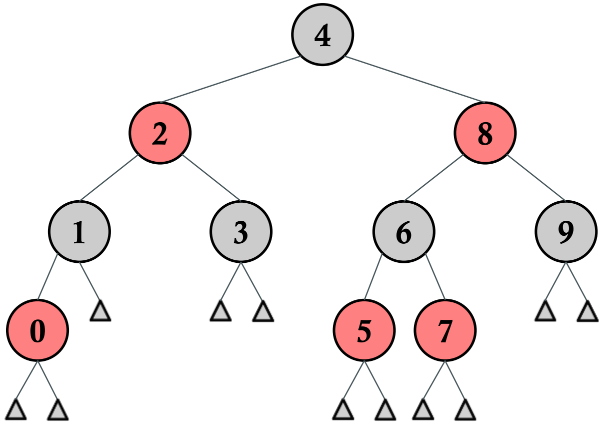
RED BLACK TREE: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree*

- A RBT is a *binary search tree*



- Properties of a BST:
  - `node > lChild && node < rChild`
  - In-order traversal returns a sorted list

- Properties of a RBT:
  - *node* has color – either red or black
  - *root* is black
  - all leaves are `NIL` nodes: *value* is `null`; *color* is black – **they are not `NULL`**
  - a red node's children and parent must all be black
  - every path from a node to any of its descendant `NIL` has the same number of black nodes
  - RBT is balanced (by regulating the colors)

- Initialization
  - Create an empty RBT/Clear a RBT: set *root* to `null`
  - Check empty: check if *root* is `null`

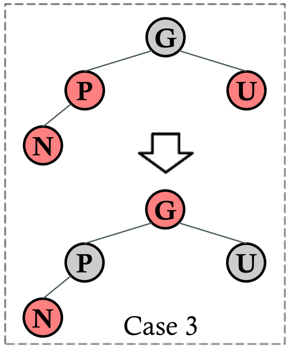
- Count the size
  - Solution 1: recursion – count the size of *root*'s subtree (set *subroot* to *root*)
    - If *subroot* is `null` or `NIL`, then return 0
    - Otherwise, count the size of *lChild*'s subtree *size*1 and the size of *rChild*'s subtree *size*2
    - Return 1 + *size*1 + *size*2
  - Solution 2 (recommended): use `int` variable as counter

- In-order traversal (recursion)
  - Traverse the *root*'s subtree (set *subroot* to *root*)
    - If *subroot* is `null` or `NIL`, then it is empty, return nothing
    - Otherwise:
      1. First, traverse the *lChild*'s subtree
      2. Then, visit *subroot*
      3. Finally, traverse the *rChild*'s subtree

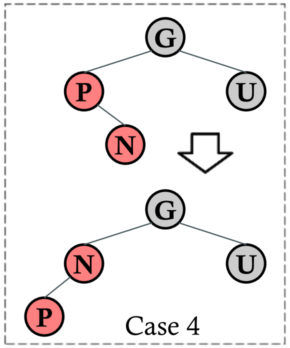
- Search for a *value*— search as a BST
  - Solution 1: recursion – search for *value* from *root*'s subtree (set *subroot* to *root*)
    - If *subroot* is `null` or `NIL`, then search failed, return `null`
    - Otherwise, if *value* = *subroot.value*, then found, return the *value*
    - Otherwise:
      - If *value* < *subroot.value*, search for *value* from *lChild*'s subtree
      - Else (*value* > *subroot.value*), search for *value* from *rChild*'s subtree
  - Solution 2: iteration – start from *root* (set *current* to *root*)
    - If *subroot* is `null` or `NIL`, then search failed, return `null`
    - Otherwise, if *value* = *suroot.value*, then found, return the *value*
    - Otherwise:
      - If *value* < *subroot.value*, set *current* to *current.lChild*
      - Else (*value* > *subroot.value*), set *current* to *current.rChild*

RED BLACK TREE: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

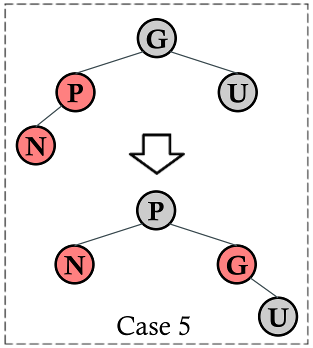
- Insert a *value*
  - Step 1: insert as a BST
    - Search the *value*, recursively or iteratively
    - If *value* found, then update *value*, and insertion done
    - Otherwise, create a new node with the *value* to replace the `NIL` node, **add two new `NIL` children to it**, and fix its color
    - If it's the first insertion, point *root* to the new node
    - Set the *node*'s color to red, and then balance the tree.
  - Step 2: balance the tree – `fixInsColor(node)`
    - Case 1, *node* is *root* (first insertion): set color to black, done
    - Case 2, *node*'s *parent* exists and is black: tree still valid, done
    - Case 3, both *parent* and *uncle* exist and are red: set their colors to black, set *grandparent*'s color (*grandparent* must exist) to red, and fix *grandparent*'s color – invoke `fixInsColor(gp)`, done
    - Case 4, *parent* exists and is red; *uncle* exists and is black:
      - If *node* is a rChild and *parent* is a *lChild*: rotate **left** on ***node***, set *node* to *node*'s *lChild*, and go on to Case 5
      - If *node* is a lChild and *parent* is a *rChild*: rotate **right** on ***node***, set *node* to *node*'s *rChild*, and go on to Case 5
    - Case 5, *parent* exist and is red; *uncle* exist and is black; both *node* and *parent* shall be *lChild/rChild*: set *parent*'s color to black and *grandparent*'s color to red, and:
      - If *node* is a lChild: rotate **right** on ***parent***, done
      - If *node* is a rChild: rotate **left** on ***parent***, done



Case 3



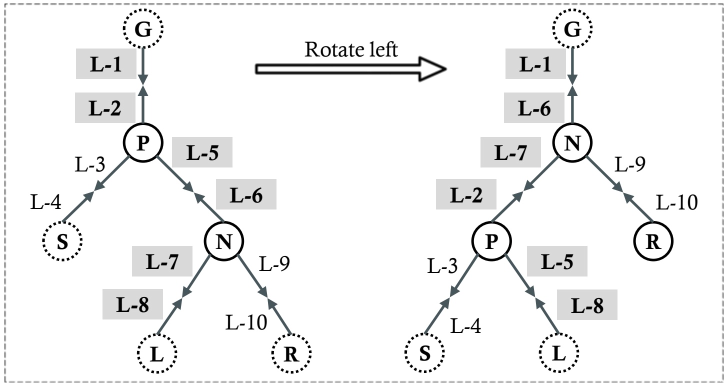
Case 4

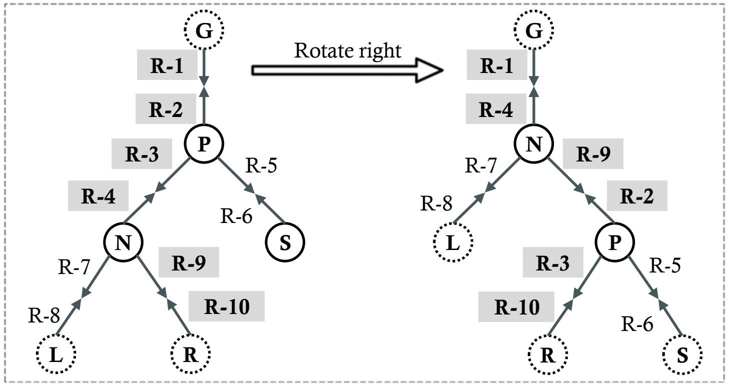


Case 5

- Rotate a binary tree on *node*
  - If *parent* is *root*, then set *root* to *node*
  - **Six** pointers need to be changed:

Rotate left	Rotate right
L-5: *parent.rChild -> node.lChild*	**R-3**: *parent.lChild -> node.rChild*
L-8: *lChild.parent -> parent* if *lChild* exists	**R-10**: *rChild.parent -> parent* if *rChild* exists
L-6, R-4: *node.parent -> grandparent*	
L-1, R-1: *grandparent.lChild -> node* if *parent* is a lChild *grandparent.rChild -> node* if *parent* is a rChild	
L-7: *node.lChild -> parent*	**R-9**: *node.rChild -> parent*
L-2, R-2: *parent.parent -> node*	


G
L-1
L-2
P
L-5
L-3
L-6
L-4
S
N
L-7
L-9
L-8
L-10
L
R
Rotate left
G
L-1
L-6
L-7
N
L-9
L-2
L-10
P
R
L-3
L-5
L-4
L-8
S
L


G
R-1
R-2
P
R-3
R-5
R-4
R-6
N
S
R-7
R-9
R-8
R-10
L
R
Rotate right
G
R-1
R-4
N
R-9
R-7
R-2
R-8
L
P
R-3
R-5
R-10
R-6
R
S