USC Viterbi
School of Engineering

CSCI 102
2-3-4 Trees and Red/Black Trees

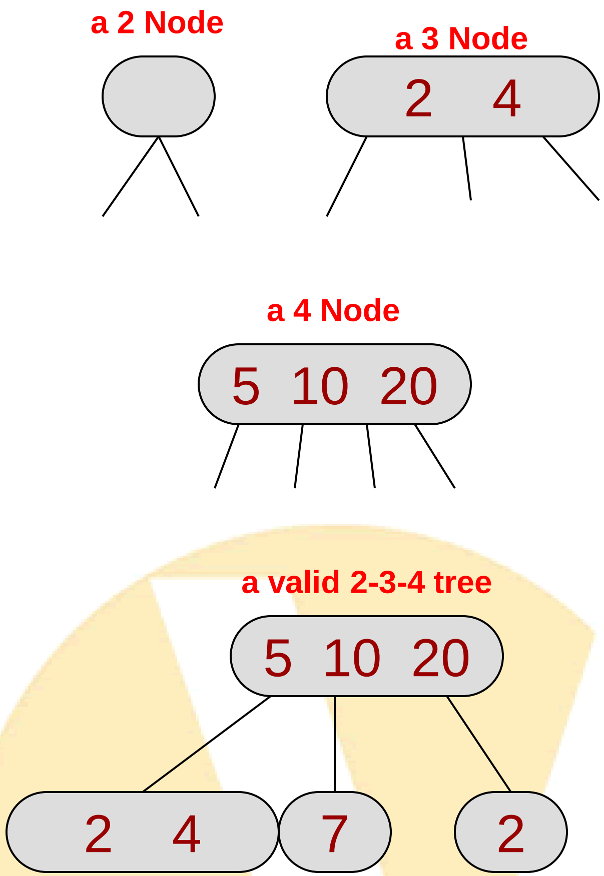
Mark Redekopp
Michael Crowley

USC

USC Viterbi School of Engineering

Definition

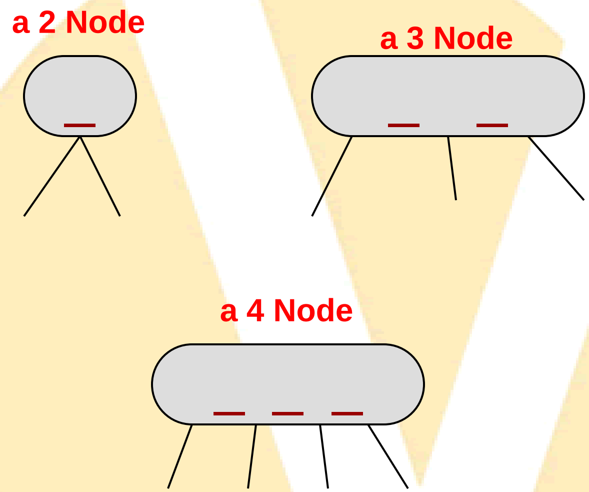
- 2-3-4 trees are very much like 2-3 trees but form the basis of a balanced, **binary** tree representation called Red-Black (RB) trees which are commonly used [used in C++ STL map & set]
  - We study them mainly to ease understanding of RB trees
- 2-3-4 Tree is a tree where
  - Non-leaf nodes have 1 value & 2 children or 2 values & 3 children or 3 values & 4 children
  - All leaves are at the same level
- Like 2-3 trees, 2-3-4 trees are always full and thus have an upper bound on their height of $\log_2(n)$



2-, 3-, & 4-Nodes

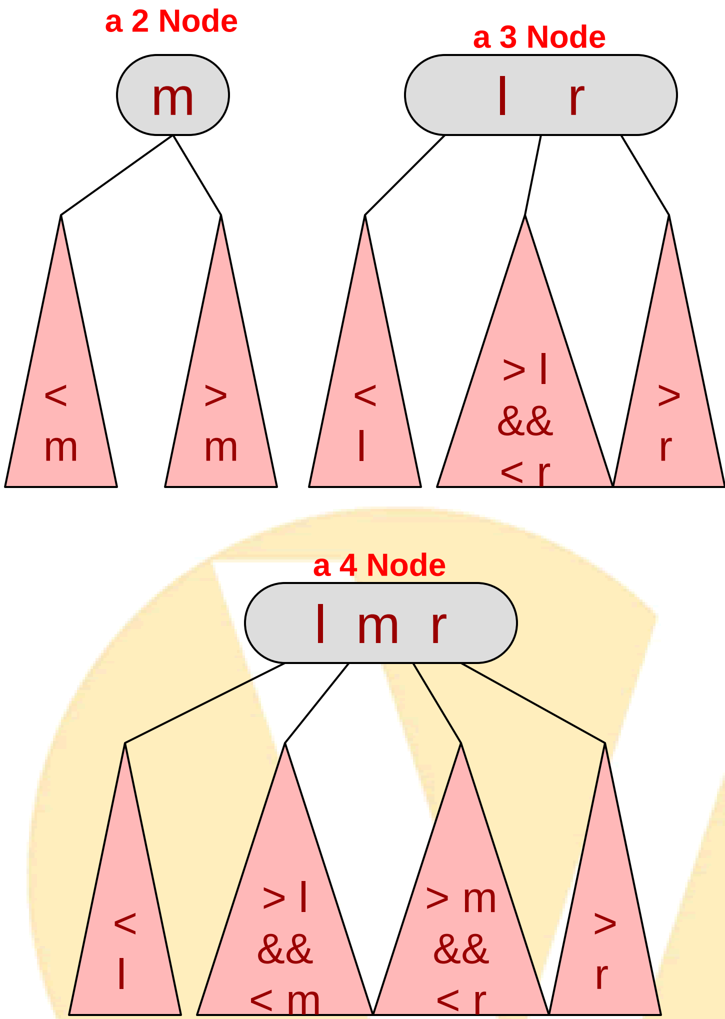
- 4-nodes require more memory and can be inefficient when the tree actually has many 2 nodes

```
template <typename T>
struct Item234 {
  T val1;
  T val2;
  T val3;
  Item234<T>* left;
  Item234<T>* midleft;
  Item234<T>* midright;
  Item234<T>* right;
  int nodeType;
};
```



2-3-4 Search Trees

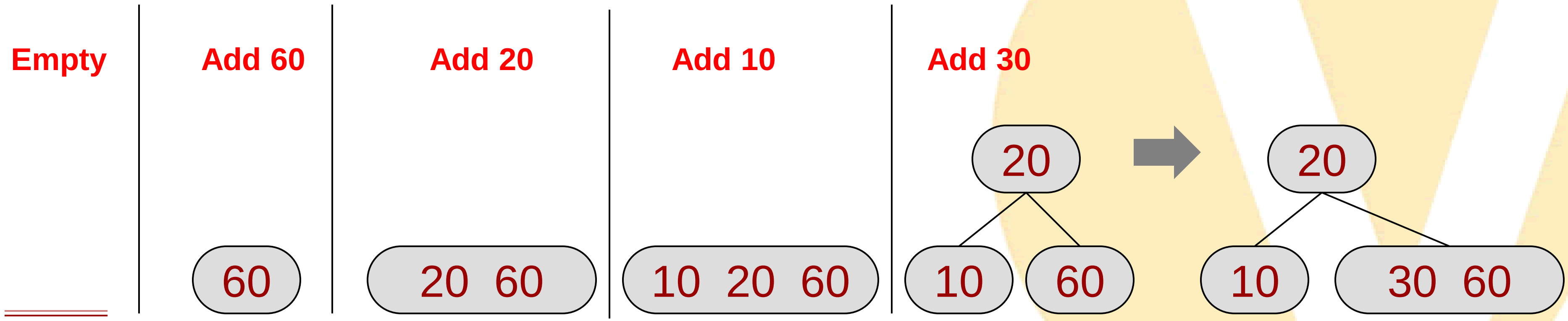
- Similar properties as a 2-3 Search Tree
- 4 Node:
  - Left subtree nodes are < *l*
  - Middle-left subtree > *l* and < *r*
  - Right subtree nodes are > *r*



2-3-4 Insertion Algorithm

- Key: Rather than search down the tree and then possibly promote and break up 4-nodes on the way back up, split 4 nodes on the way down
- To insert a value,
  - 1. If node is a 4-node
    - Split the 3 values into a left 2-node, a right 2-node, and promote the middle element to the parent of the node (which definitely has room) attaching children appropriately
    - Continue on to next node in search order
  - 2a. If node is a leaf, insert the value
  - 2b. Else continue on to the next node in search tree order
- Insert 60, 20, 10, 30, 25, 50, 80

Key: 4-nodes get split as you walk down thus, a parent will always have room for a value

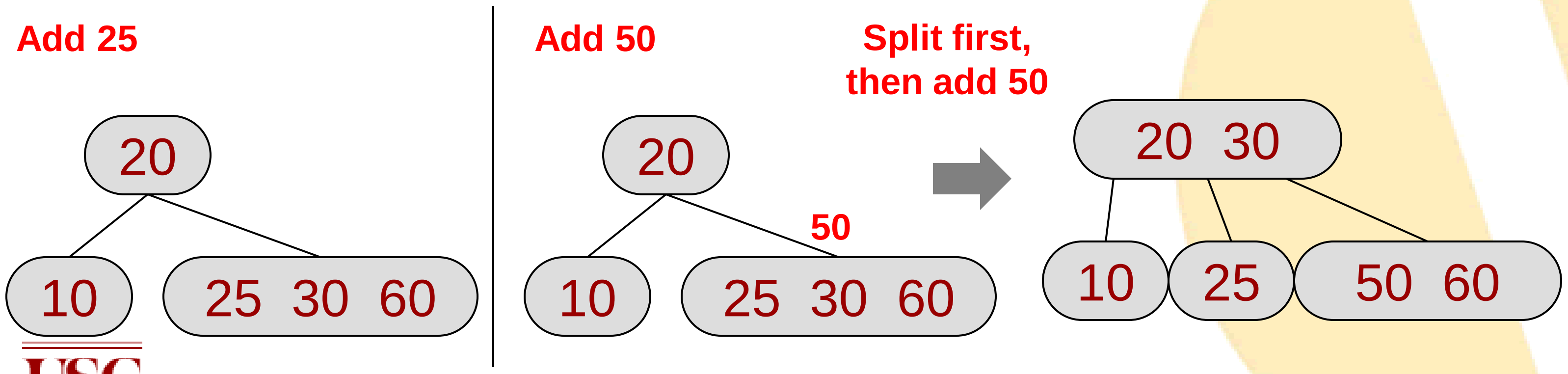


2-3-4 Insertion Algorithm

- Key: Split 4 nodes on the way down
- To insert a value,
  - 1. If node is a 4-node
    - Split the 3 values into a left 2-node, a right 2-node, and promote the middle element to the parent of the node (which definitely has room) attaching children appropriately
    - Continue on to next node in search order
  - 2a. If node is a leaf, insert the value
  - 2b. Else continue on to the next node in search tree order

Key: 4-nodes get split as you walk down thus, a parent will always have room for a value

- Insert 60, 20, 10, 30, 25, 50, 80

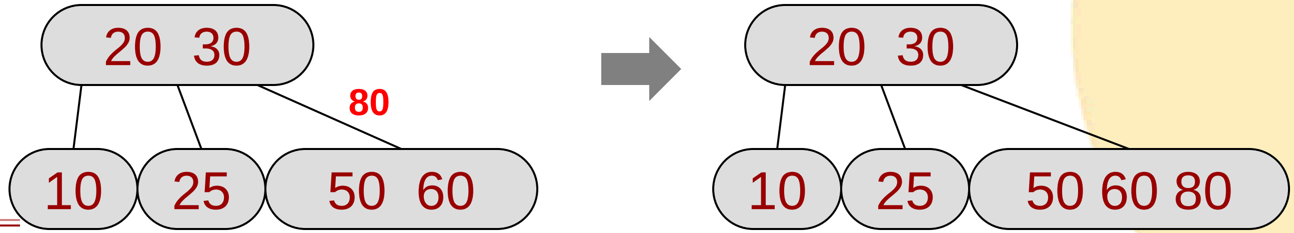


2-3-4 Insertion Algorithm

- Key: Split 4 nodes on the way down
- To insert a value,
  - 1. If node is a 4-node
    - Split the 3 values into a left 2-node, a right 2-node, and promote the middle element to the parent of the node (which definitely has room) attaching children appropriately
    - Continue on to next node in search order
  - 2a. If node is a leaf, insert the value
  - 2b. Else continue on to the next node in search tree order
- Insert 60, 20, 10, 30, 25, 50, 80

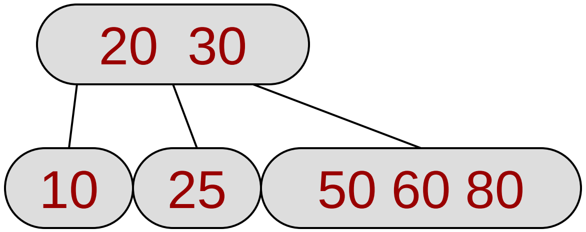
Key: 4-nodes get split as you walk down thus, a parent will always have room for a value

Add 80



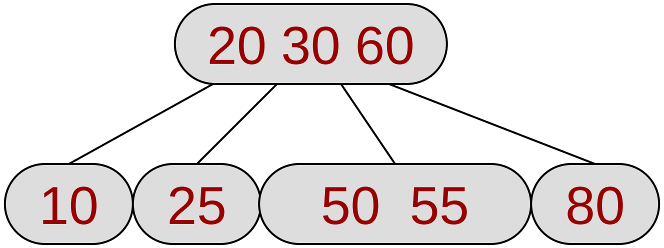
2-3-4 Insertion Exercise 1

Add 55



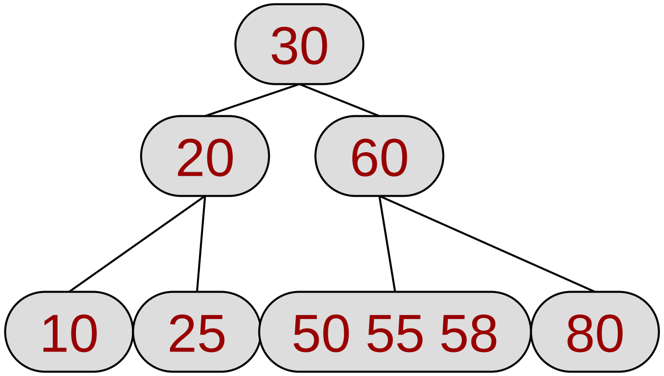
2-3-4 Insertion Exercise 2

Add 58



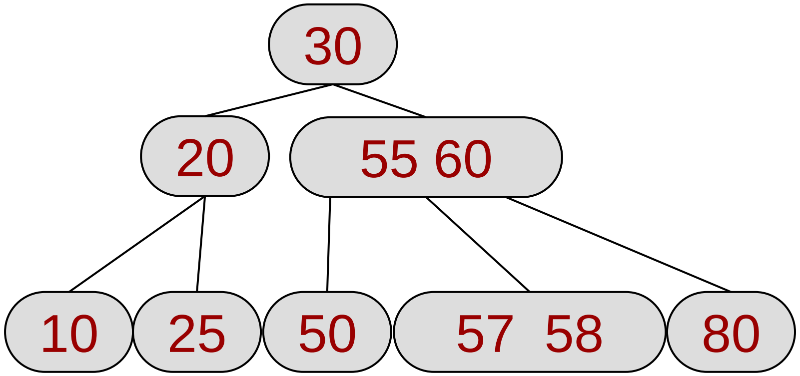
2-3-4 Insertion Exercise 3

Add 57



2-3-4 Insertion Exercise 3

Resulting Tree



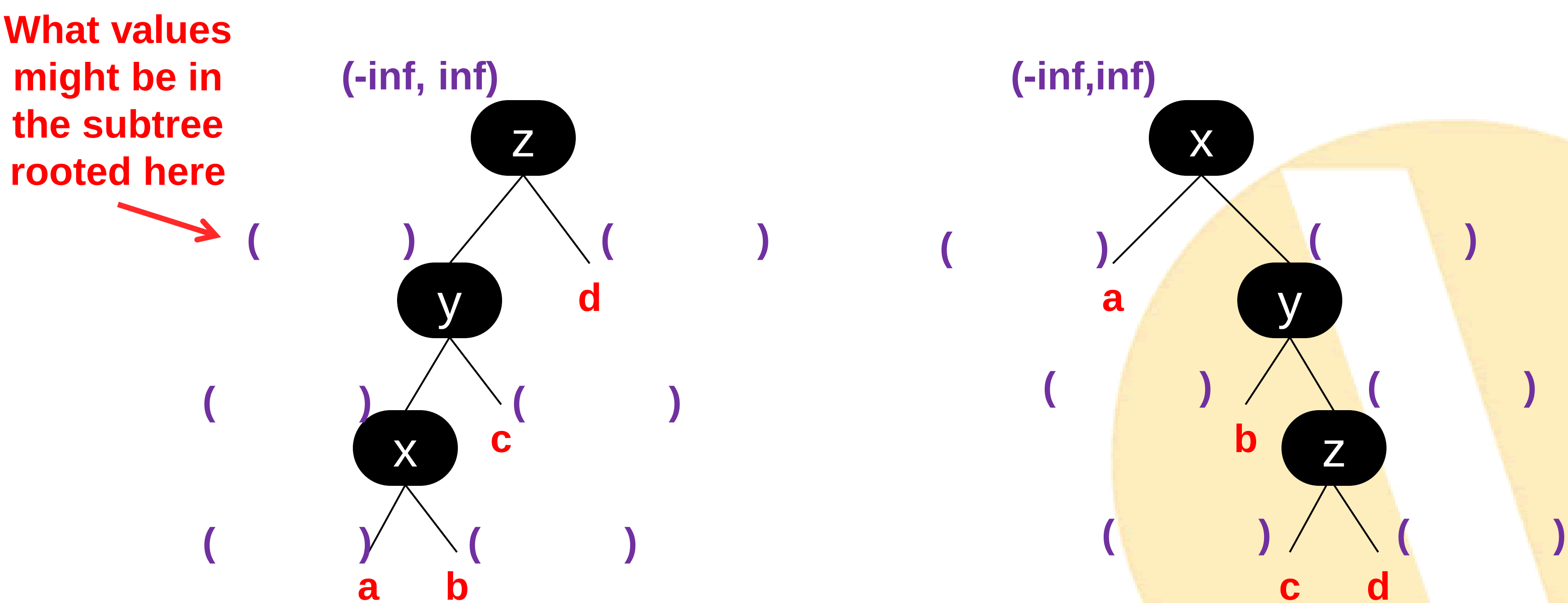
2-3-4 Tree Resources

- http://ultrastudio.org/en/2-3-4_tree

TREE ROTATIONS

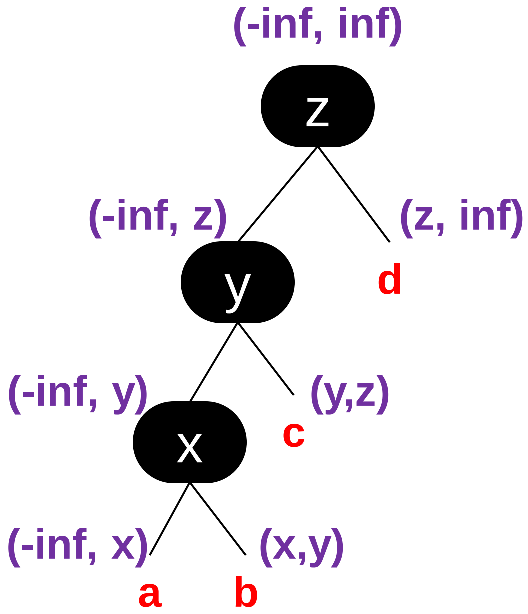
BST Subtree Ranges

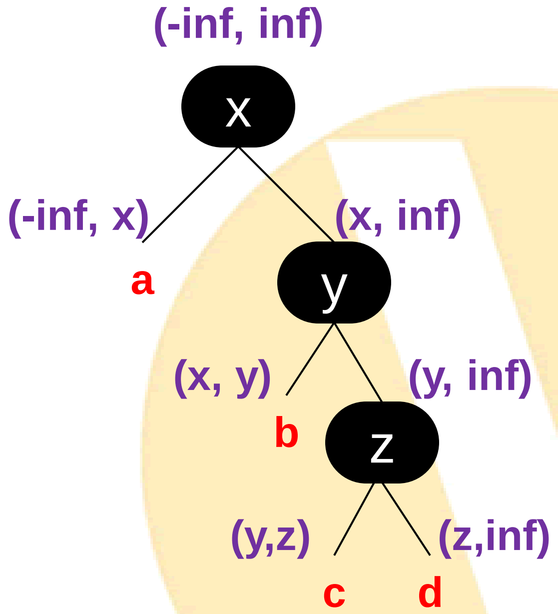
- Consider a binary search tree, what range of values could be in the subtree rooted at each node
  - At the root, any value could be in the "subtree"
  - At the first left child?
  - At the first right child?



BST Subtree Ranges

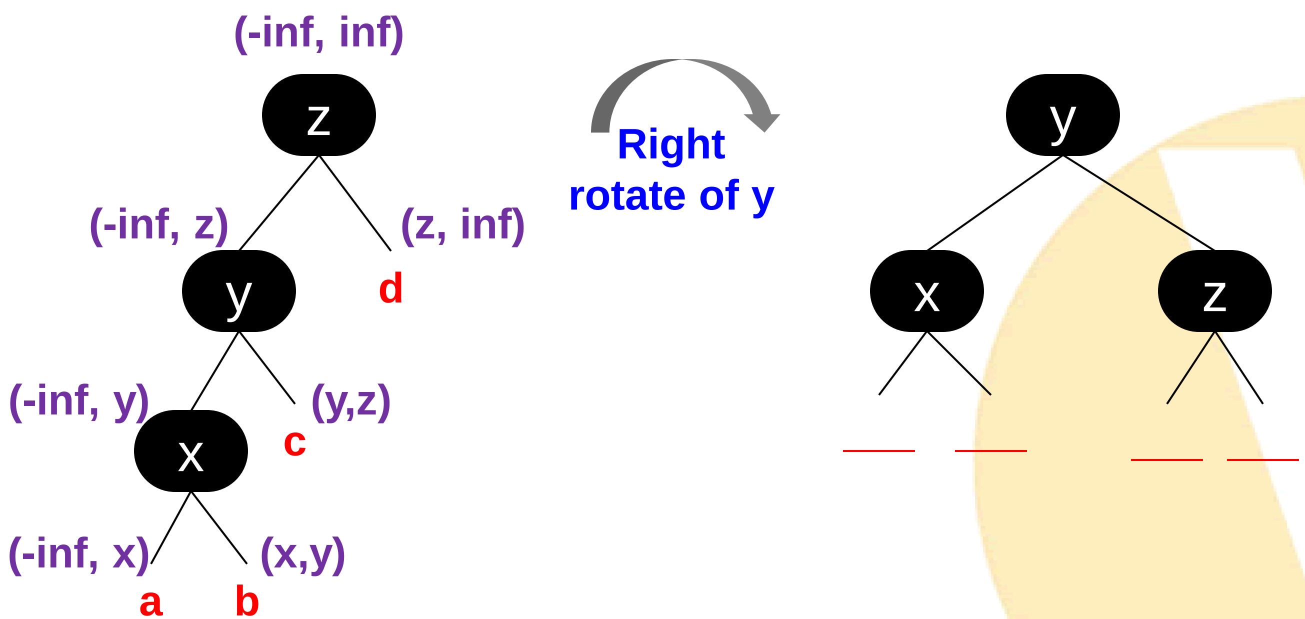
- Consider a binary search tree, what range of values could be in the subtree rooted at each node
  - At the root, any value could be in the "subtree"
  - At the first left child?
  - At the first right child?





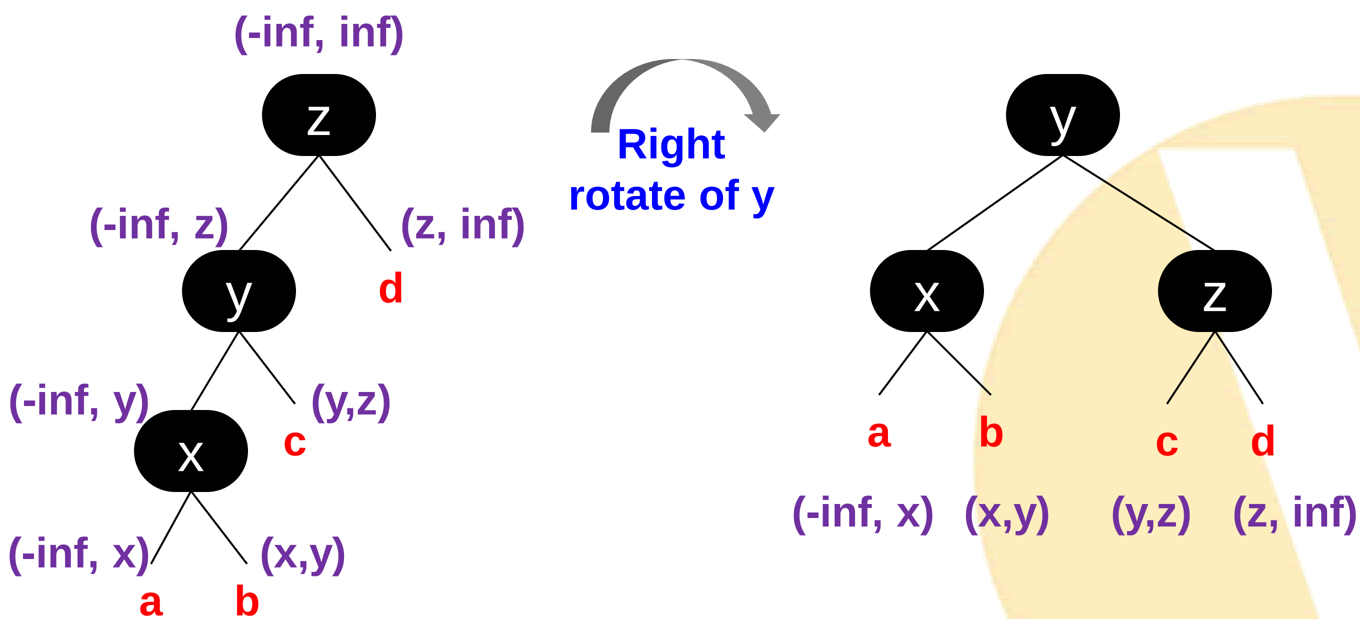
Right Rotation

- Define a right rotation as taking a left child, making it the parent and making the original parent the new right child
- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it…



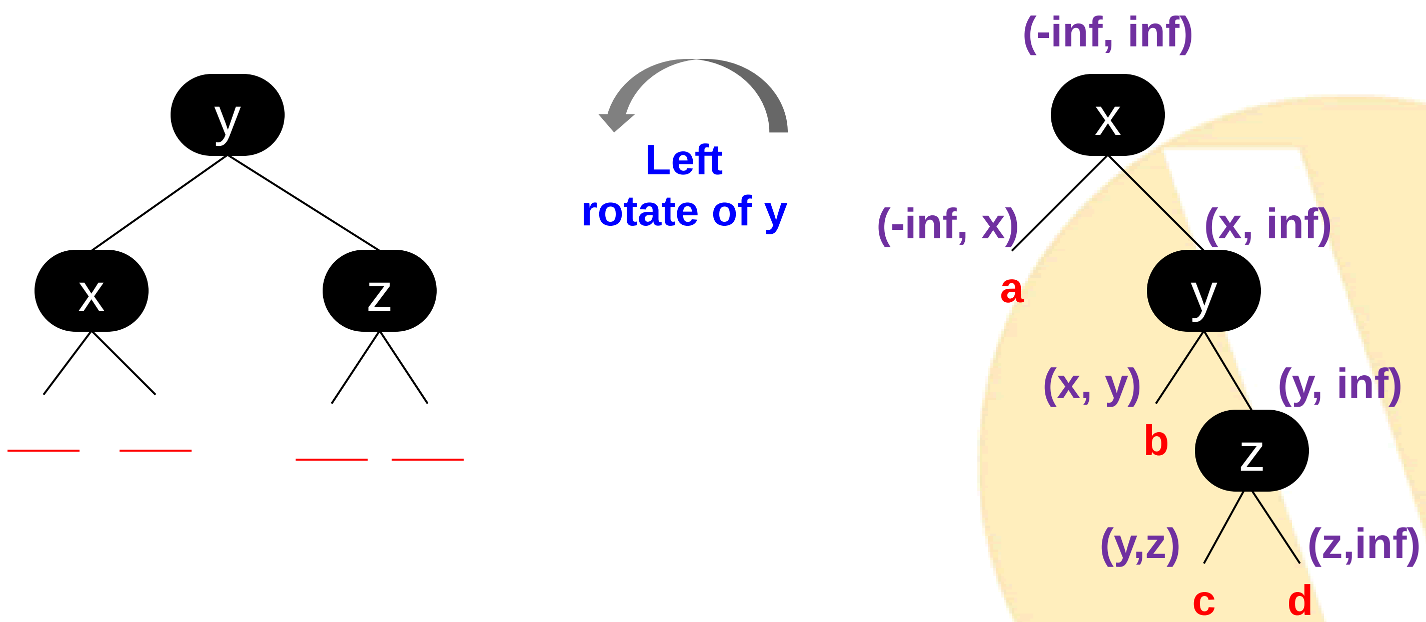
Right Rotation

- Define a right rotation as taking a left child, making it the parent and making the original parent the new right child
- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it…



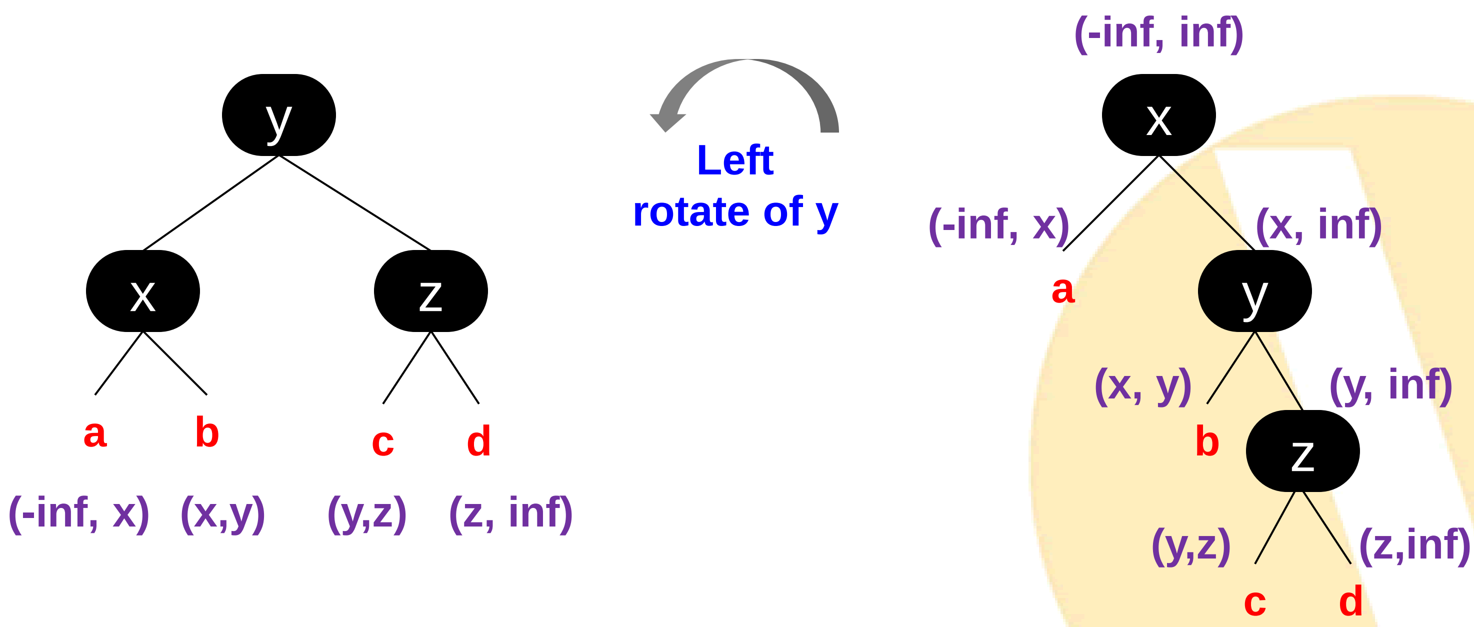
Left Rotation

- Define a left rotation as taking a right child, making it the parent and making the original parent the new left child
- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it…



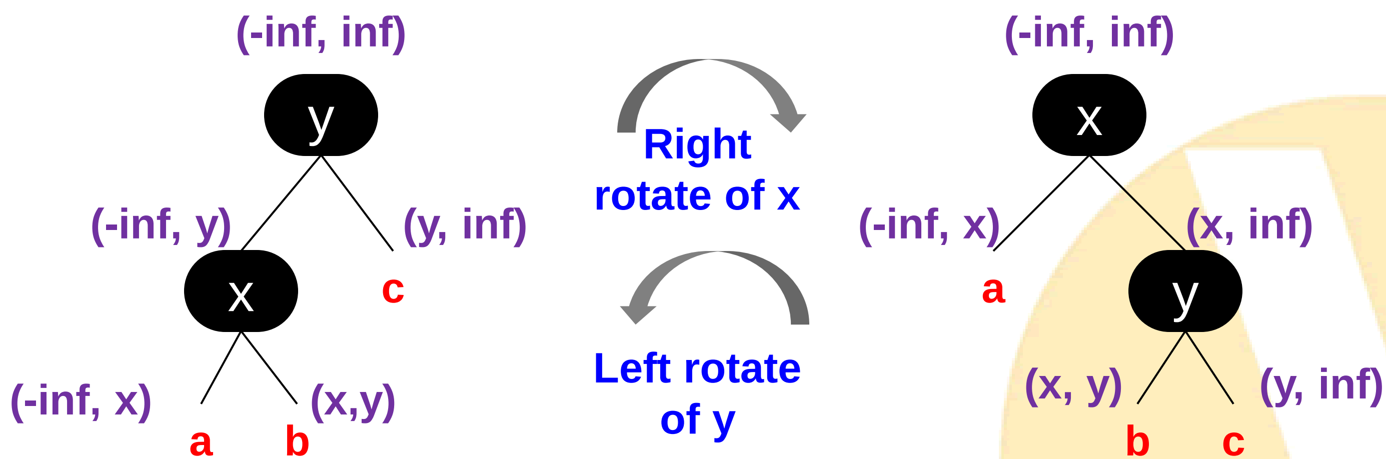
Left Rotation

- Define a left rotation as taking a right child, making it the parent and making the original parent the new left child
- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it…



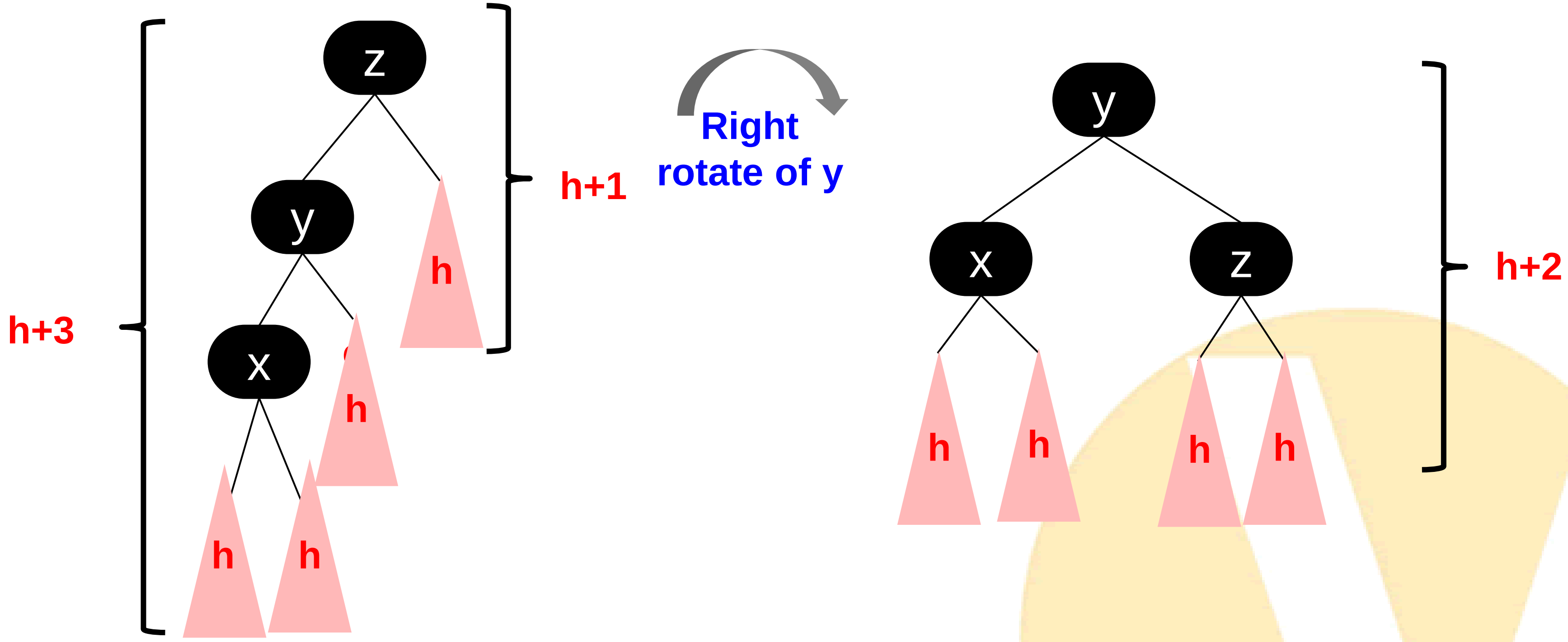
Rotations

- Define a right rotation as taking a left child, making it the parent and making the original parent the new right child
- Where do subtrees a, b, and c belong?
  - Use their ranges to reason about it…



Rotation's Effect on Height

- When we rotate, it serves to re-balance the tree



"Balanced" Binary Search Trees

RED BLACK TREES

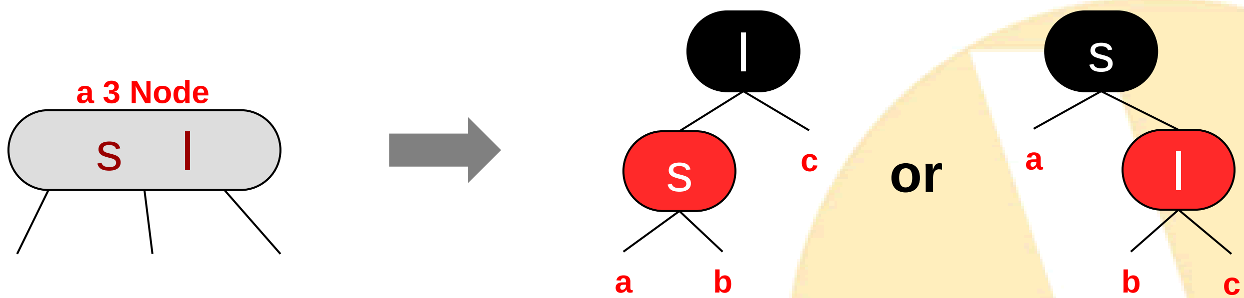
Red Black Trees

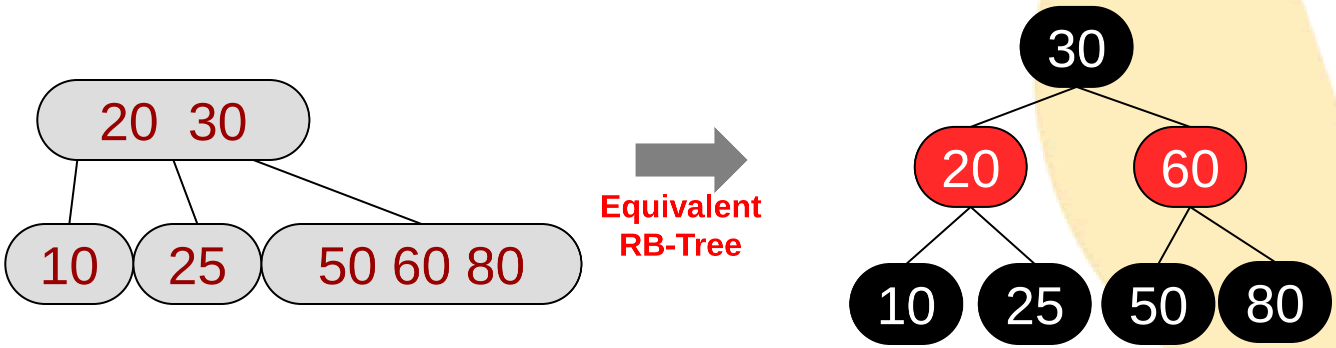
- A red-black tree is a binary search tree
  - Only 2 nodes (no 3- or 4-nodes)
  - Can be build from a 2-3-4 tree directly by converting each 3- and 4- nodes to multiple 2-nodes
- All 2-nodes means no wasted storage overheads
- Yields a "balanced" BST
- "Balanced" means that the height of an RB-Tree is at MOST **twice** the height of a 2-3-4 tree
  - Recall, height of 2-3-4 tree had an upper bound of $\log_2(n)$
  - Thus height or an RB-Tree is bounded by $2*\log_2 n$ which is still $O(\log_2(n))$

Red Black Trees

- A Red-Black tree is a "transformed" 2-3-4 tree
  - 3- and 4- nodes get converted to 2 nodes as follows
  - Red nodes are always ones that would join with their parent to become a 3- or 4-node in a 2-3-4 tree

S = Small
M = Median
L = Large



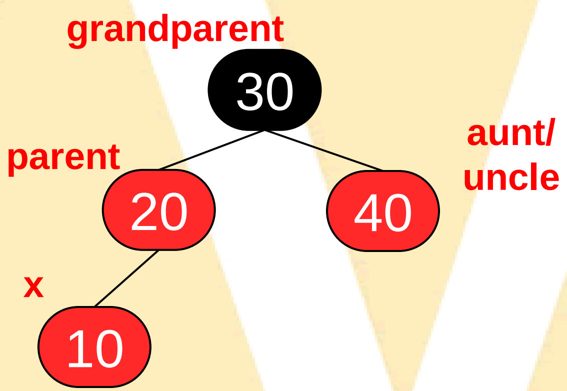


Red-Black Tree Properties

- Valid RB-Trees maintain the invariants that…
- 1. No path from root to leaf has two consecutive red nodes (i.e. a parent and its child cannot both be red)
  - Since red nodes are just the extra values of a 3- or 4-node from 2-3-4 trees you can't have 2 consecutive red nodes
- 2. Every path from leaf to root has the same number of black children
  - Recall, 2-3-4 trees are full (same height from leaf to root for all paths)
  - Also remember each 2, 3-, or 4- nodes turns into a black node ***plus*** 0, 1, or 2 red node children
- 3. (Usually) the root is black

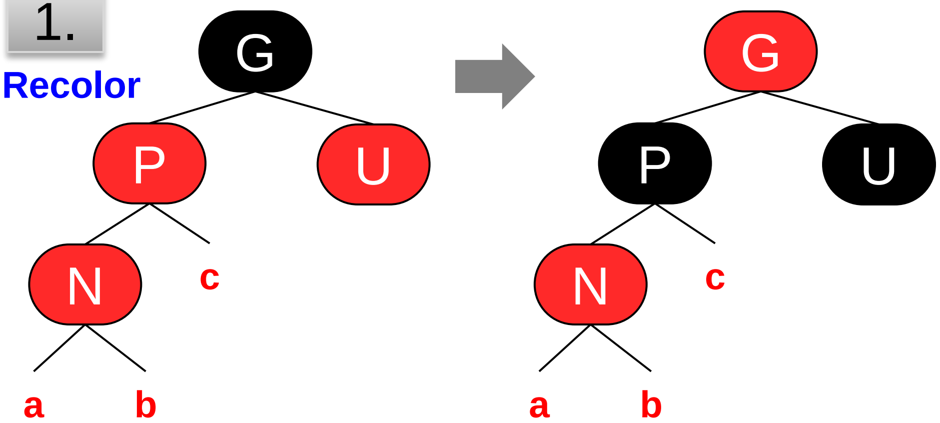
Red-Black Insertion

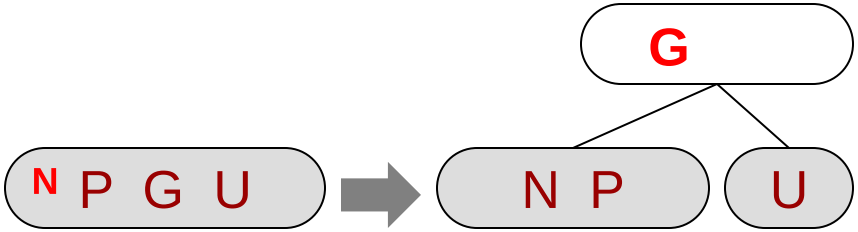
- Insertion Algorithm:
  - 1. Insert node into normal BST location (at a leaf location) and color it RED
  - 2a. If the node's parent is black (i.e. the leaf used to be a 2-node) then DONE (i.e. you now have what was a 3- or 4-node)
  - 2b. Else perform fixTree transformations then repeat step 2 on the parent or grandparent (whoever is red)
- fixTree involves either
  - recoloring or
  - 1 or 2 rotations and recoloring
- Which case of fixTree you perform depends on the color of the new node's "aunt/uncle"



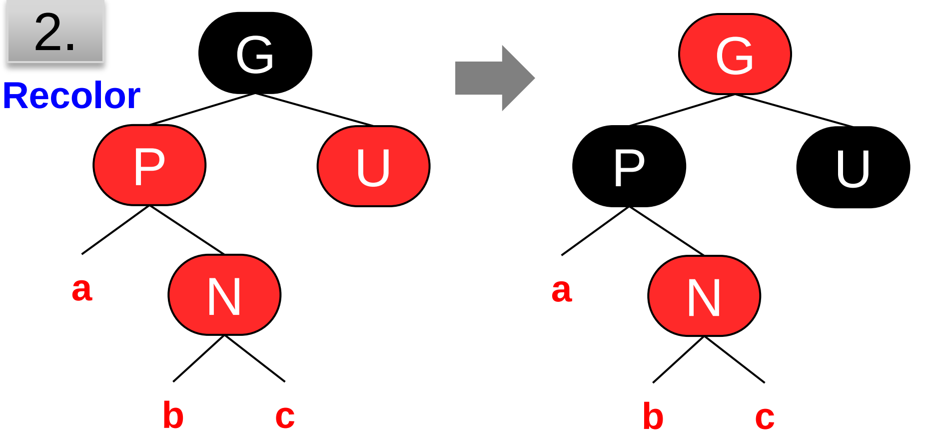
fixTree Cases

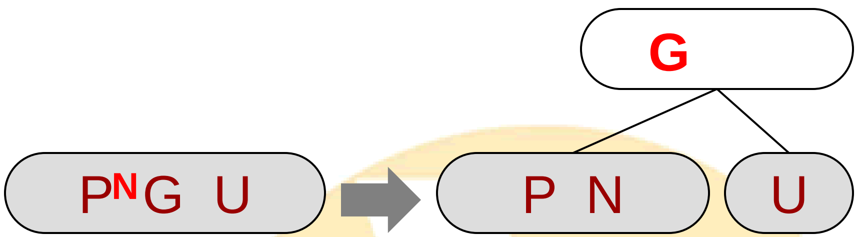
1.
Recolor



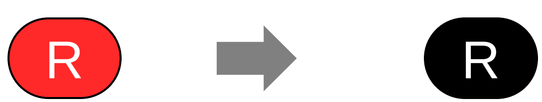


2.
Recolor





3.
Recolor Root

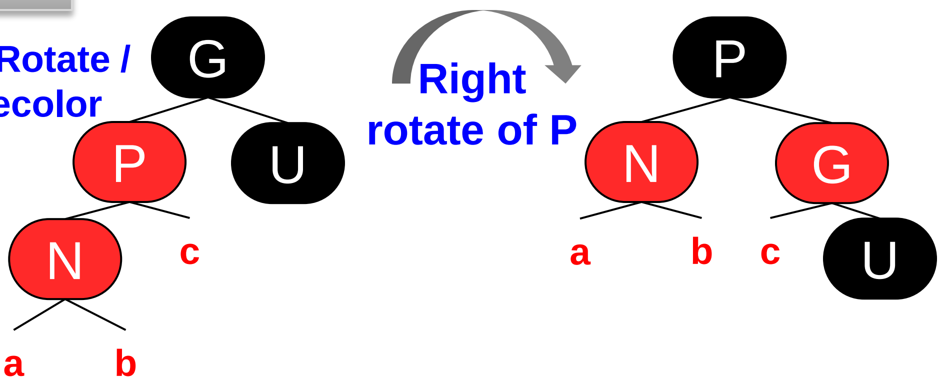


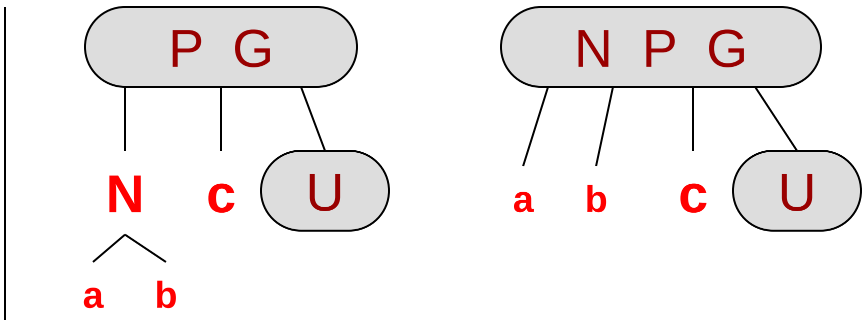
Note: For insertion/removal algorithm we consider non-existent leaf nodes as black nodes

fixTree Cases

4.

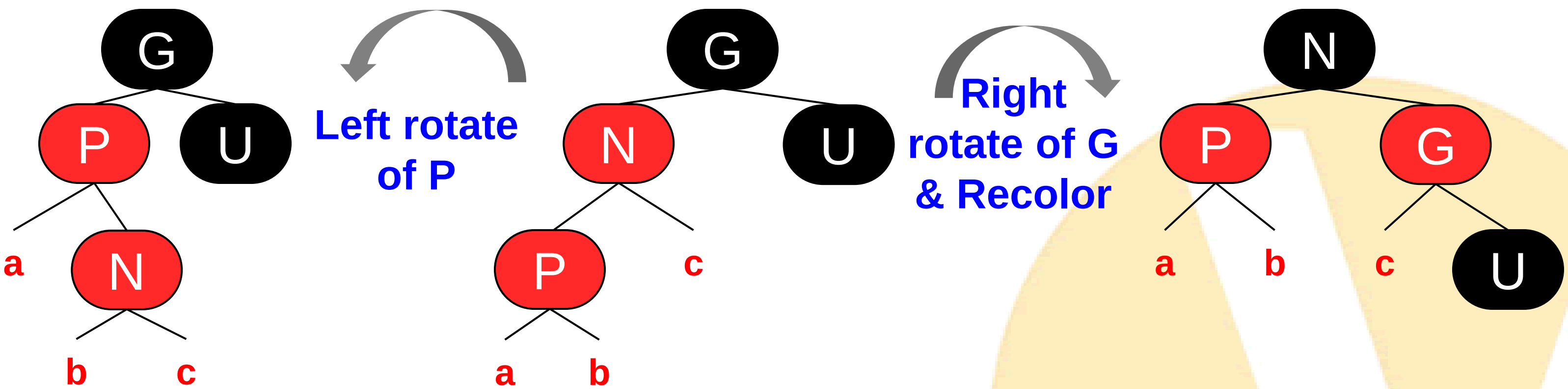
1 Rotate / Recolor

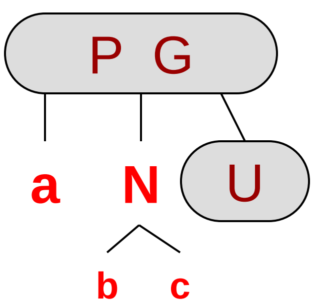


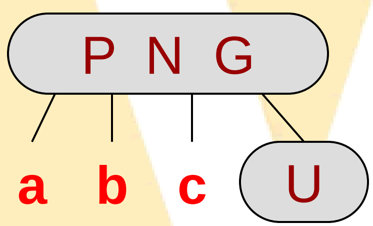


5.

2 Rotates / Recolor



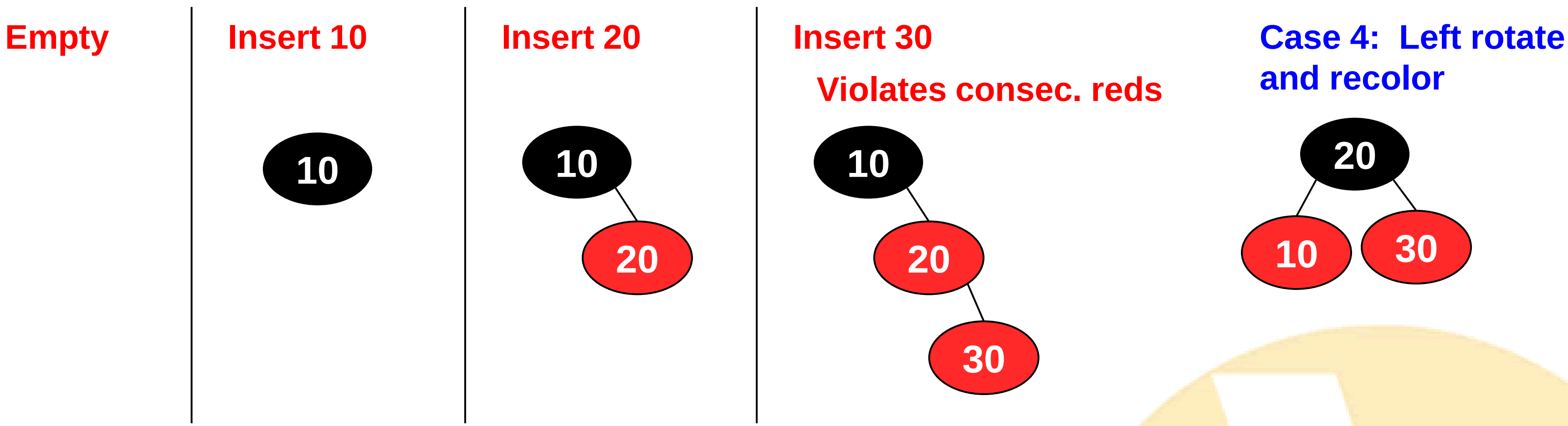


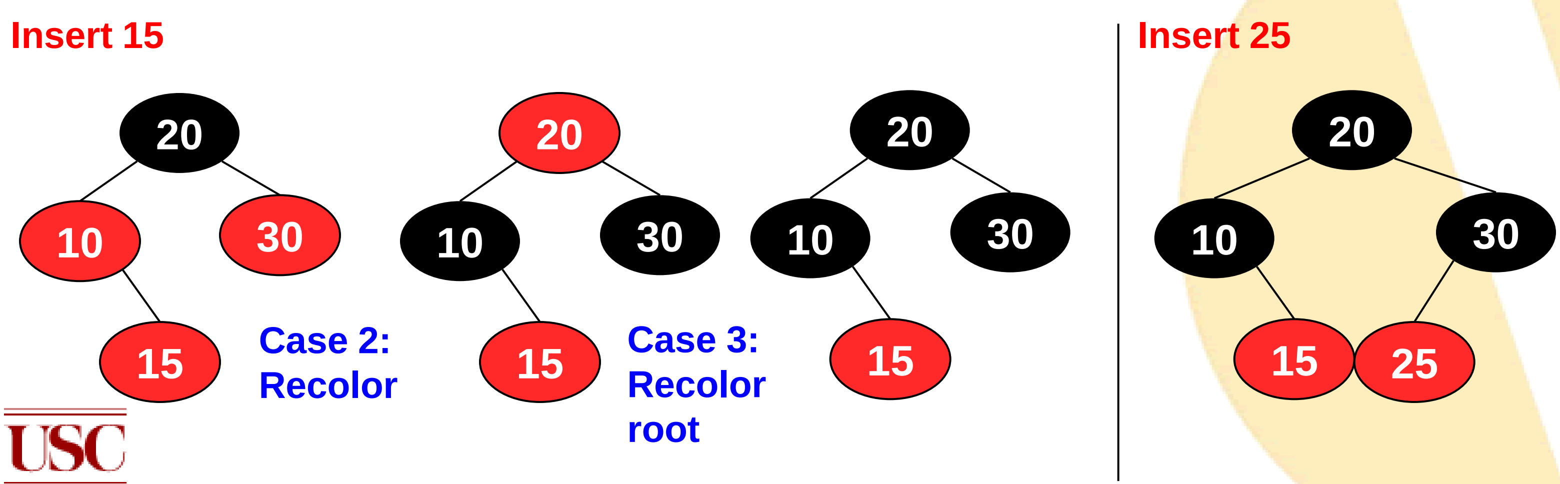


www.cse.ohio-state.edu/~gurari/course/cis680Ch11.html

Insertion

- Insert 10, 20, 30, 15, 25, 12, 5, 3, 8

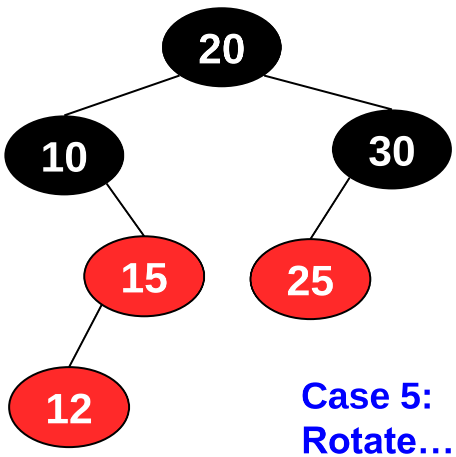




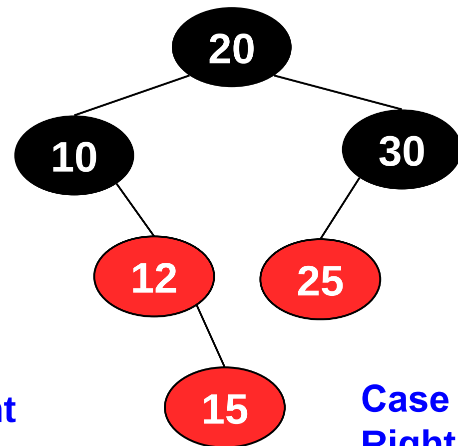
Insertion

- Insert 10, 20, 30, 15, 25, 12, 5, 3, 8

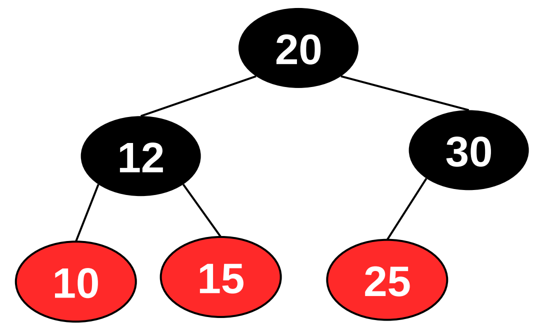
Insert 12



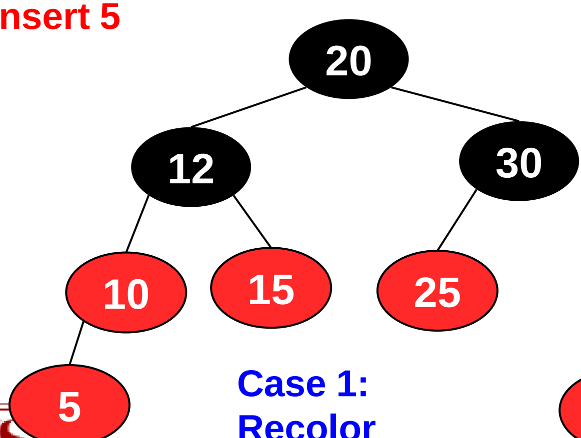
Case 5: Right Rotate…



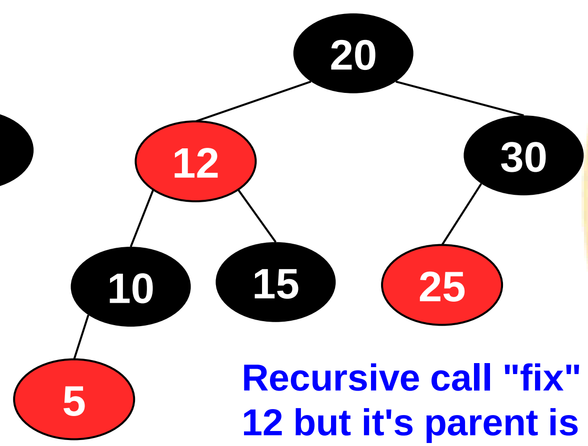
Case 5: … Right Rotate and recolor



Insert 5



Case 1: Recolor

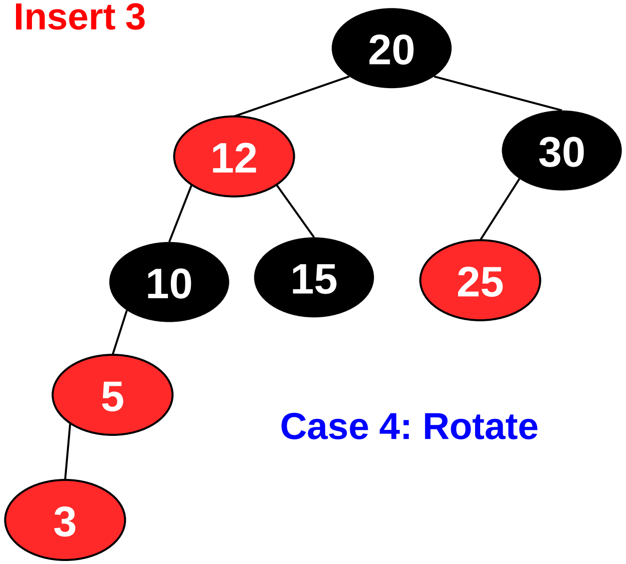


Recursive call "fix" on 12 but it's parent is black so we're done

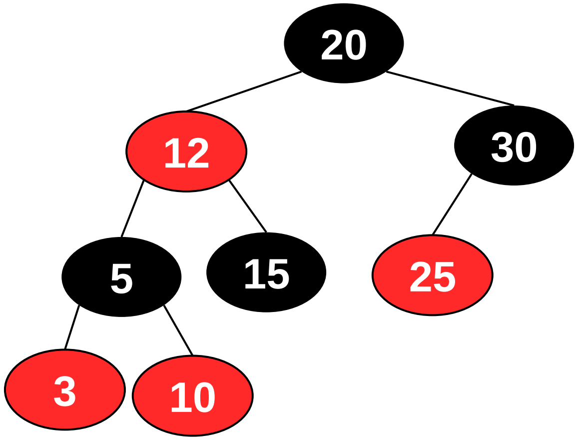
Insertion

- Insert 10, 20, 30, 15, 25, 12, 5, 3, 8

Insert 3



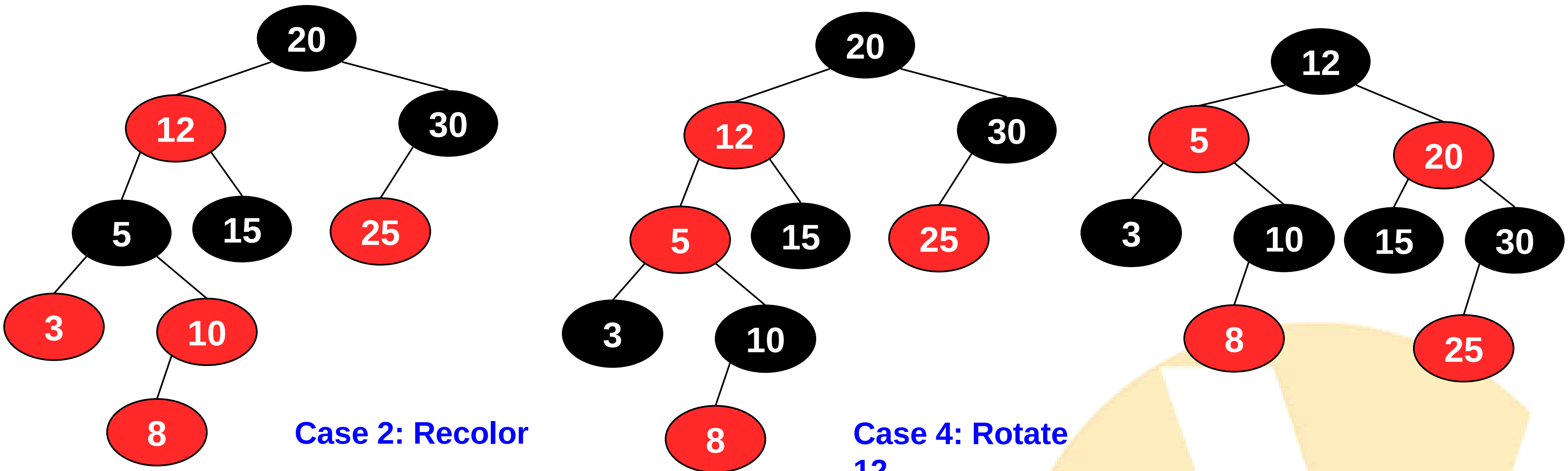
Case 4: Rotate



Insertion

- Insert 10, 20, 30, 15, 25, 12, 5, 3, 8

Insert 8

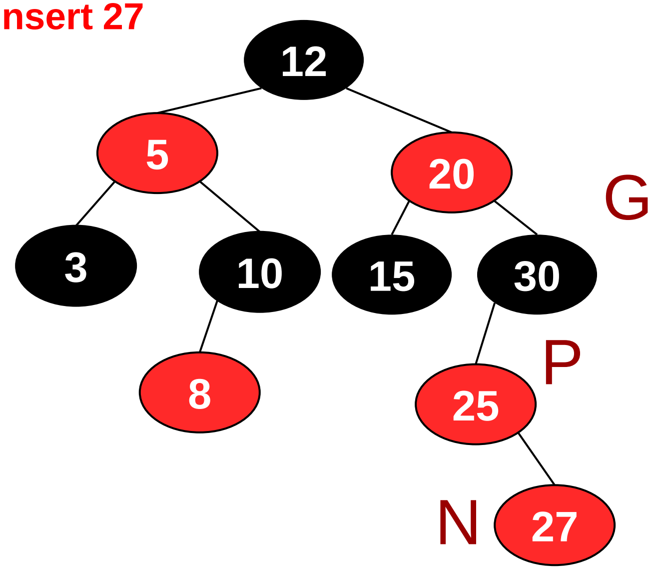


Case 2: Recolor

Case 4: Rotate 12

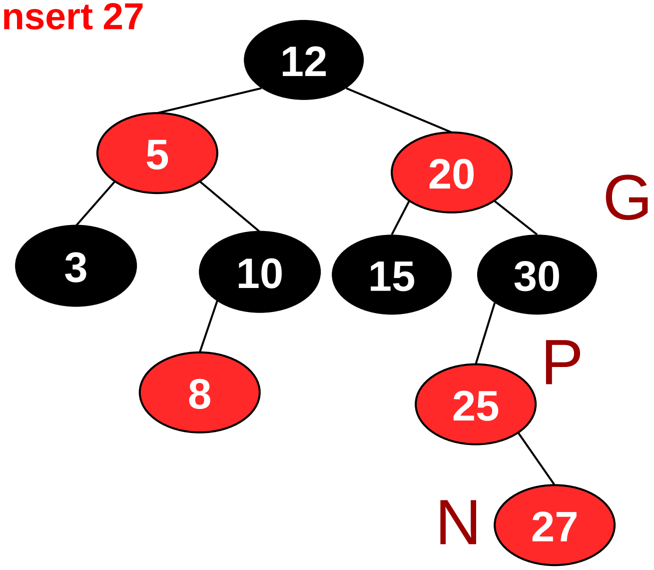
Insertion Exercise 1

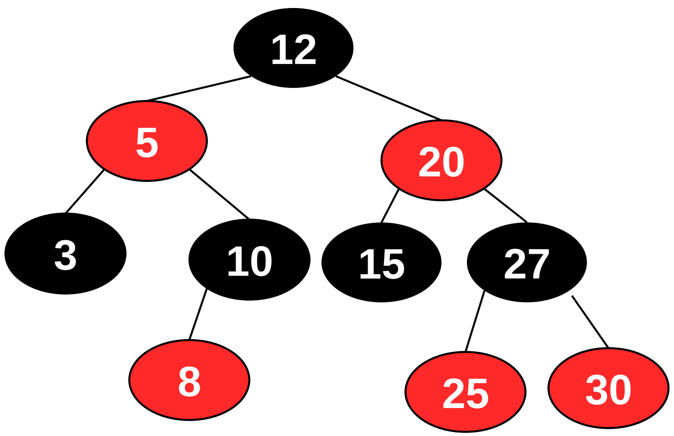
Insert 27



Insertion Exercise 1

Insert 27



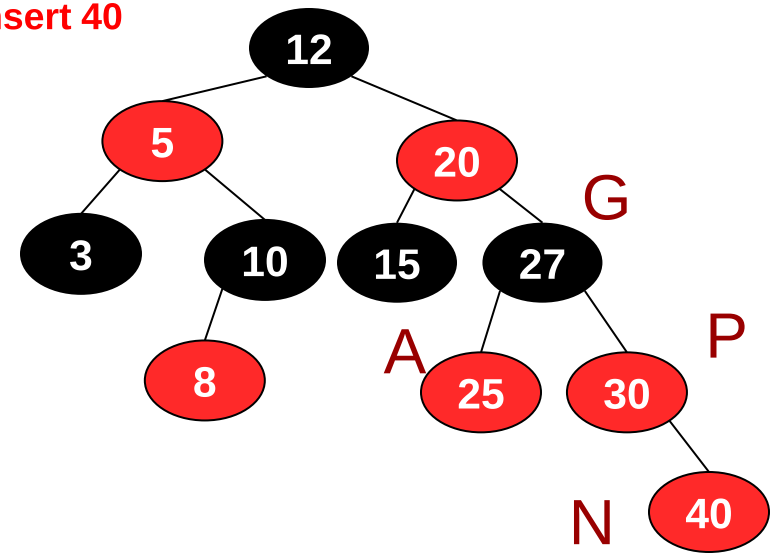


This is case 5.

1. Left rotate around P
2. Right rotate around N
3. Recolor

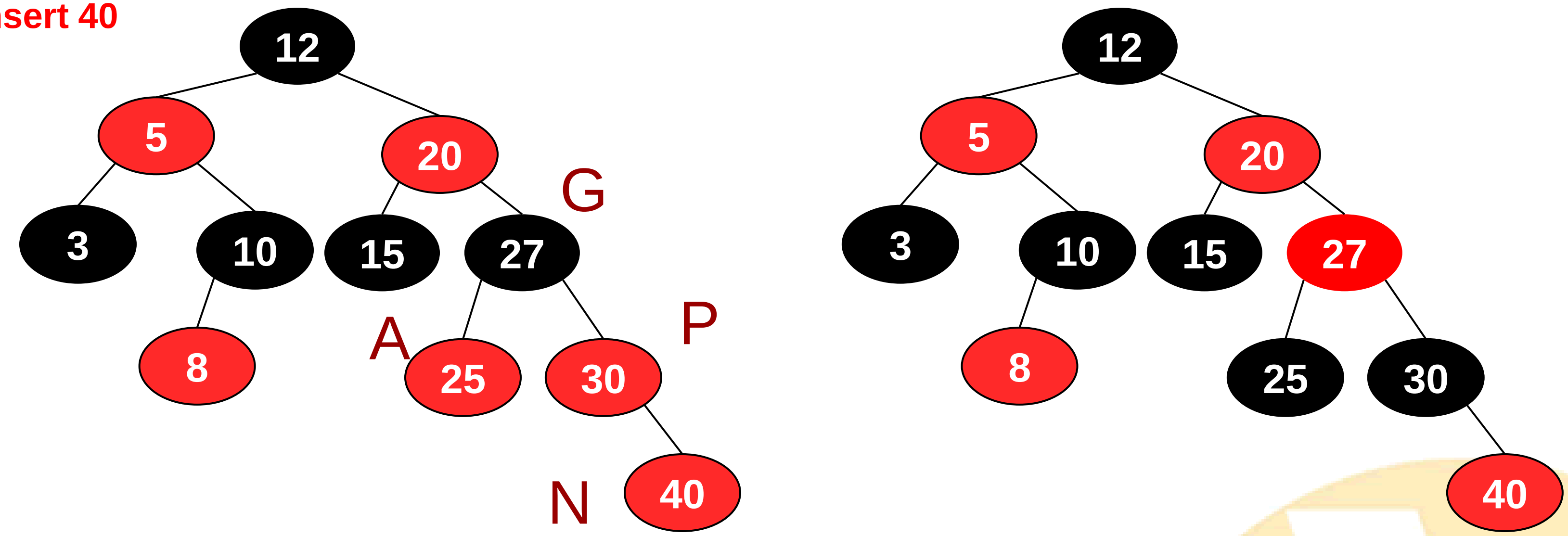
Insertion Exercise 2

Insert 40



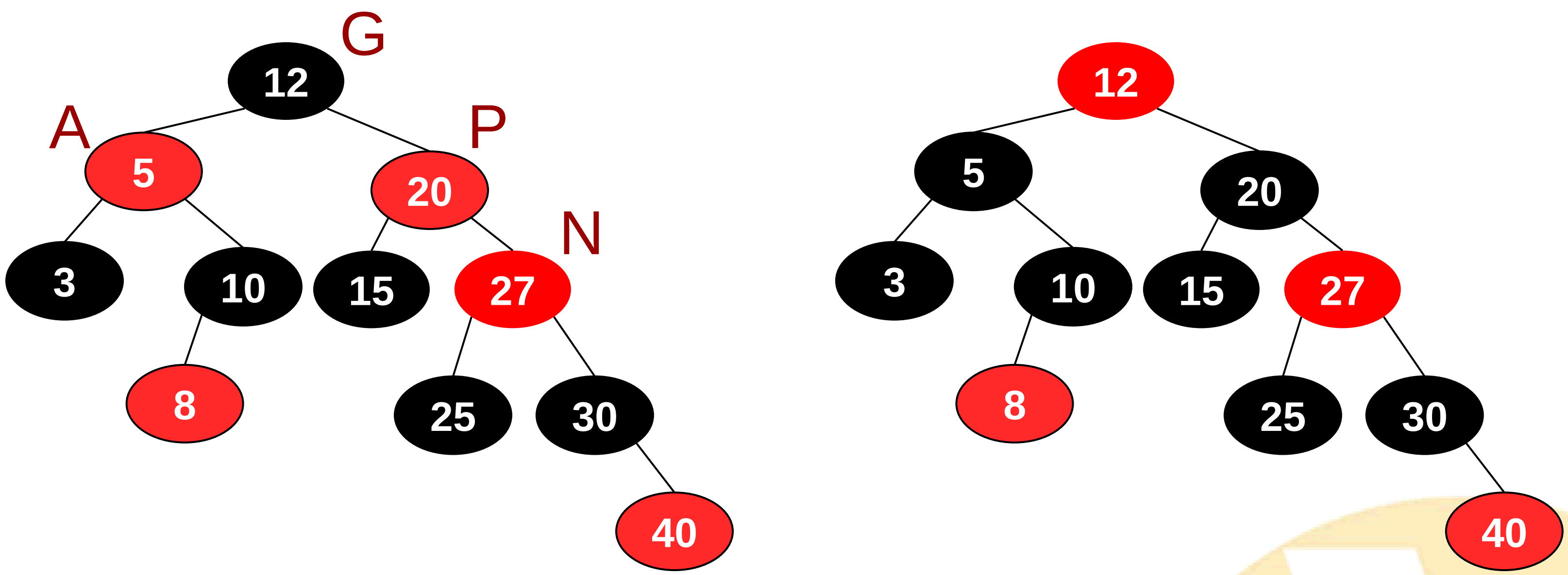
Insertion Exercise 2

Insert 40



Aunt and Parent are the same color. So recolor aunt, parent, and grandparent.

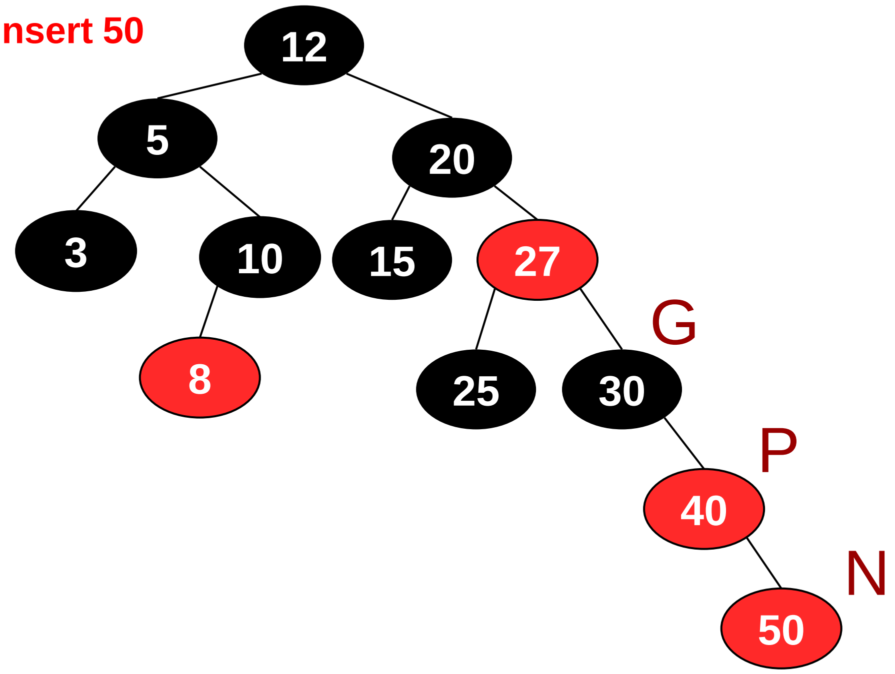
Insertion Exercise 2



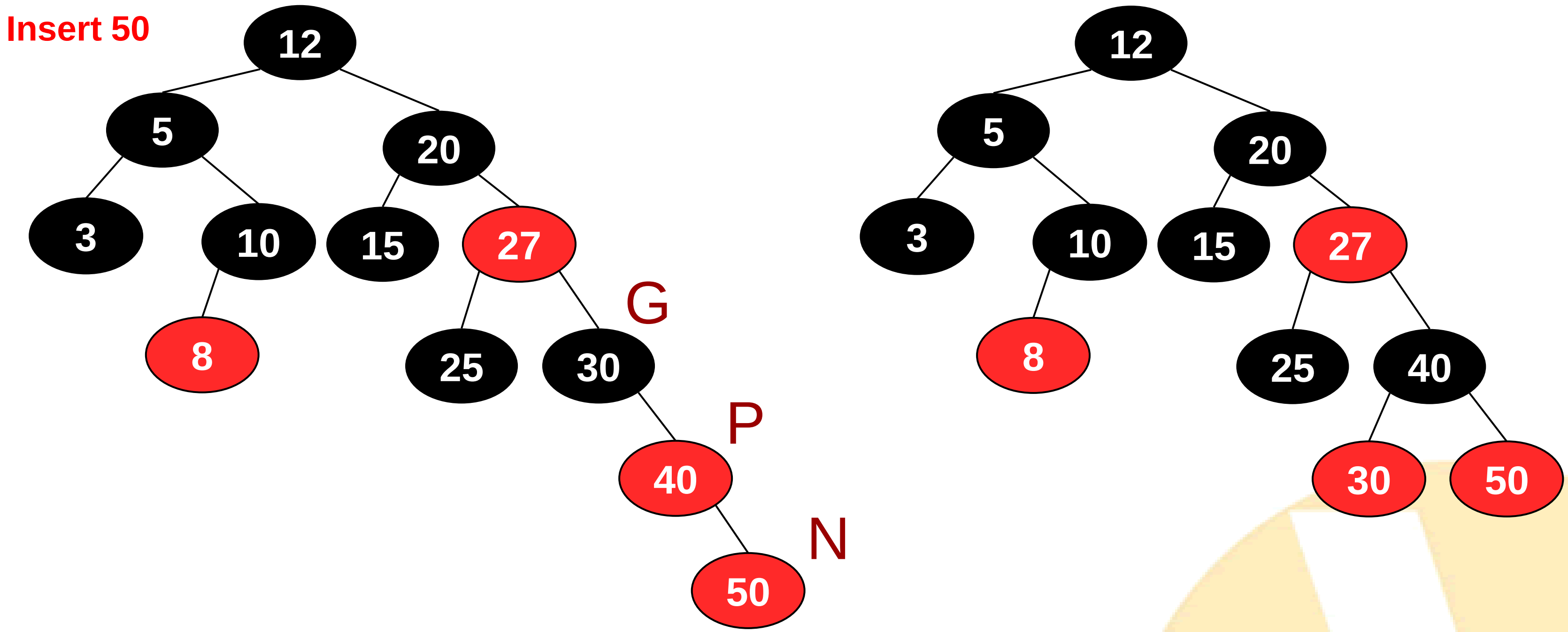
Aunt and Parent are the same color. So recolor aunt, parent, and grandparent.

Insertion Exercise 3

Insert 50



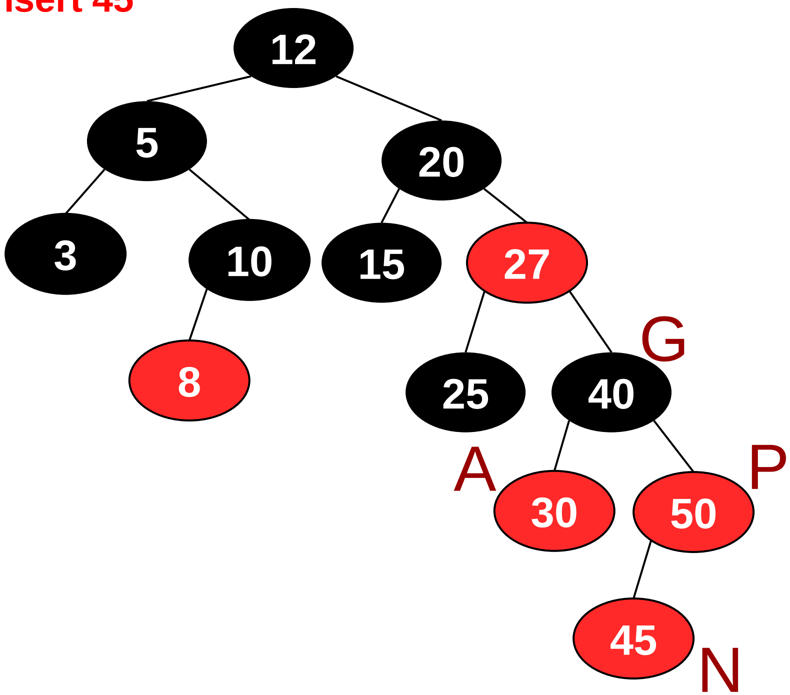
Insertion Exercise 3



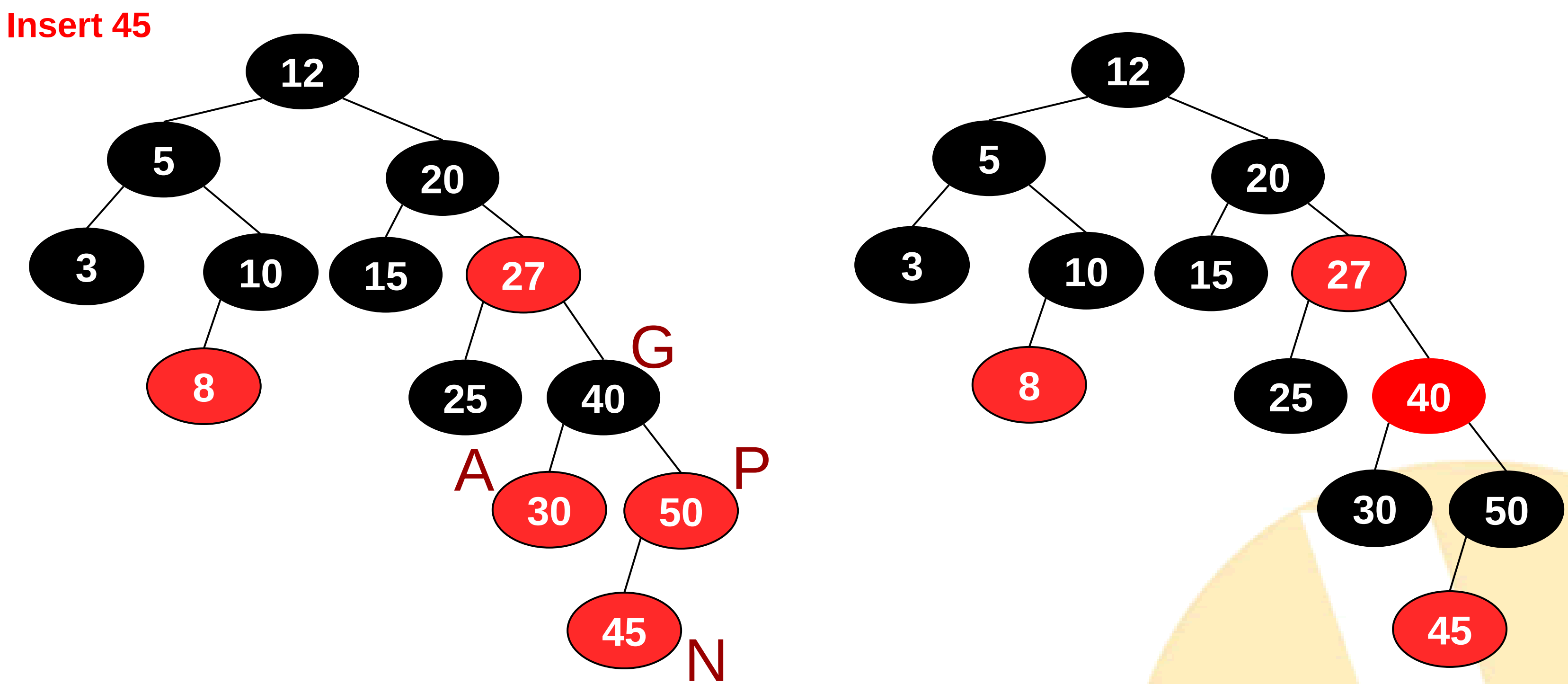
Remember, empty nodes are black.
Do a left rotation around P and recolor.

Insertion Exercise 4

Insert 45

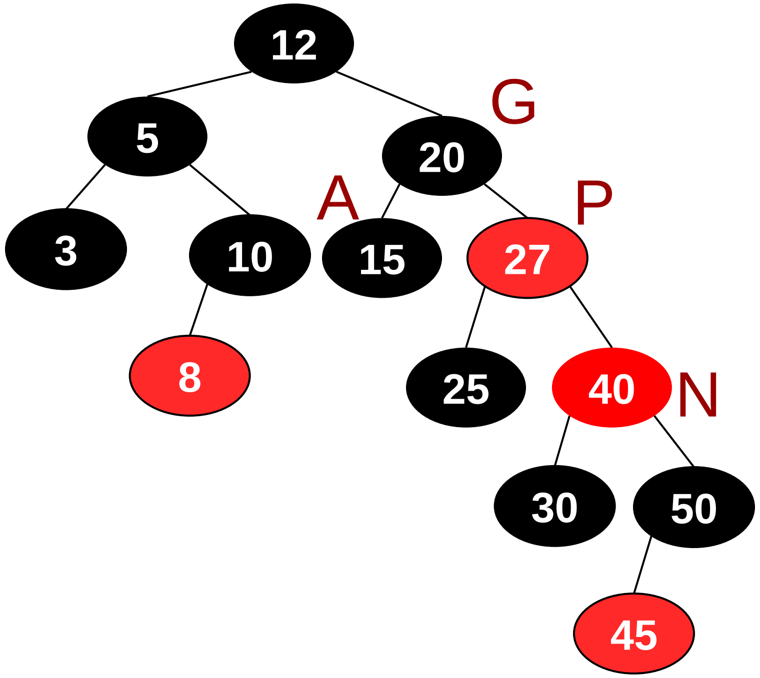


Insertion Exercise 4

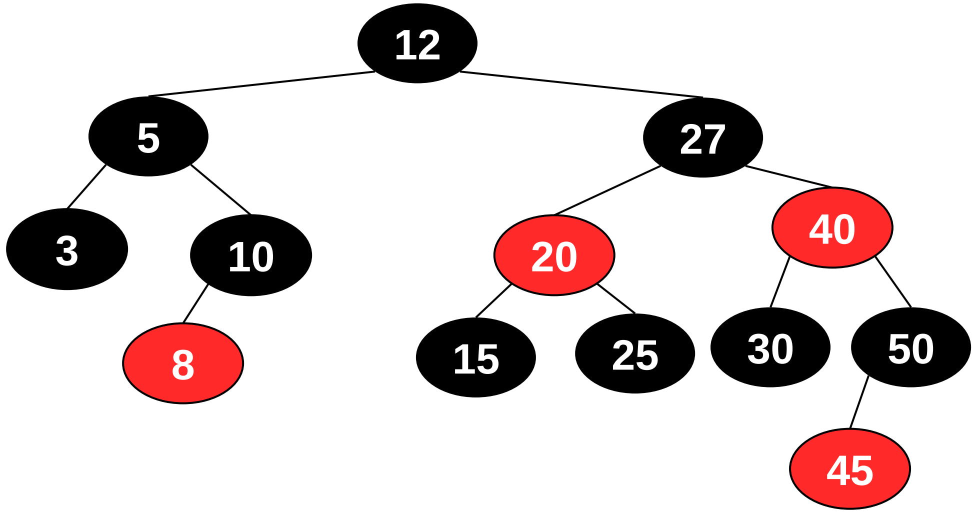


Aunt and Parent are the same color.
Just recolor.

Insertion Exercise 4

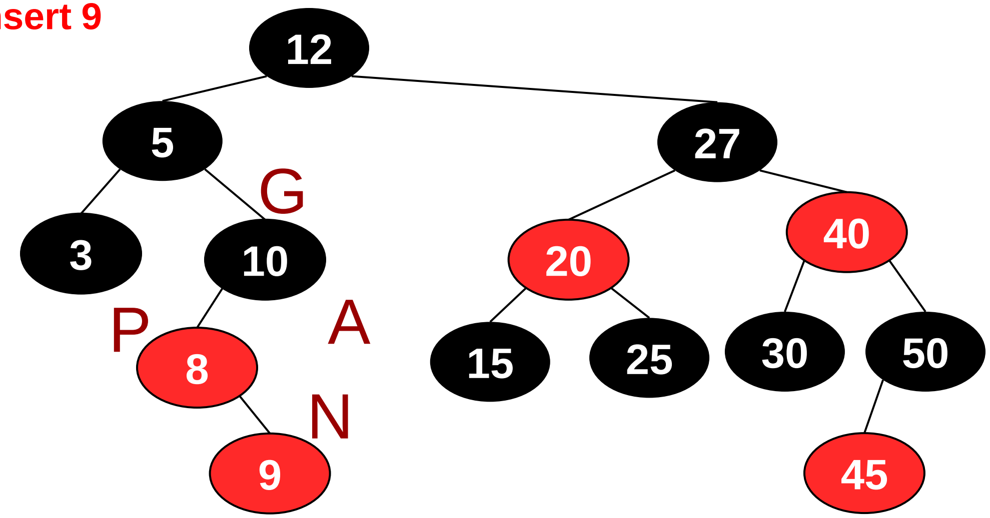


Final Result

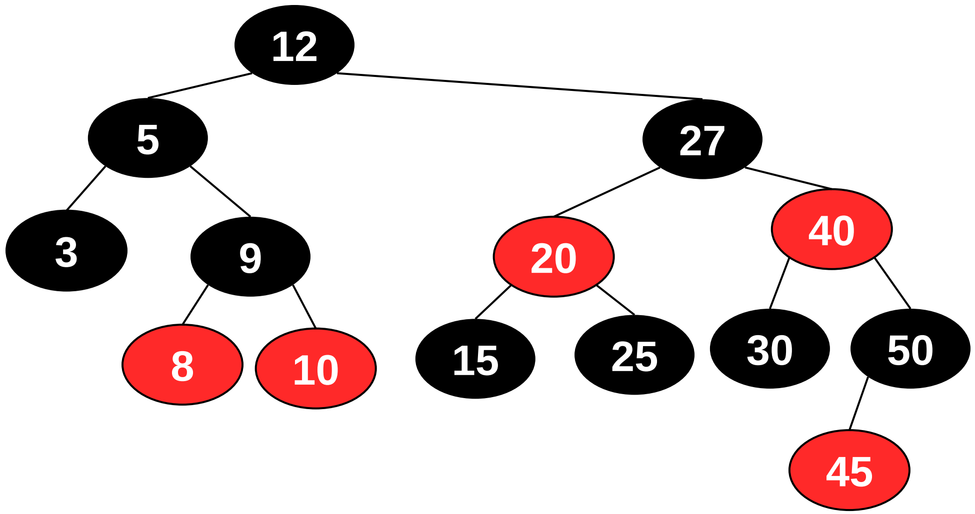


Insertion Exercise 5

Insert 9



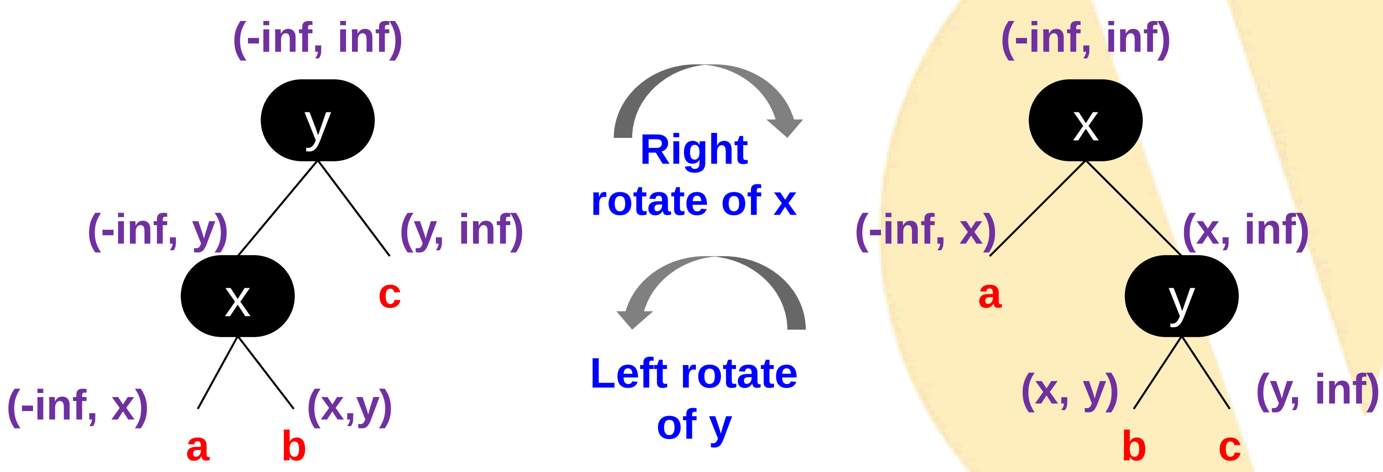
Insertion Exercise 5



RB TREE IMPLEMENTATION

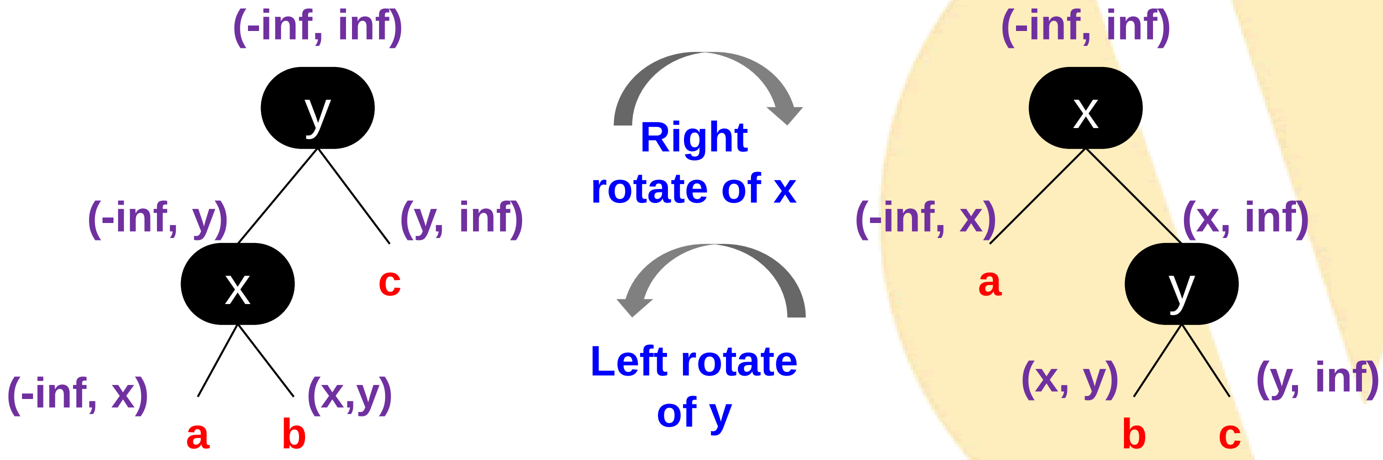
Hints

- Implement private methods:
  - findMyUncle()
  - AmIaRightChild()
  - AmIaLeftChild()
  - RightRotate
  - LeftRotate
    - Need to change x's parent, y's parent, b's parent, x's right, y's left, x's parent's left or right, and maybe root



Hints

- You have to fix the tree after insertion if…
- Watch out for traversing NULL pointers
  – node->parent->parent
  – However, if you need to fix the tree your grandparent…
- Cases break down on uncle's color
  – If an uncle doesn't exist (i.e. is NULL), he is (color?)…

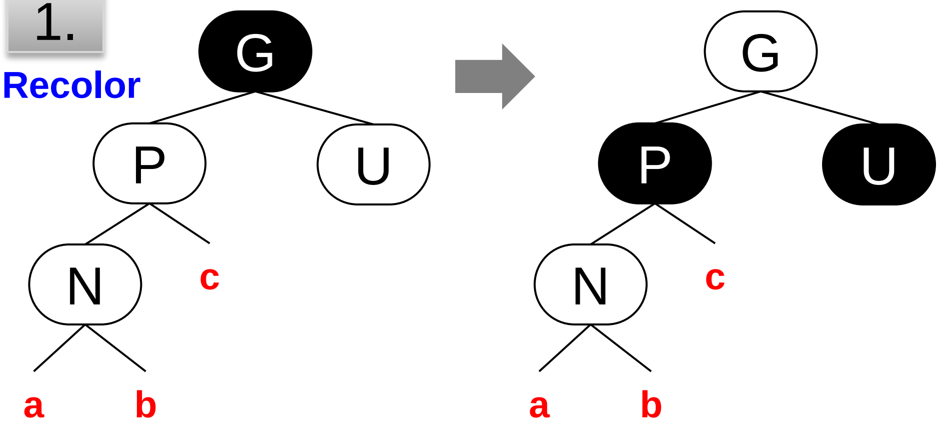


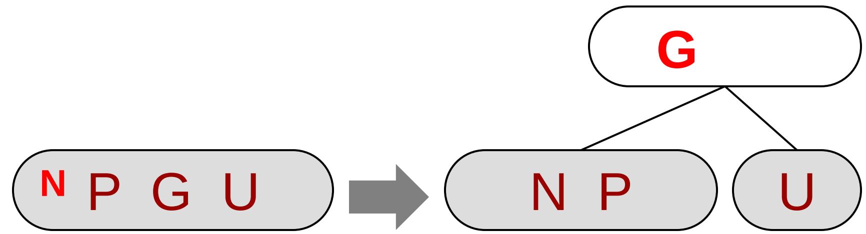
FOR PRINT

fixTree Cases

1.

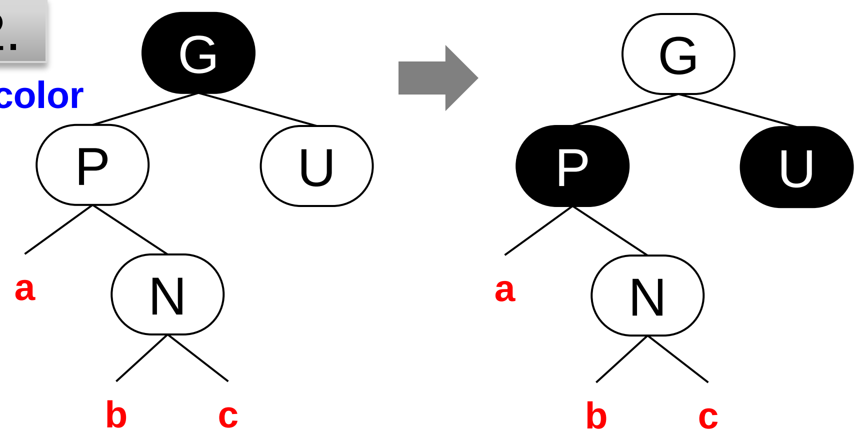
Recolor

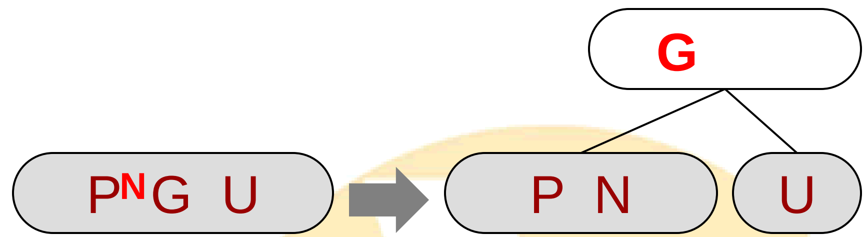




2.

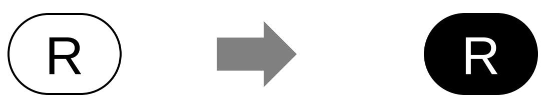
Recolor





3.

Recolor Root

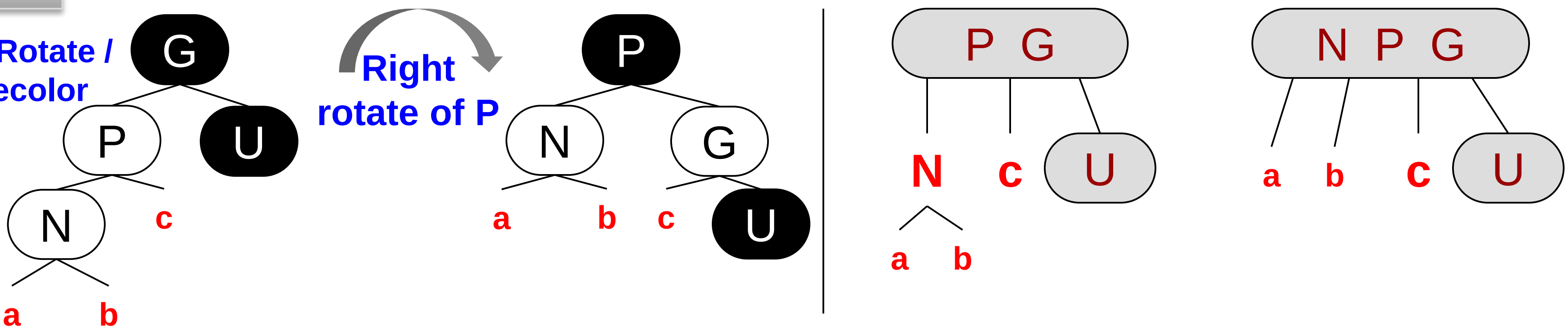


Note: For insertion/removal algorithm we consider non-existent leaf nodes as black nodes

fixTree Cases

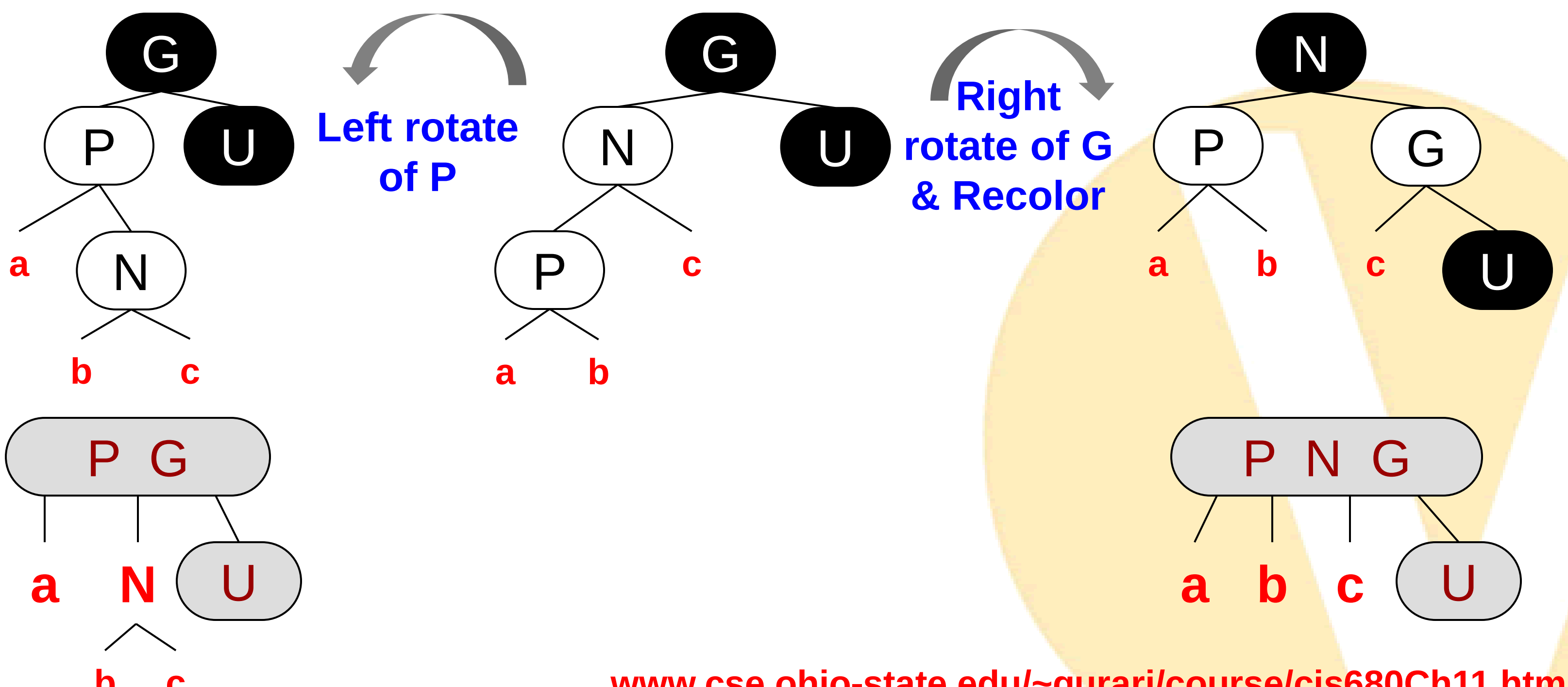
4.

1 Rotate / Recolor



5.

2 Rotates / Recolor



www.cse.ohio-state.edu/~gurari/course/cis680Ch11.html