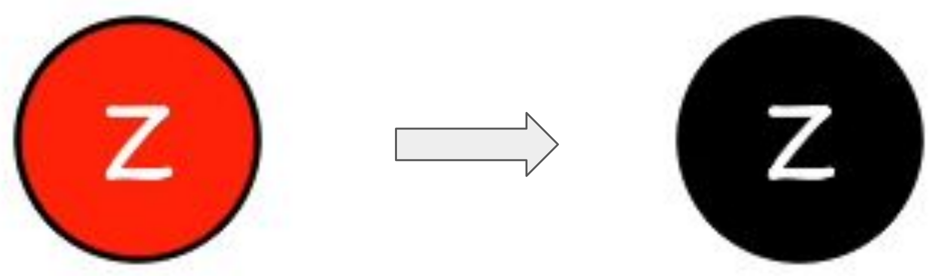
Insert Strategy

1. Insert Z and color it red
2. Recolor and rotate nodes to fix violation

4 Insert scenarios

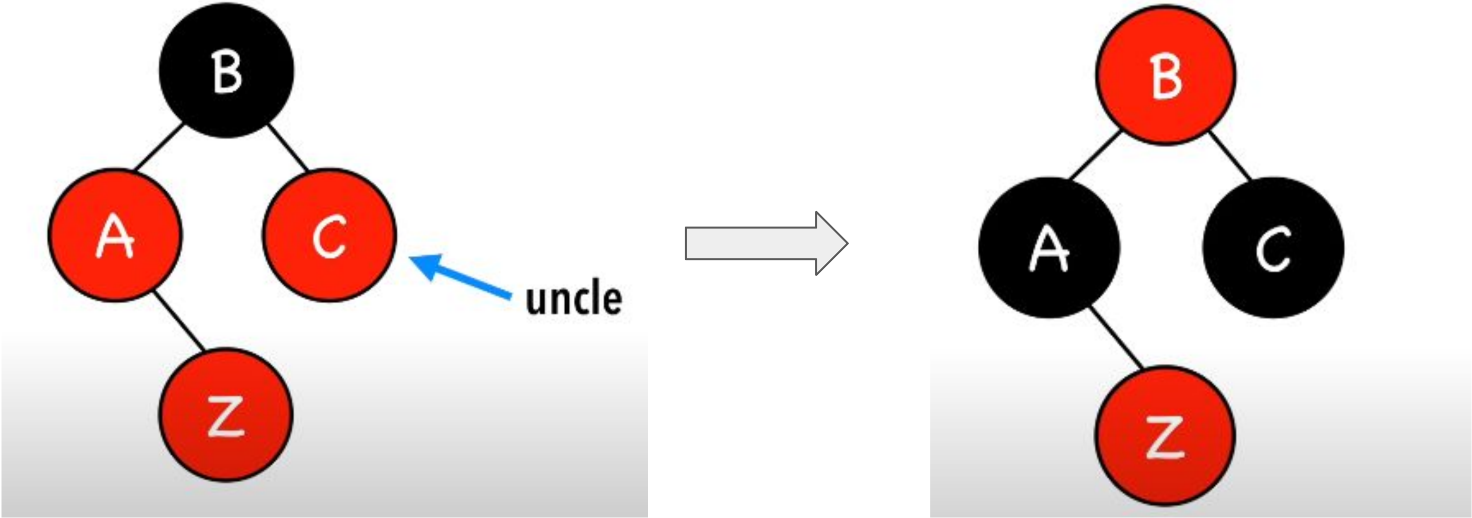
1. Z = root -> **color black**
2. Z . uncle = **red** -> **recolor**
3. Z . uncle = **black** (triangle) -> **rotate Z.parent**
4. Z . uncle = **black** (line) -> **rotate Z.grandparent & recolor**

Case 1: Z is root



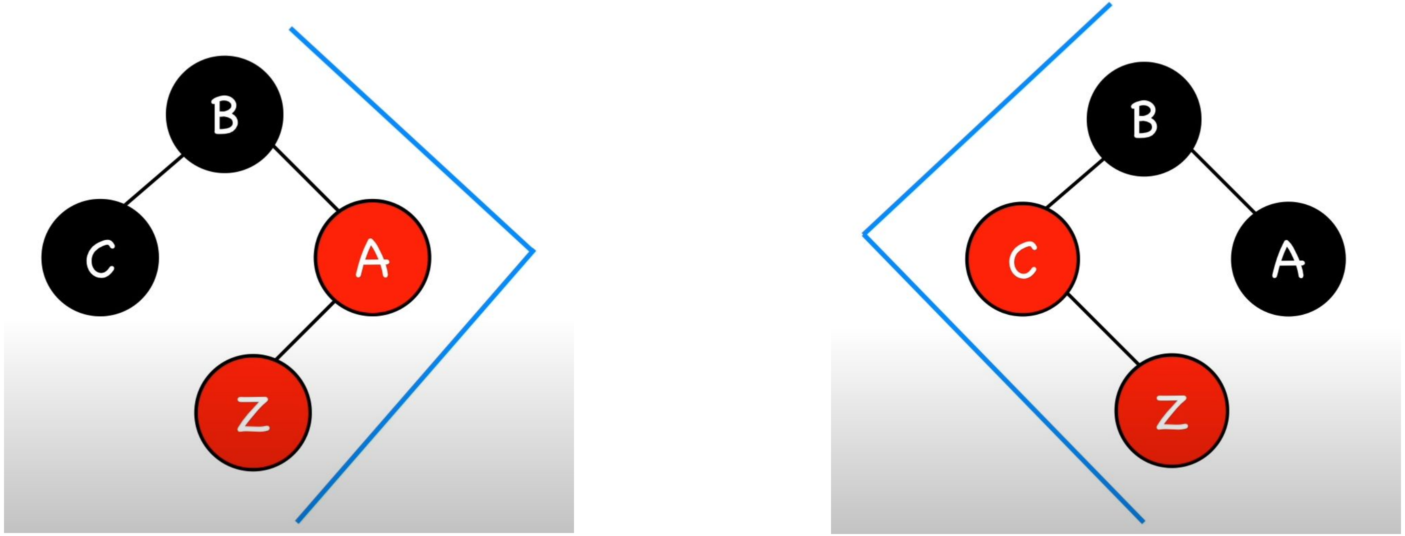
Solution: color black

Case 2: Z.uncle is red

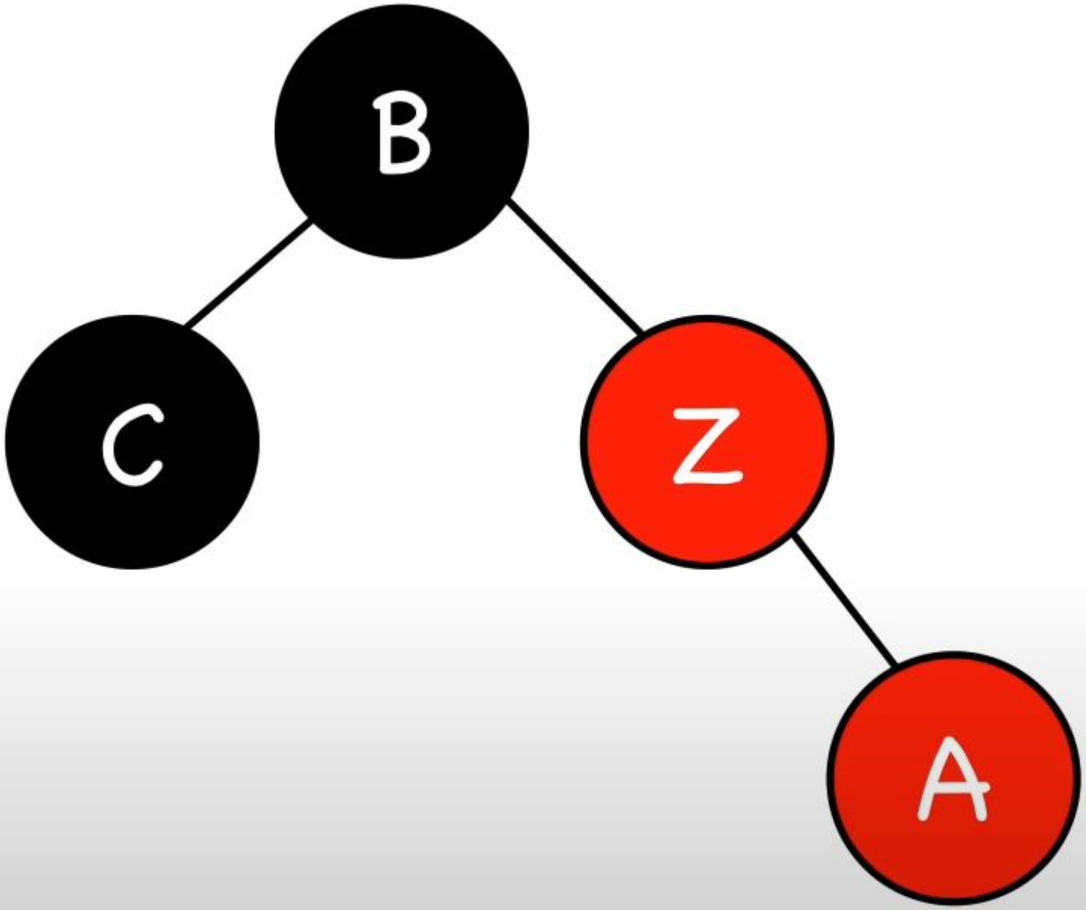


Solution: recolor

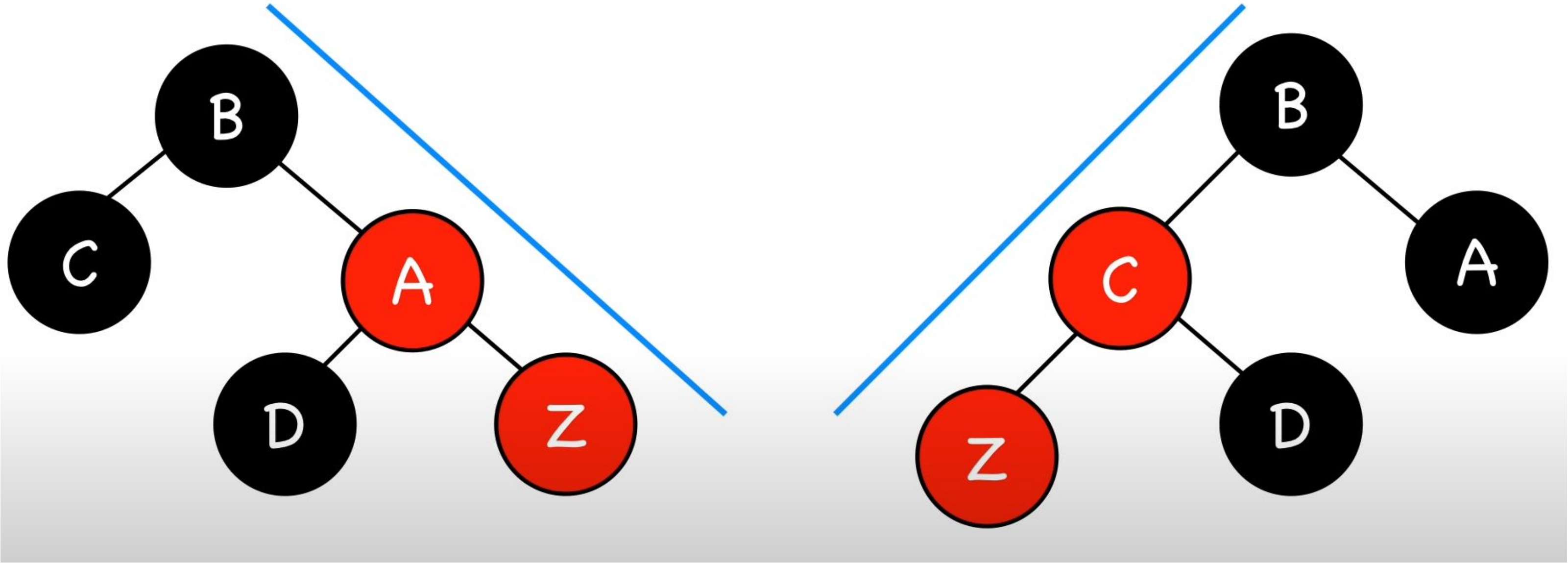
Case 3: Z.uncle is black (triangle)



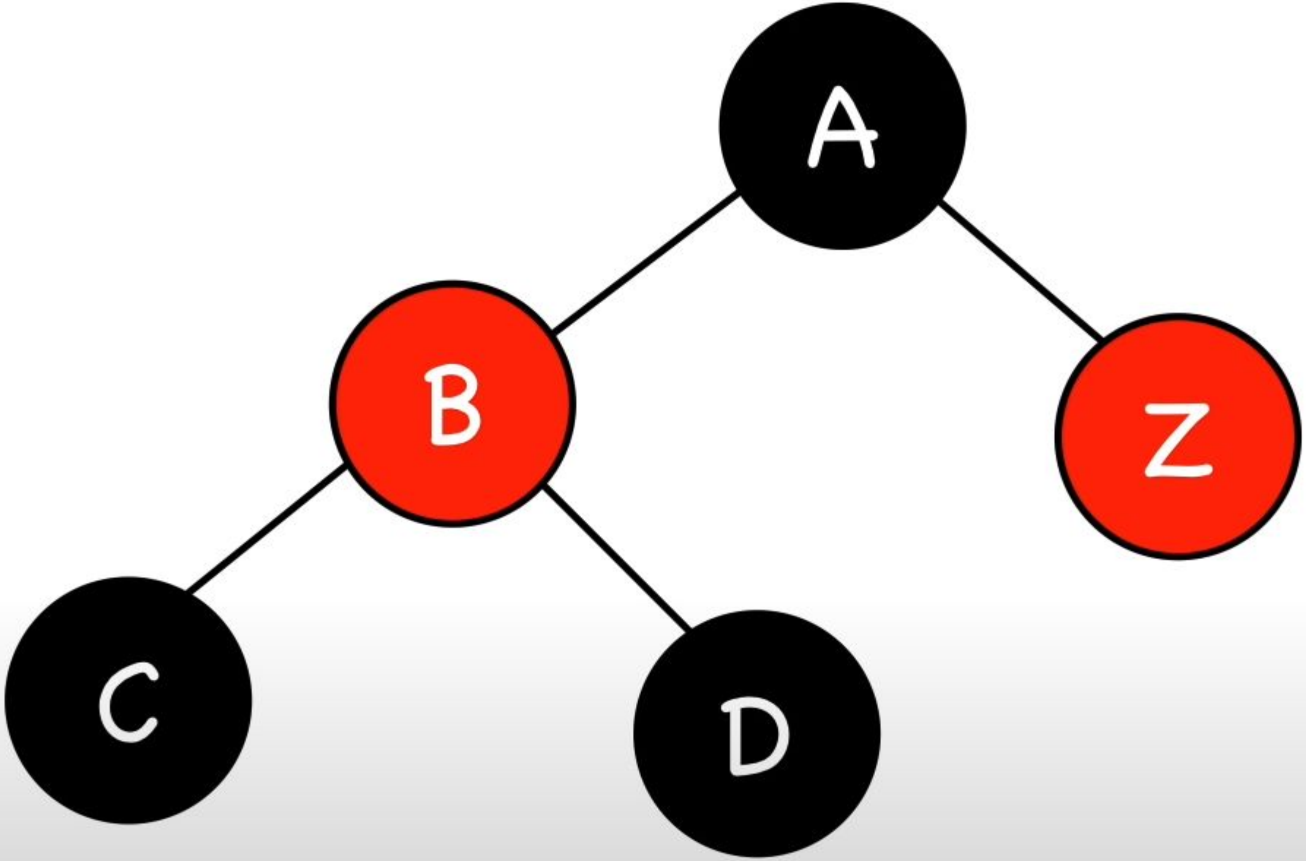
Solution: rotate Z.parent


B
C
Z
A

Case 4: Z.uncle is black (line)



Solution: rotate Z.grandparent and recolor


A
B
Z
C
D

Operations

- Search
- Insert
- **Remove**

1. Change Node with the immediately right node.
2. Cases after swapping:

    A) Node has 2 NIL children and is RED. **-> Delete Node.**

    B) Node has 1 NIL children and is RED. **-> Replace** Node **with its** non-child**.**

    C) Node has 1 RED child and is **BLACK. ->Replace** Node **with its** child, **Recolor to BLACK**

    D) Node has 1 **BLACK** child and is **BLACK -> Replace** Node **with its** child. **Double-black** child. **Transform to a normal black.**

Time and Space Complexity

Time:

Search O(log n)

Insert O(log n)

Remove O(log n)

This is because, although the red-black tree isn't as perfectly balanced as the AVL tree, it is still guaranteed to have a maximum total height of 2log(n+1).

Space:

O(n)

Same as BST, but one extra storage bit for the color.

Advantages:

1. Red black tree are useful when we need insertion and deletion relatively frequent.
2. Red-black trees are self-balancing so these operations are guaranteed to be O(logn).
3. They have relatively low constants in a wide variety of scenarios.

Disadvantages:

Even if you don't need ordered data, red-black trees might not be perfect - in particular, not if you are implementing a disk-based database. In disk-based I/O, seeking is expensive compared to sequential block reading, and the goal is therefore to minimize the number of disk accesses.

Applications

- The process scheduler in Linux uses Red Black Trees.
- C++ STL: map, multimap, multiset.

References

- https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

- Michael Sambol.(2016). Red-black trees in 4 minutes — The basics. https://www.youtube.com/watch?v=qvZGUFHWChY&list=PL9xmBV_5YoZNqDI8qfOZgzbqahCUmUEin&index=1

Thanks!!!!