Red Black Trees

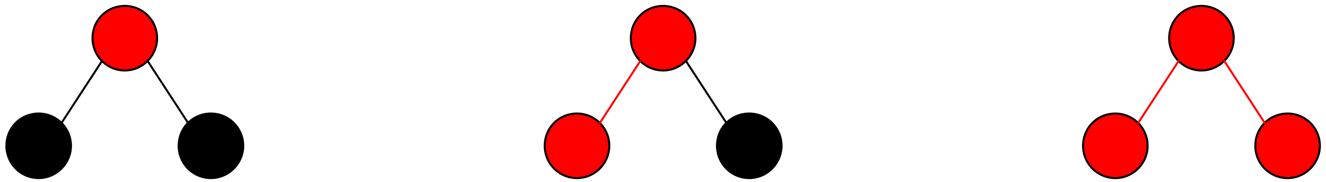
Prepared by
Dr. Annushree Bablani

A balanced binary search tree- Review

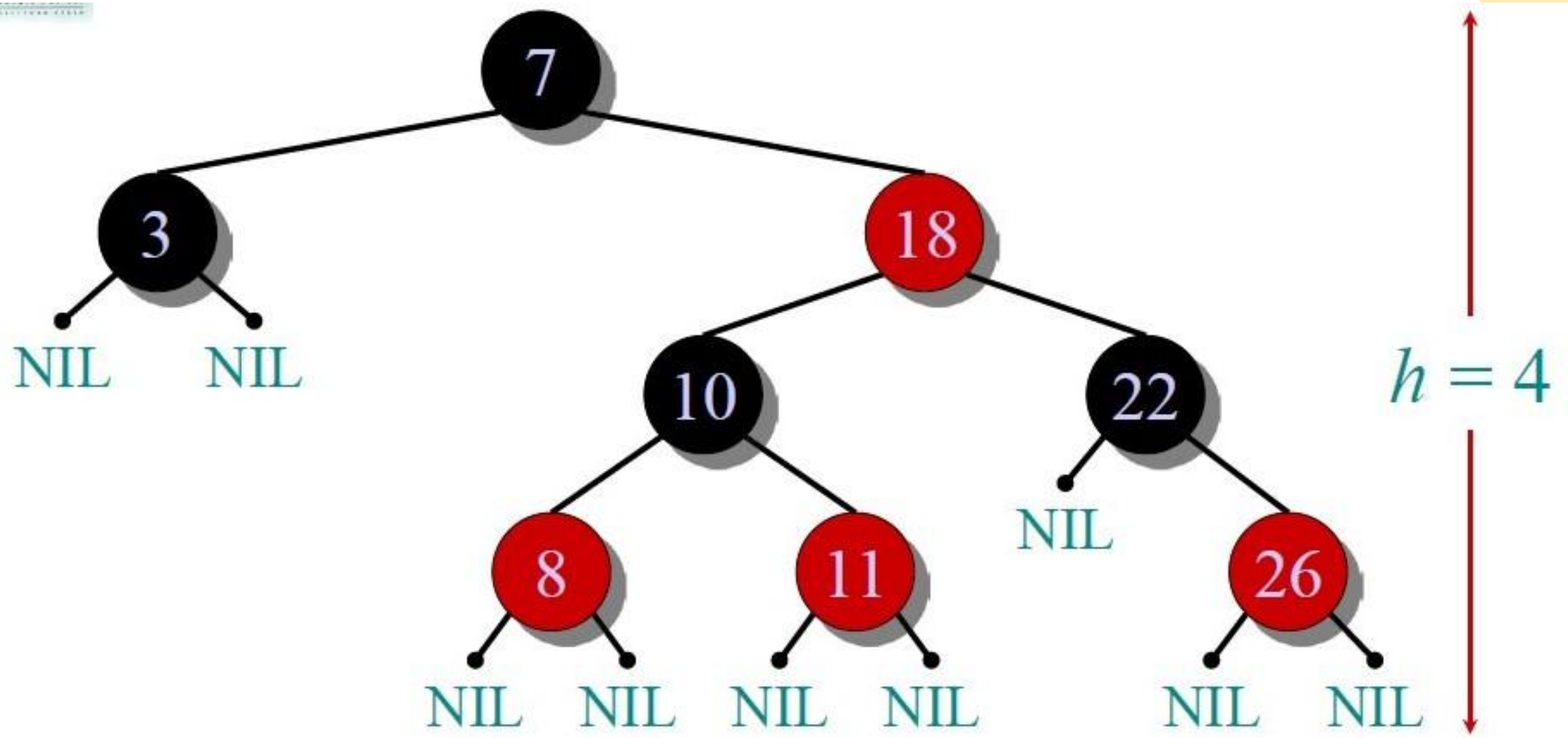
- Binary Search Tree (BST) is a good data structure for searching algorithm
- It supports
  - Search, find predecessor, find successor, find minimum, find maximum, insertion, deletion
- The performance of BST is related to its height h
  - All the operations are O(h)
- We want a balanced binary search tree
  - Height of the tree is O(log n)
- Red-Black Tree is one of the balanced binary search tree

Properties of Red-Black Trees

- Every node is either red or black
- The root is black
- If a node is red, then both its children are black

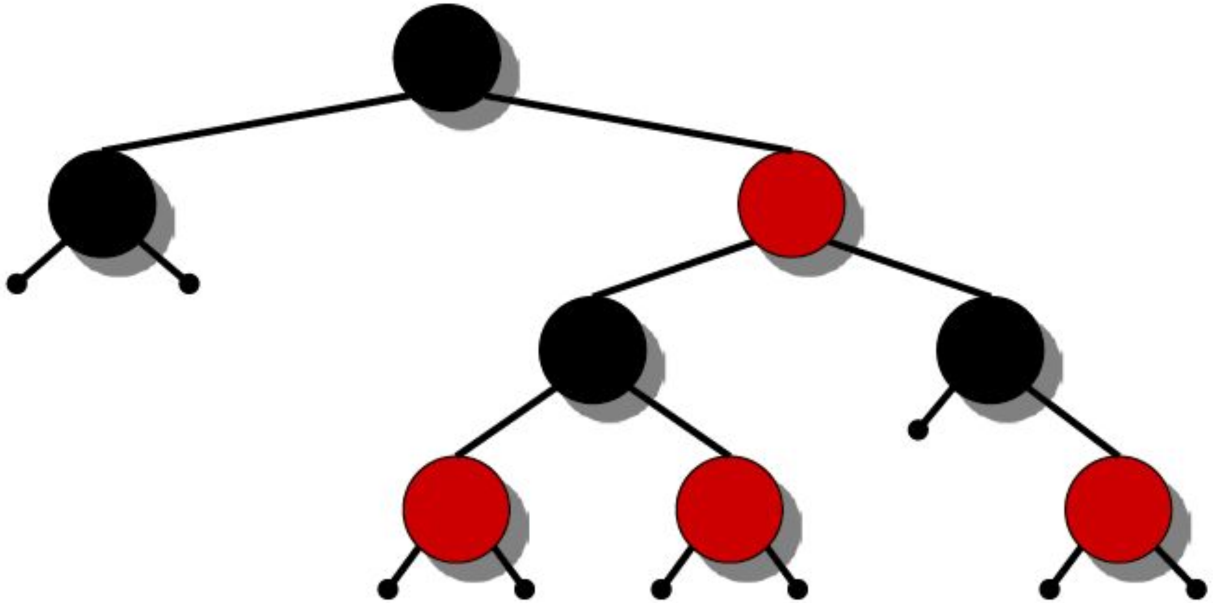


- For each node, all path from the node to descendant leaves contain the same number of black nodes
    - All path from the node have the same black height


7
3
18
NIL
NIL
10
22
h = 4
8
11
NIL
26
NIL
NIL
NIL
NIL
NIL
NIL

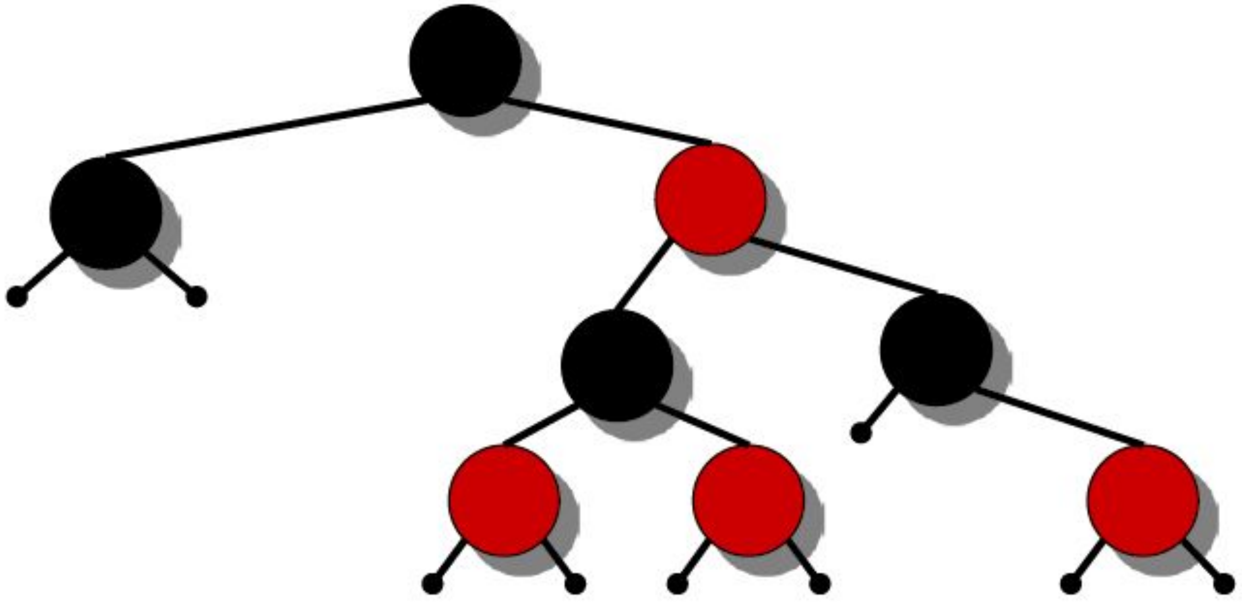
Height of a red-black tree

- A red-black tree with n keys has height $h \leq 2 \lg(n + 1)$.
- **INTUITION:**
    - Merge red nodes into their black parents.



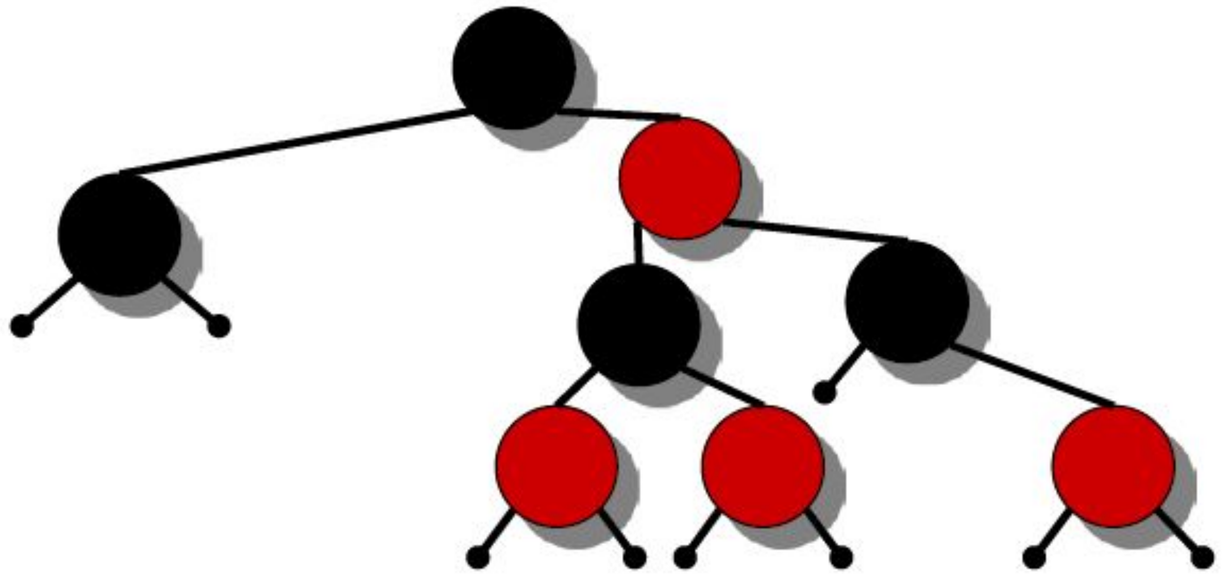
Height of a red-black tree

- A red-black tree with n keys has height $h \leq 2\lg(n+1)$.
- **INTUITION:**
  - Merge red nodes into their black parents.



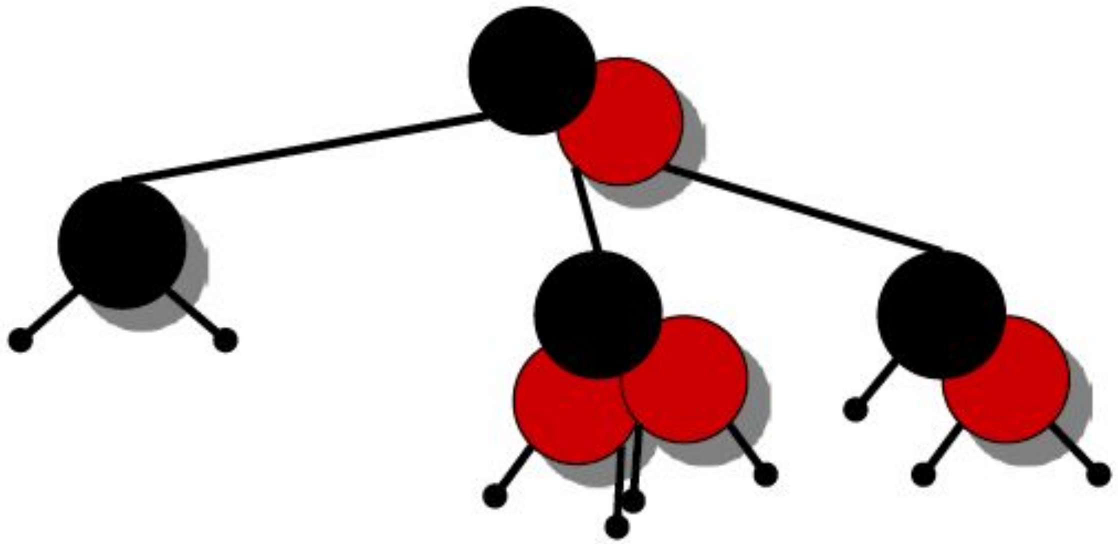
Height of a red-black tree

- A red-black tree with n keys has height $h \leq 2 \lg(n + 1)$.
- **INTUITION:**
    - Merge red nodes into their black parents.



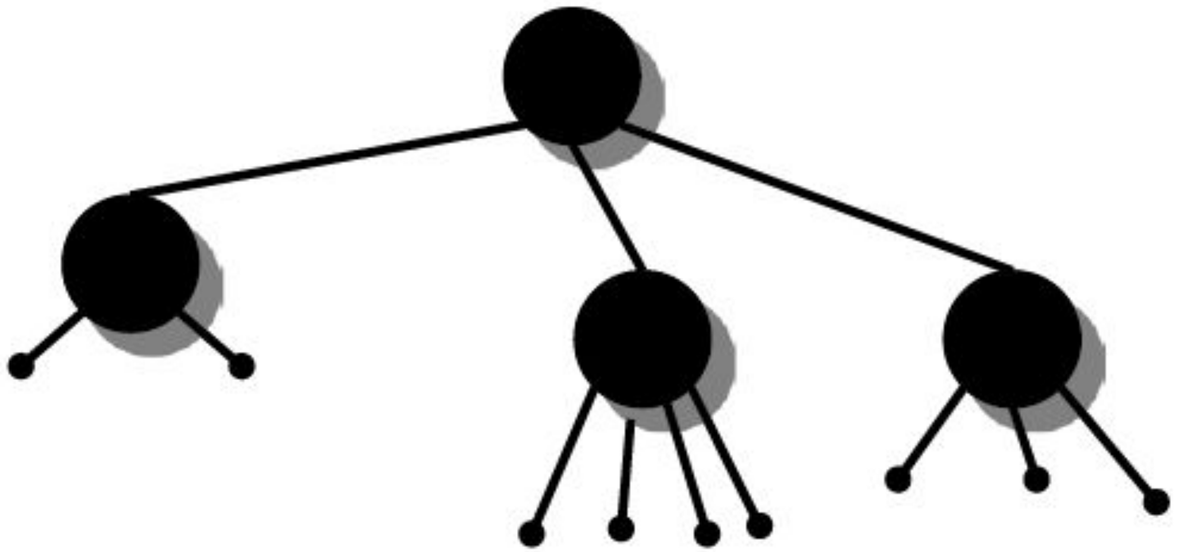
Height of a red-black tree

- A red-black tree with n keys has height $h \leq 2 \lg(n + 1)$.
- **INTUITION:**
  - Merge red nodes into their black parents.



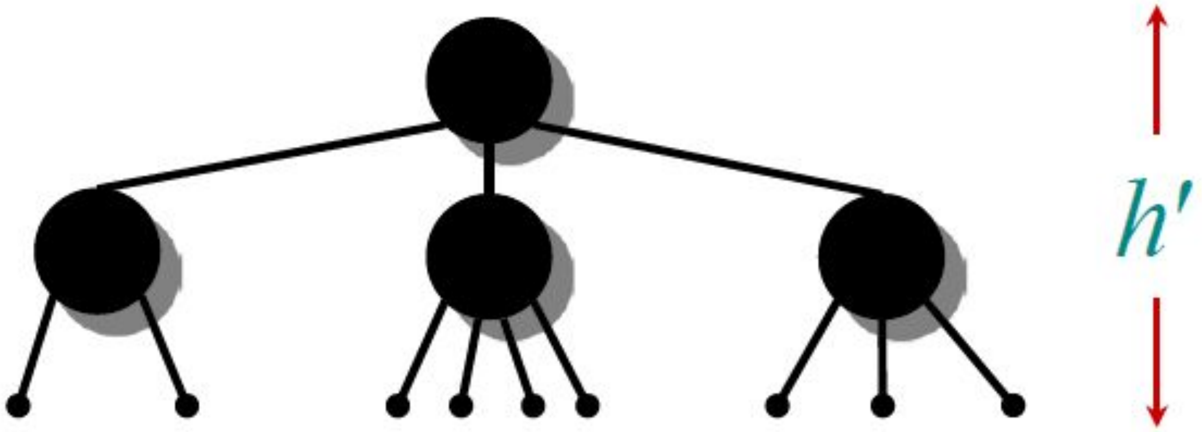
Height of a red-black tree

- A red-black tree with n keys has height $h \leq 2 \lg(n + 1)$.
- **INTUITION:**
  - Merge red nodes into their black parents.



Height of a red-black tree

- A red-black tree with n keys has height $h \leq 2 \lg(n + 1)$.
- **INTUITION:**
  - Merge red nodes into their black parents.
  - This process produces a tree in which each node has 2, 3, or 4 children.
  - The 2-3-4 tree has uniform depth h' of leaves.

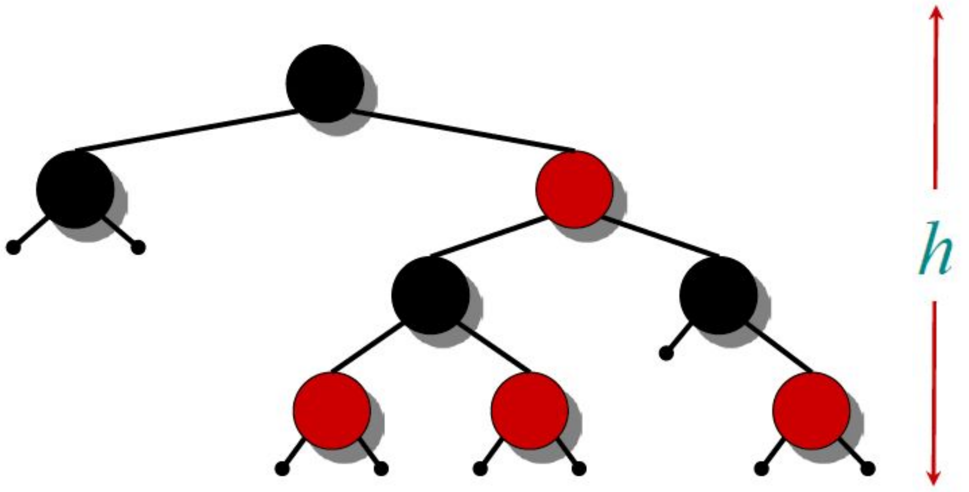


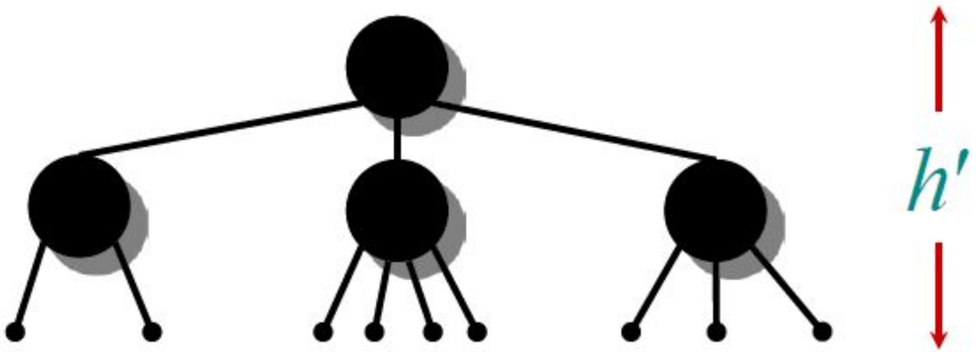
Height of a red-black tree

- We have $h' \geq h/2$, since at most half the leaves on any path are red.

- The number of leaves in each tree is $n + 1 \Rightarrow n + 1 \geq 2^{h'} \Rightarrow \lg(n + 1) \geq h' \geq h/2 \Rightarrow h \leq 2\lg(n + 1)$.

- The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in $O(\lg n)$ time on a red-black tree with n nodes.



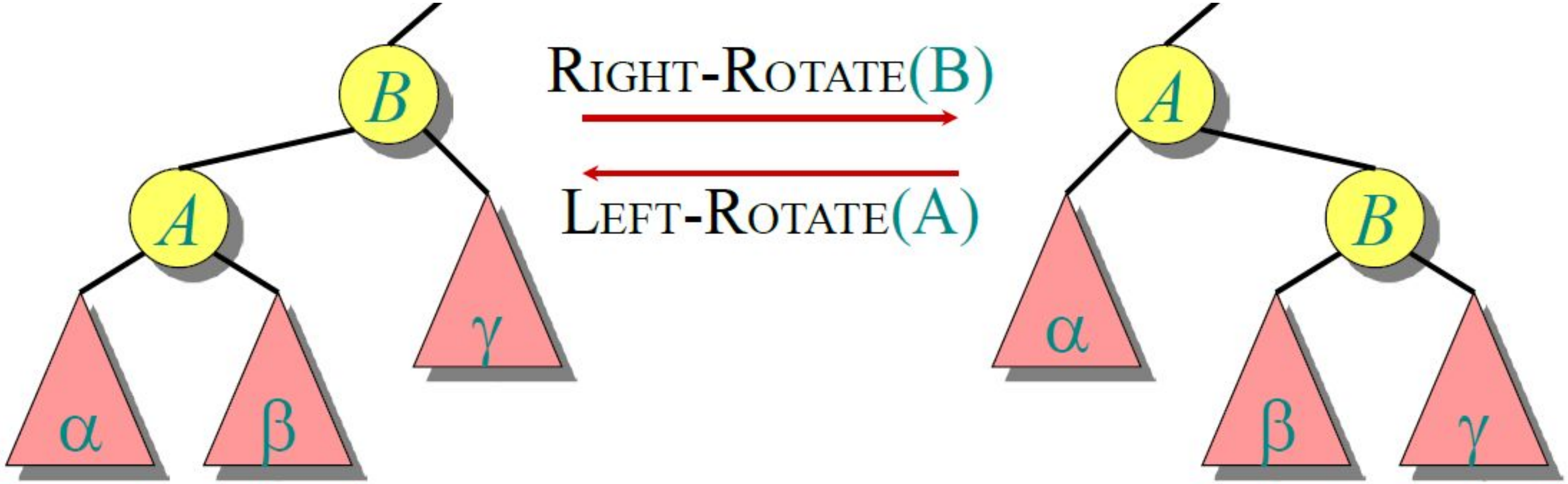


Modifying operations

- The operations INSERT and DELETE cause modifications to the red-black tree:
  - the operations causes violation of the red-black properties
  - Colour of some nodes to be changed
  - restructuring the links of the tree via ***"rotations"***.

Rotations

- Rotations maintain the in-order ordering of keys:
  - $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c$.
- A rotation can be performed in $O(1)$ time.



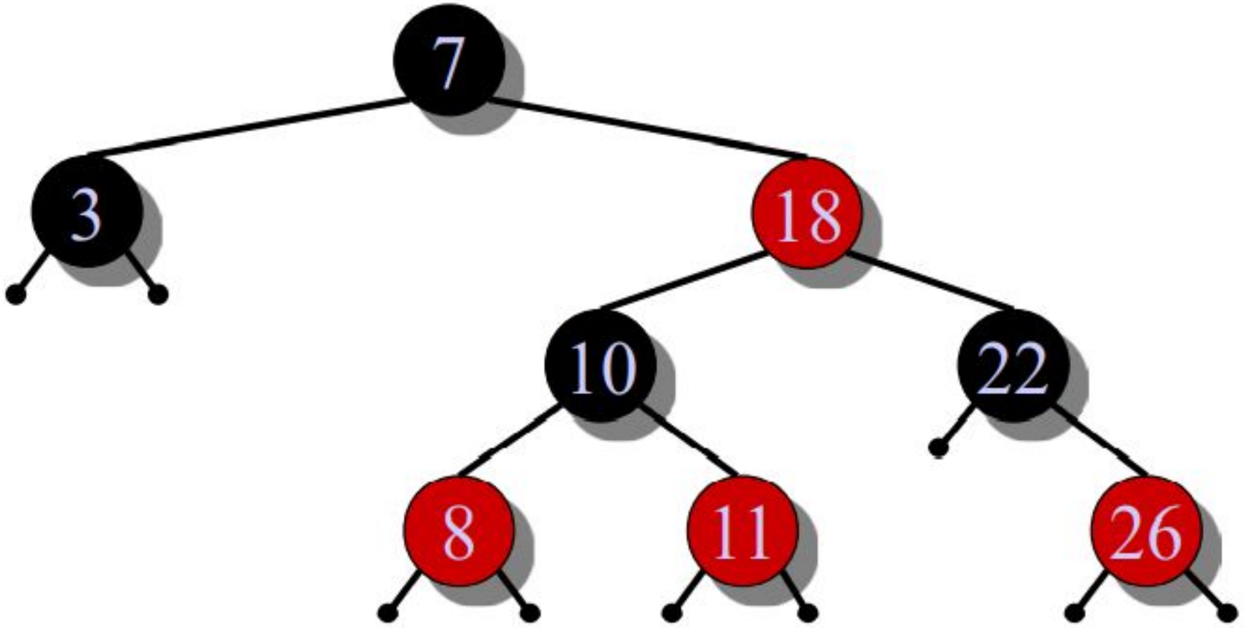
Rotations

Left Rotate (T, A) pseudoocode

```
B=A.right
A.right=B.left
If B.left≠T.nil
        B.left.p=A
B.p=A.p
If A.p==T.nil
        T.root=B
Elseif A==A.p.left
        A.p.left=A
Else A.p.right=B
B.left=A            // A on B's Left
A.p=B
```

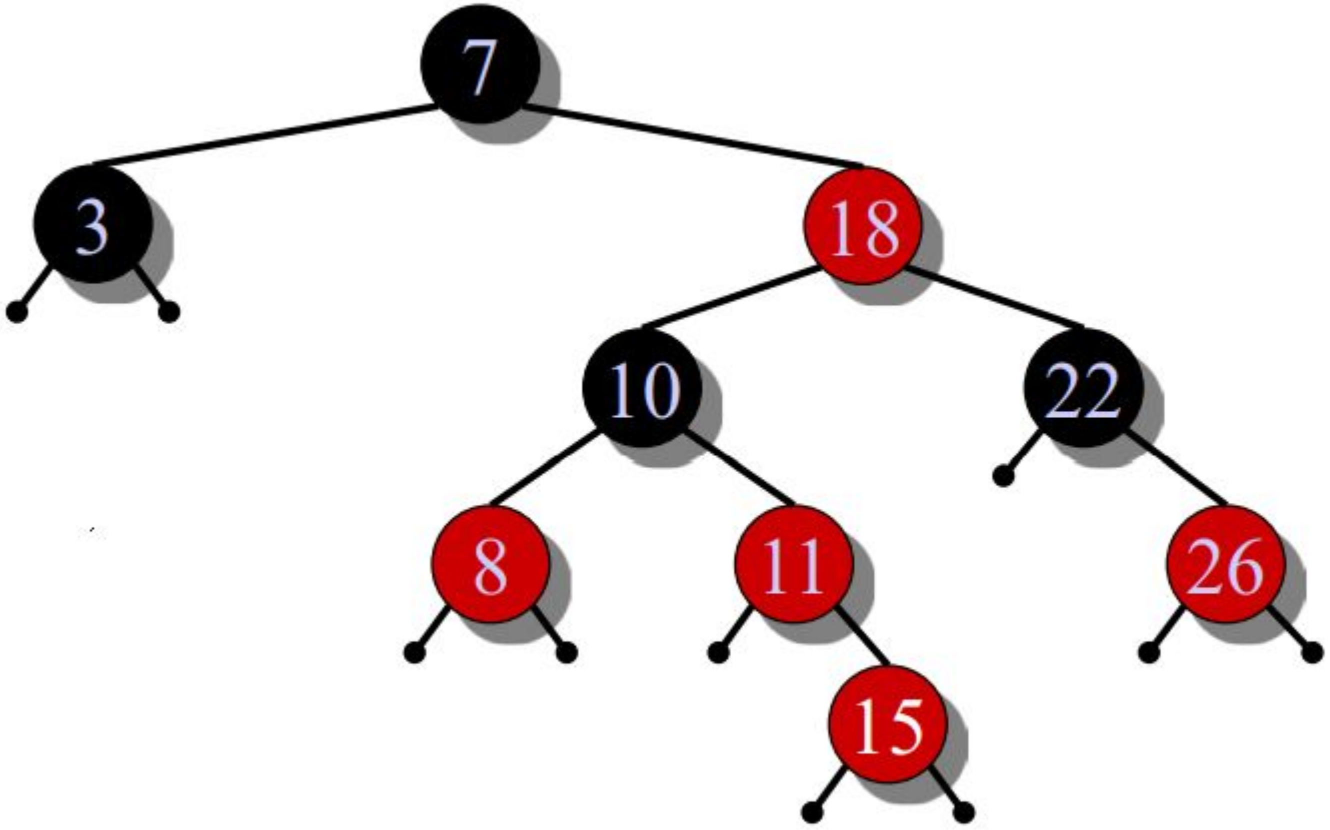
Insertion into a red-black tree

- IDEA:
  - Insert x in tree. Color x red. Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring



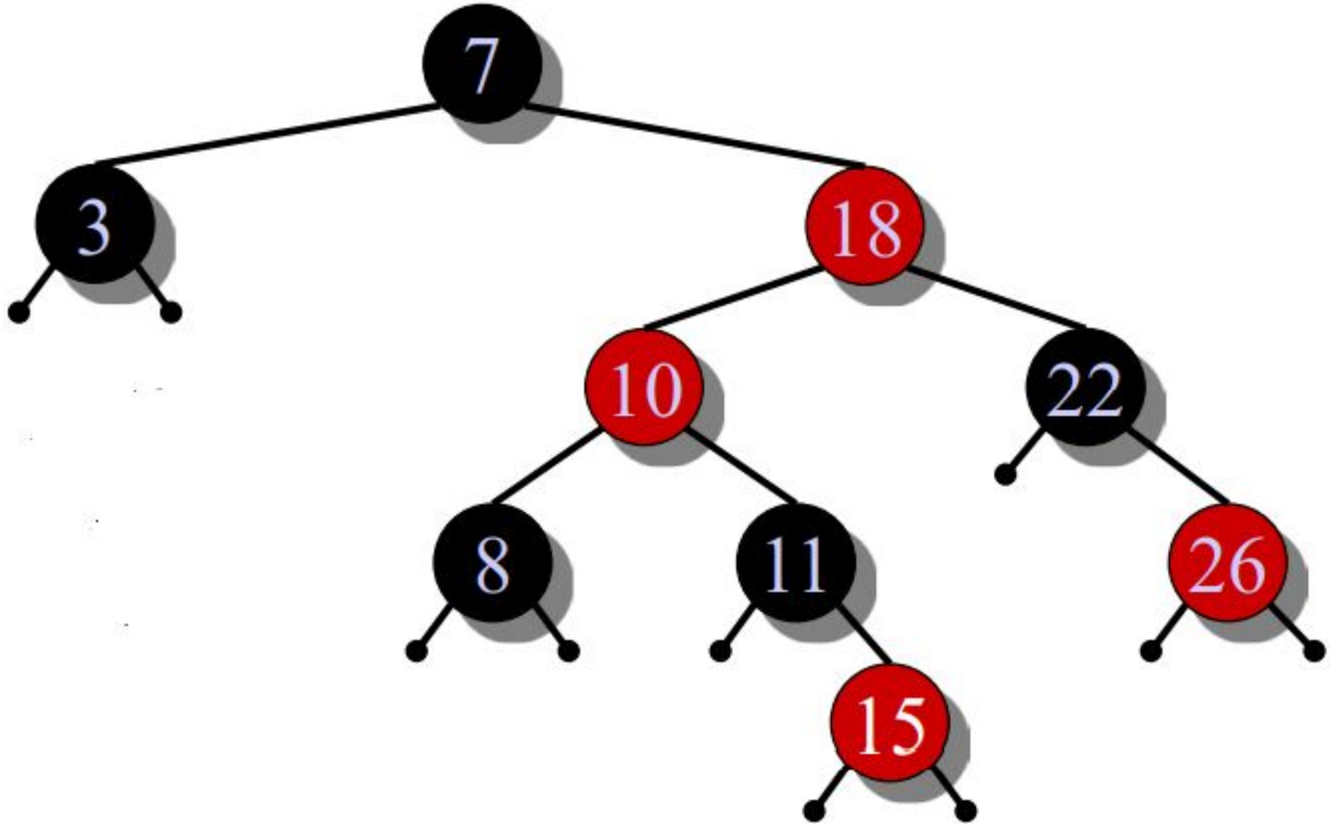
Insertion into a red-black tree

- IDEA:
  - Insert x in tree. Color x red. Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring
- Example:
  - Insert x =15.



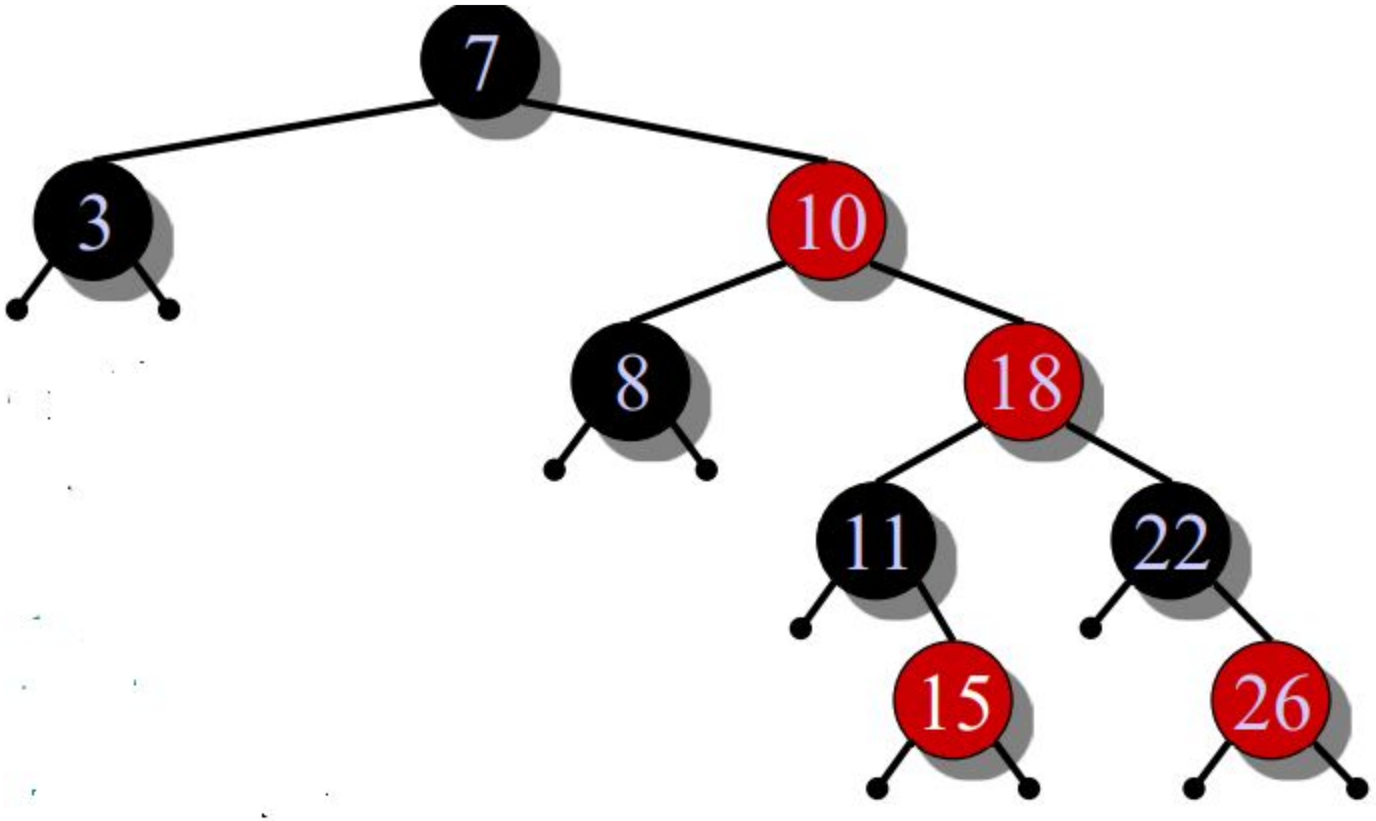
Insertion into a red-black tree

- IDEA:
  - Insert x in tree. Color x red. Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring
- Example:
  - Insert x =15.
  - Recolor, moving the violation up the tree.
  - RIGHT-ROTATE(18).



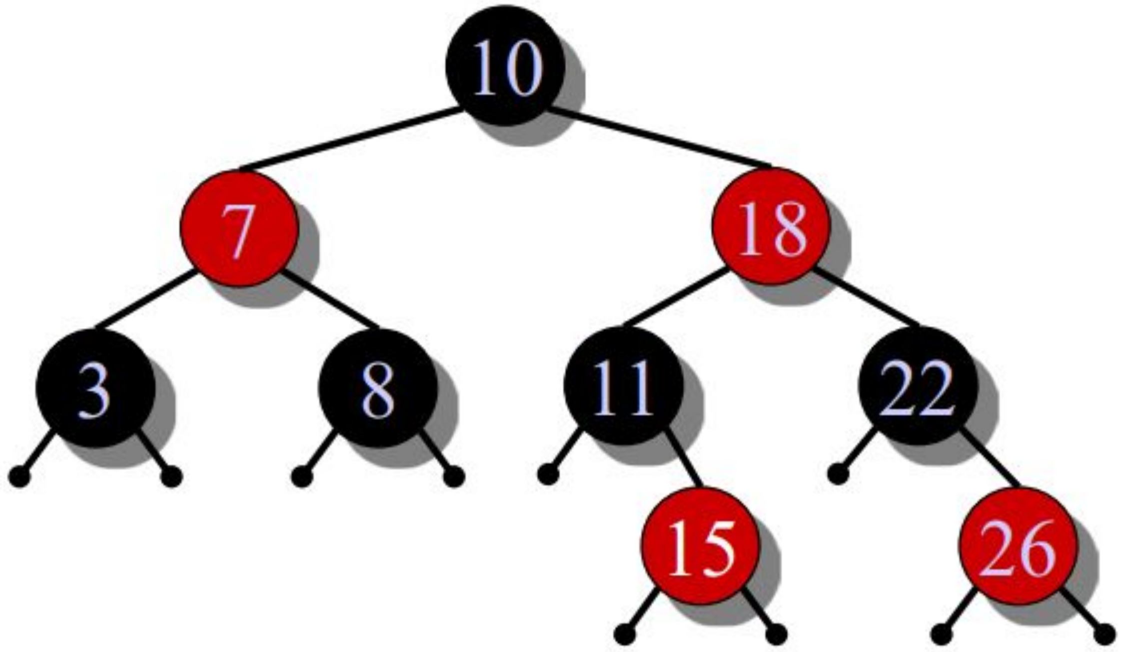
Insertion into a red-black tree

- IDEA:
    - Insert x in tree. Color x red. Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring
- Example:
    - Insert x =15.
    - Recolor, moving the violation up the tree.
    - RIGHT-ROTATE(18).
    - LEFT-ROTATE(7) and recolor.



Insertion into a red-black tree

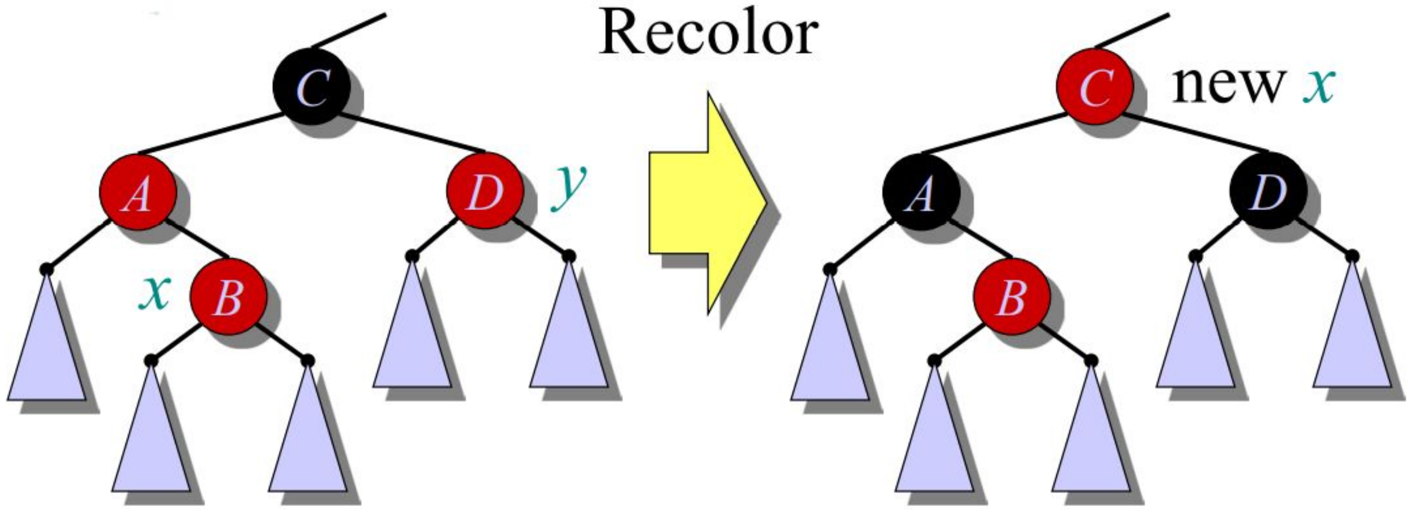
- IDEA:
  - Insert x in tree. Color x red. Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring
- Example:
  - Insert x =15.
  - Recolor, moving the violation up the tree.
  - RIGHT-ROTATE(18).
  - LEFT-ROTATE(7) and recolor.



Insertion into a red-black tree- Pseudocode

```
RB-INSERT(T, x)
    TREE-INSERT(T, x)
    color[x] ← RED                      //only RB property 3 can be violated
    while x ≠ root[T] and color[p[x]] = RED
        do if p[x] = left[p[p[x]]
            then y ← right[p[p[x]]          // y = aunt/uncle of x
                if color[y] = RED
                then 〈Case 1〉
                else if x = right[p[x]]
                    then 〈Case 2〉            // Case 2 falls into Case 3
                    〈Case 3〉
            else 〈“then” clause with “left” and “right” swapped〉
    color[root[T]] ← BLACK
```

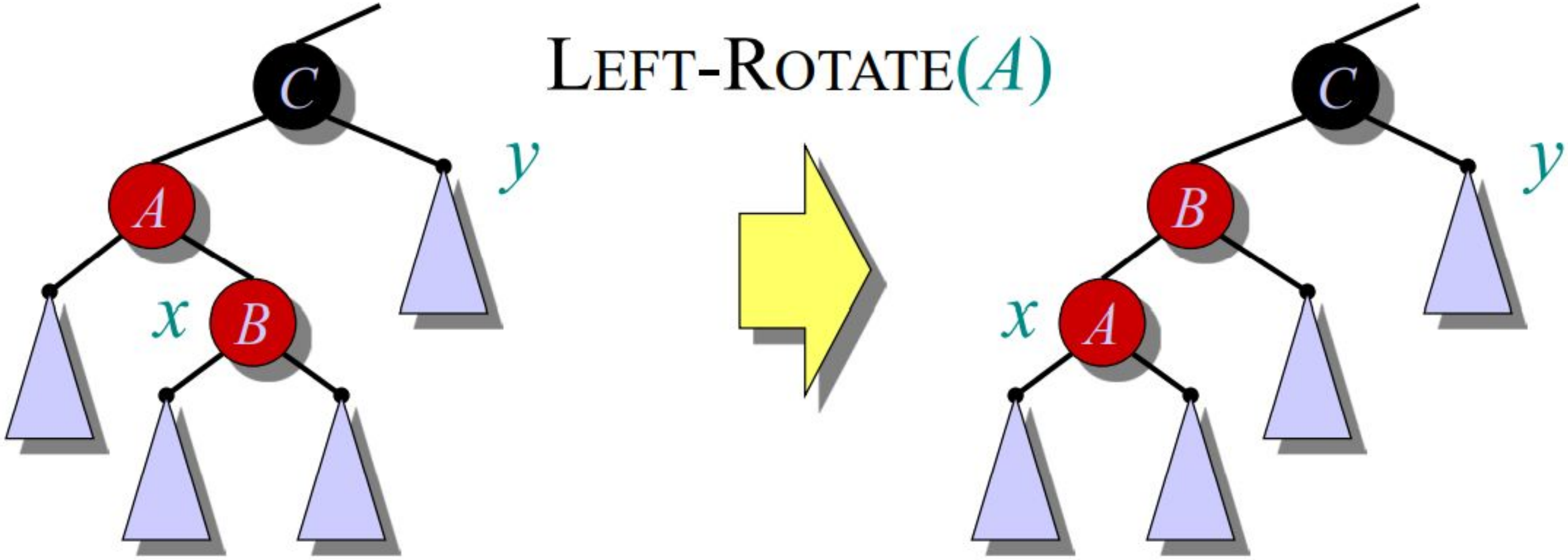
Insertion into a red-black tree- Case-1



(Or, children of A are swapped.)

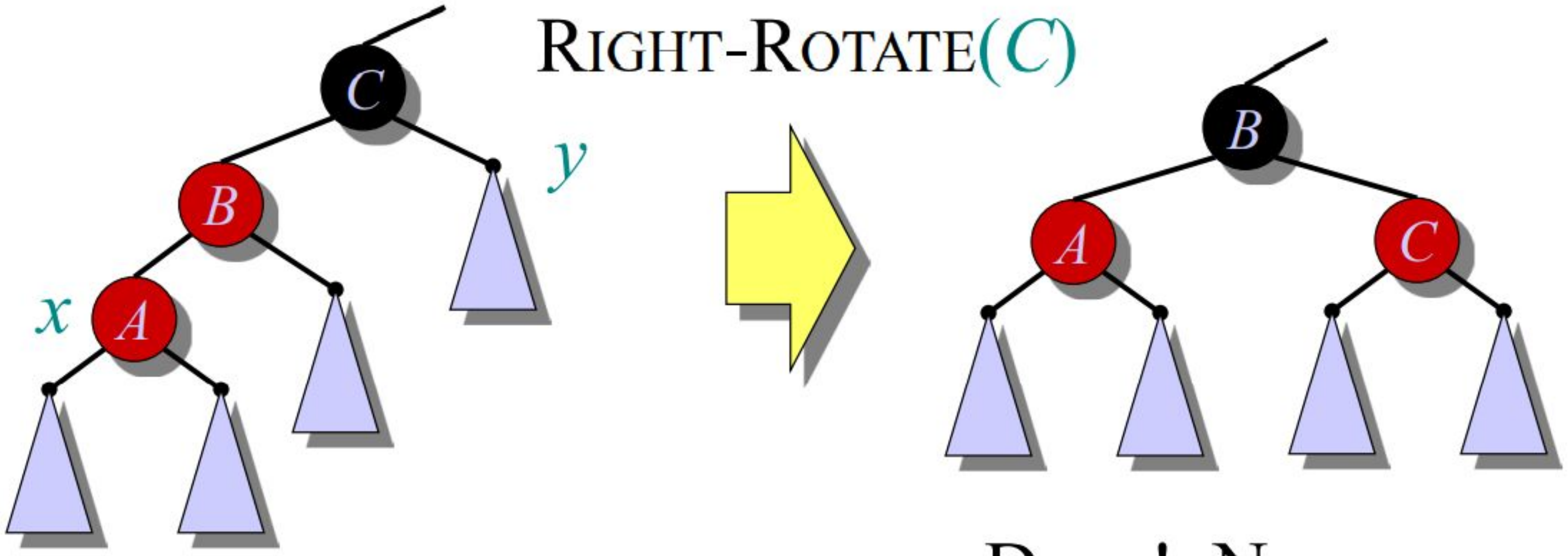
Push C's black onto A and D, and recurse, since C's parent may be red.

Insertion into a red-black tree- Case-2



Transform to Case 3.

Insertion into a red-black tree- Case-3



Done! No more violations of RB property 3 are possible.

Analysis

- Go up the tree performing Case 1, which only recolors nodes.
- If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.
- Running time: $O(\lg n)$ with $O(1)$ rotations.
- RB-DELETE — same asymptotic running time and number of rotations as RB-INSERT