CSCI 104
B-Trees (2-3, 2-3-4) and Red/Black Trees

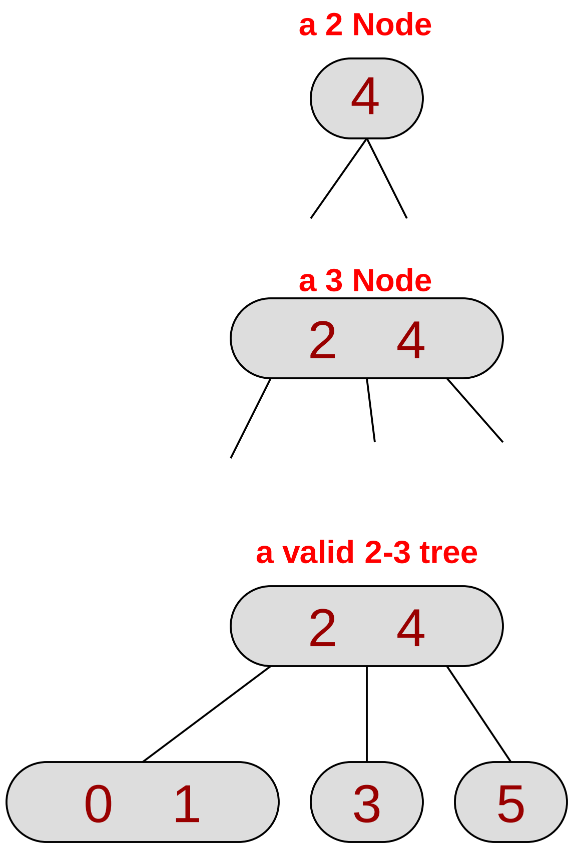
Mark Redekopp
David Kempe

An example of B-Trees

2-3 TREES

Definition

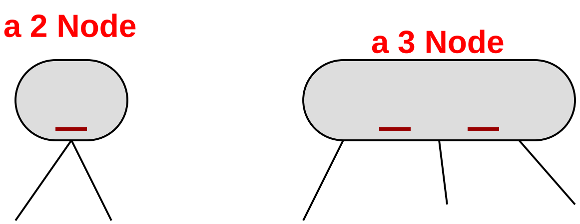
- 2-3 Tree is a tree where
  - Non-leaf nodes have 1 value & 2 children or 2 values and 3 children
  - All leaves are at the same level
- Following the line of reasoning…
  - All leaves at the same level with internal nodes having at least 2 children implies a (**full / complete**) tree
    - FULL (Recall complete just means the lower level is filled left to right but not full)
  - A full tree with n nodes implies…
    - Height that is bounded by $\log_2(n)$



Implementation of 2- & 3-Nodes

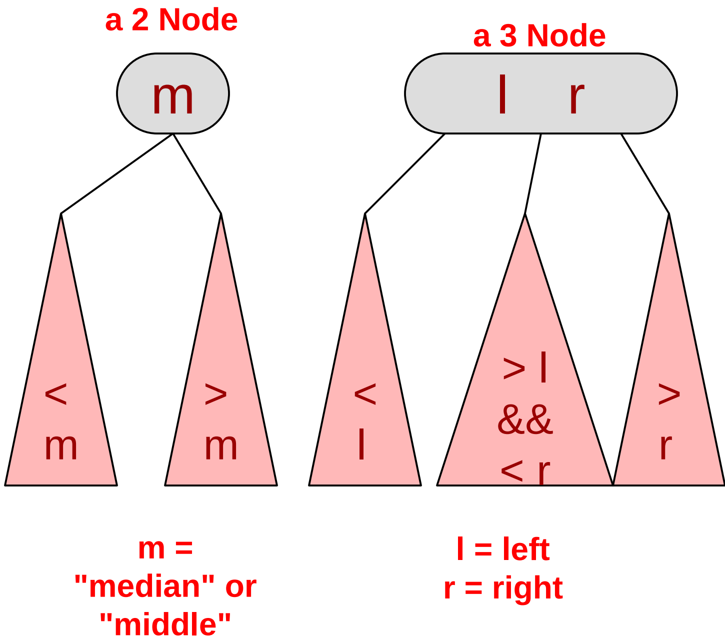
- You will see that at different times 2 nodes may have to be upgraded to 3 nodes
- To model these nodes we plan for the worst case…a 3 node
- This requires wasted storage for 2 nodes

```
template <typename T>
struct Item23 {
  T val1;
  T val2;
  Item23<T>* left;
  Item23<T>* mid;
  Item23<T>* right;
  bool twoNode;
};
```



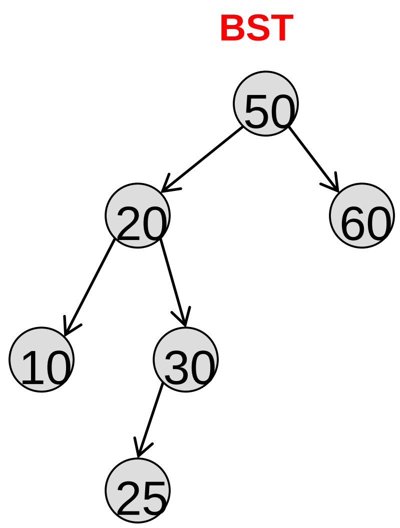
2-3 Search Trees

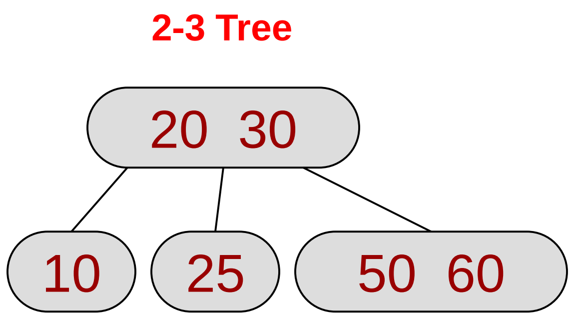
- Similar properties as a BST
- 2-3 Search Tree
  - If a 2 Node with value, *m*
    - Left subtree nodes are < node value
    - Right subtree nodes are > node value
  - If a 3 Node with value, *l* and *r*
    - Left subtree nodes are < *l*
    - Middle subtree > *l* and < *r*
    - Right subtree nodes are > *r*
- 2-3 Trees are almost always used as search trees, so from now on if we say 2-3 tree we mean 2-3 search tree



2-3 Search Tree

- Binary search tree compared to 2-3 tree
- Check if 55 is in the tree?

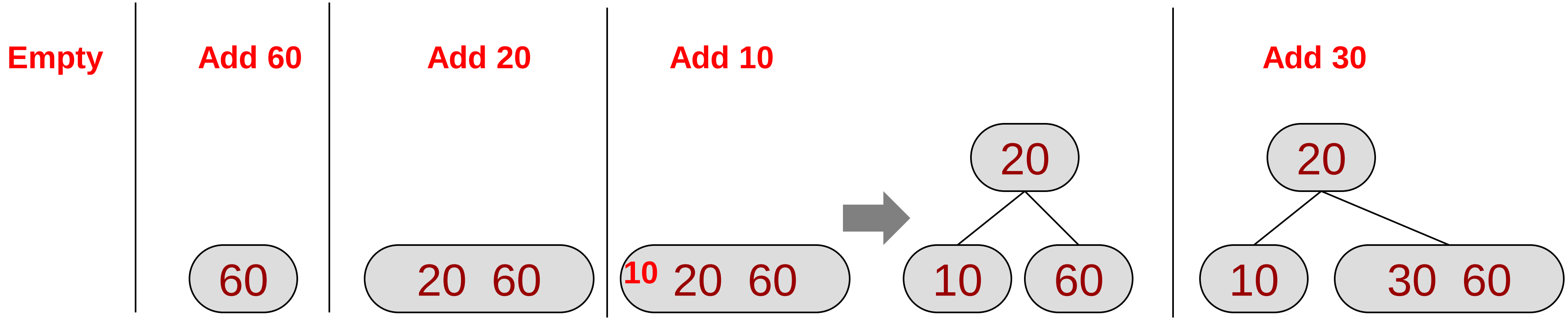




2-3 Insertion Algorithm

- Key: Since all leaves must be at the same level ("leaves always have their feet on the ground"), insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2-nodes with the smallest value as the left, biggest as the right, and median value promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 60, 20, 10, 30, 25, 50, 80

Key: Any time a node accumulates 3 values, split it into single valued nodes (i.e. 2-nodes) and promote the median

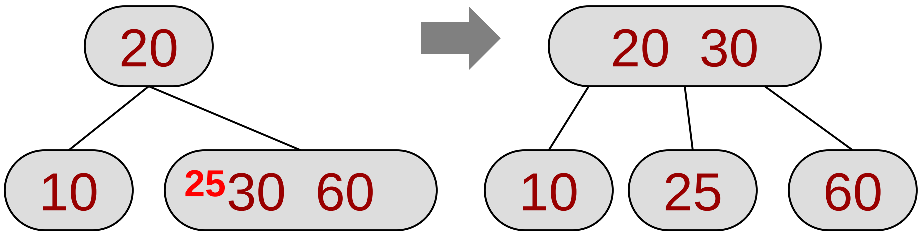


2-3 Insertion Algorithm

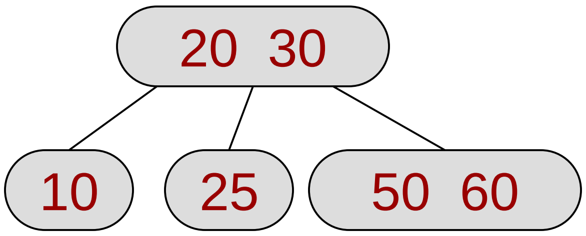
- Key: Since all leaves must be at the same level ("leaves always have their feet on the ground"), insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2-nodes with the smallest value as the left, biggest as the right, and median value promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 60, 20, 10, 30, 25, 50, 80

Key: Any time a node accumulates 3 values, split it into single valued nodes (i.e. 2-nodes) and promote the median

Add 25



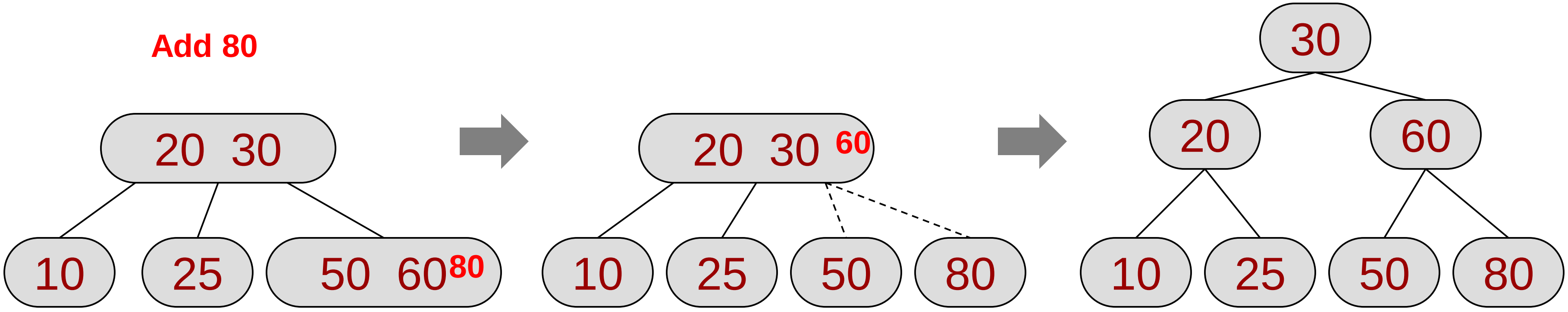
Add 50



2-3 Insertion Algorithm

- Key: Since all leaves must be at the same level ("leaves always have their feet on the ground"), insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2 nodes with the smallest value as the left, biggest as the right, and median value promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 60, 20, 10, 30, 25, 50, 80

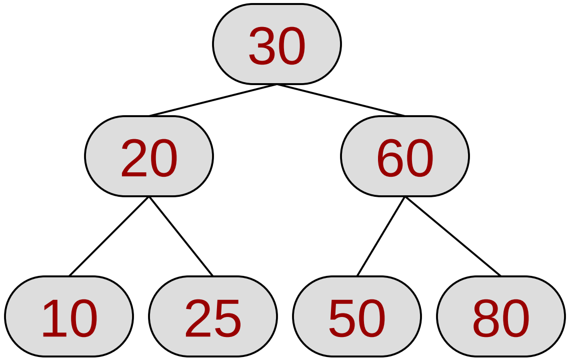
Key: Any time a node accumulates 3 values, split it into single valued nodes (i.e. 2-nodes) and promote the median



2-3 Insertion Algorithm

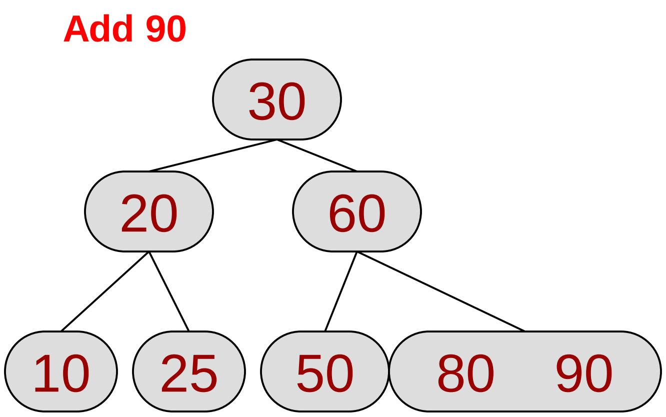
- Key: Since all leaves must be at the same level ("leaves always have their feet on the ground"), insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2 nodes with the smallest value as the left, biggest as the right, and median value promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 90,91,92, 93

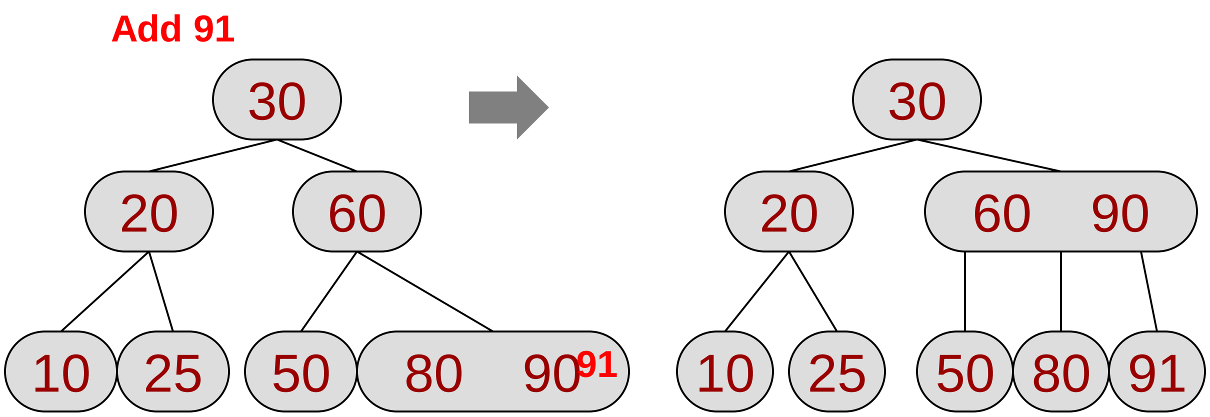
Add 90



2-3 Insertion Algorithm

- Key: Since all leaves must be at the same level, insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2 nodes with the smallest value as the left, biggest as the right, and median value should be promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 90,91,92,93

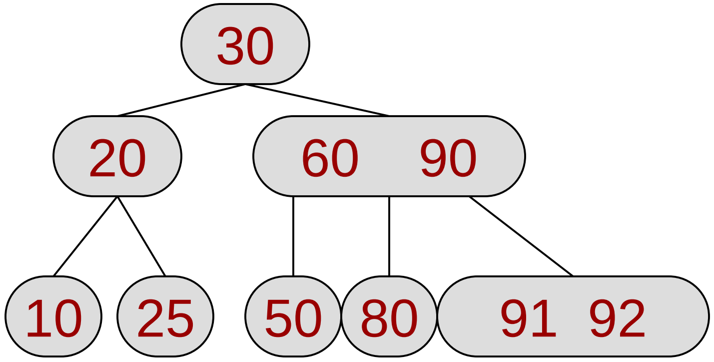




2-3 Insertion Algorithm

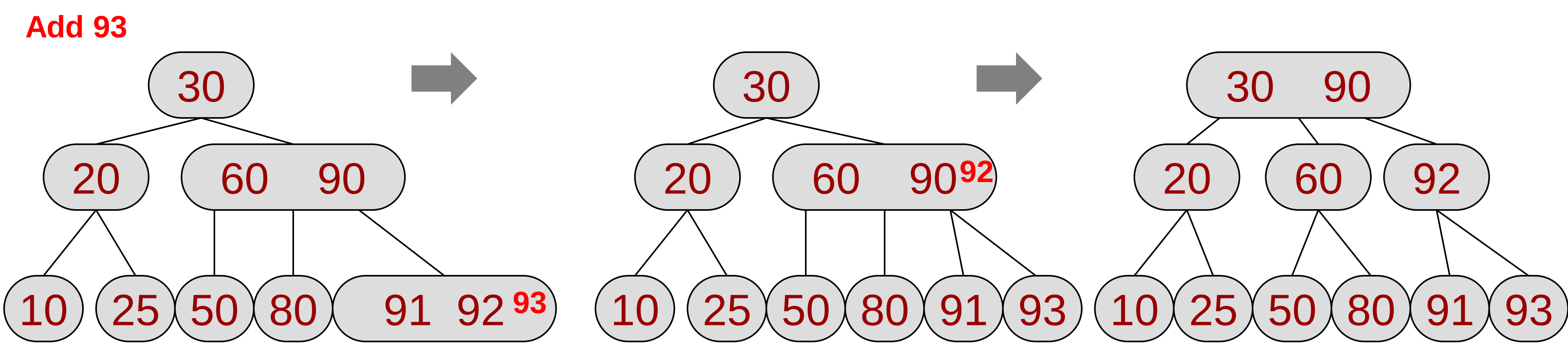
- Key: Since all leaves must be at the same level, insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2 nodes with the smallest value as the left, biggest as the right, and median value should be promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 90,91,92,93

Add 92



2-3 Insertion Algorithm

- Key: Since all leaves must be at the same level, insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2 nodes with the smallest value as the left, biggest as the right, and median value should be promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 90,91,92,93



2-3 Tree Removal

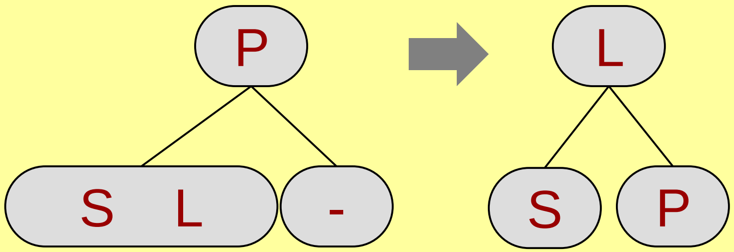
- Key: 2-3 Trees must remain "full" (leaf nodes all at the same level)
- Remove
  - 1. Find data item to remove
  - 2. If data item is not in a leaf node, find in-order successor (which is in a leaf node) and swap values (it's safe to put successor in your location)
  - 3. Remove item from the leaf node
  - 4. If leaf node is now empty, call fixTree(leafNode)

Another key: Want to get item to remove down to a leaf and then work up the tree

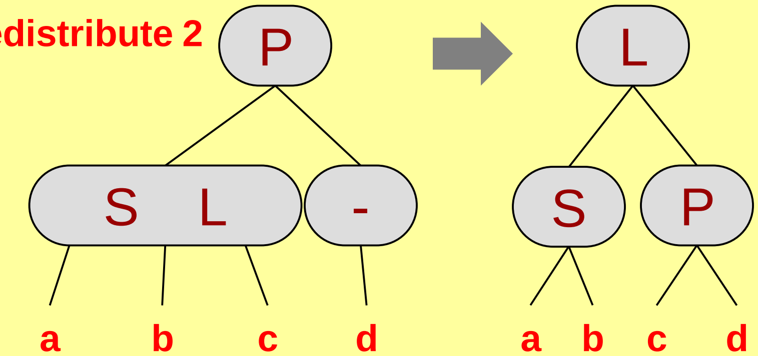
- fixTree(n)
  - If n is root, delete root and return
  - Let p be the parent of n
  - If a sibling of n has two items
    - Redistribute items between n, sibling, and p and move any appropriate child from sibling to n
  - Else
    - Choose a sibling, s, of n and bring an item from p into s redistributing any children of n to s
    - Remove node n
    - If parent is empty, fixTree(p)

Remove Cases

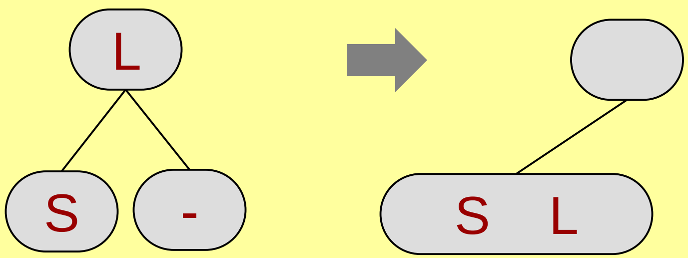
Redistribute 1



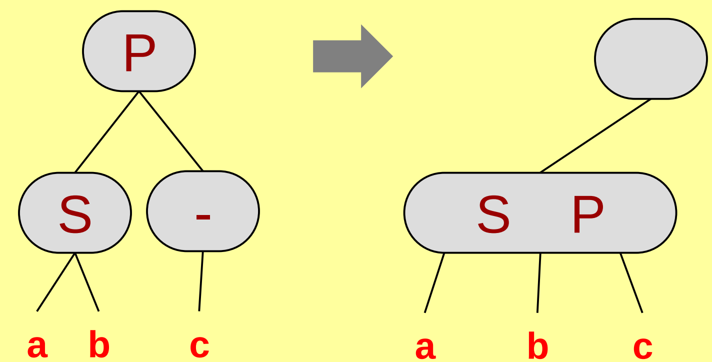
Redistribute 2



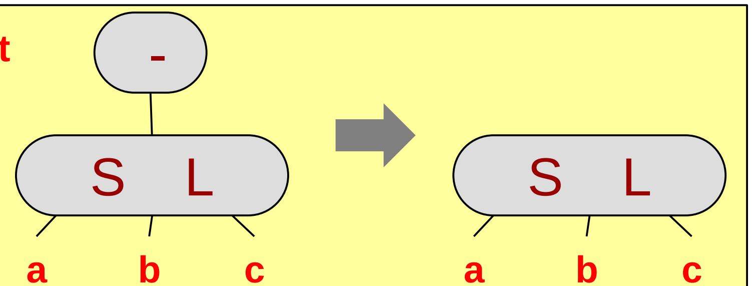
Merge 1



Merge 2



Empty root

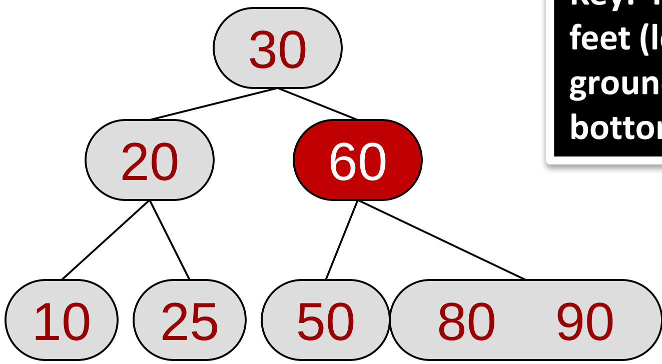


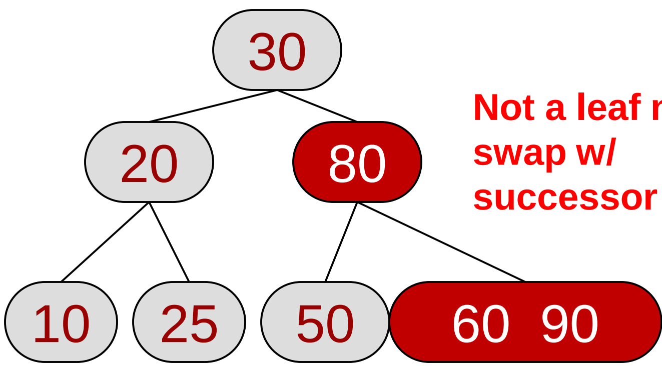
P = parent
S = smaller
L = larger

Remove Examples

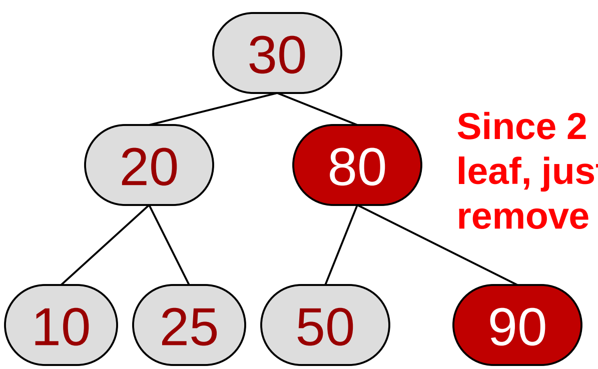
Key: Keep all your feet (leaves) on the ground (on the bottom row)

Remove 60



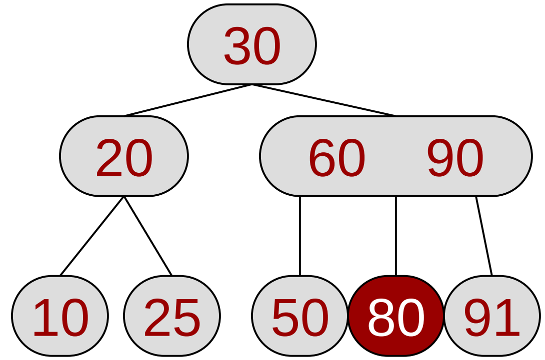


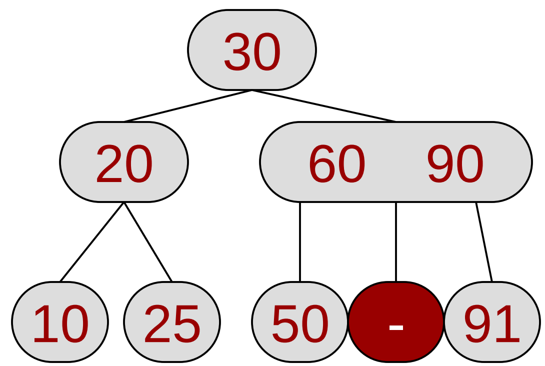
Not a leaf node so swap w/ successor at leaf



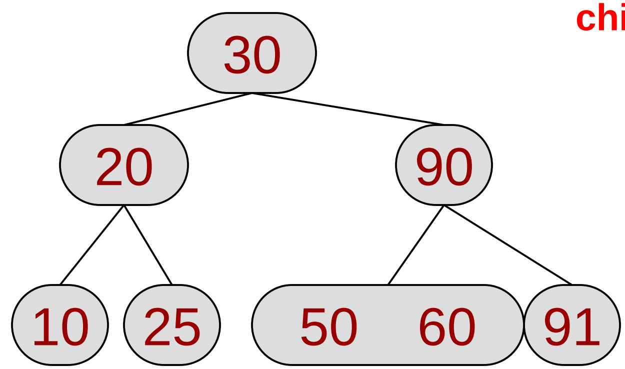
Since 2 items at leaf, just remove 60

Remove 80





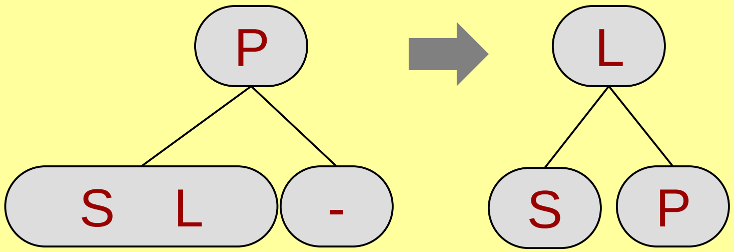
Can't just delete because a 3-node would have only 2 children



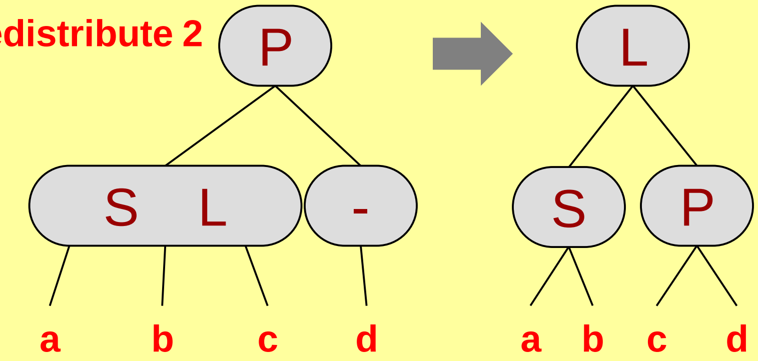
Rotate 60 down into 50 to make a 3-node at the leaf and 2-node parent

Remove Cases

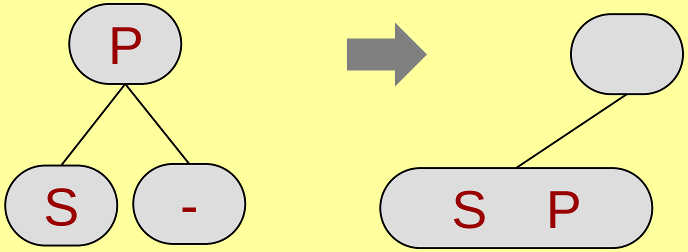
Redistribute 1



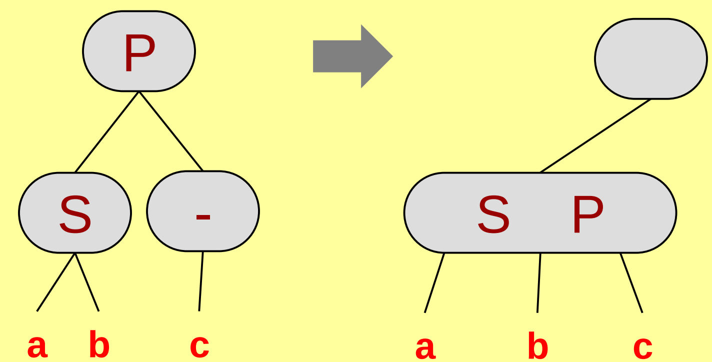
Redistribute 2



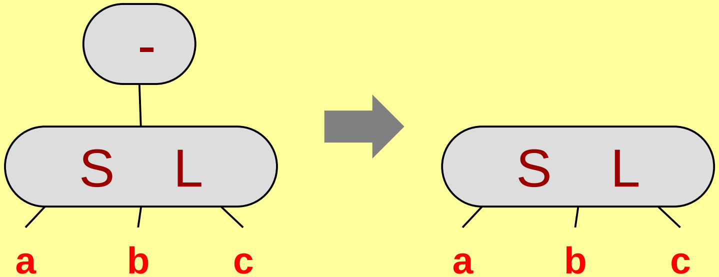
Merge 1



Merge 2

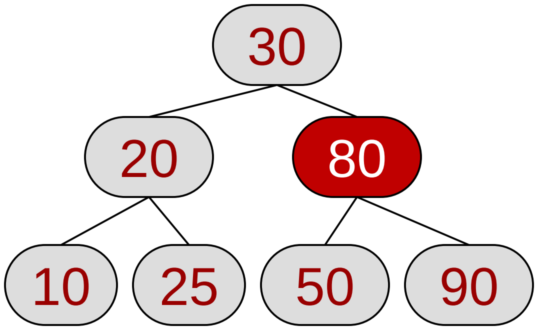


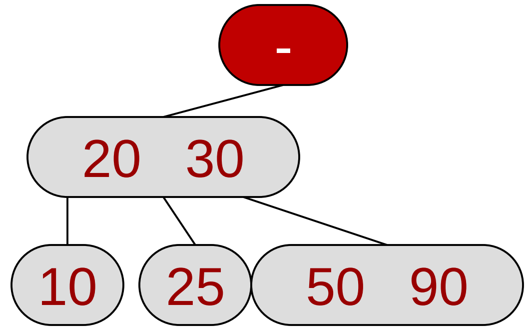
Empty root



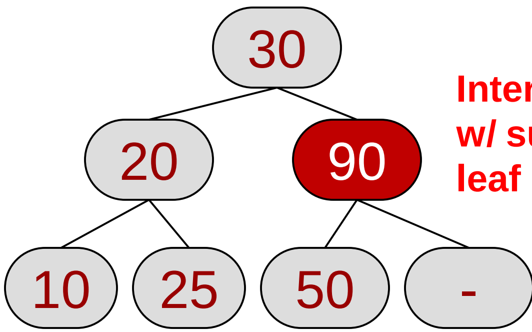
Remove Examples

Remove 80

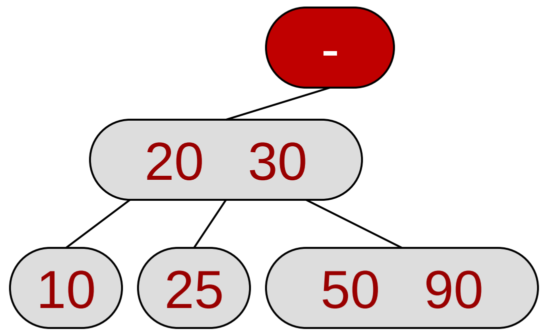




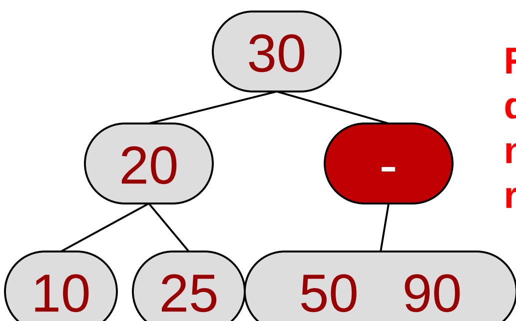
Rotate parent down and empty node up, then recurse



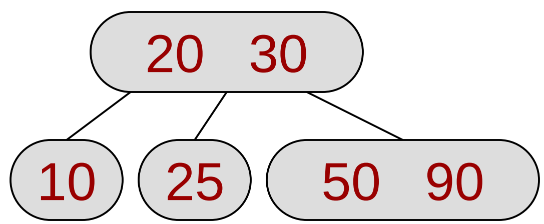
Internal so swap w/ successor at leaf



Remove root and thus height of tree decreases

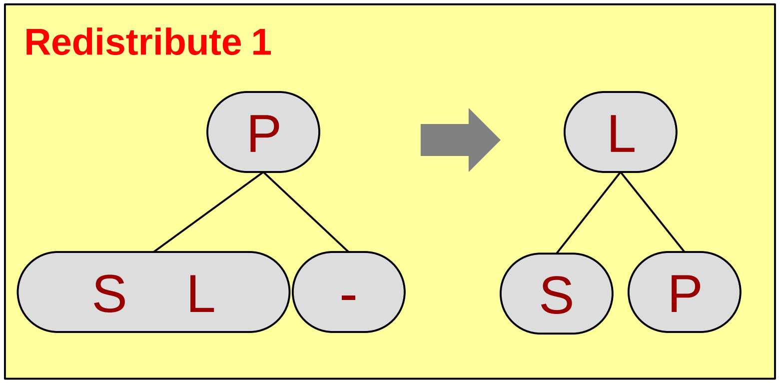


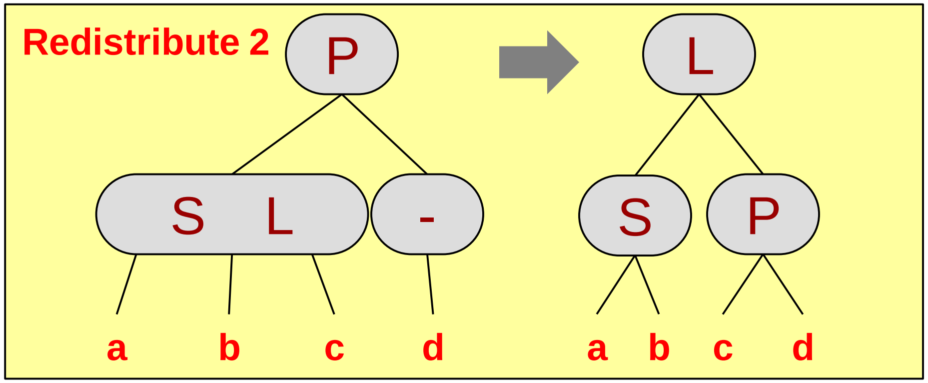
Rotate parent down and empty node up, then recurse

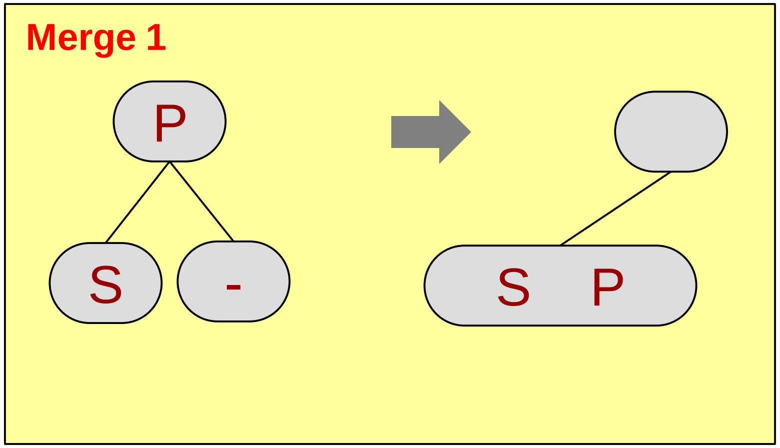


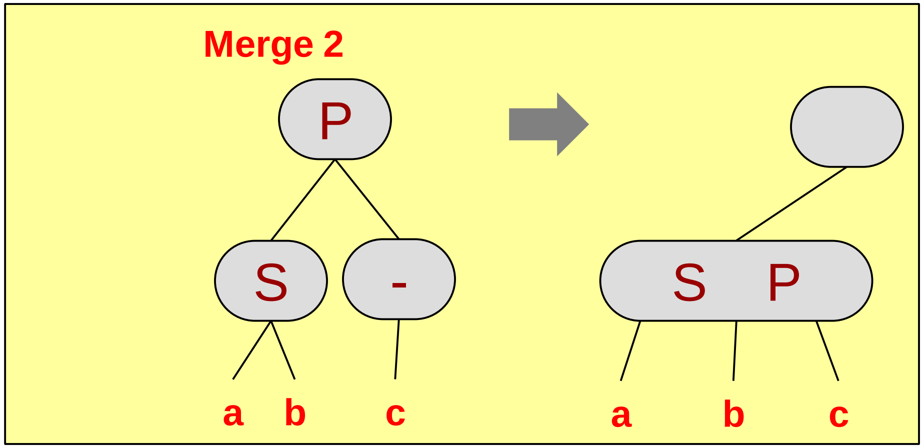
USC Viterbi School of Engineering

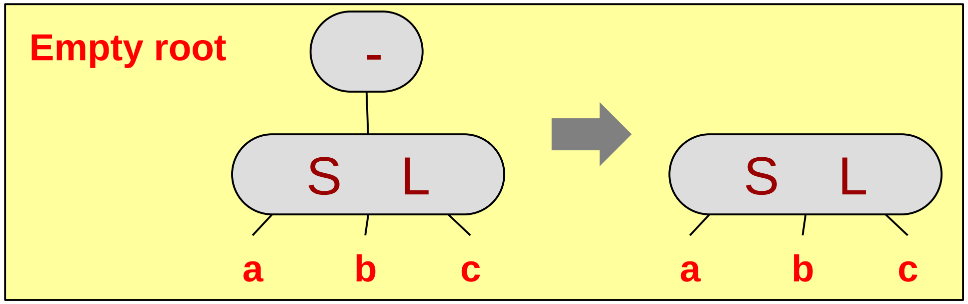
Remove Cases







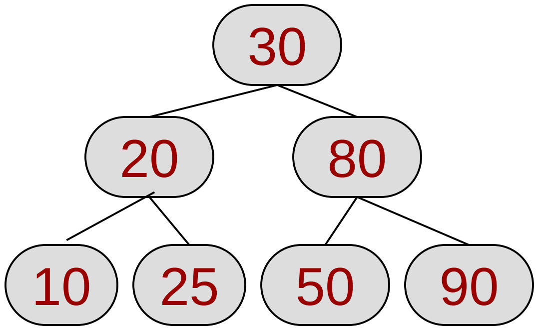


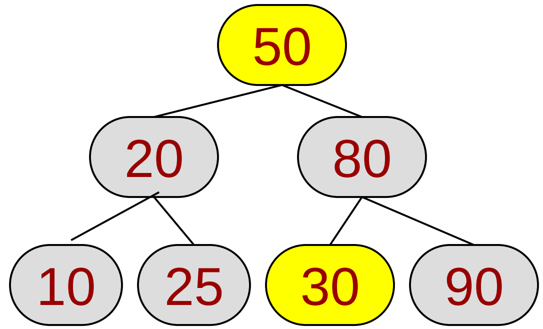


USC Viterbi School of Engineering

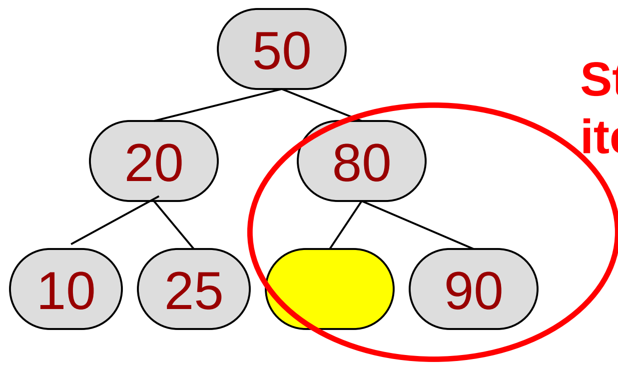
Remove Exercise 1

Remove 30

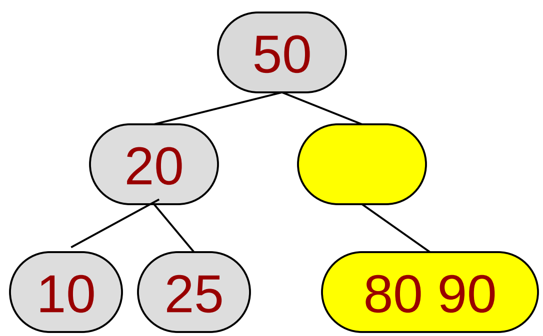




Step 1: Not a leaf, so swap with successor

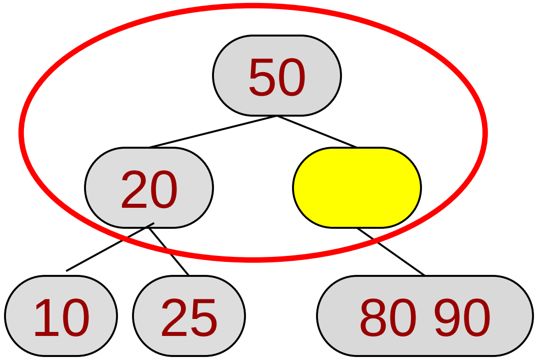


Step 2: Remove item from node

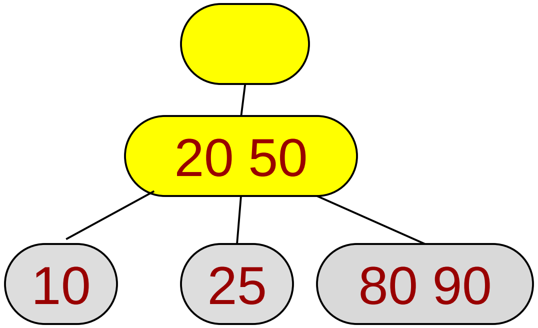


Step 3: Two values and 3 nodes, so merge. Must maintain levels.

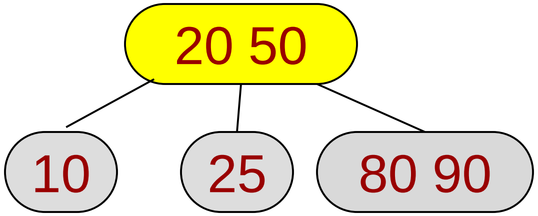
Remove Exercise 1 (cont.)



Start over with the empty parent. Do another merge



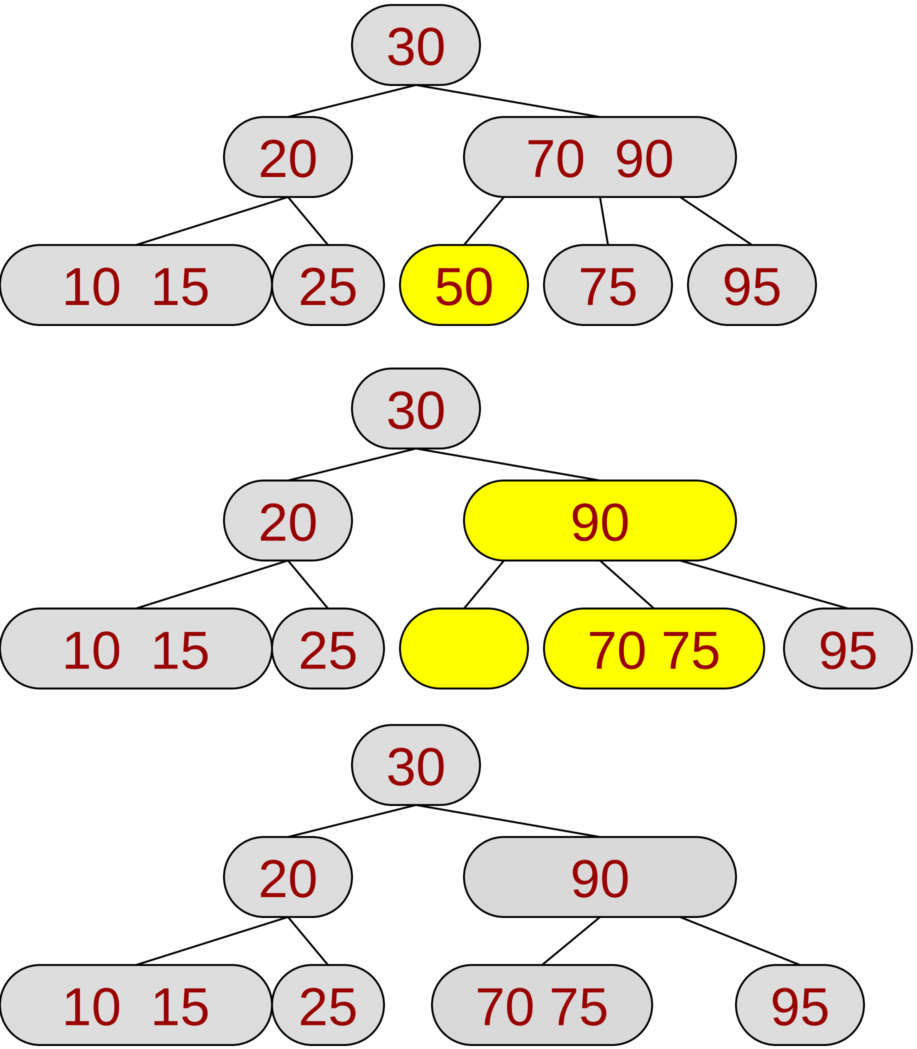
Step 4: Merge values



Step 5: Can delete the empty root node.

Remove Exercise 2

Remove 50



Step 1: It's a leaf node, so no need to find successor. Remove the item from node.

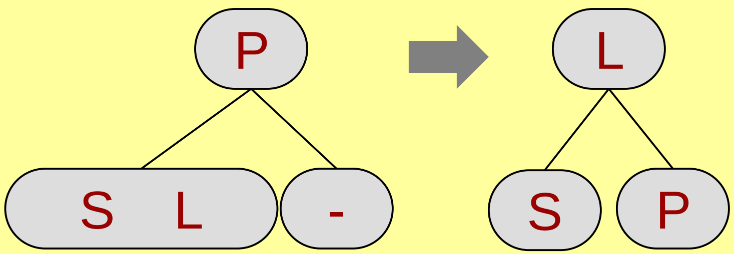
Step 2: Since no 3-node children, push a value of parent into a child.

Step 3: Delete the node.

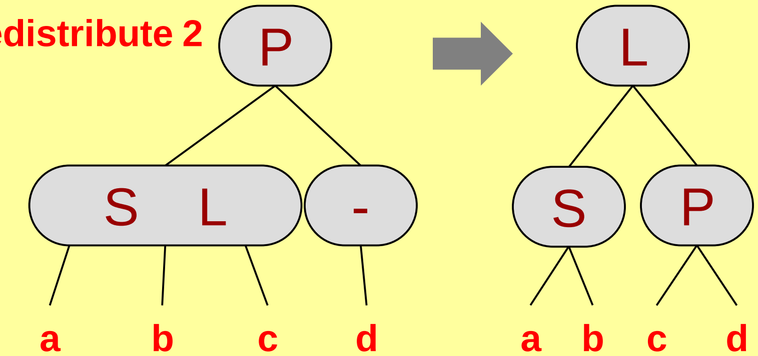
USC Viterbi School of Engineering

Remove Cases

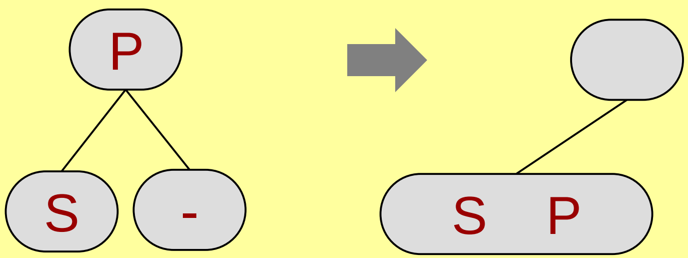
Redistribute 1



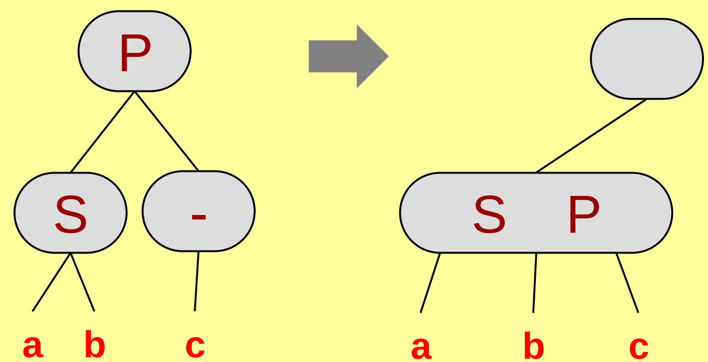
Redistribute 2



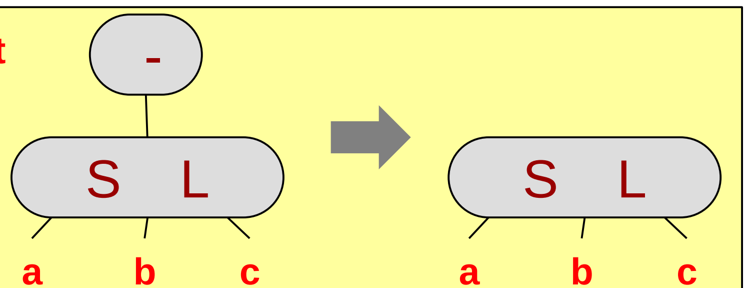
Merge 1



Merge 2

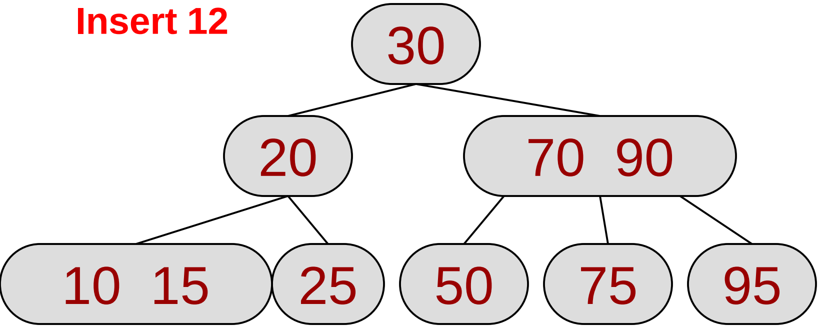


Empty root



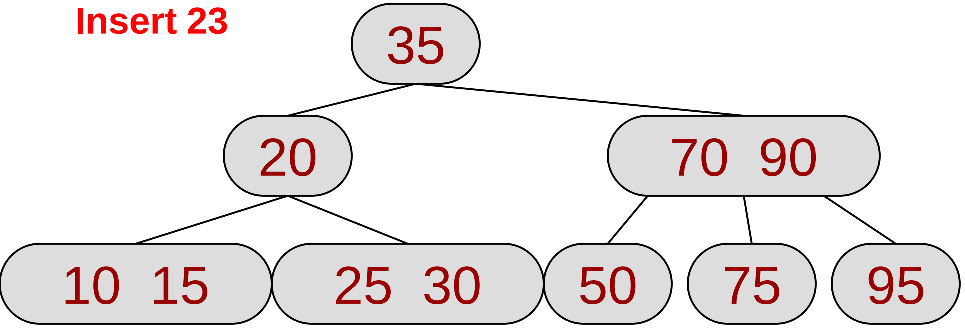
Insertion Exercise 1

Insert 12



Insertion Exercise 2

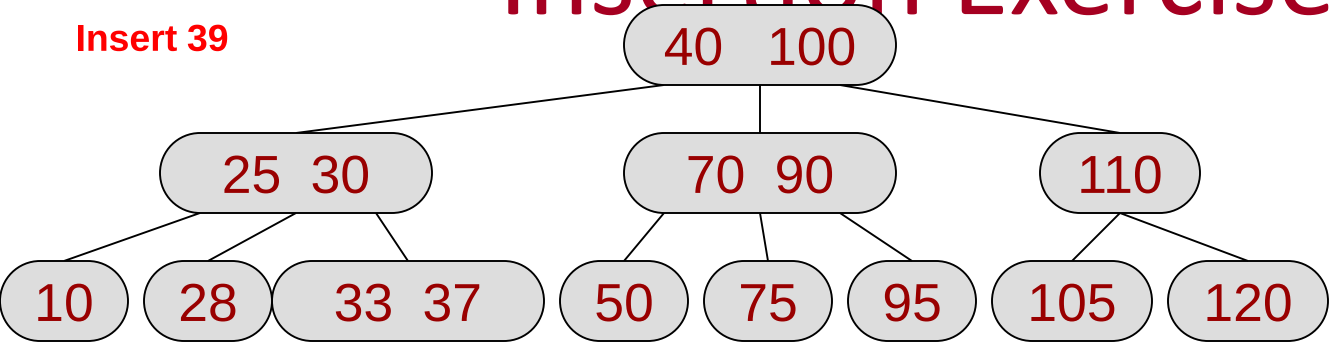
Insert 23



USC Viterbi School of Engineering

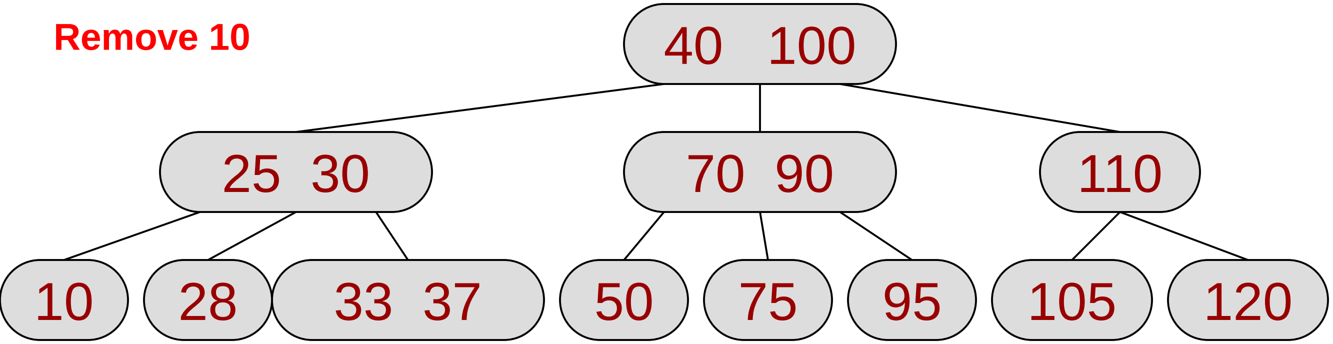
Insertion Exercise 3

Insert 39



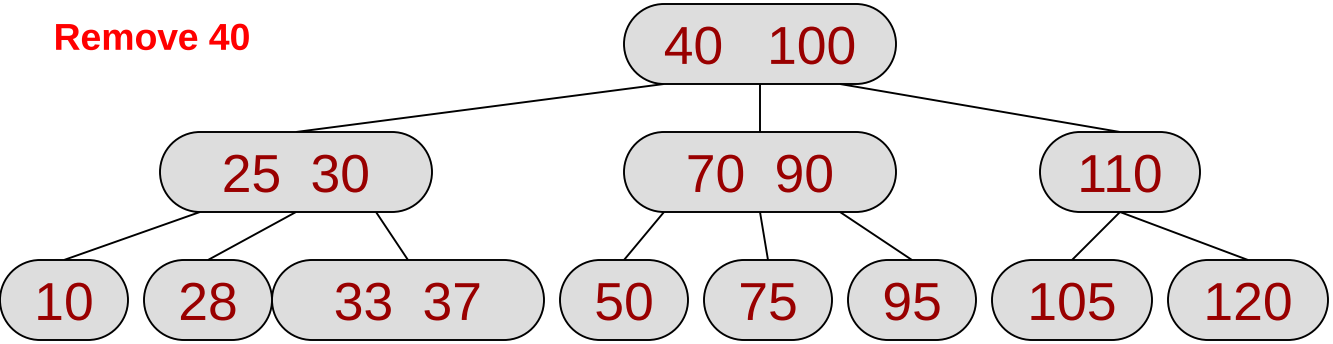
Removal Exercise 4

Remove 10



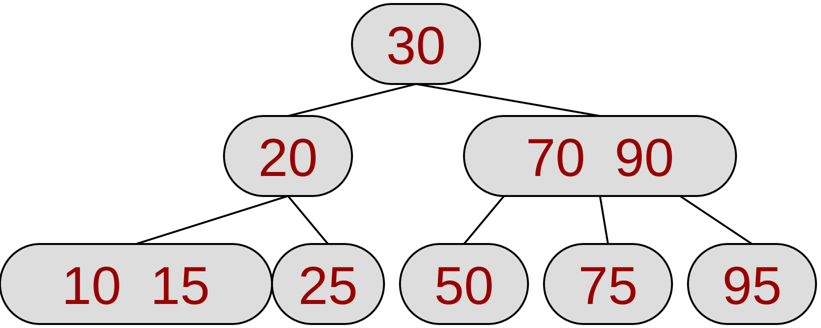
Removal Exercise 5

Remove 40



Removal Exercise 6

Remove 30

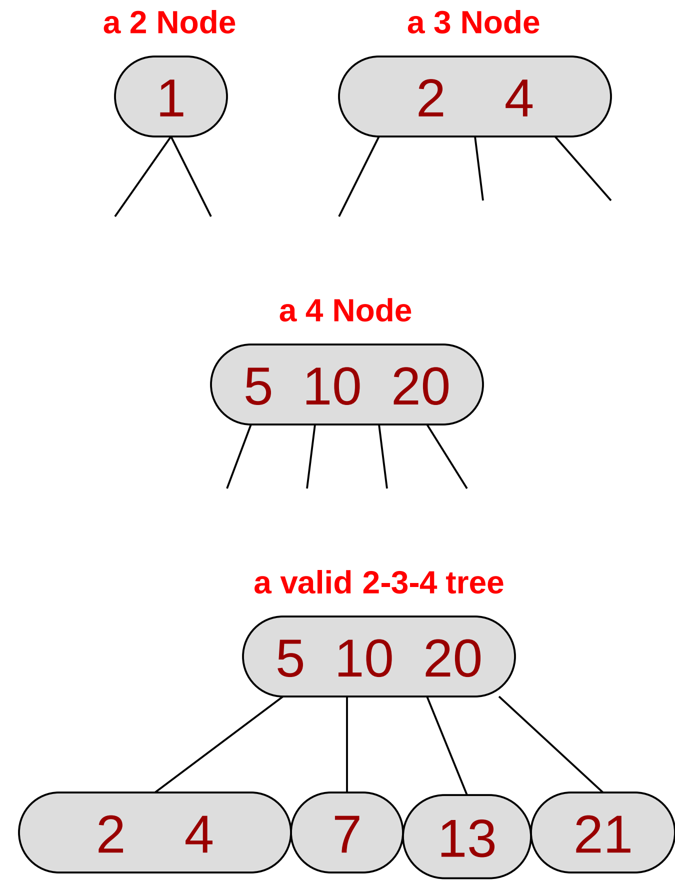


Other Resources

- http://www.cs.usfca.edu/~galles/visualization/BTree.html

Definition

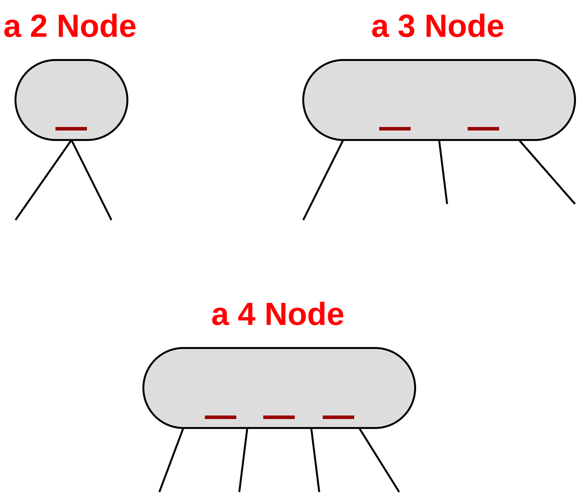
- 2-3-4 trees are very much like 2-3 trees but form the basis of a balanced, **binary** tree representation called Red-Black (RB) trees which are commonly used [used in C++ STL map & set]
  - We study them mainly to ease understanding of RB trees
- 2-3-4 Tree is a tree where
  - Non-leaf nodes have 1 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

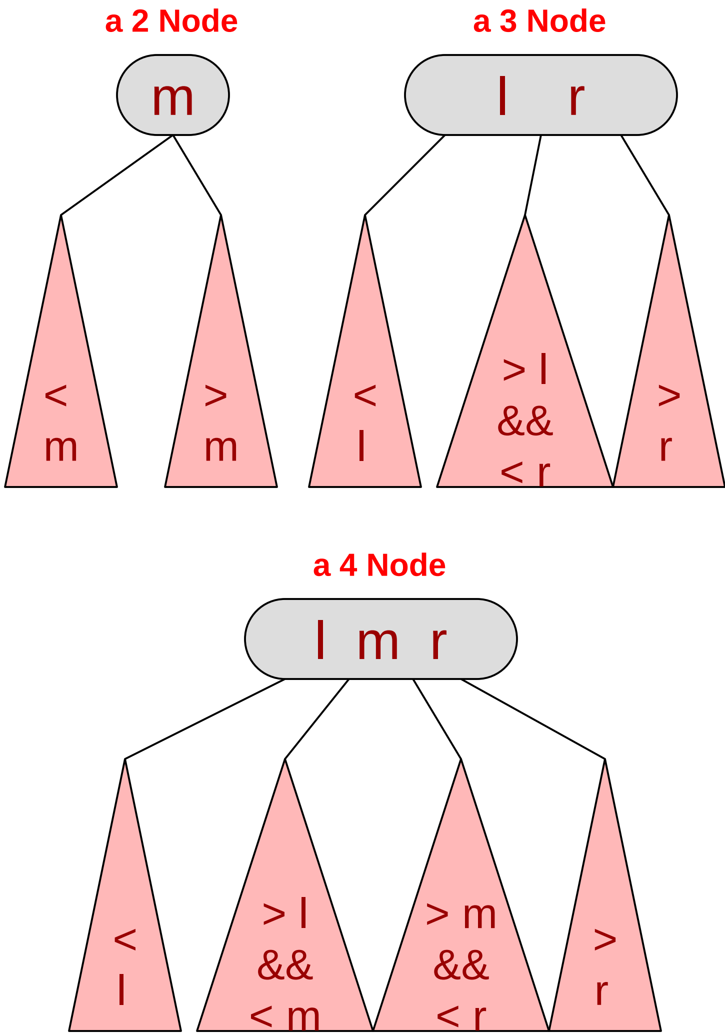
- 4-nodes require more memory and can be inefficient when the tree actually has many 2 nodes

```
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

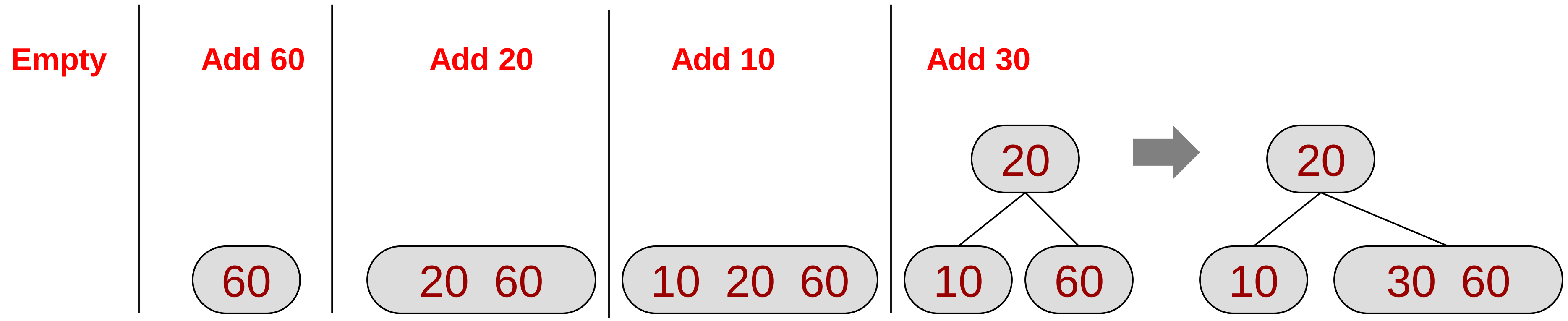
- Similar properties as a 2-3 Search Tree
- 4 Node:
  - Left subtree nodes are < *l*
  - Middle-left subtree > *l* and < *r*
  - Right subtree nodes are > *r*



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 leaf will always have room for a value

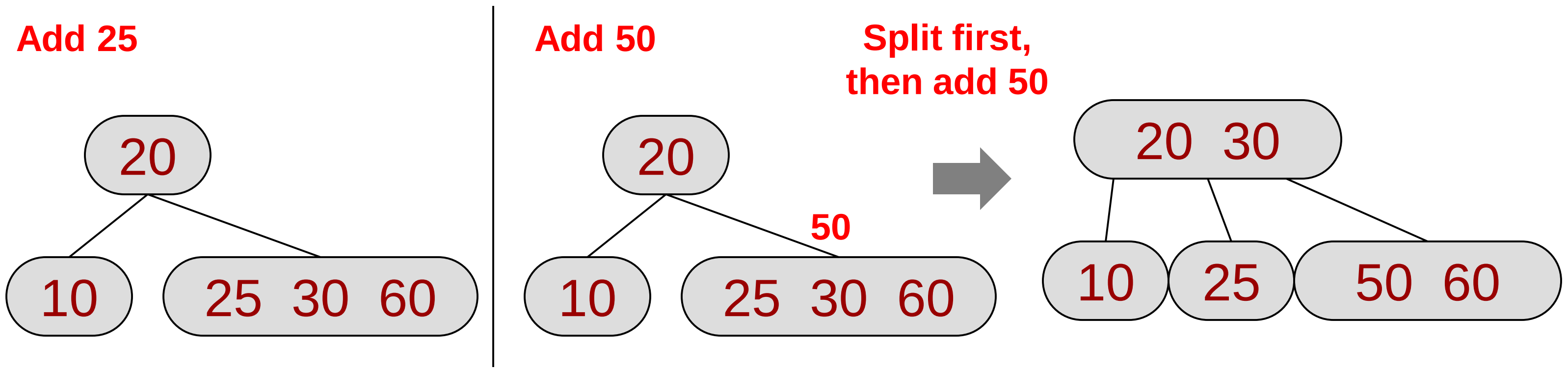


2-3-4 Insertion Algorithm

- 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 leaf will always have room for a value

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

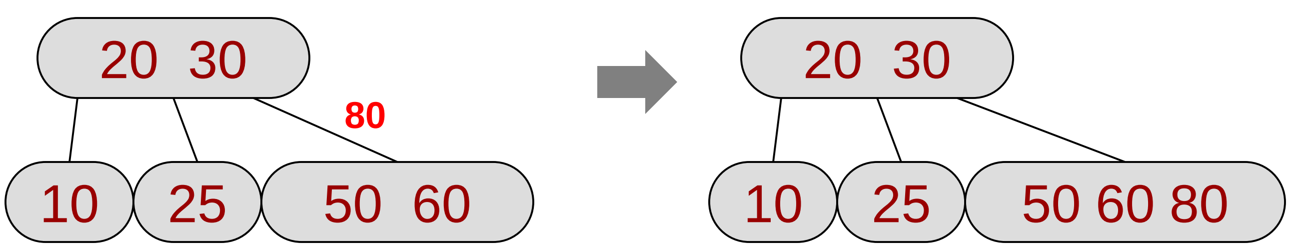


2-3-4 Insertion Algorithm

- 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

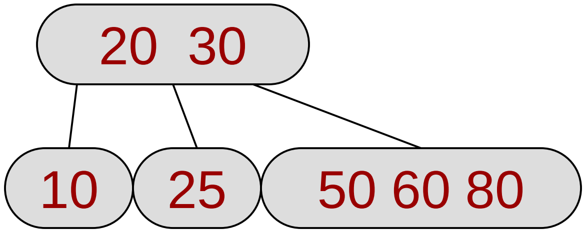
Key: 4-nodes get split as you walk down thus, a leaf will always have room for a value

Add 80



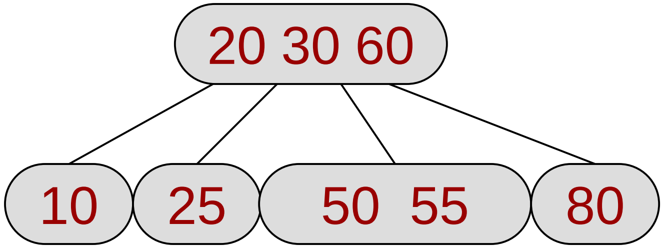
2-3-4 Insertion Exercise 1

Add 55



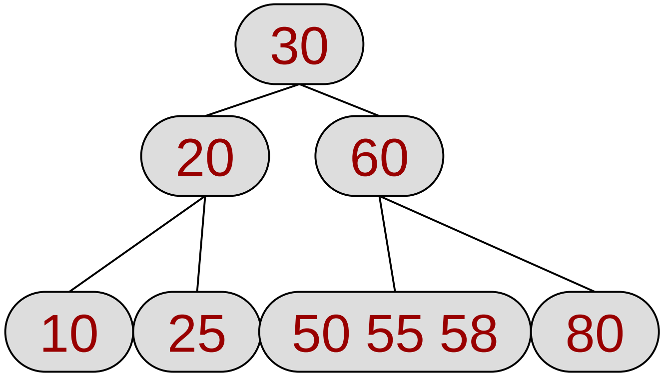
2-3-4 Insertion Exercise 2

Add 58



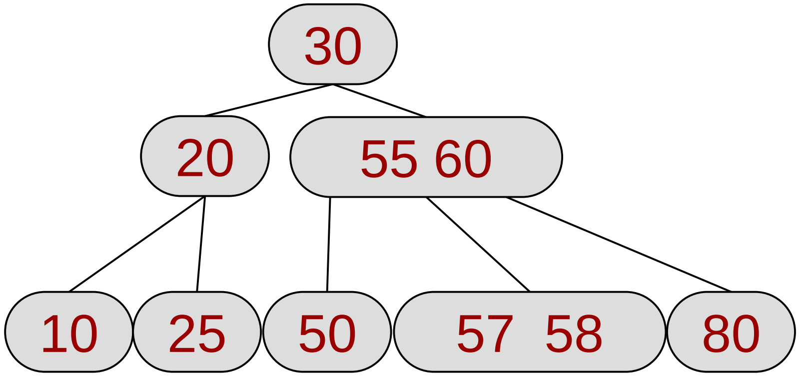
2-3-4 Insertion Exercise 3

Add 57



2-3-4 Insertion Exercise 3

Resulting Tree



B-Trees

- 2-3 and 2-3-4 trees are just instances of a more general data structure known as B-Trees
- Define minimum number of children (degree) for non-leaf nodes, d
  - Non-root nodes must have at least d-1 keys and d children
  - All nodes must have at most 2d-1 keys and 2d children
  - 2-3-4 Tree (d=2)
- Used for disk-based storage and indexing with large value of d to account for large random-access lookup time but fast sequential access time of secondary storage

B Tree Resources

- https://www.cs.usfca.edu/~galles/visualization/BTree.html
- http://ultrastudio.org/en/2-3-4_tree

"Balanced" Binary Search Trees

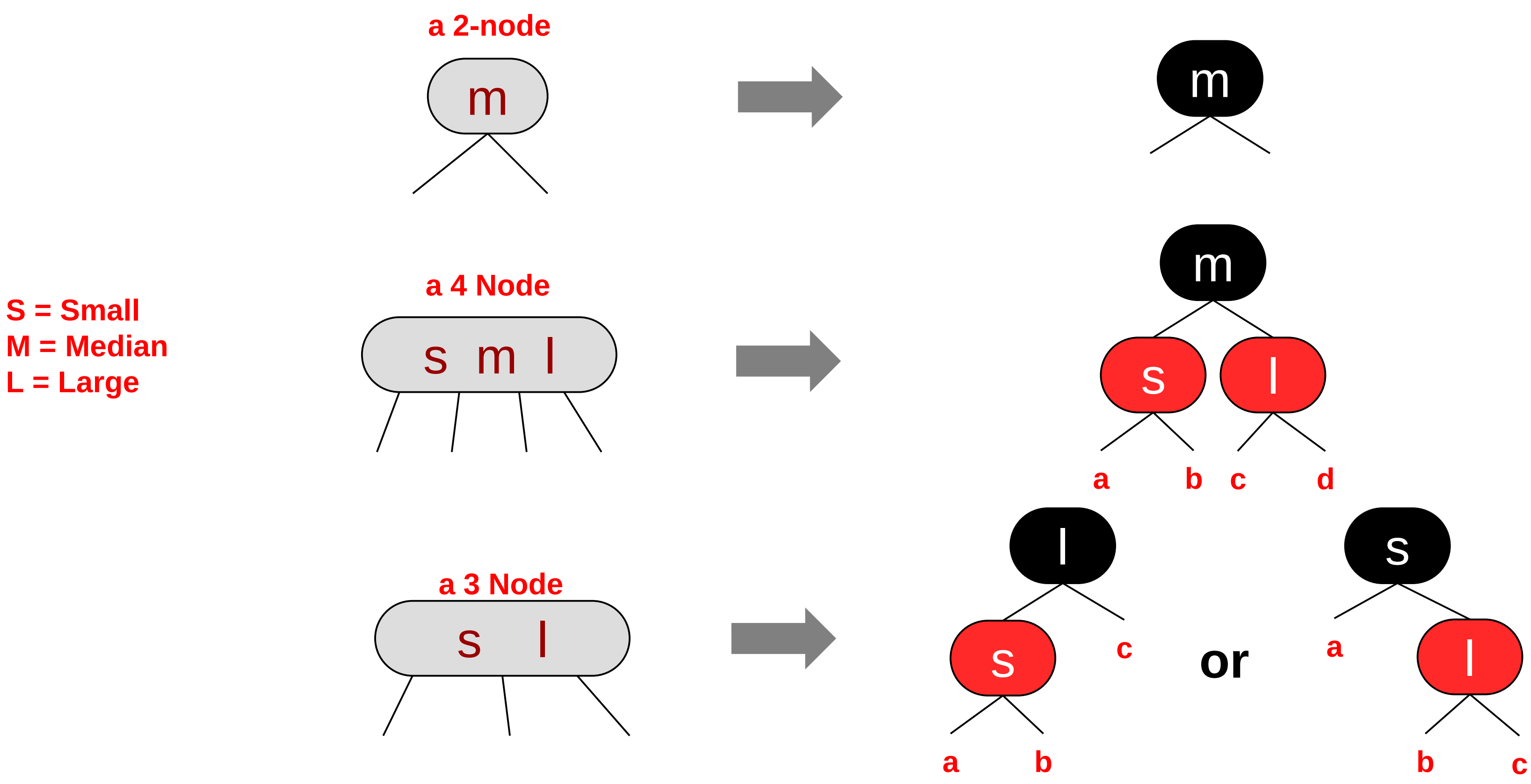
RED BLACK TREES

Red Black Trees

- A red-black tree is a binary search tree
  - Only 2 nodes (no 3- or 4-nodes)
  - Can be built 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 $O(\log_2(n))$

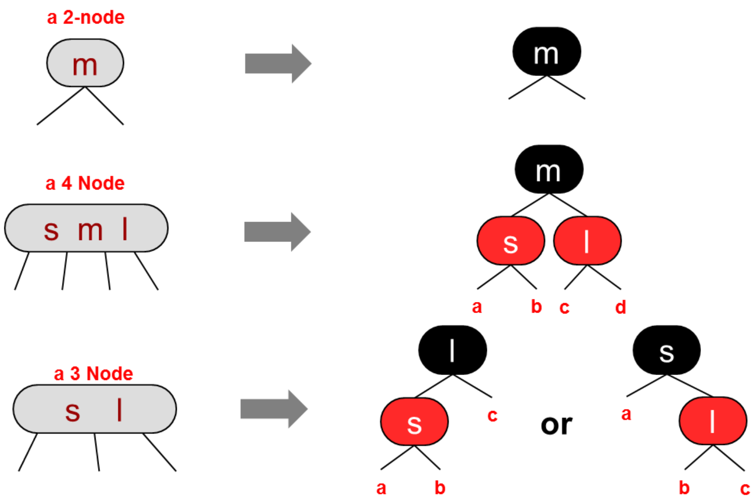
Red Black and 2-3-4 Tree Correspondence

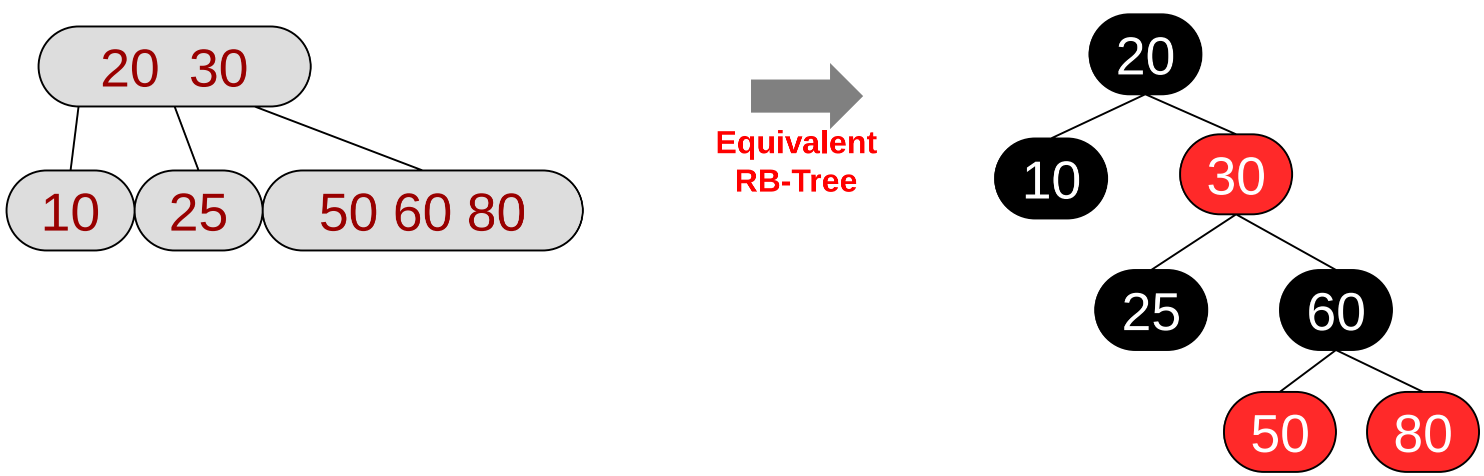
- Every 2-, 3-, and 4-node can be converted to…
  - At least 1 black node and 1 or 2 red children of the black node
  - Red nodes are always ones that would join with their parent to become a 3- or 4-node in a 2-3-4 tree



Red Black Trees

- Below is a 2-3-4 tree and how it can be represented as a directly corresponding RB-Tree
- Notice at most each 2-3-4 node expands to **2** level of red/black nodes
- **Q:** Thus if the height of the 2-3-4 tree was bound by $\log_2 n$, then the height of an RB-tree is bounded by?
- **A:** $2*\log_2 n = O(\log_2 n)$



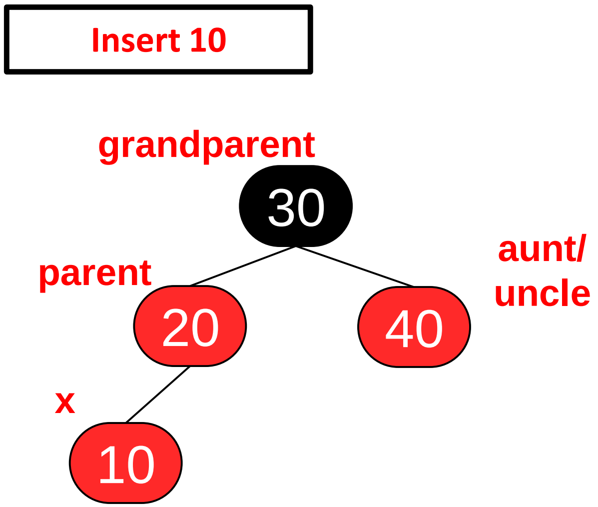


Red-Black Tree Properties

- 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 nodes
  - 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. At the end of an operation the root should always be black
- 4. We can imagine leaf nodes as having 2 non-existent (NULL) black children if it helps

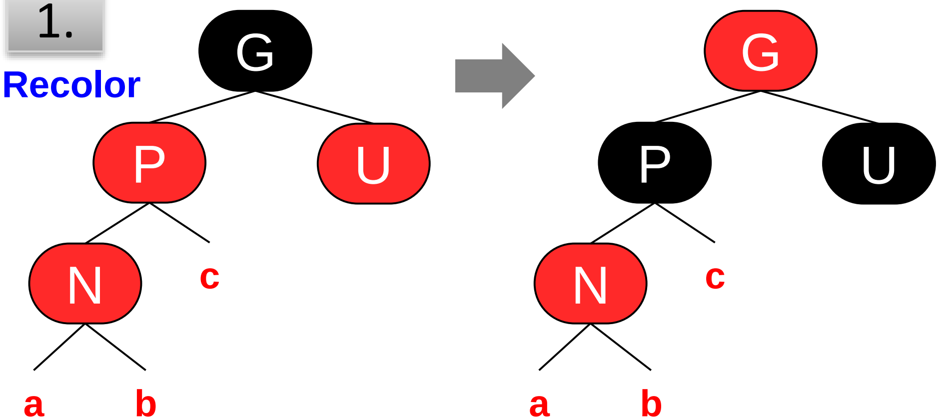
Red-Black Insertion

- 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"

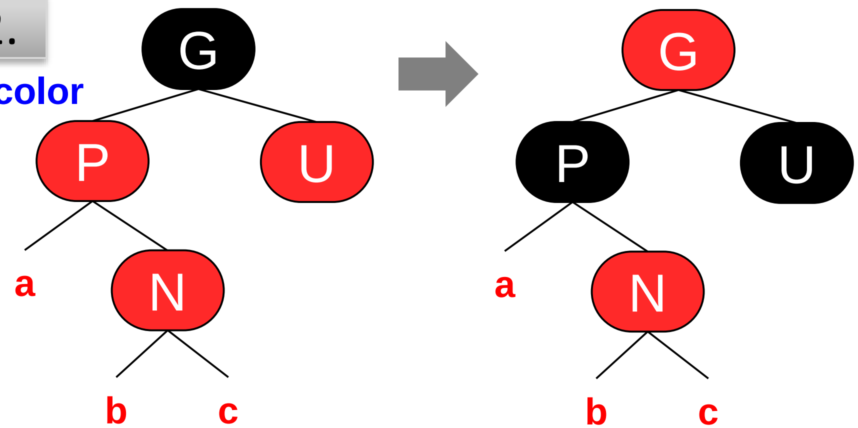


fixTree Cases

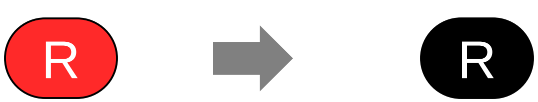
1.
Recolor

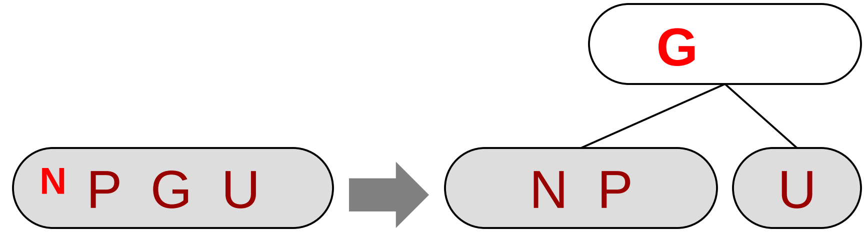


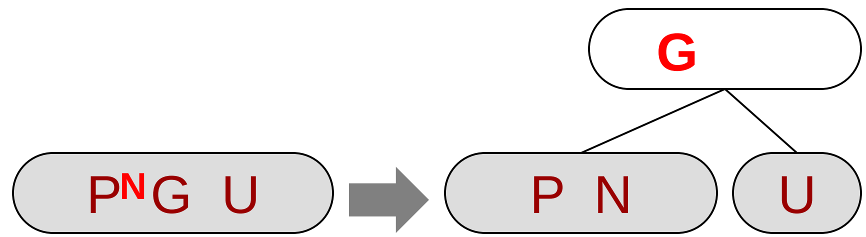
2.
Recolor



3.
Recolor Root





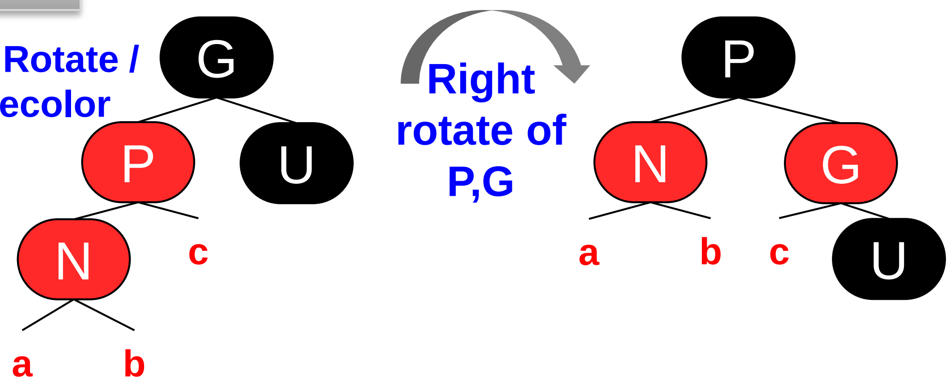


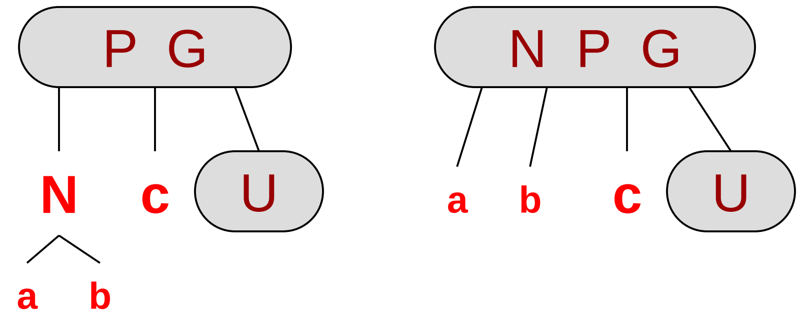
Note: For insertion/removal algorithm we consider non-existent leaf nodes as black nodes

fixTree Cases

4.

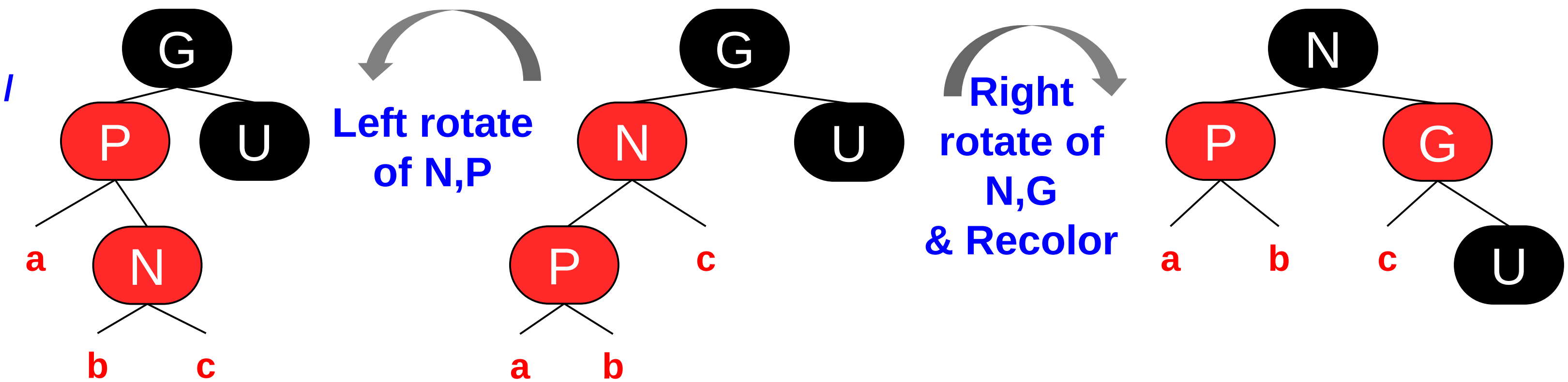
1 Rotate / Recolor

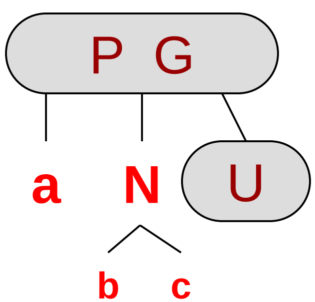


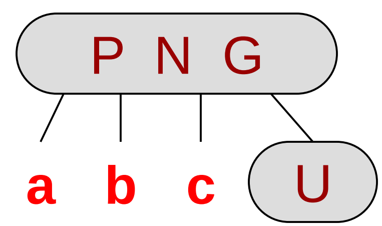


5.

2 Rotates / Recolor



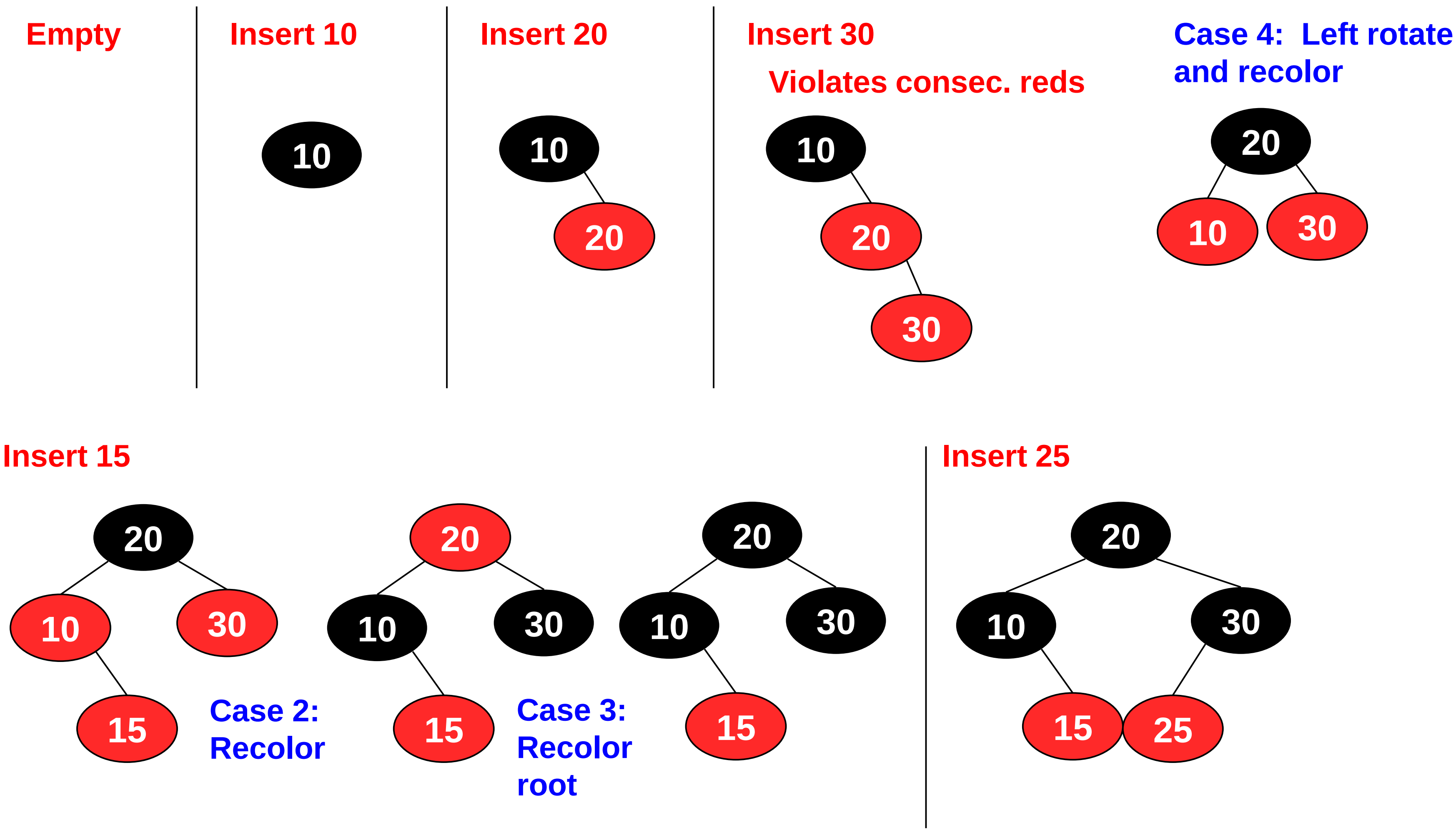




www.cse.ohio-state.edu/~gurari/course/cis680Ch11.html

Insertion

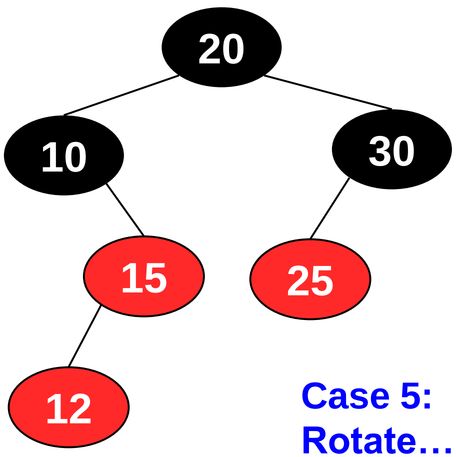
- Insert 10, 20, 30, 15, 25, 12, 5, 3, 8



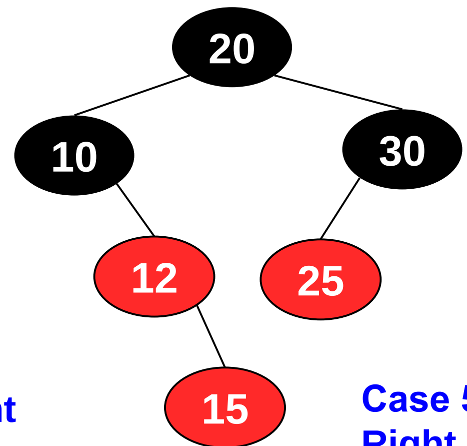
Insertion

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

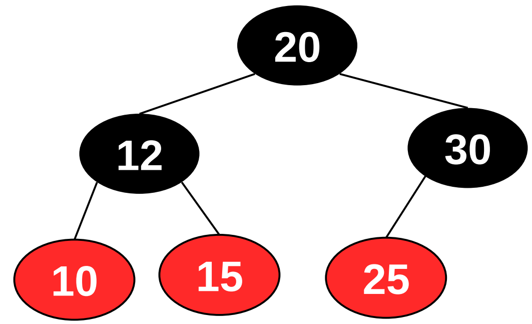
Insert 12



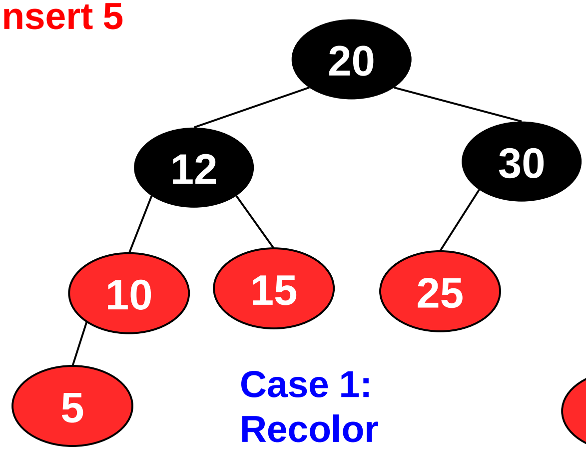
Case 5: Right Rotate…



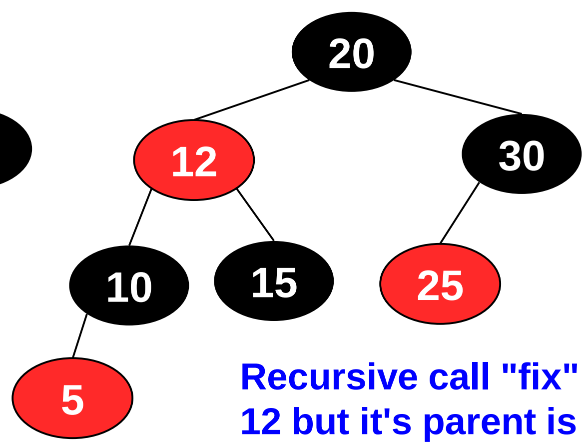
Case 5: … Right Rotate and recolor



Insert 5



Case 1: Recolor

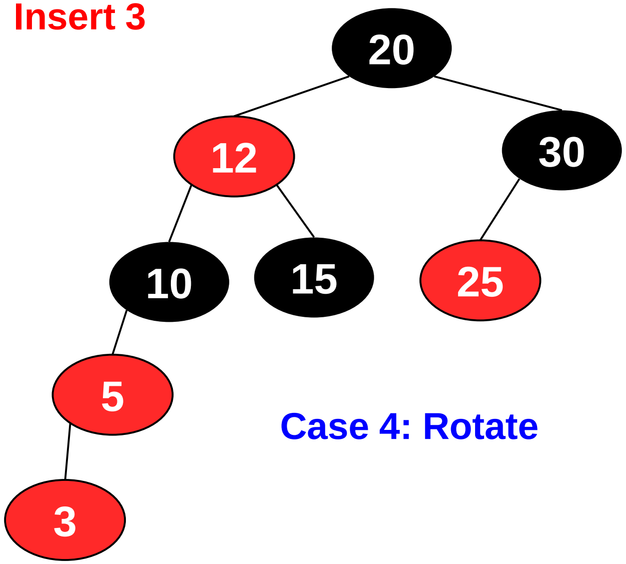


Recursive call "fix" on 12 but it's parent is black so we're done

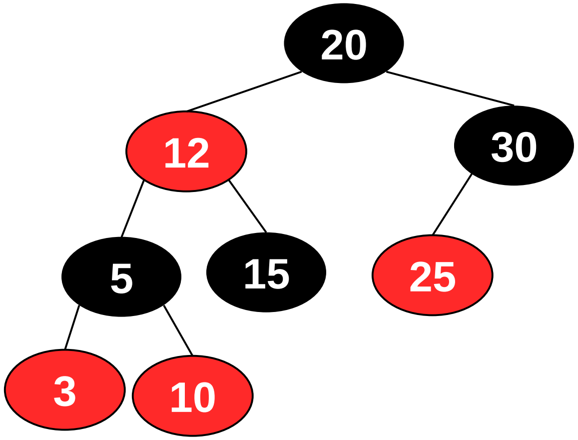
Insertion

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

Insert 3

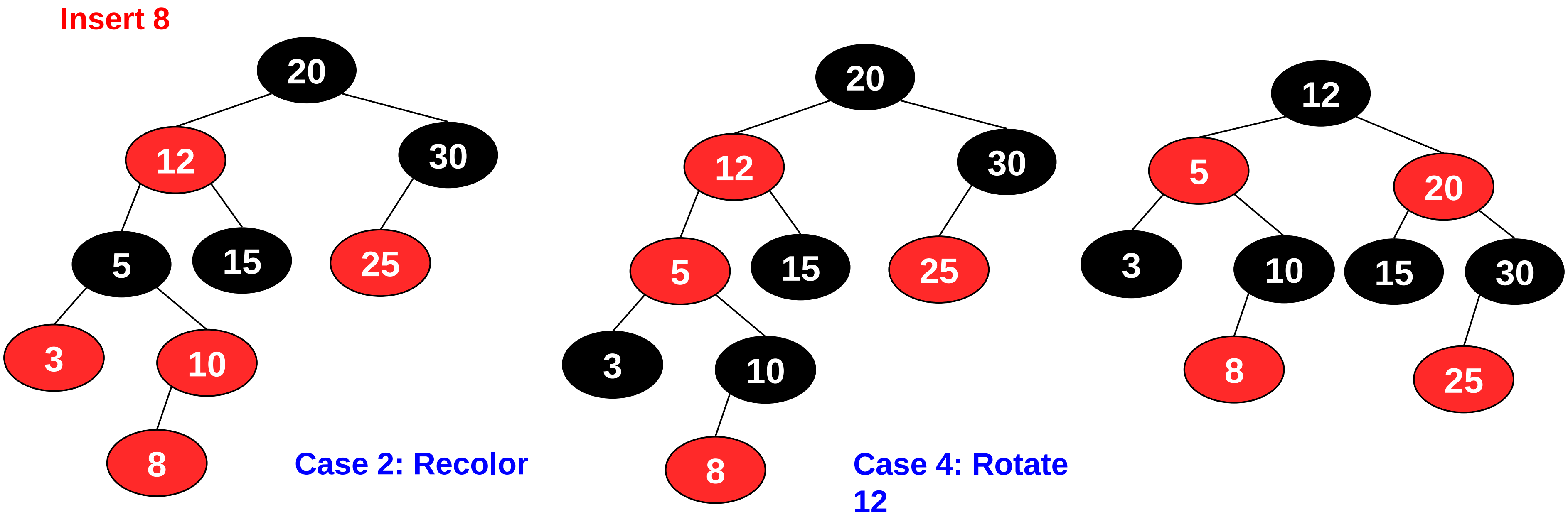


Case 4: Rotate



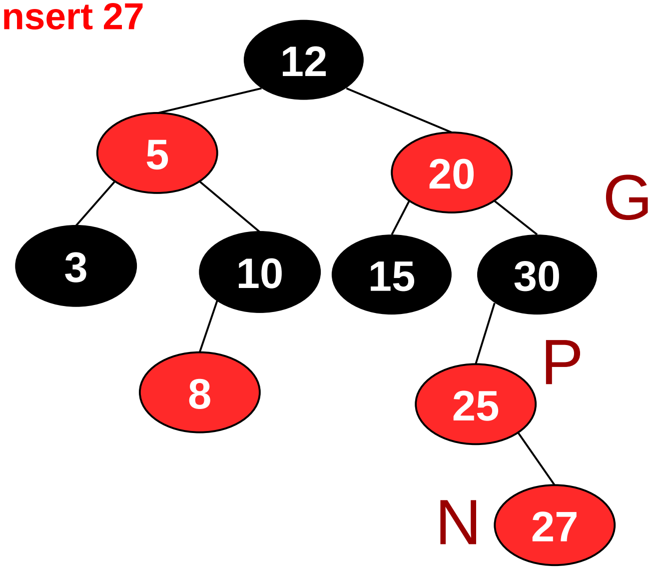
Insertion

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

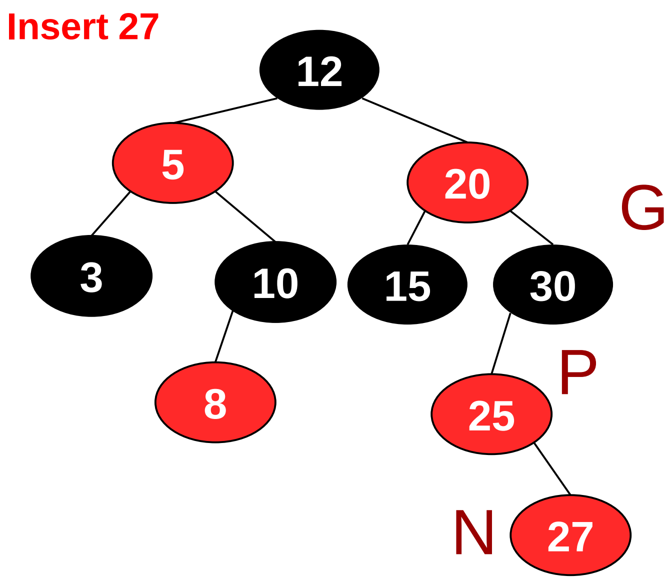


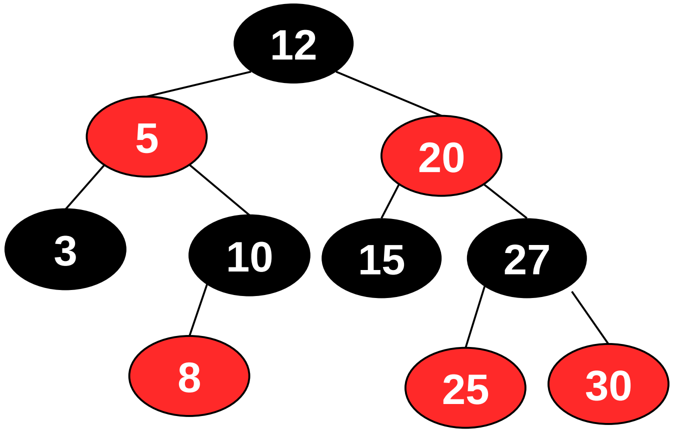
Insertion Exercise 1

Insert 27



Insertion Exercise 1



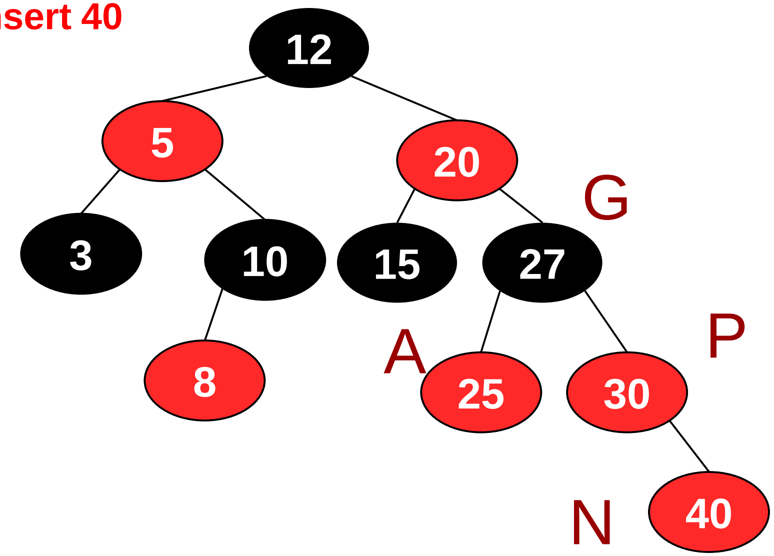


This is case 5.

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

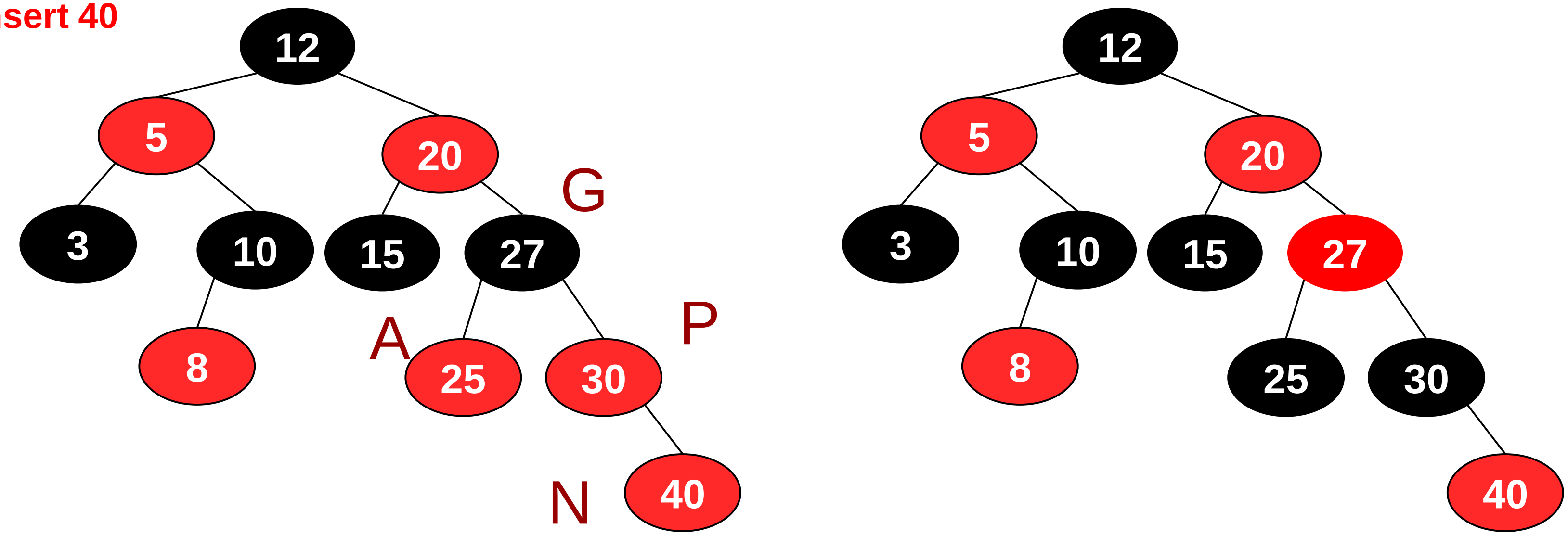
Insertion Exercise 2

Insert 40



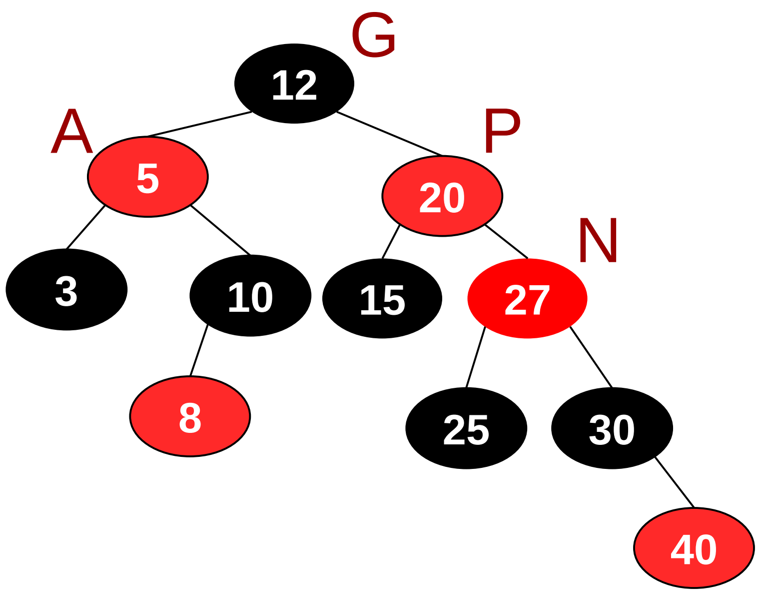
Insertion Exercise 2

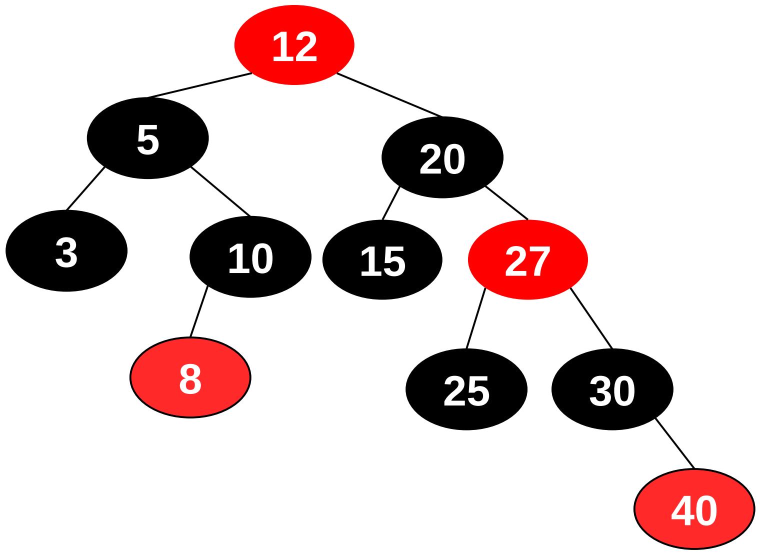
Insert 40



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

Insertion Exercise 2

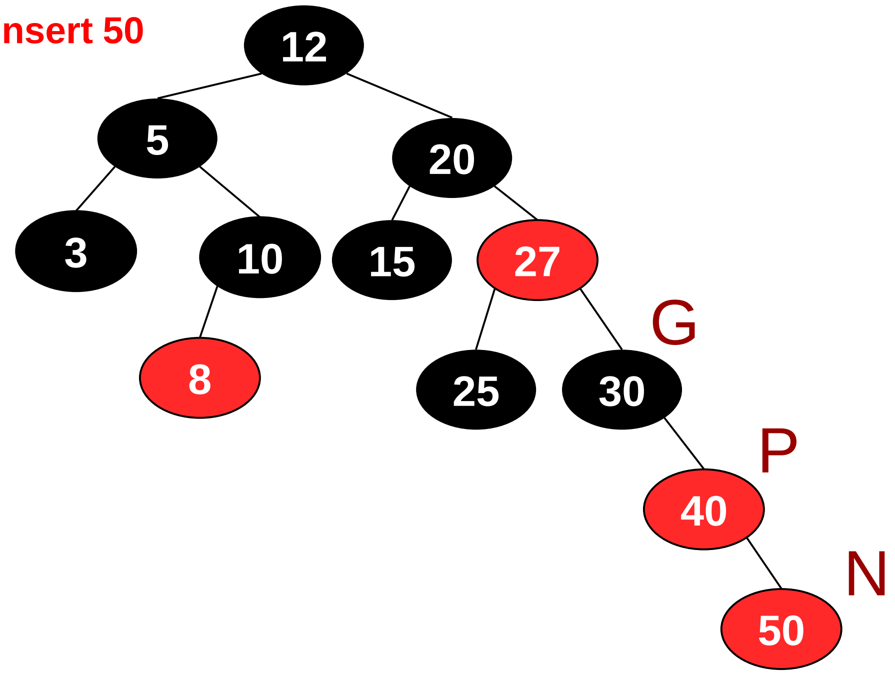




Aunt and Parent are the same color. So recolor aunt, parent, and grandparent.

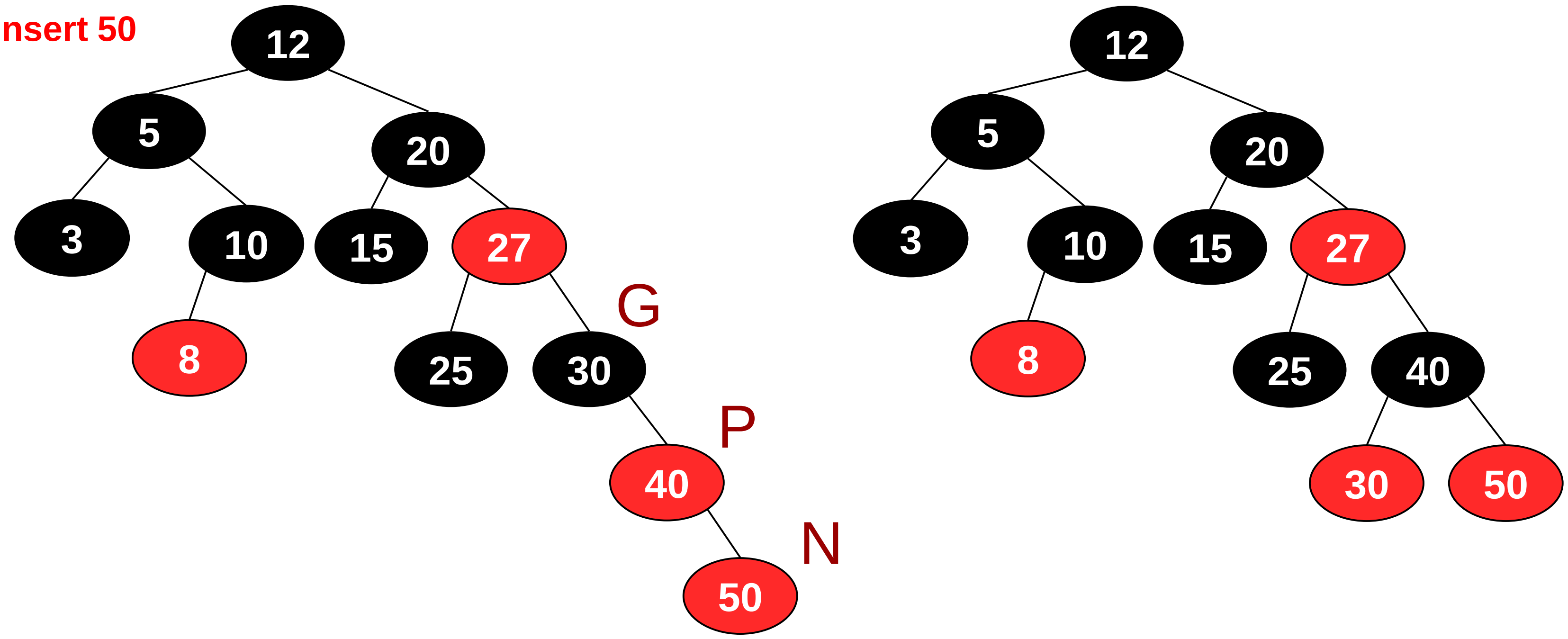
Insertion Exercise 3

Insert 50



Insertion Exercise 3

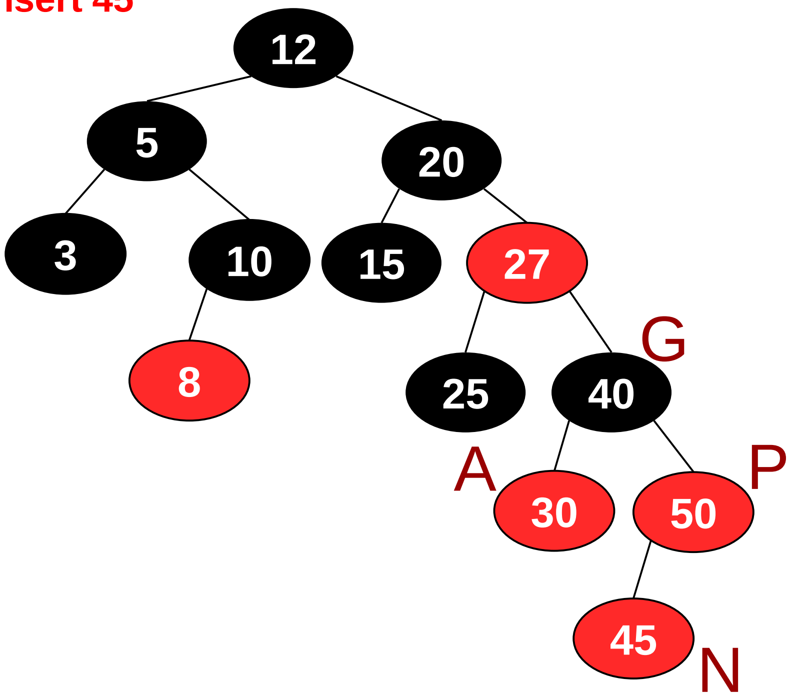
Insert 50



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

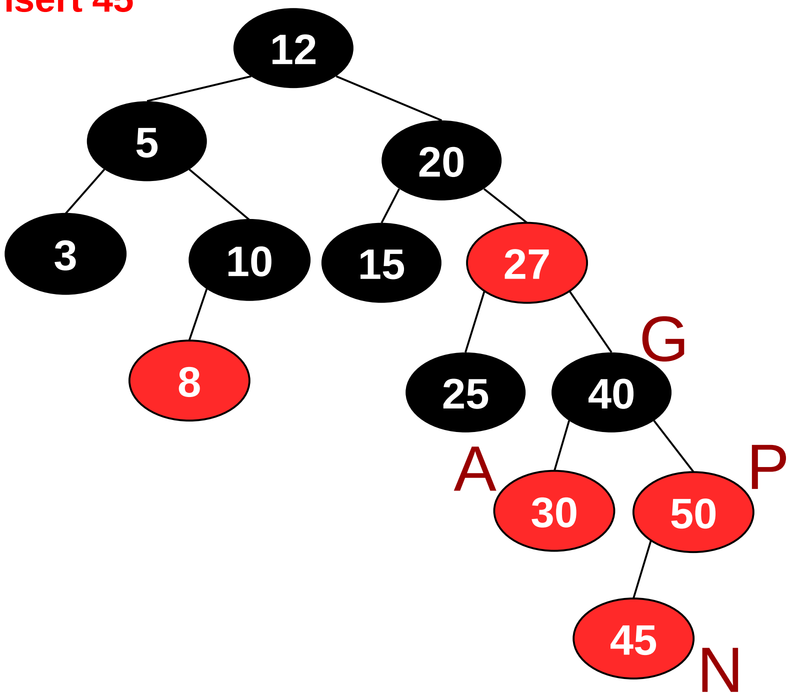
Insertion Exercise 4

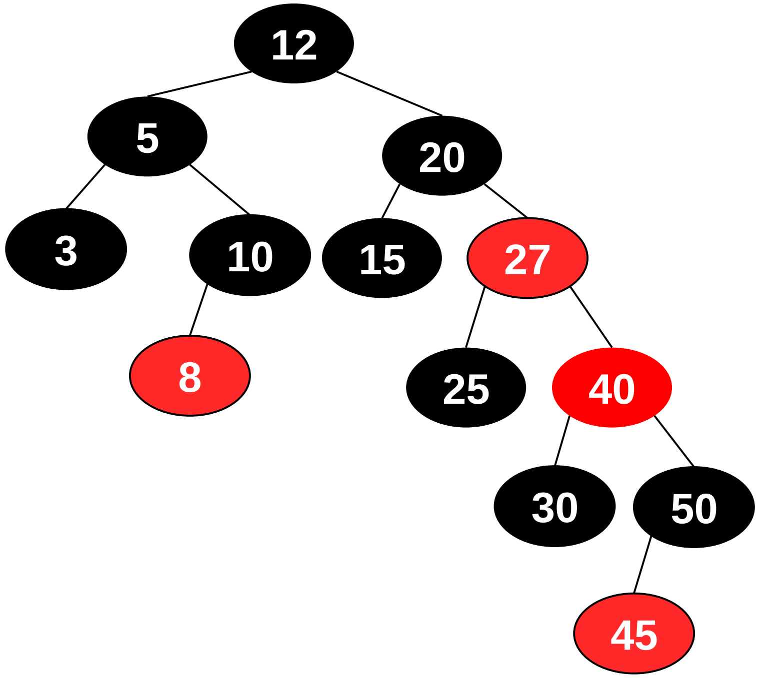
Insert 45



Insertion Exercise 4

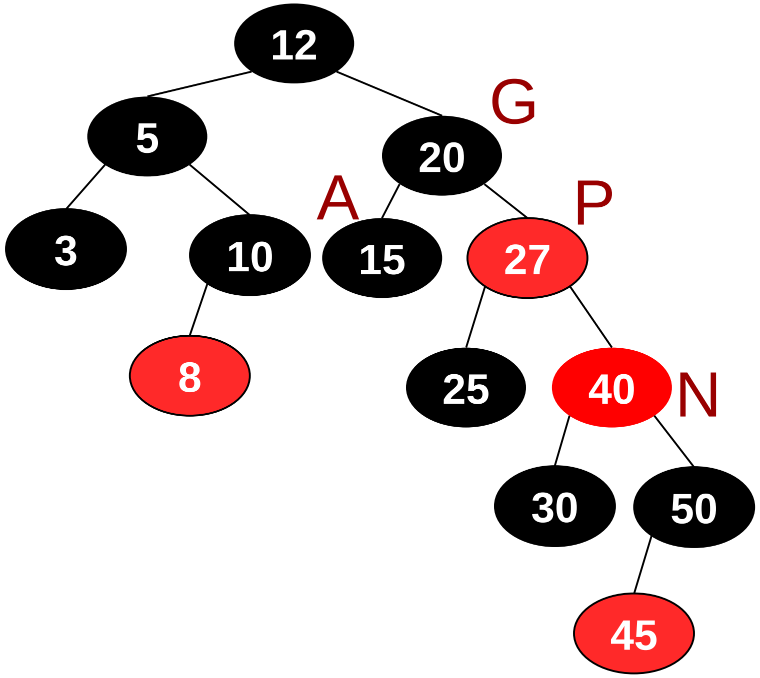
Insert 45



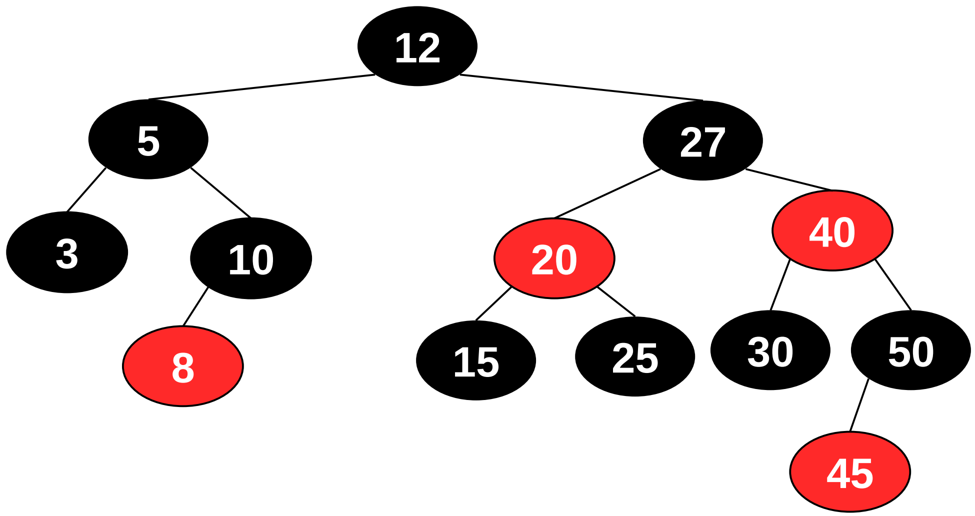


Aunt and Parent are the same color.
Just recolor.

Insertion Exercise 4

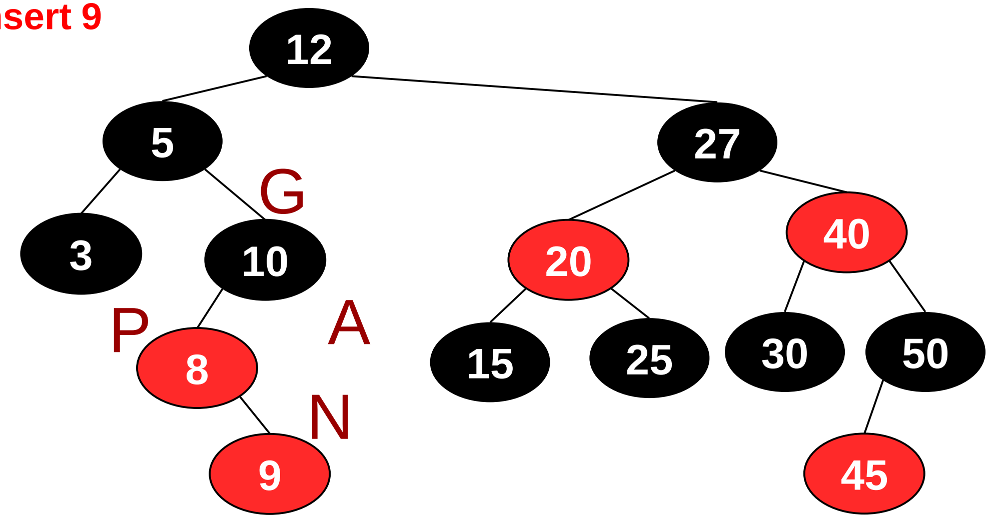


Final Result

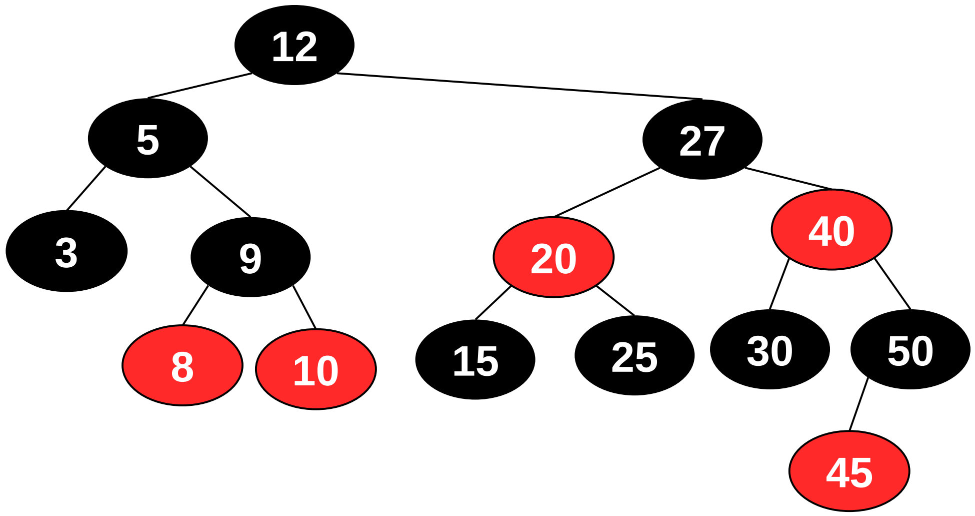


Insertion Exercise 5

Insert 9



Insertion Exercise 5



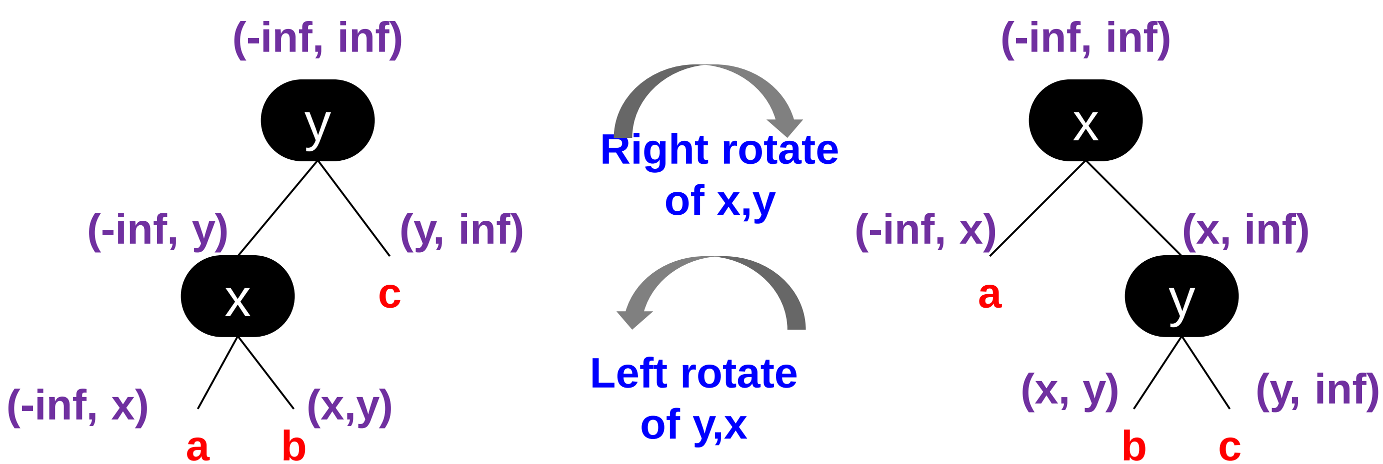
RB-Tree Visualization & Links

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

RB TREE IMPLEMENTATION

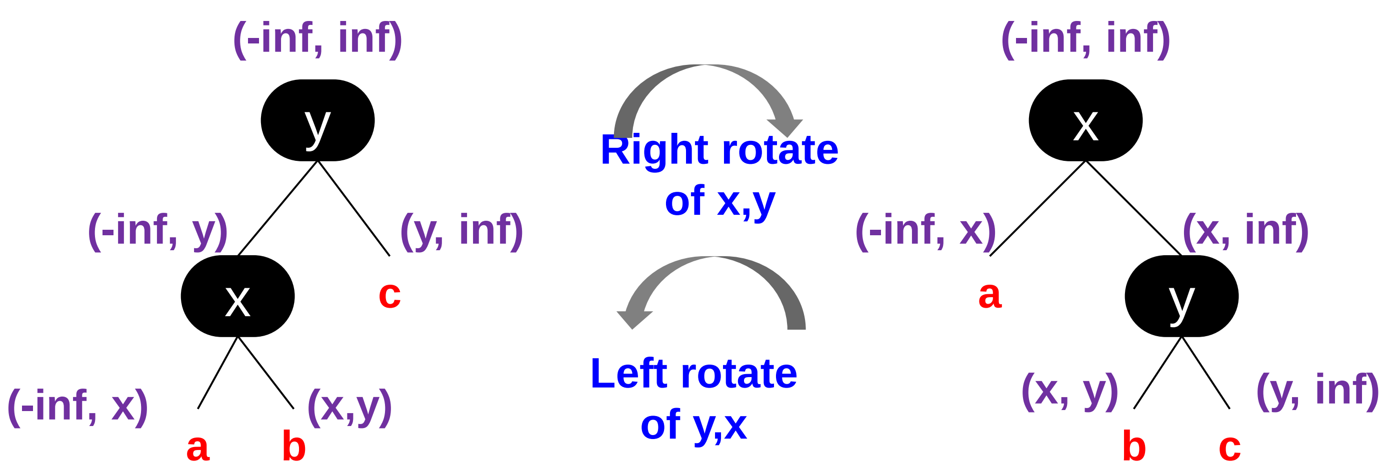
Hints

- 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

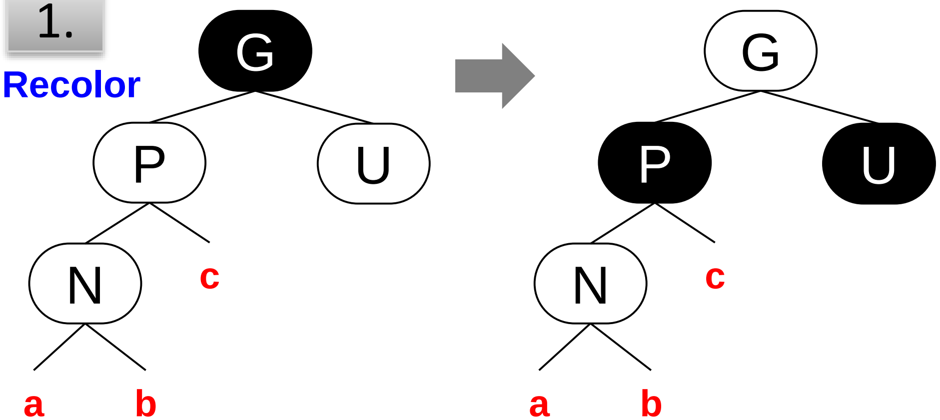
- You have to fix the tree after insertion if...
- Watch out for traversing NULL pointers
  - node->parent->parent
  - However, if you need to fix the tree your grandparent...
- Cases break down on uncle's color
  - If an uncle doesn't exist (i.e. is NULL), he is (color?)...



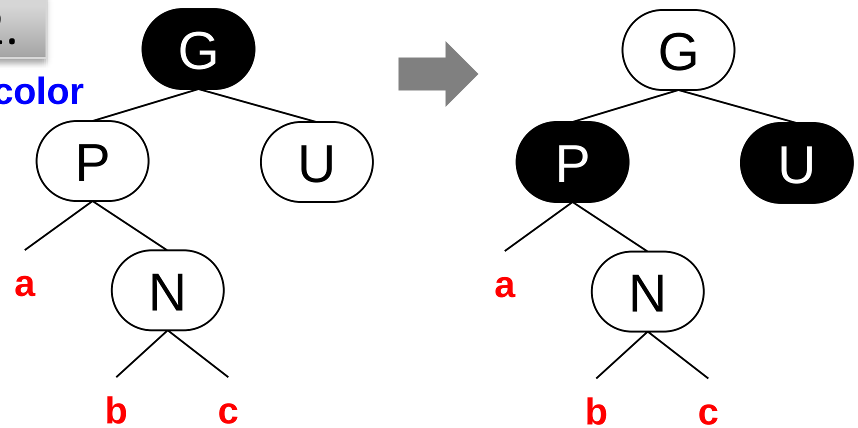
FOR PRINT

fixTree Cases

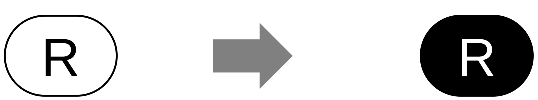
1\.
Recolor

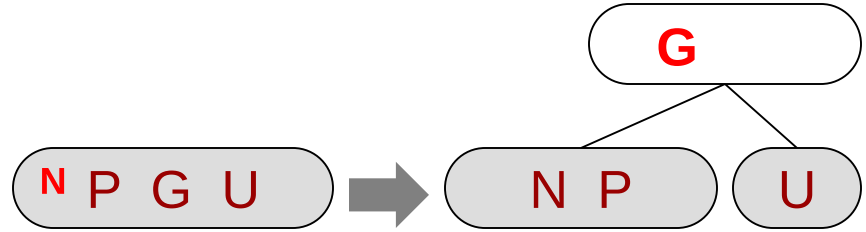


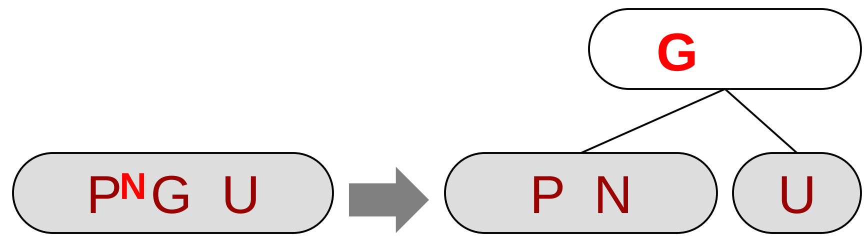
2\.
Recolor



3\.
Recolor Root





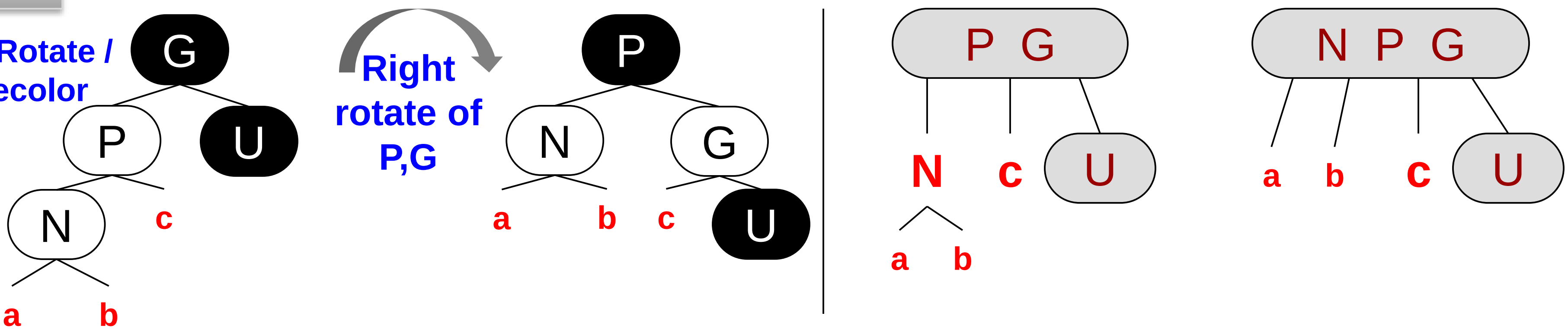


Note: For insertion/removal algorithm we consider non-existent leaf nodes as black nodes

fixTree Cases

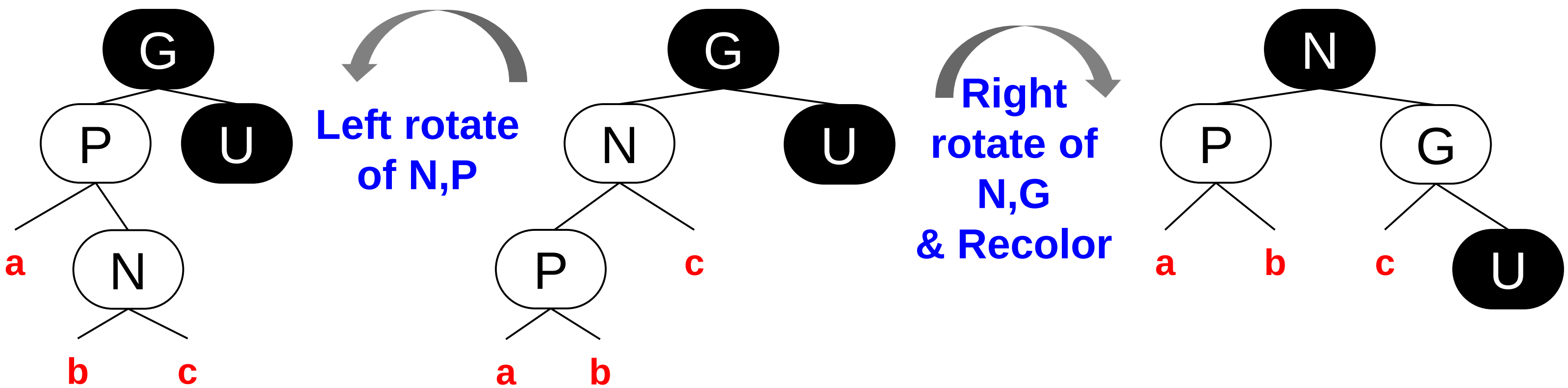
4.

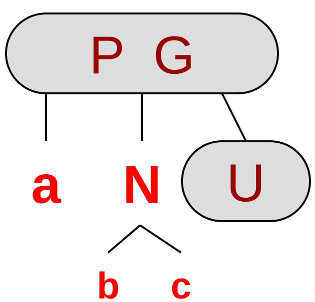
1 Rotate / Recolor

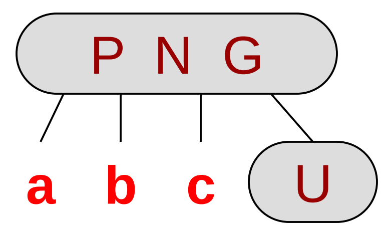


5.

2 Rotates / Recolor







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