Red-black trees

Mahdi Ebrahimi

Some slides adapted from
Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo

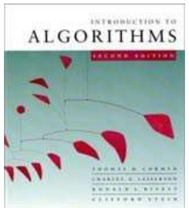
ece.uwaterloo.ca
dwharder@alumni.uwaterloo.ca

https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/#trees-and-hierarchical-orders

Outline

In this topic, we will cover:

- The idea behind a red-black tree
- Defining balance
- Insertions
- The benefits of red-black trees over AVL trees



Balanced search trees

Balanced search tree: A search-tree data structure for which a height of $O(\lg n)$ is guaranteed when implementing a dynamic set of n items.

Examples:
- AVL trees
- Red-black trees
- 2-3 trees
- 2-3-4 trees
- B-trees/B^+-trees

Red-black trees

This data structure requires an extra one-bit color field in each node.

Red-black properties:

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).

Red Black Trees

- A BST with more complex algorithms to ensure balance
- Each node is labeled as Red or Black.
- Path: A unique series of links (edges) traverses from the root to each node.
  - The number of edges (links) that must be followed is the path length
- In Red Black trees paths from the root to elements with 0 or 1 child are of particular interest

Red-Black Trees

A red black tree “colours” each node within a tree either red or black

- This can be represented by a single bit
- In AVL trees, balancing restricts the difference in heights to at most one
- For red-black trees, we have a different set of rules related to the colours of the nodes

Red-Black Trees

A little manipulation shows that the worst-case height simplifies to

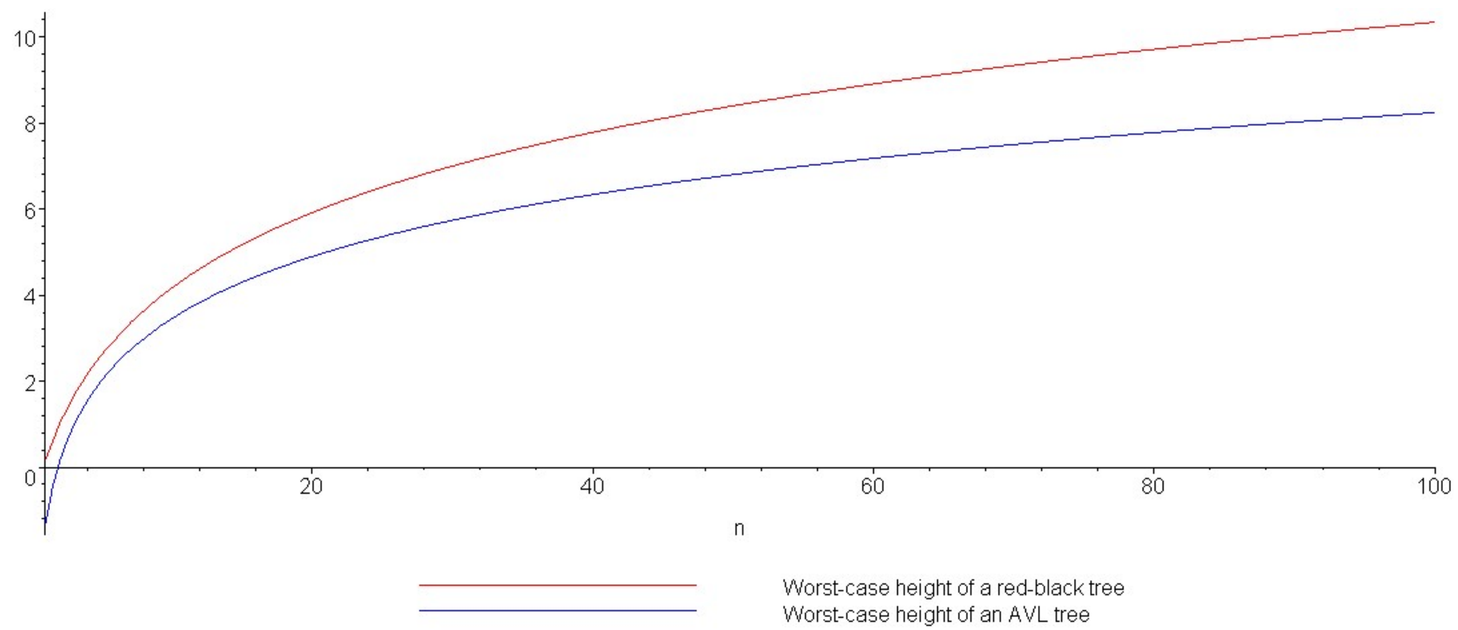
$$h_{\text{worst}} = 2\,\lg(n + 2) - 3$$

This grows quicker than the worst-case height for an AVL tree

$$h_{\text{worst}} = \log_{\phi}(\,n\,) - 1.3277$$

Red-Black Trees

Plotting the growth of the height of the worst-case red-black tree (red) versus the worst-case AVL tree (blue) demonstrates this:



Red-Black Trees

This table shows the number of nodes in a worst-case trees for the given heights

Thus, an AVL tree with 131070 nodes has a height of 23 while a red-black tree could have a height as large as 31

Height	AVL Tree	Red-Black Tree
1	2	2
3	7	6
5	20	14
7	54	30
9	143	62
11	376	126
13	986	254
15	2583	510
17	6764	1022
19	17710	2046
21	46367	4094
23	121492	8190
25	317810	16382
27	832039	32766
29	2178308	65534
31	5702886	131070
33	14930351	262142

Red-Black Trees

Comparing red-black trees with AVL trees, we note that:

- Red-black trees require one extra bit per node
- AVL trees require one byte per node (assuming the height will never exceed 255)
  - aside: we can reduce this to two bits, storing one of -1, 0, or 1 indicating that the node is left heavy, balanced, or right heavy, respectively

Red-Black Trees

AVL trees are not as deep in the worst case as are red-black trees

- therefore AVL trees will perform better when numerous searches are being performed,
- however, insertions and deletions will require:
  - more rotations with AVL trees, and
  - require recursions from and back to the root
- thus AVL trees will perform worse in situations where there are numerous insertions and deletions

Insertions

We will consider two types of insertions:

- bottom-up (insertion at the leaves), and
- top-down (insertion at the root)
  - This part is optional, and you can study by yourself.

The first will be instructional and we will use it to derive the second case

Bottom-Up Insertions

After an insertion is performed, we must satisfy all the rules of a red-black tree:

1. The root must be black,
2. If a node is red, its children must be black, and
3. Each path from a node to any of its descendants which are is not a full node (*i.e.*, two children) must have the same number of black nodes

The first and second rules are local: they affect a node and its neighbours

The third rule is global: adding a new black node anywhere will cause all of its ancestors to become unbalanced

Bottom-Up Insertions

Thus, when we add a new node, we will add a node so as to break the global rule:

- the new node must be red

We will then travel up the tree to the root fixing the requirement that the children of a red node must be black

Bottom-Up Insertions

If the parent of the inserted node is already black, we are done

- Otherwise, we must correct the problem

We will look at two cases:

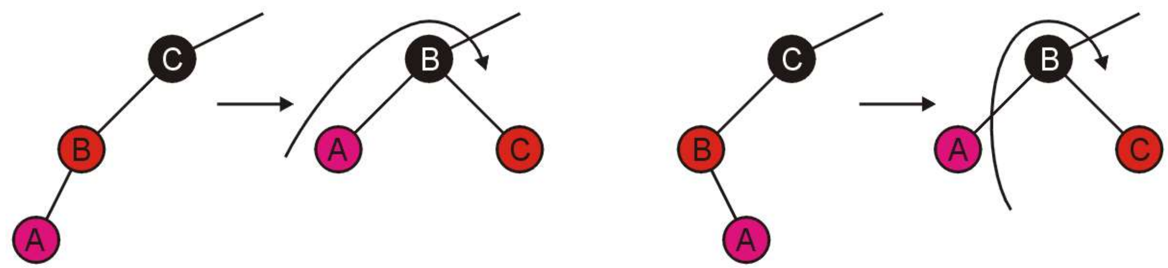
- the initial insertion, and
- the recursive steps back to the root

Bottom-Up Insertions

For the initial insertion, there are two possible cases:

- the grandparent has one child (the parent), or
- the grandparent has two children (both red)

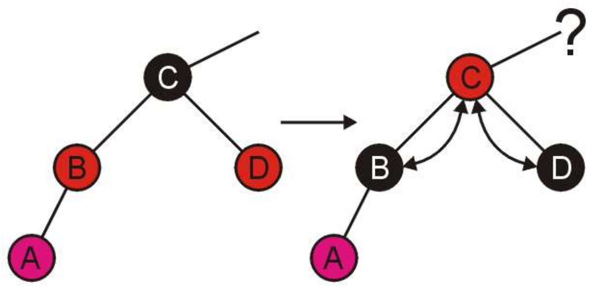
Inserting A, the first case can be fixed with a rotation:



Consequently, we are finished...

Bottom-Up Insertions

The second case can be fixed more easily, just swap the colours:



Unfortunately, we now may cause a problem between the parent and the grandparent....

Bottom-Up Insertions

Fortunately, dealing with problems caused within the tree are identical to the problems at the leaf nodes

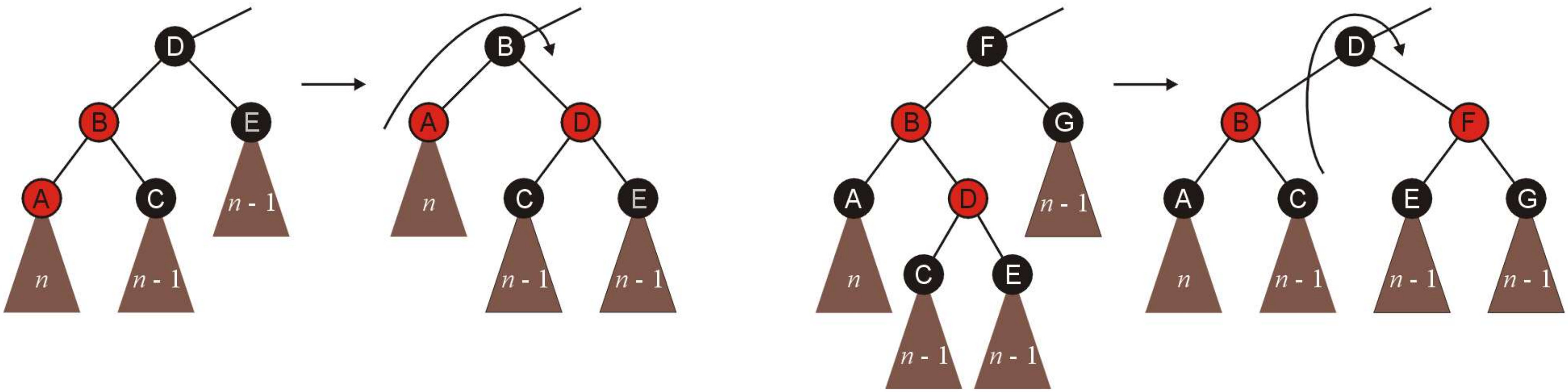
Like before, there are two cases:

- the grandparent has one child (the parent), or
- the grandparent has two children (both red)

Bottom-Up Insertions

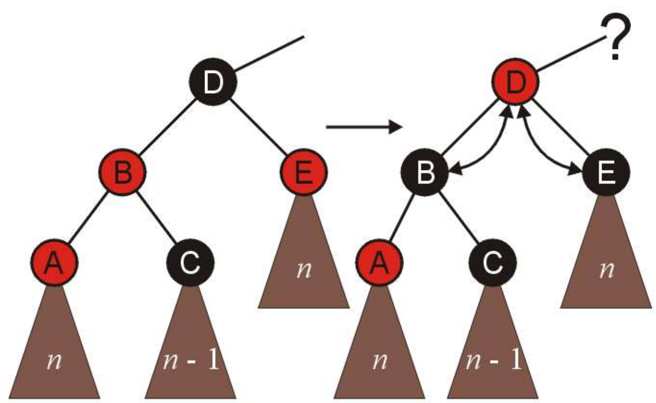
Suppose that A and D, respectively were swapped

In both these cases, we perform similar rotations as before, and we are finished



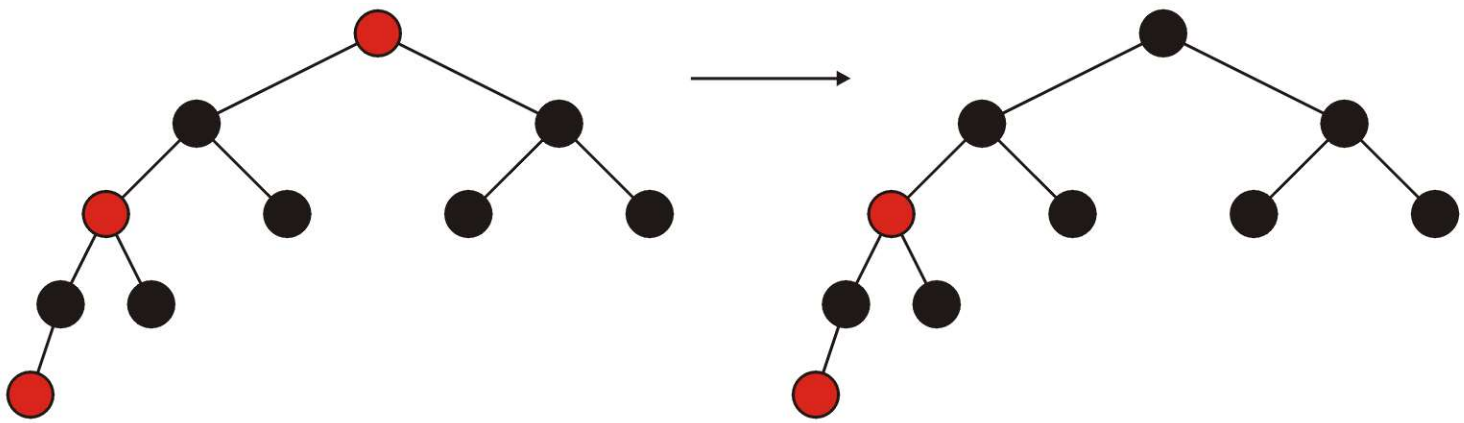
Bottom-Up Insertions

In the other case, where both children of the grandparent are red, we simply swap colours, and recurs back to the root



Bottom-Up Insertions

If, at the end, the root is red, it can be coloured black

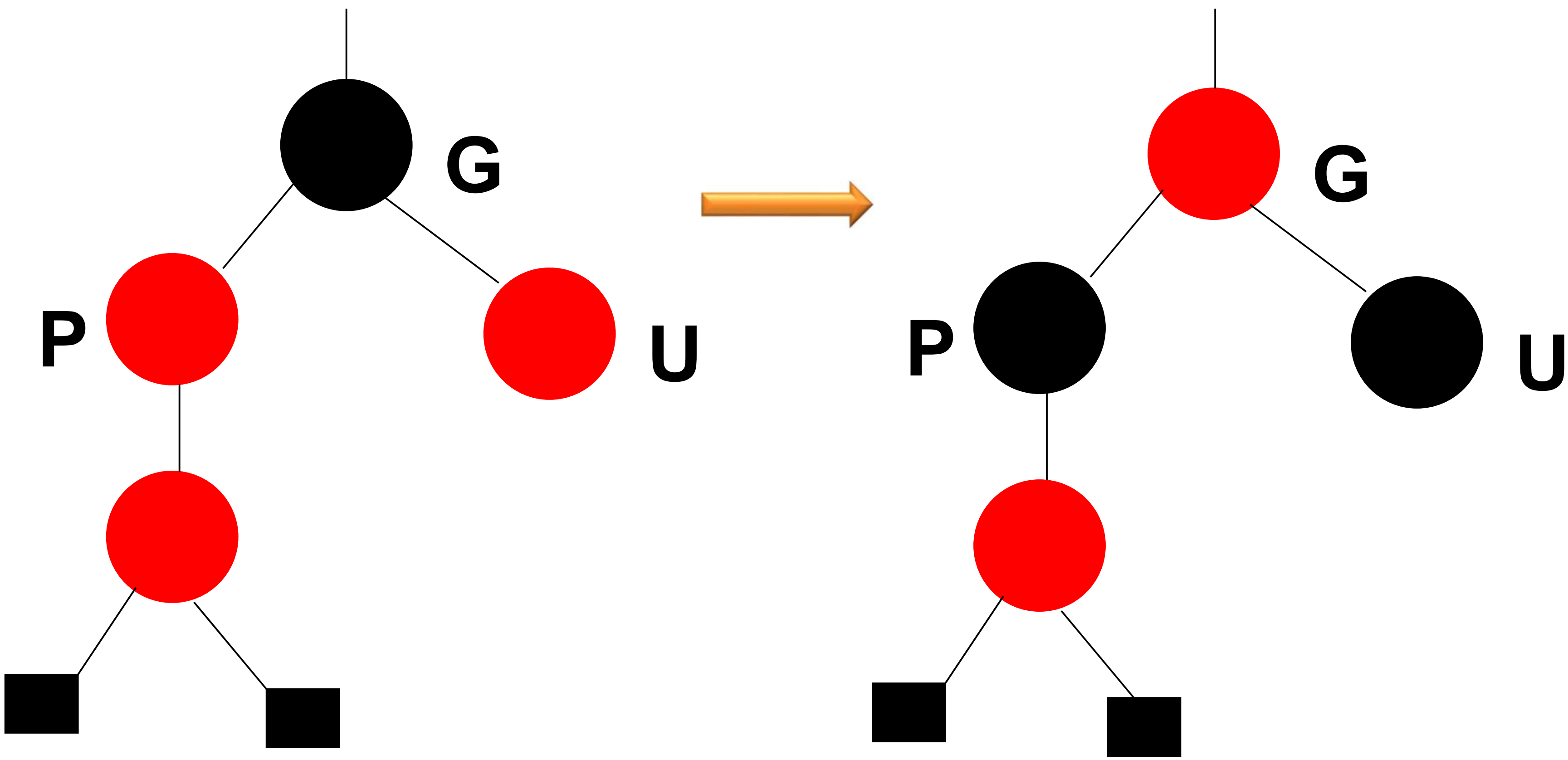


Red Black Trees Insertion

- Step 1 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.
- Step 2 - If tree is not Empty then insert the newNode as leaf node with color **Red**.
- Step 3 - If the parent of newNode is **Black** then exit from the operation.
- Step 4 - If the parent of newNode is **Red** then check the color of parentnode's sibling of newNode (Uncle's newNode).
  - Step 4.a - If it is colored **Black** or NULL then make suitable rotation and recolor.
  - Step 4.b - If it is colored **Red** then perform recolor and also check if parent's parent of newNode (grandparent's newNode) is not root node then recolor it and recheck (repeat the same until tree becomes Red Black Tree.)

We have 3 cases for insertion

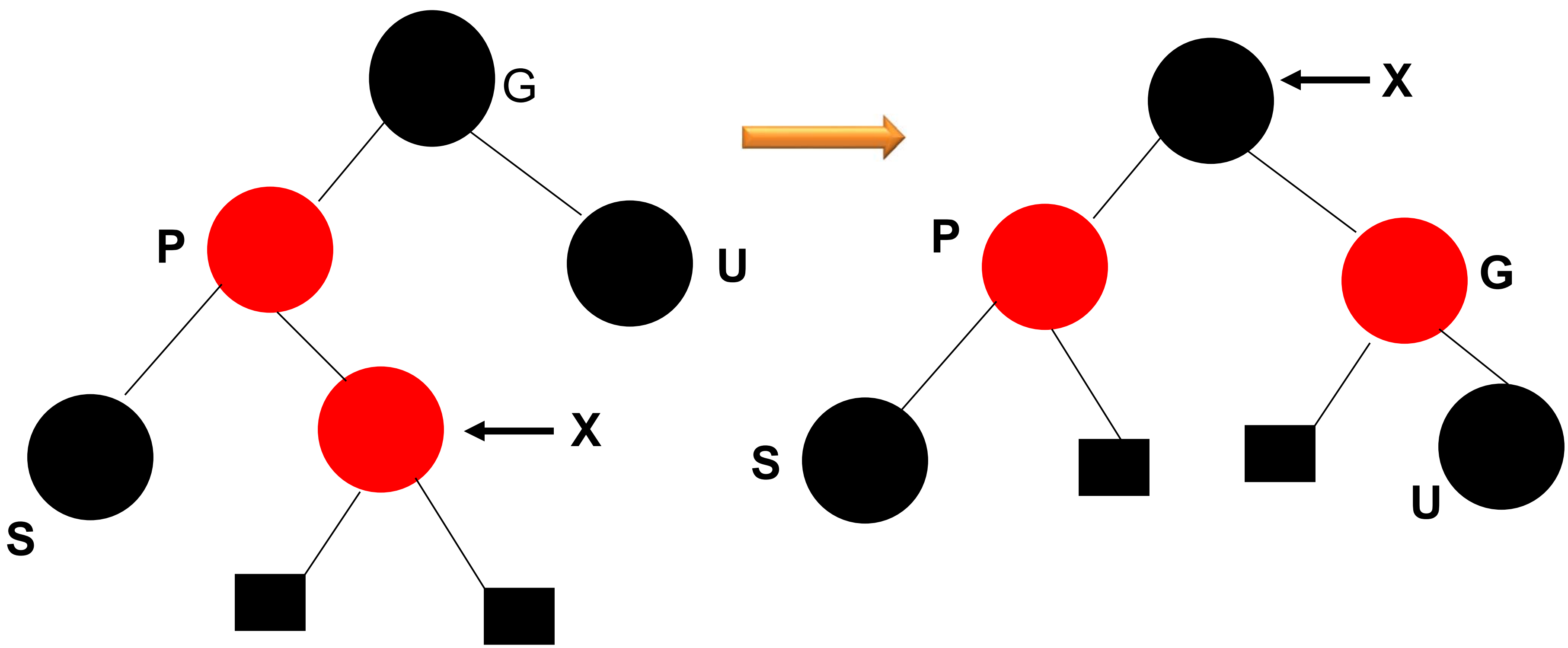
Case 1: Recolor (uncle is red)



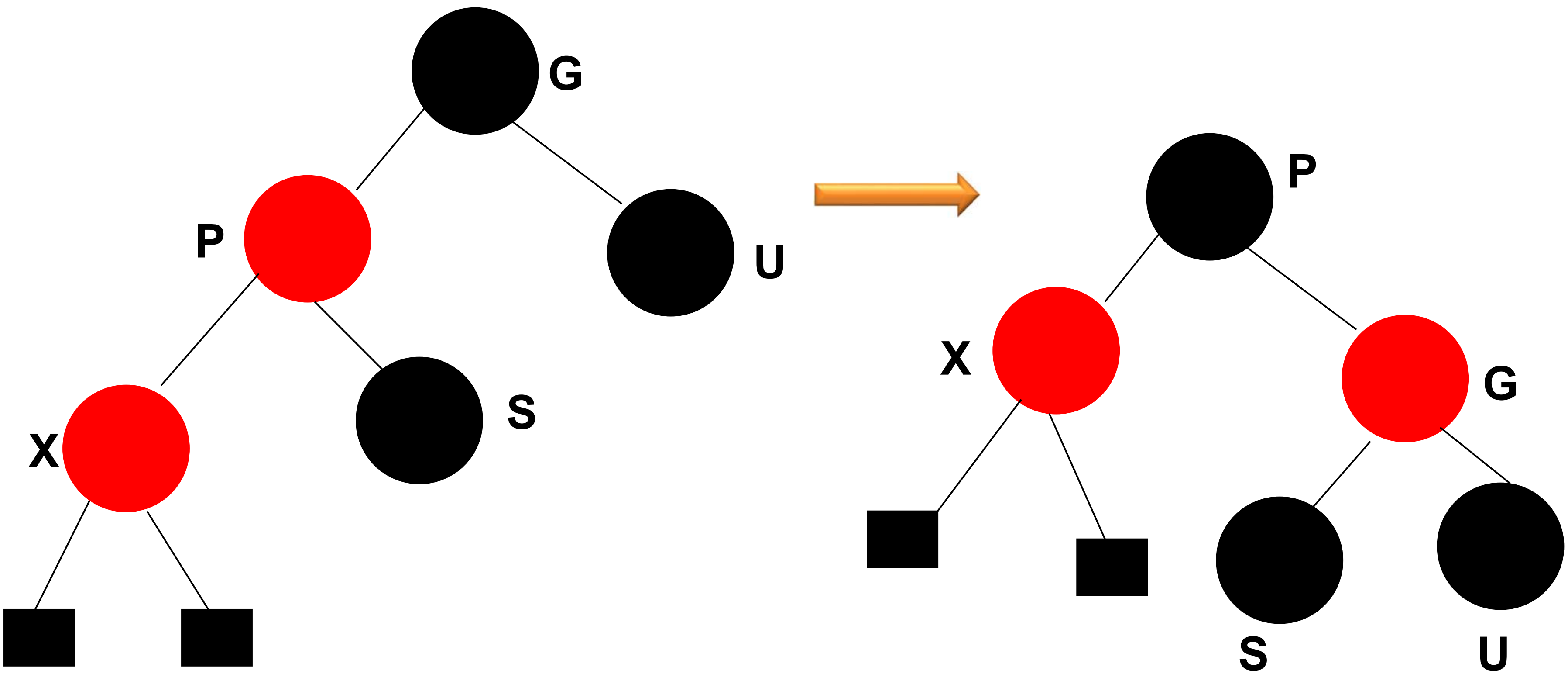
Case 2:

Double Rotate: X around P then X around G.

Recolor G and X

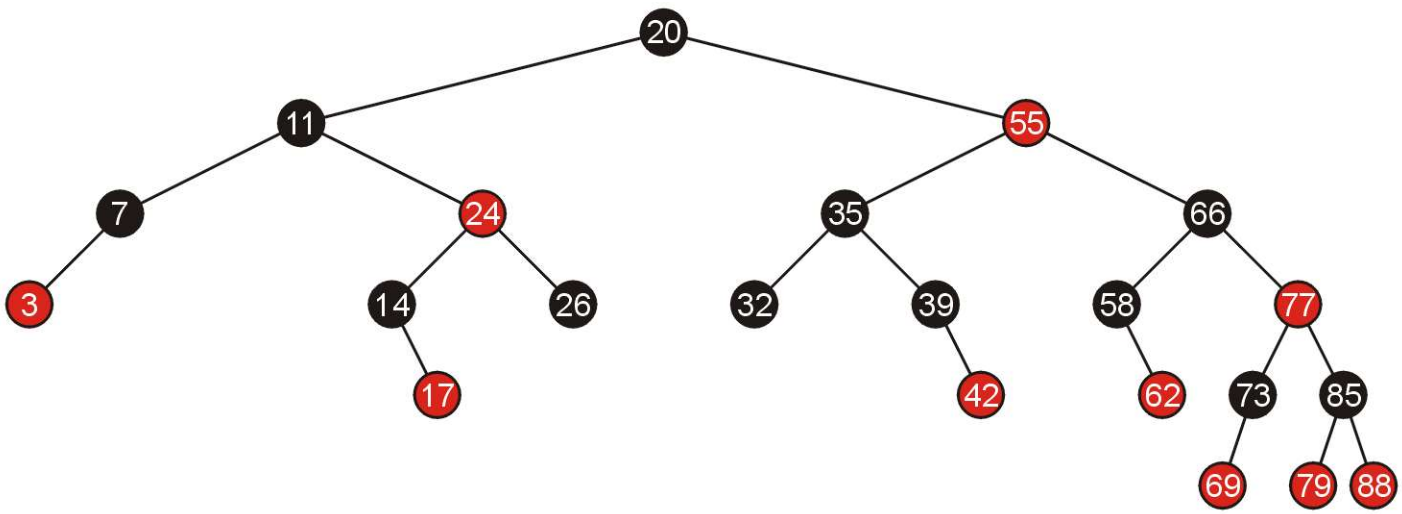


Case 3:
Single Rotate P around G
Recolor P and G



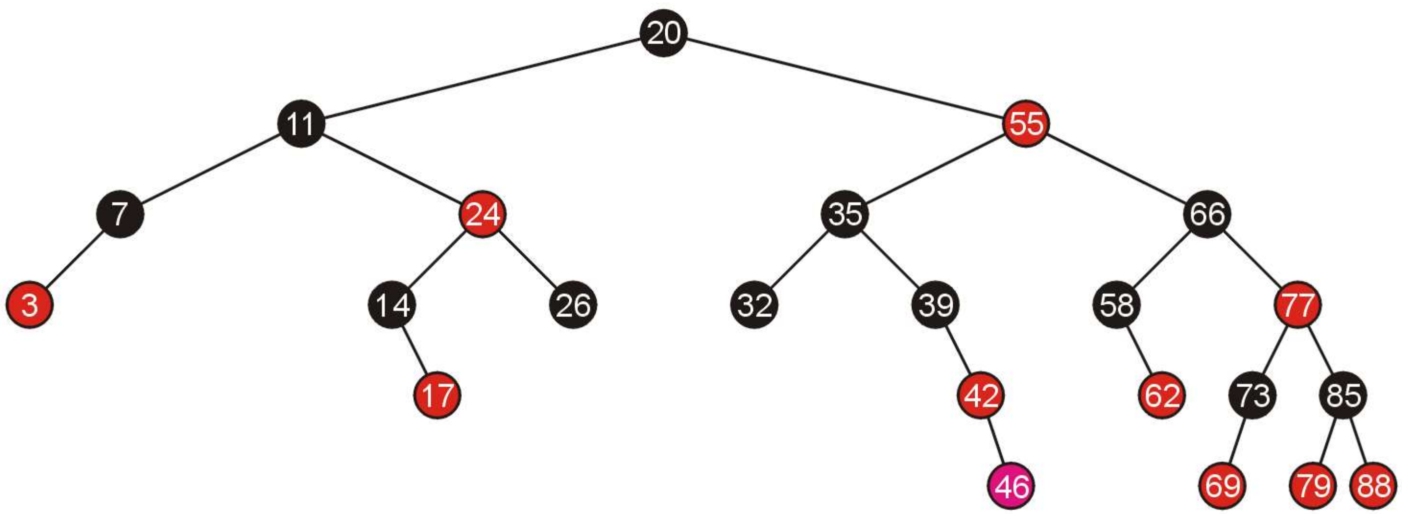
Examples of Insertions

Given the following red-black tree, we will make a number of insertions



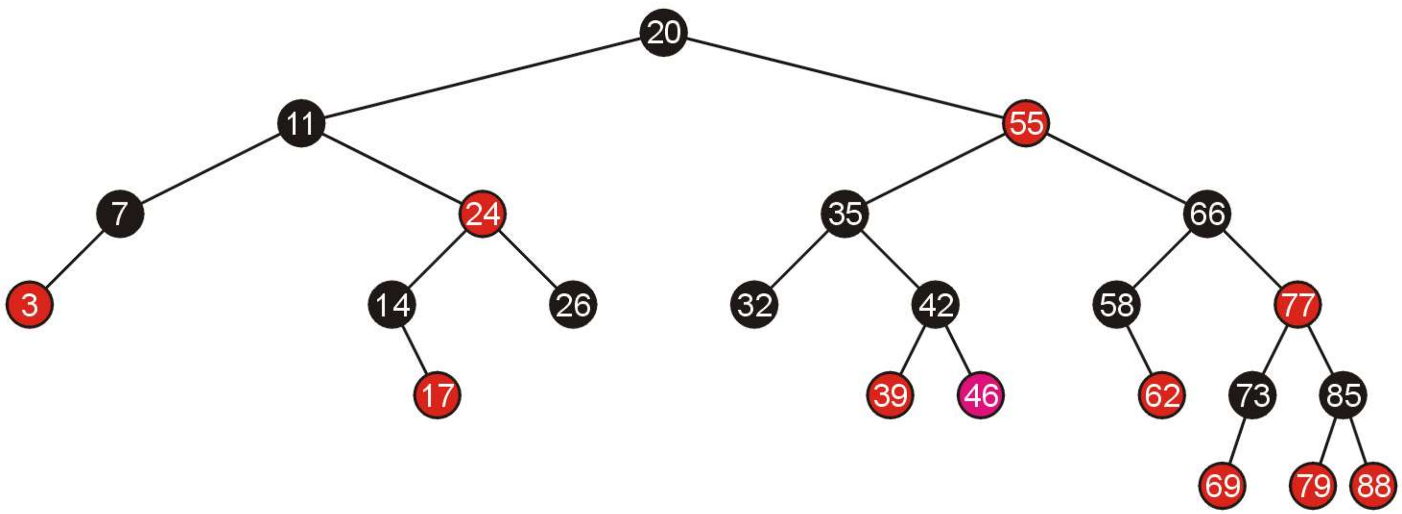
Examples of Insertions

Adding 46 creates a red-red pair which can be corrected with a single rotation



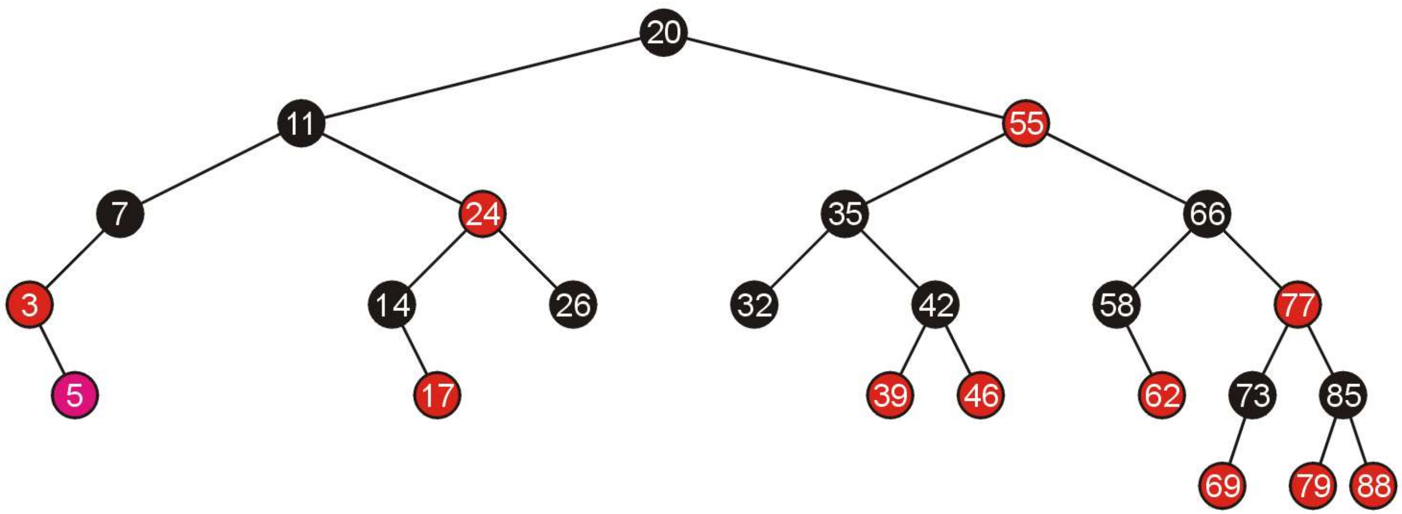
Examples of Insertions

Because the pivot is still black, we are finished



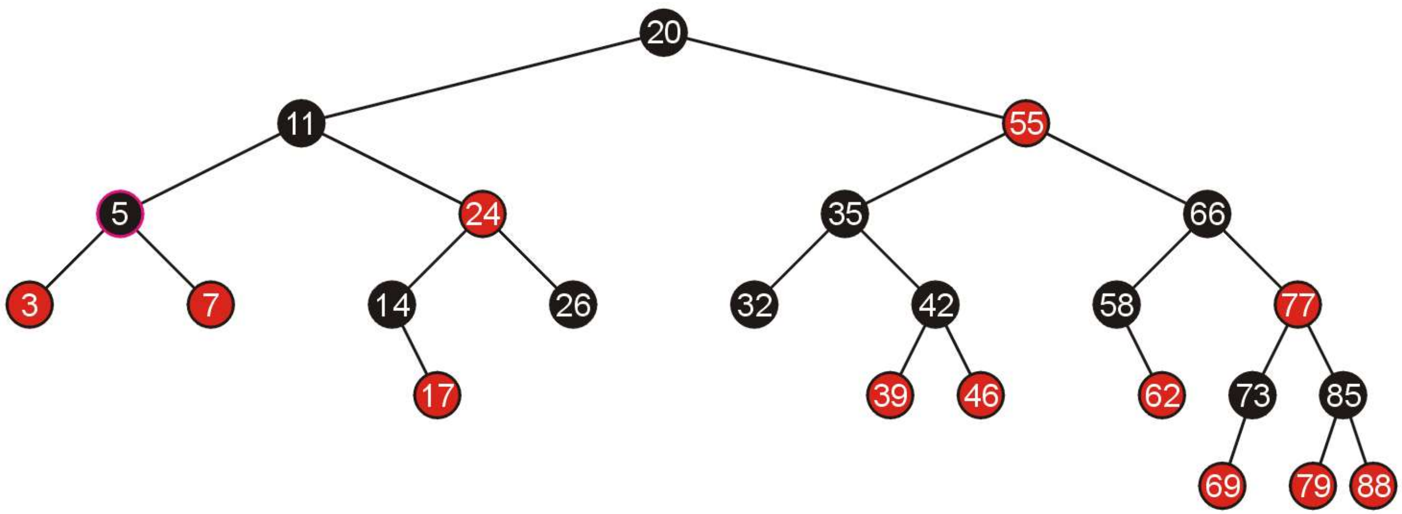
Examples of Insertions

Similarly, adding 5 requires a single rotation



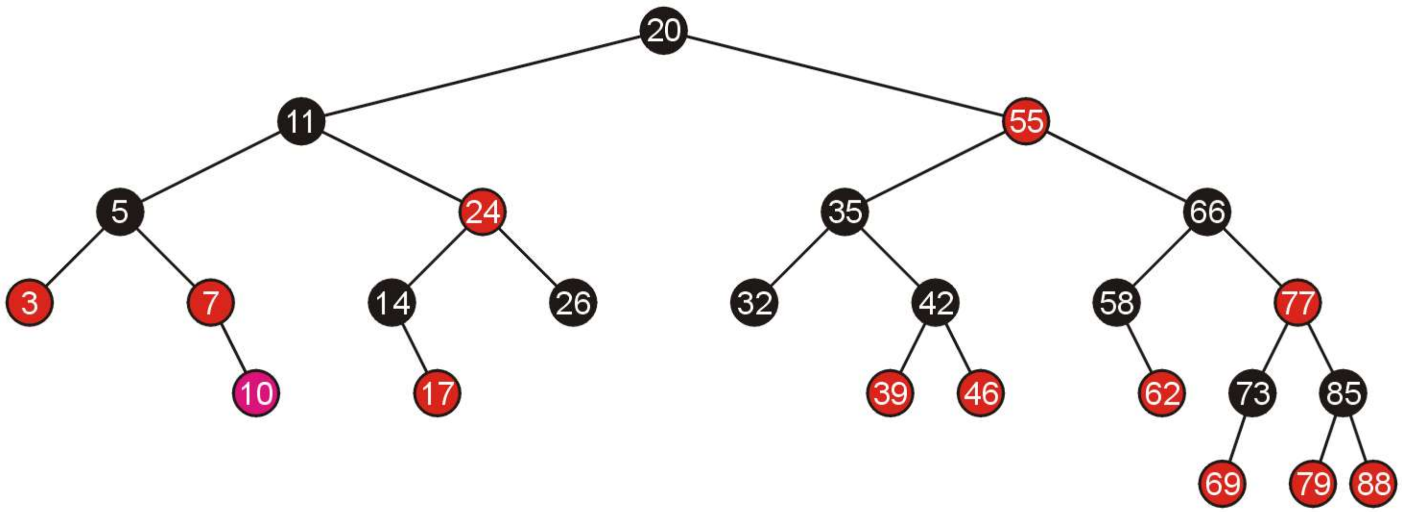
Examples of Insertions

Which again, does not require any additional work



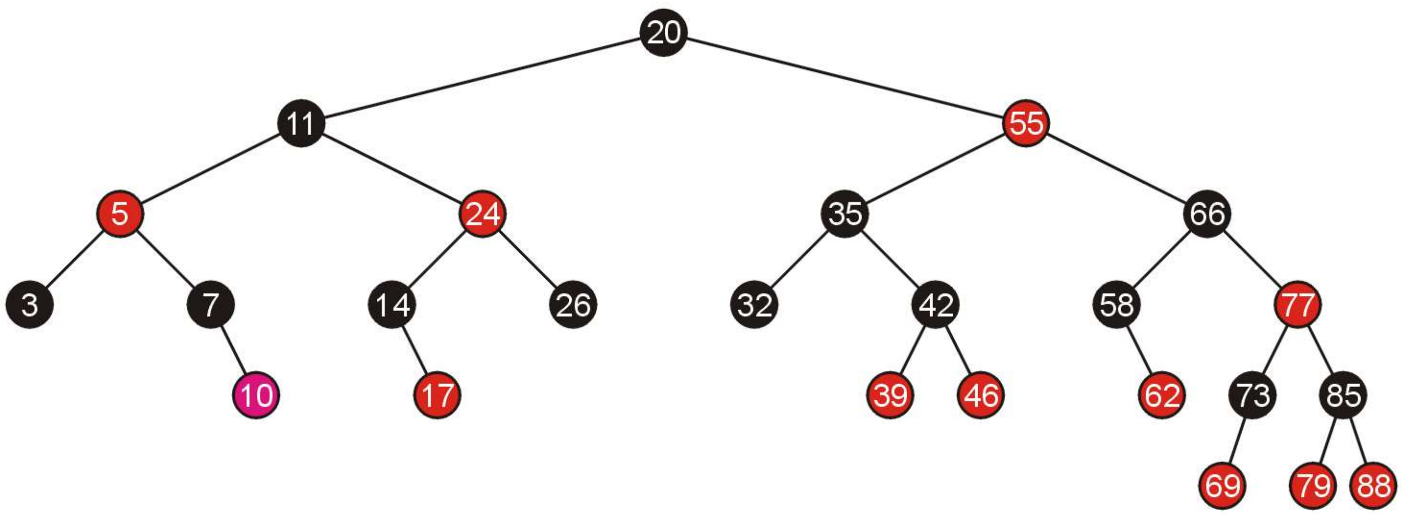
Examples of Insertions

Adding 10 allows us to simply swap the colour of the grand parent and the parent and the parent’s sibling



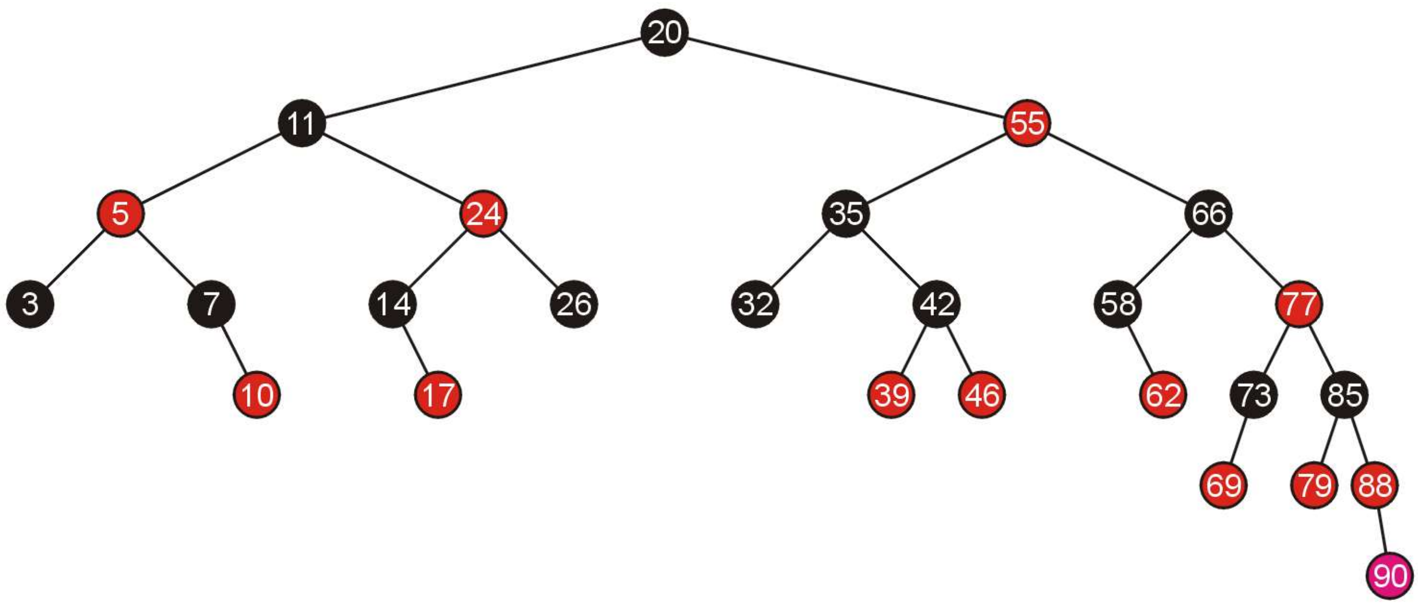
Examples of Insertions

Because the parent of 5 is black, we are finished



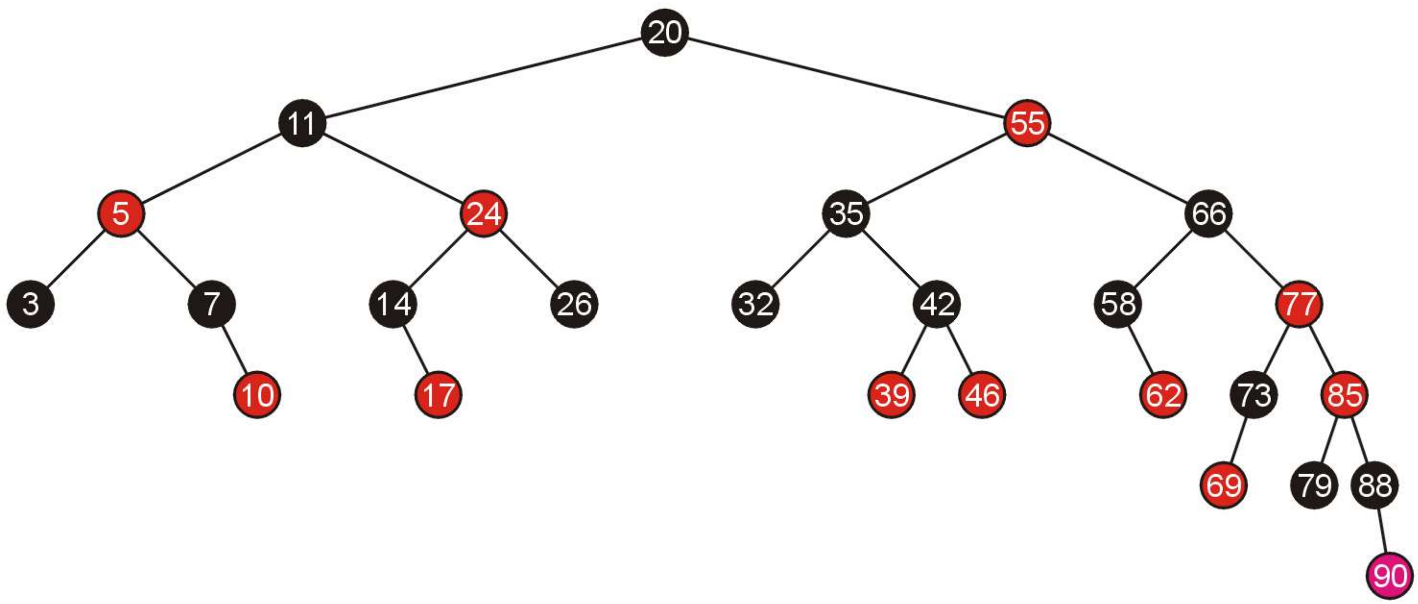
Examples of Insertions

Adding 90 again requires us to swap the colours of the grandparent and its two children



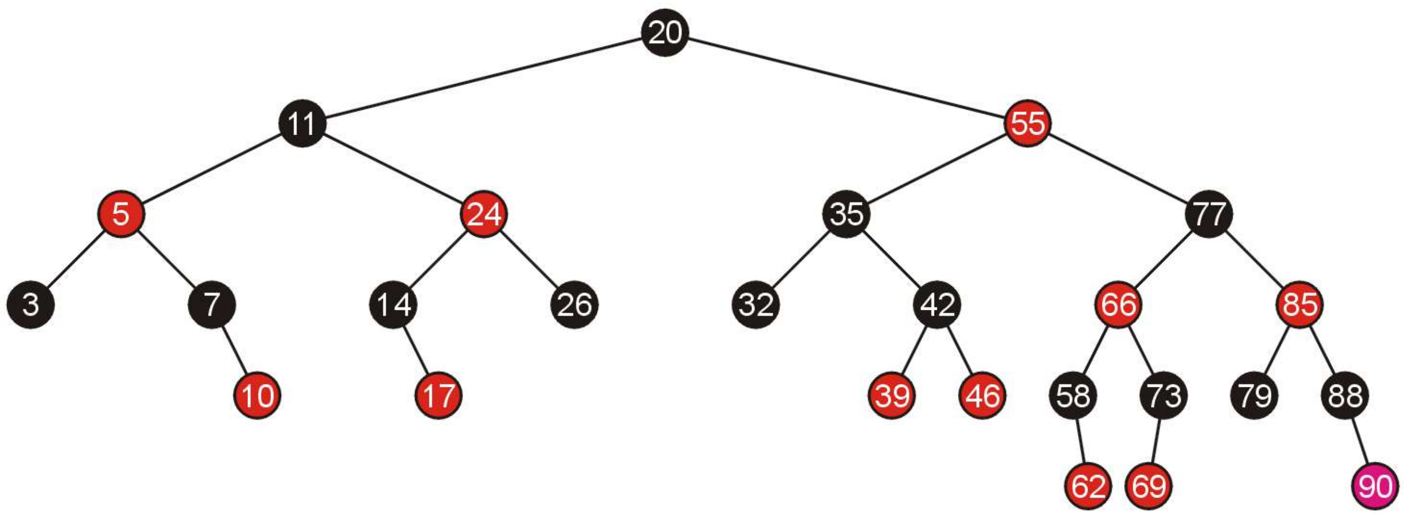
Examples of Insertions

This causes a red-red parent-child pair, which now requires a rotation



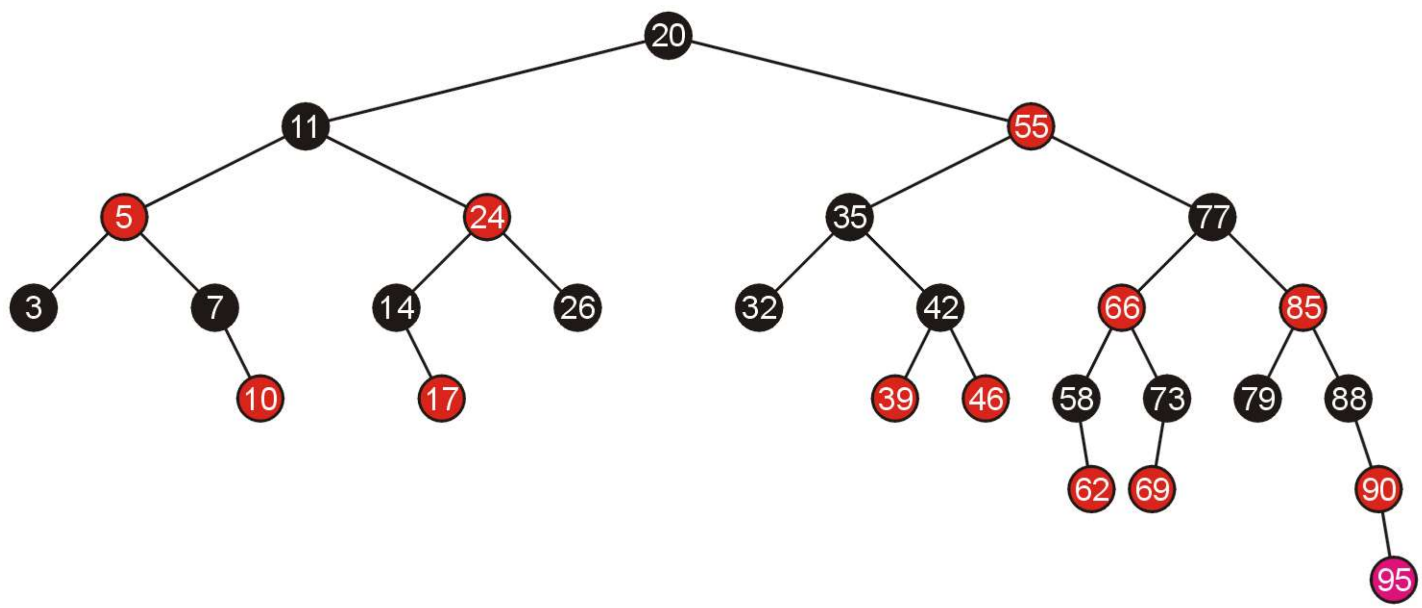
Examples of Insertions

A rotation does not require any subsequent modifications, so we are finished



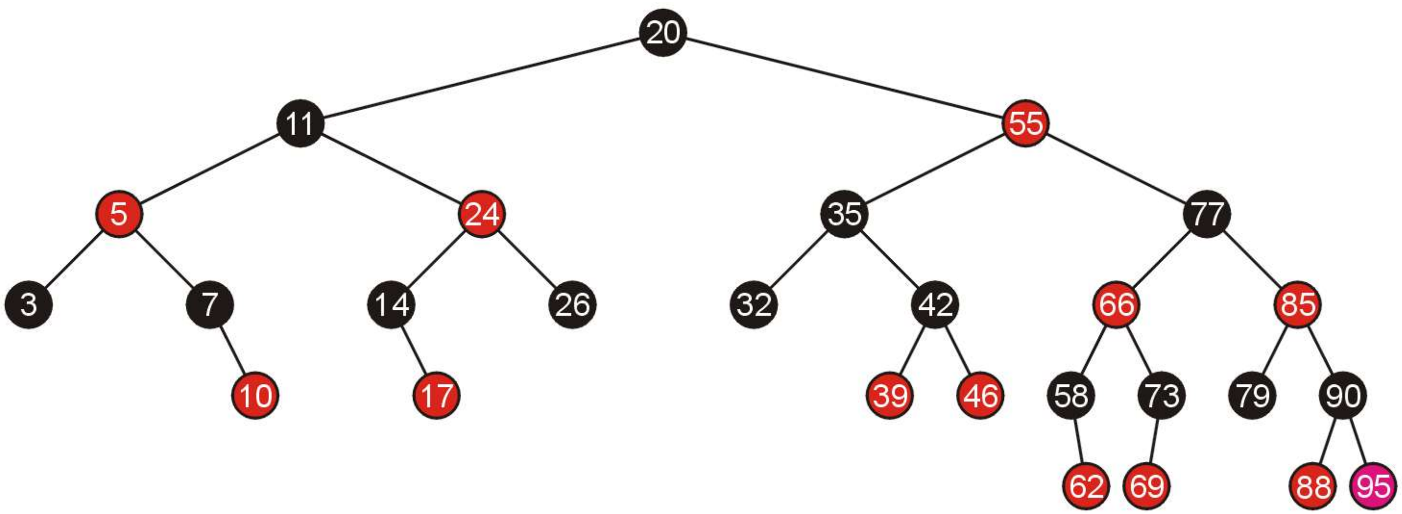
Examples of Insertions

Inserting 95 requires a single rotation



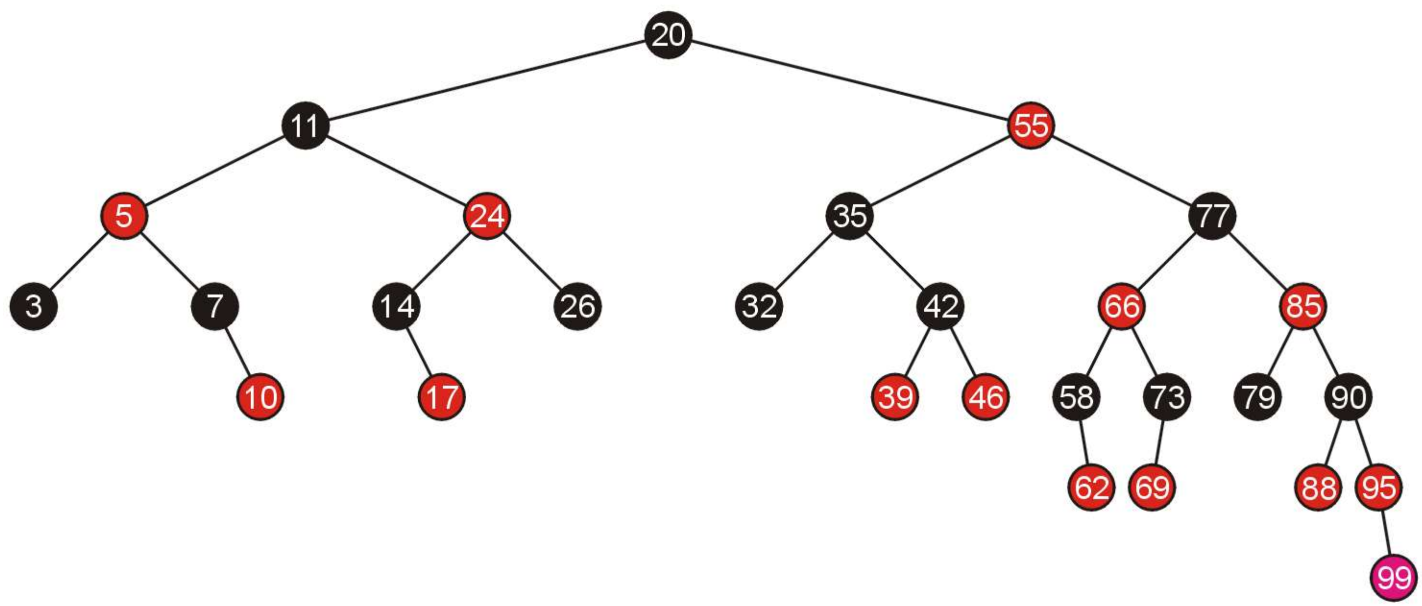
Examples of Insertions

And consequently, we are finished



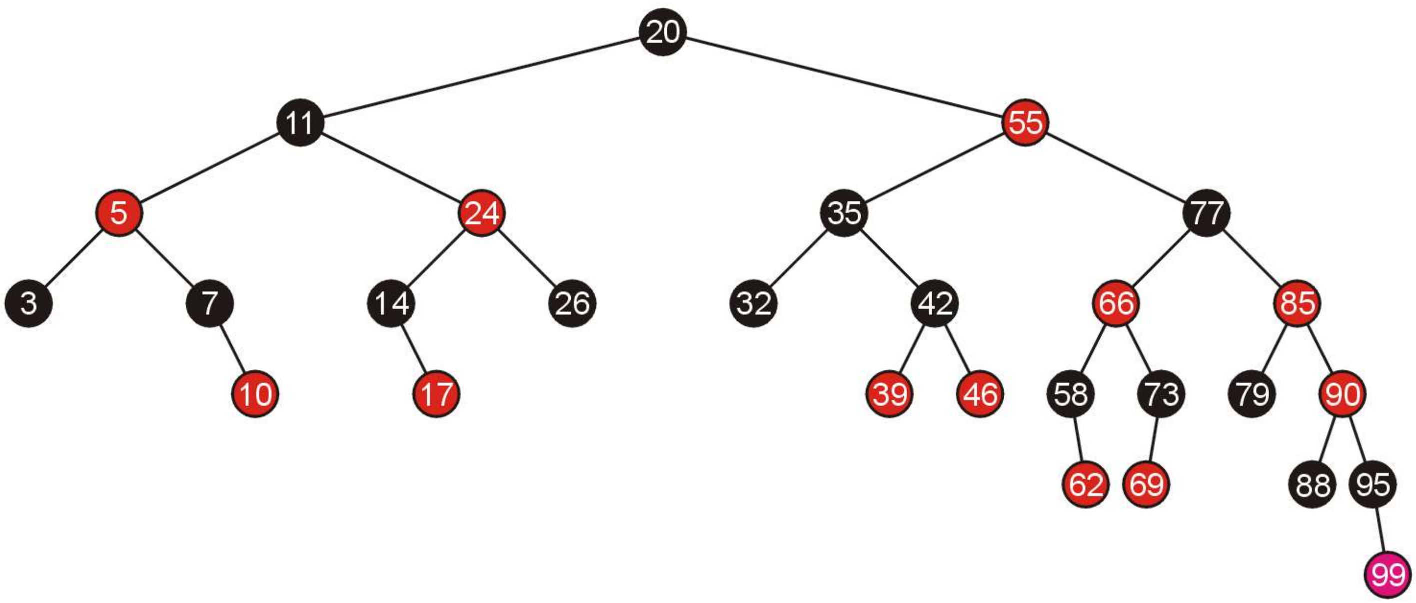
Examples of Insertions

Adding 99 requires us to swap the colours of its grandparent and the grandparent’s children



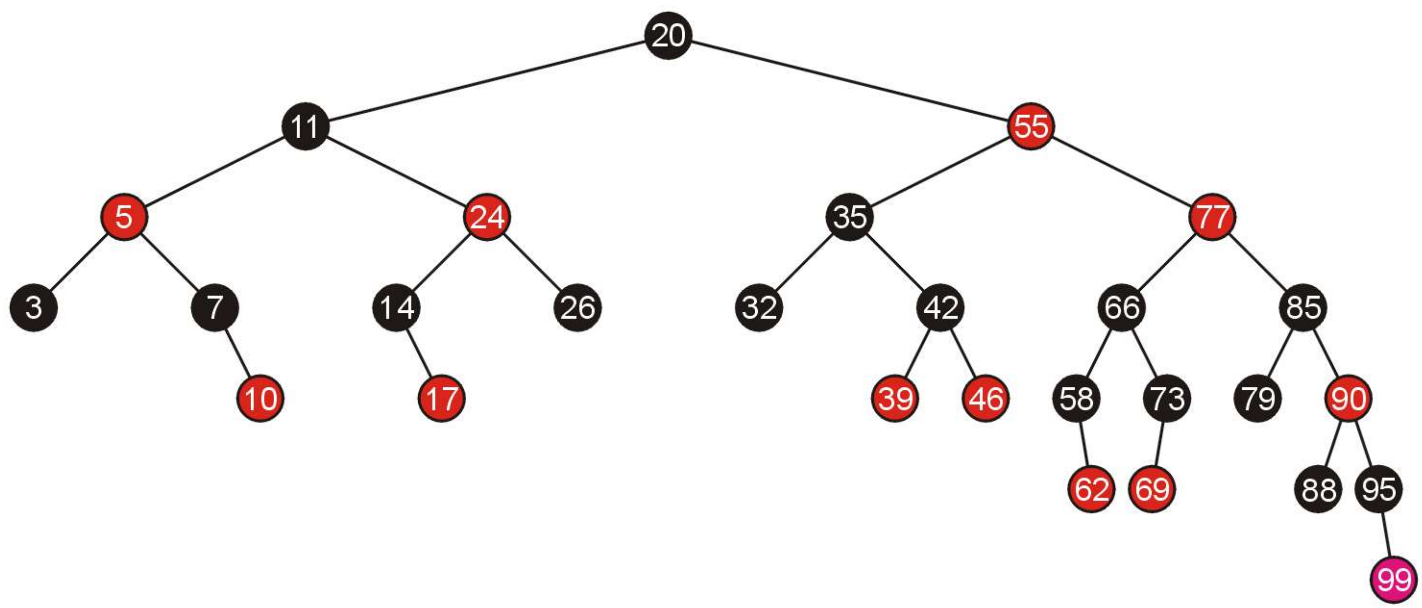
Examples of Insertions

This causes another red-red child-parent conflict between 85 and 90 which must be fixed, again by swapping colours



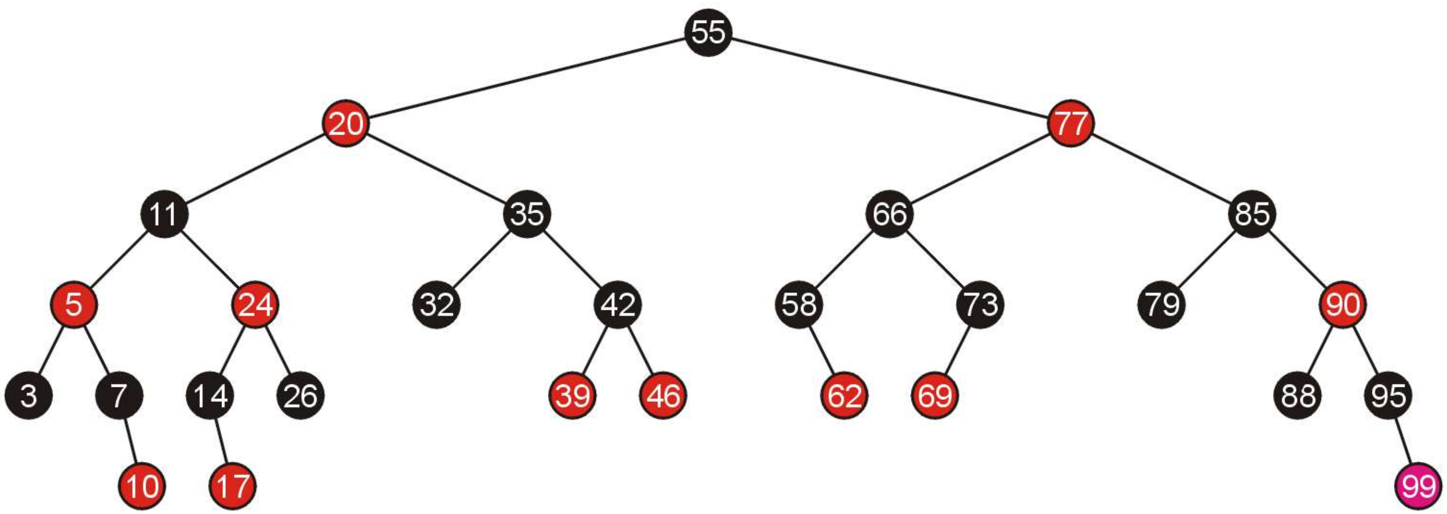
Examples of Insertions

This results in another red-red parent-child conflict, this time, requiring a rotation



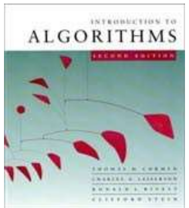
Examples of Insertions

Thus, the rotation solves the problem



Example 2

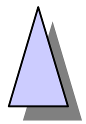
- Create a red-black tree by inserting the following sequence of numbers 8, 18, 5, 15, 17, 25, 40, and 80

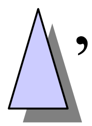


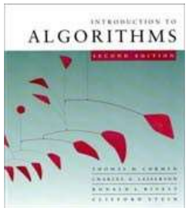
Pseudocode

RB-INSERT(T, x) TREE-INSERT(T, x)
$color[x] \leftarrow$ RED ◁ only RB property 3 can be violated
while $x \neq root[T]$ and $color[p[x]] =$ RED
  do if $p[x] = left[p[p[x]]]$
    then $y \leftarrow right[p[p[x]]]$ ◁ 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⟩
 $color[root[T]] \leftarrow$ BLACK

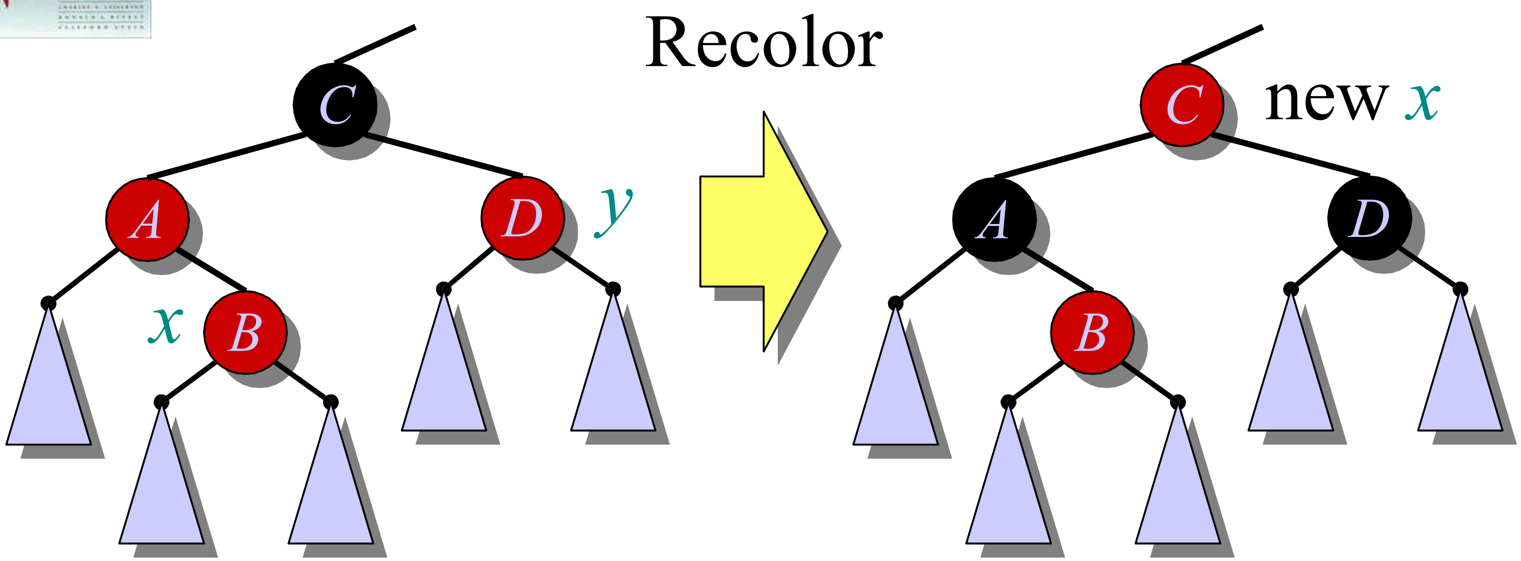
Graphical notation

Let  denote a subtree with a black root.

All ’s have the same black-height.



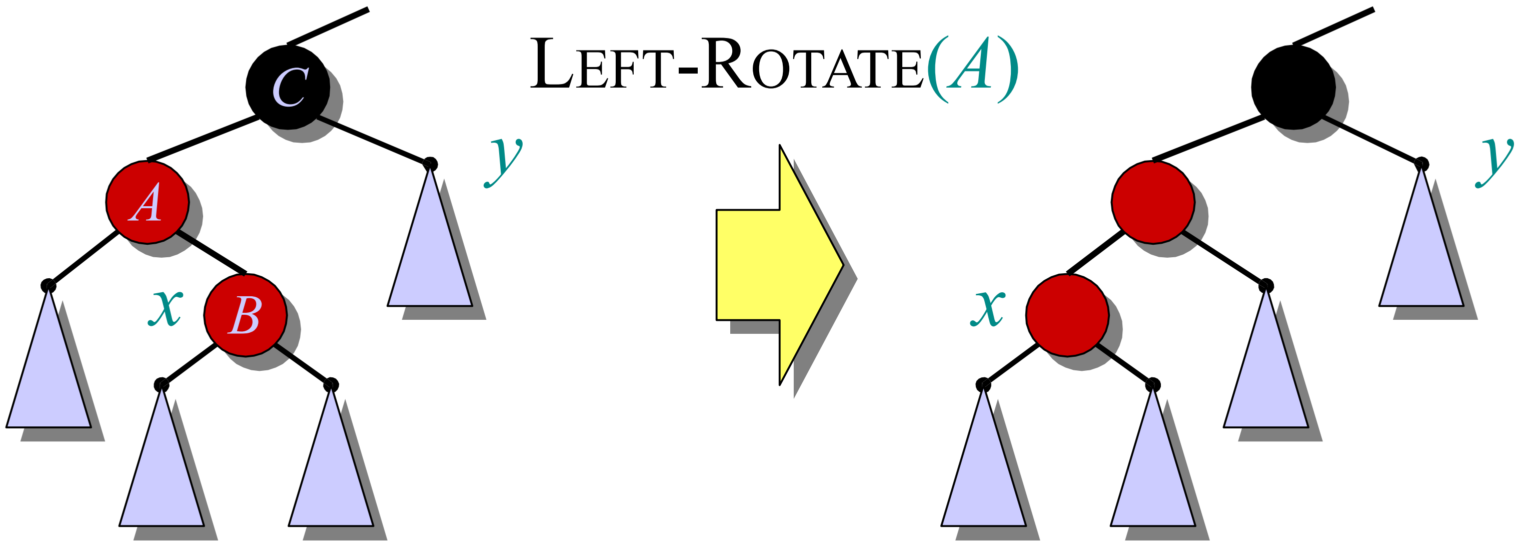
Case 1



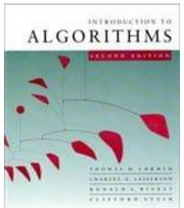
(Or, children of *A* are swapped.)

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

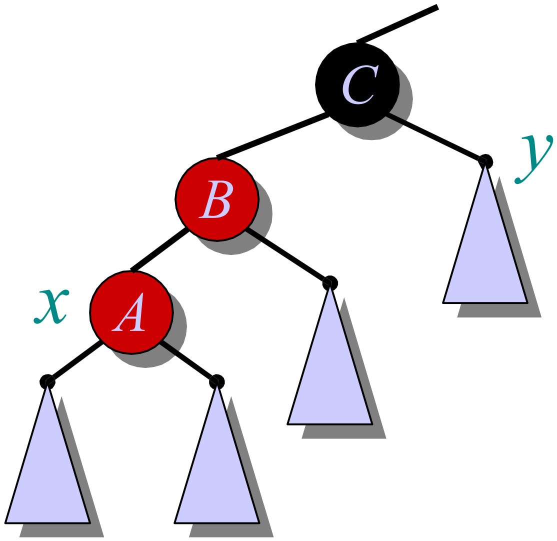
Case 2



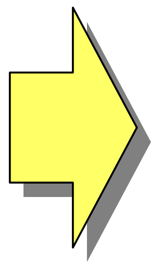
Transform to Case 3.

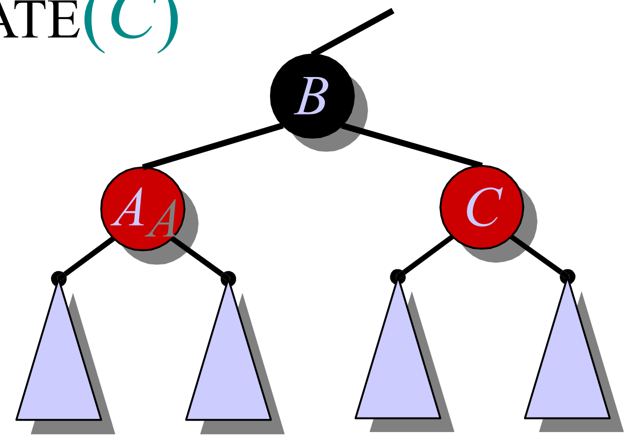


Case 3



RIGHT-ROTATE(C)





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.

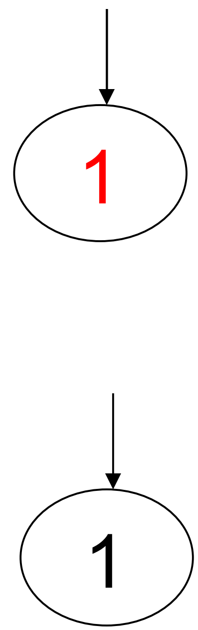
Running time: $O(\lg n)$ with $O(1)$ rotations.

RB-DELETE — same asymptotic running time and number of rotations as RB-INSERT.

Example of Inserting Sorted Numbers

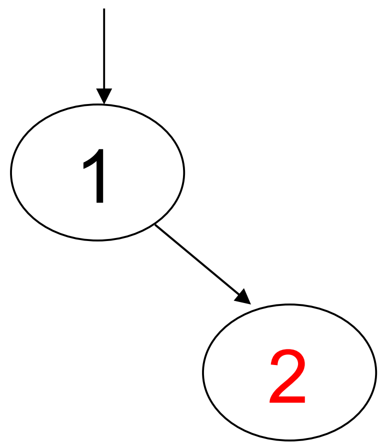
- 1 2 3 4 5 6 7 8 9 10

Insert 1. A leaf so red. Realize it is root so recolor to black.

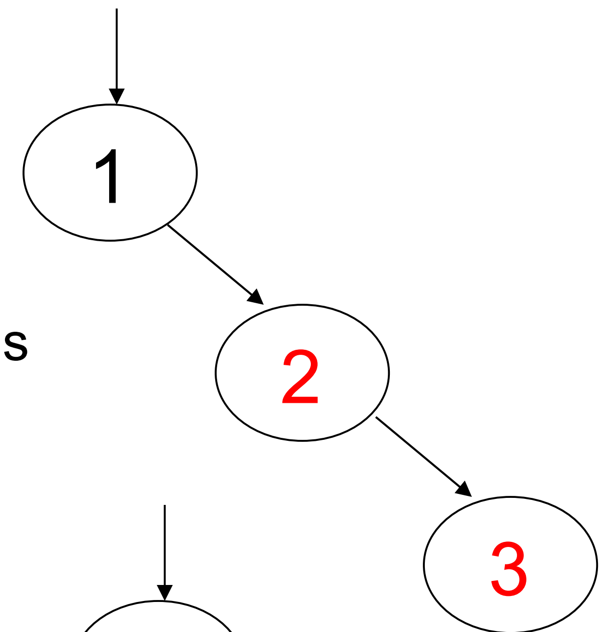


Insert 2

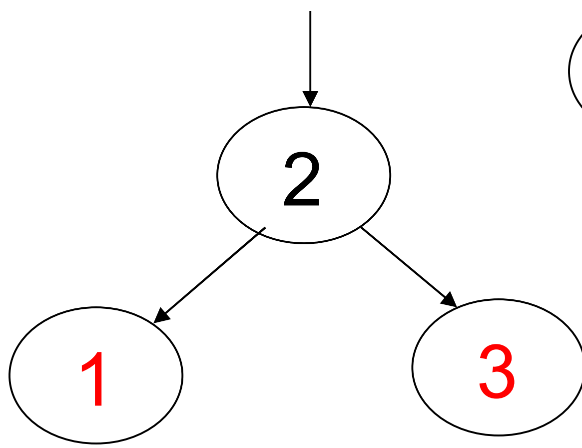
make 2 red. Parent is black so done.



Insert 3

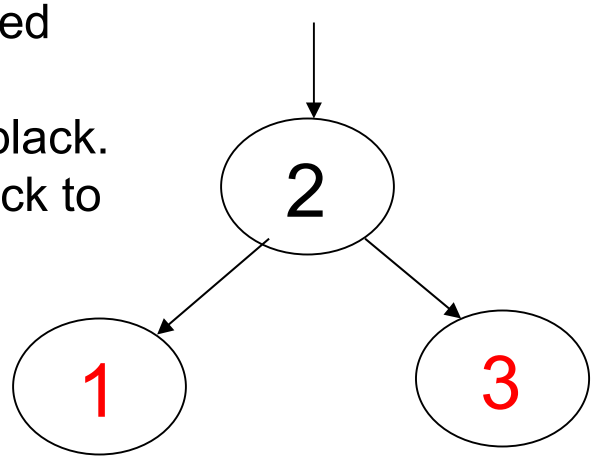


Insert 3. Parent is red.
Parent's sibling is black (null) 3 is outside relative to grandparent.
Rotate parent and grandparent

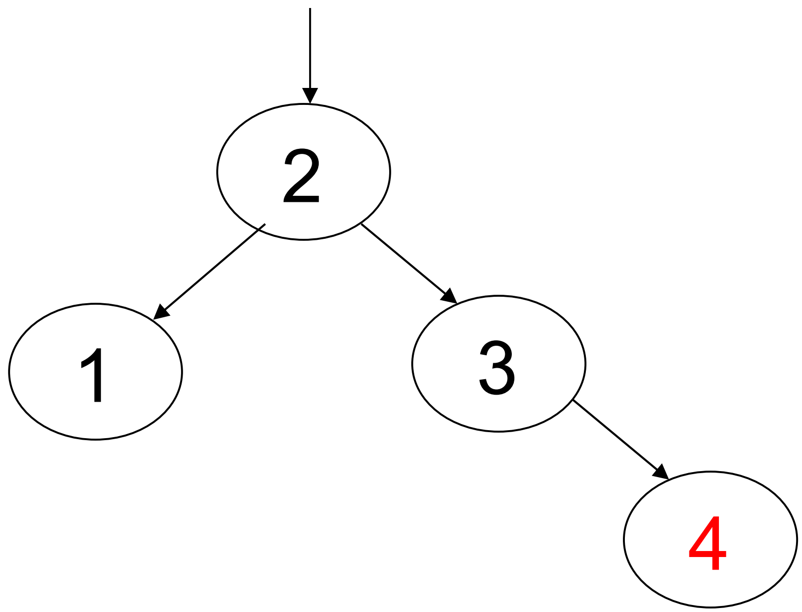


Insert 4

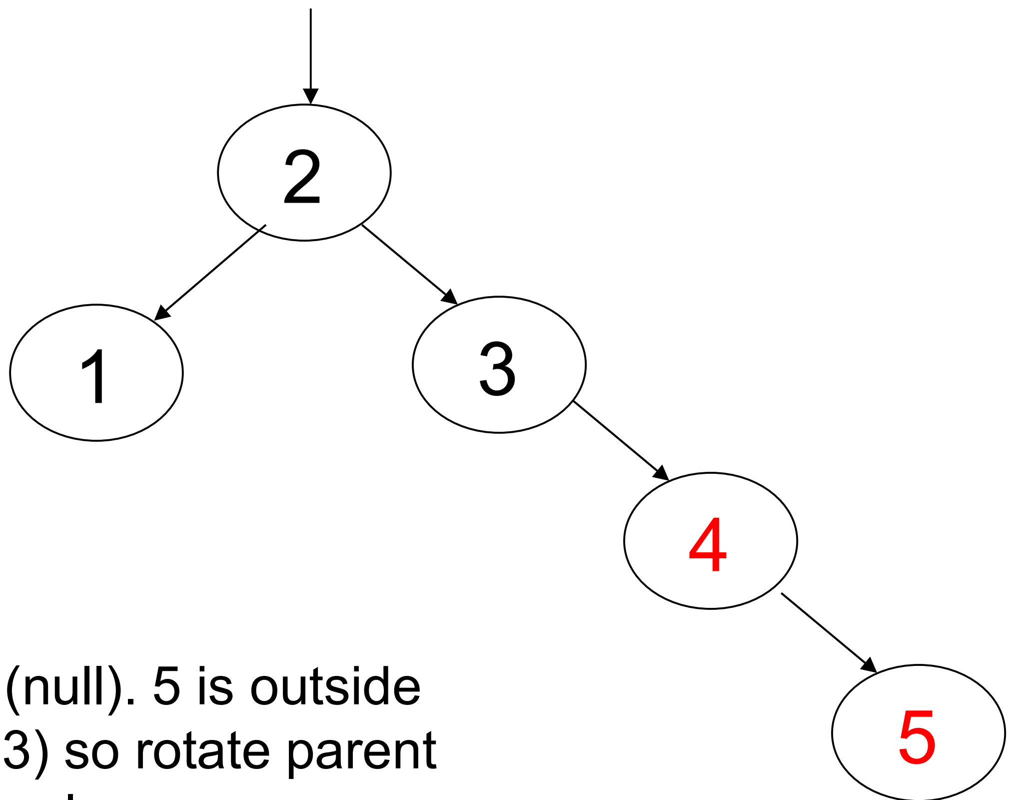
On way down see 2 with 2 red children.
Recolor 2 red and children black.
Realize 2 is root so color back to black



When adding 4 parent is black so done.

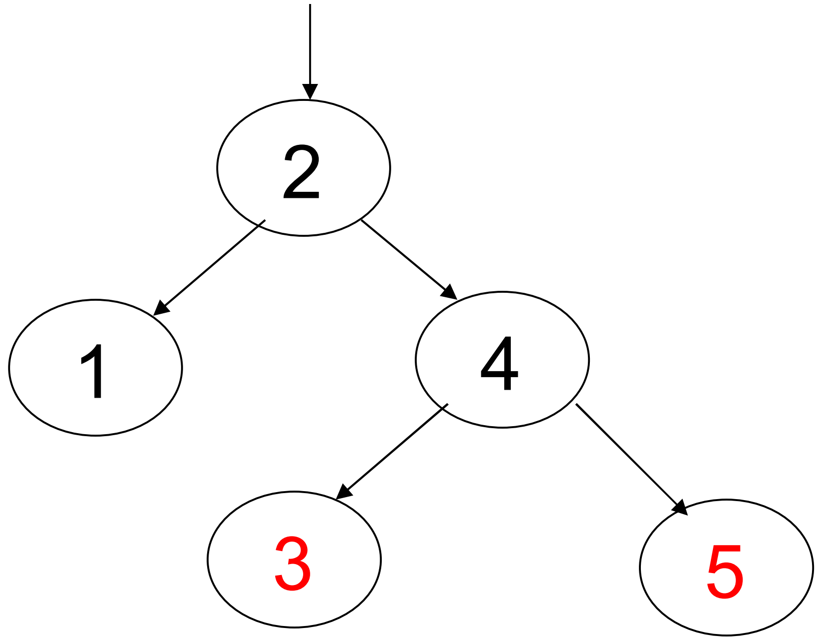


Insert 5

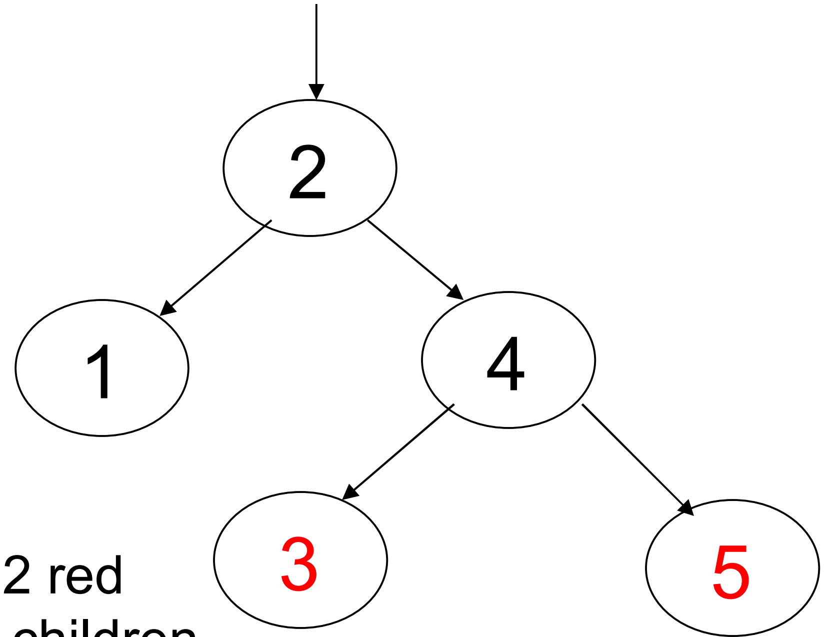


5's parent is red.
Parent's sibling is black (null). 5 is outside relative to grandparent (3) so rotate parent and grandparent then recolor

Finish insert of 5



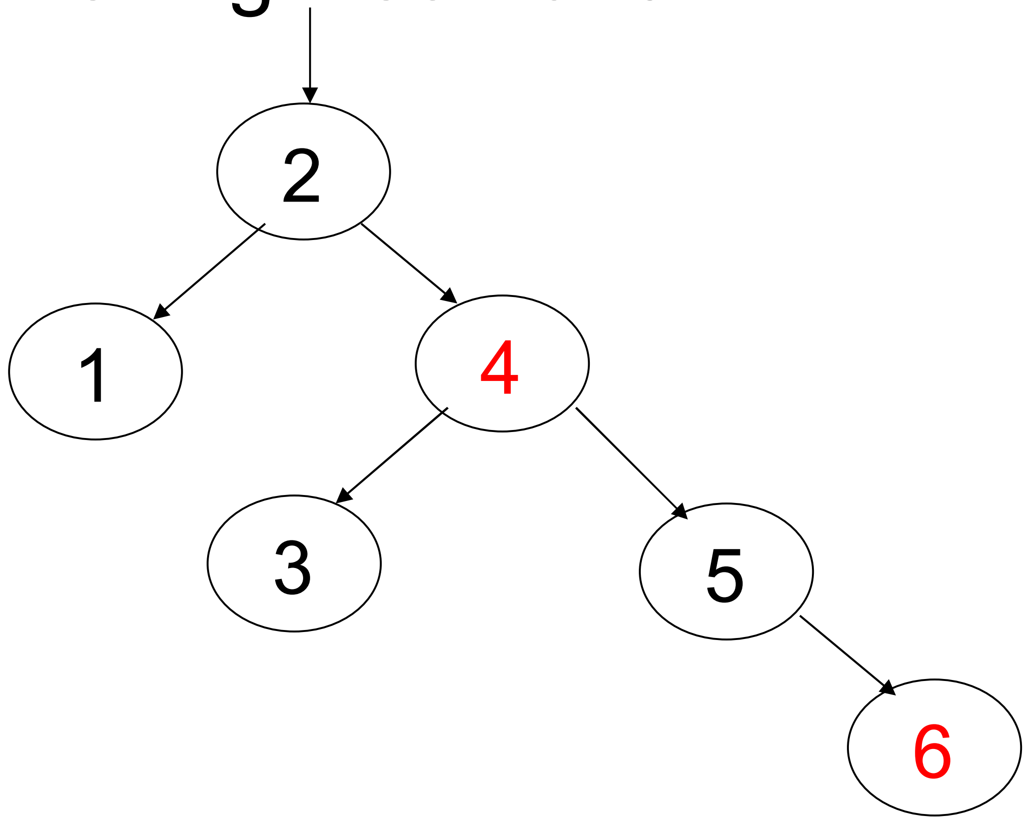
Insert 6



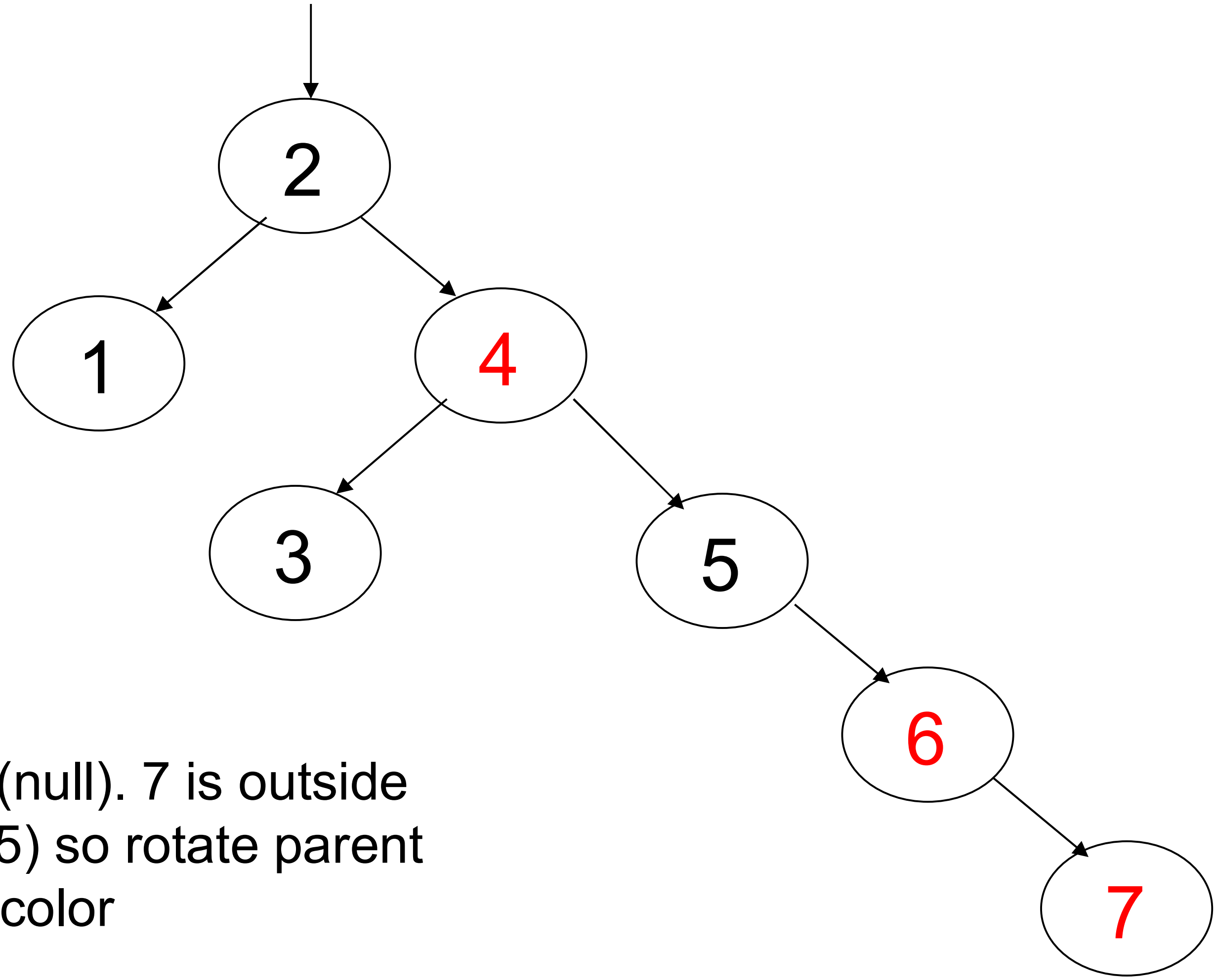
On way down see 4 with 2 red children. Make 4 red and children black. 4's parent is black so no problem.

Finishing insert of 6

6's parent is black so done.

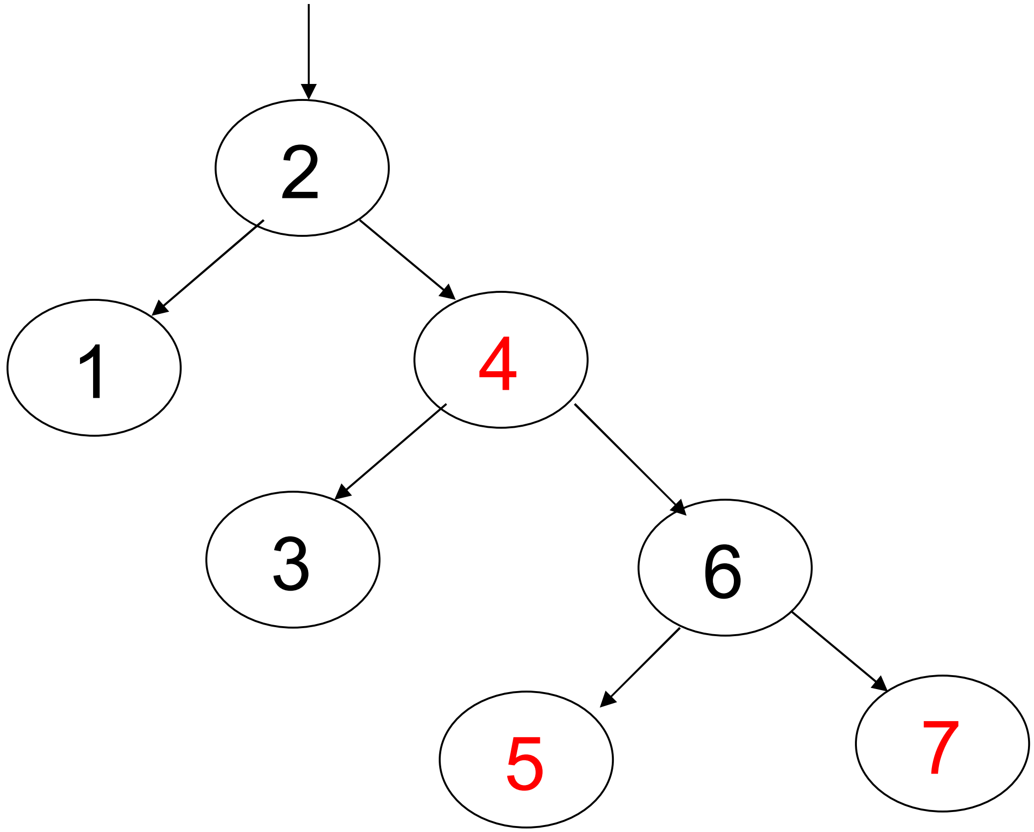


Insert 7

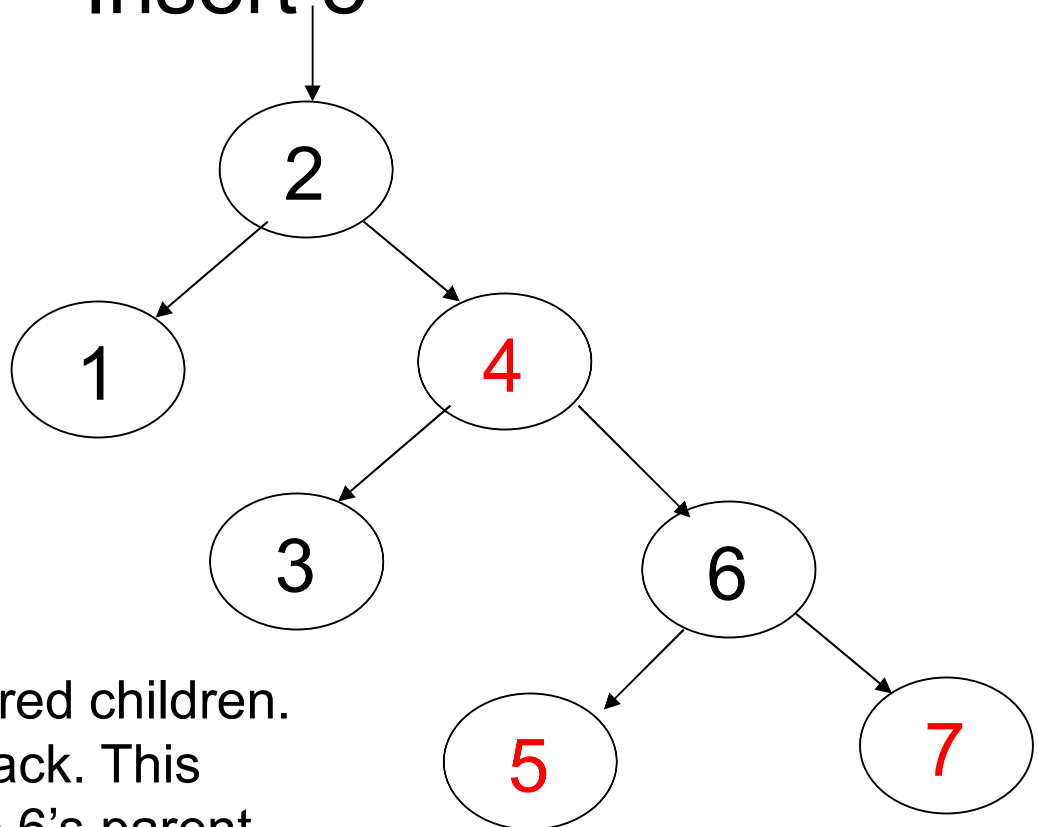


7's parent is red.
Parent's sibling is black (null). 7 is outside relative to grandparent (5) so rotate parent and grandparent then recolor

Finish insert of 7



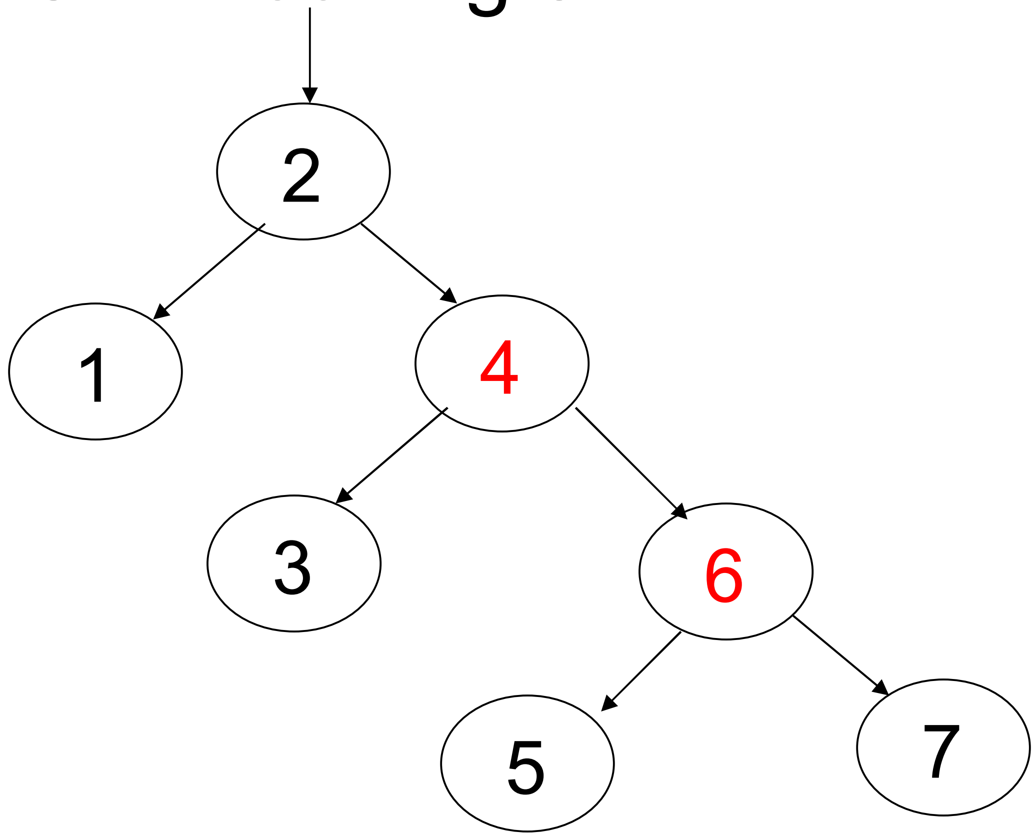
Insert 8



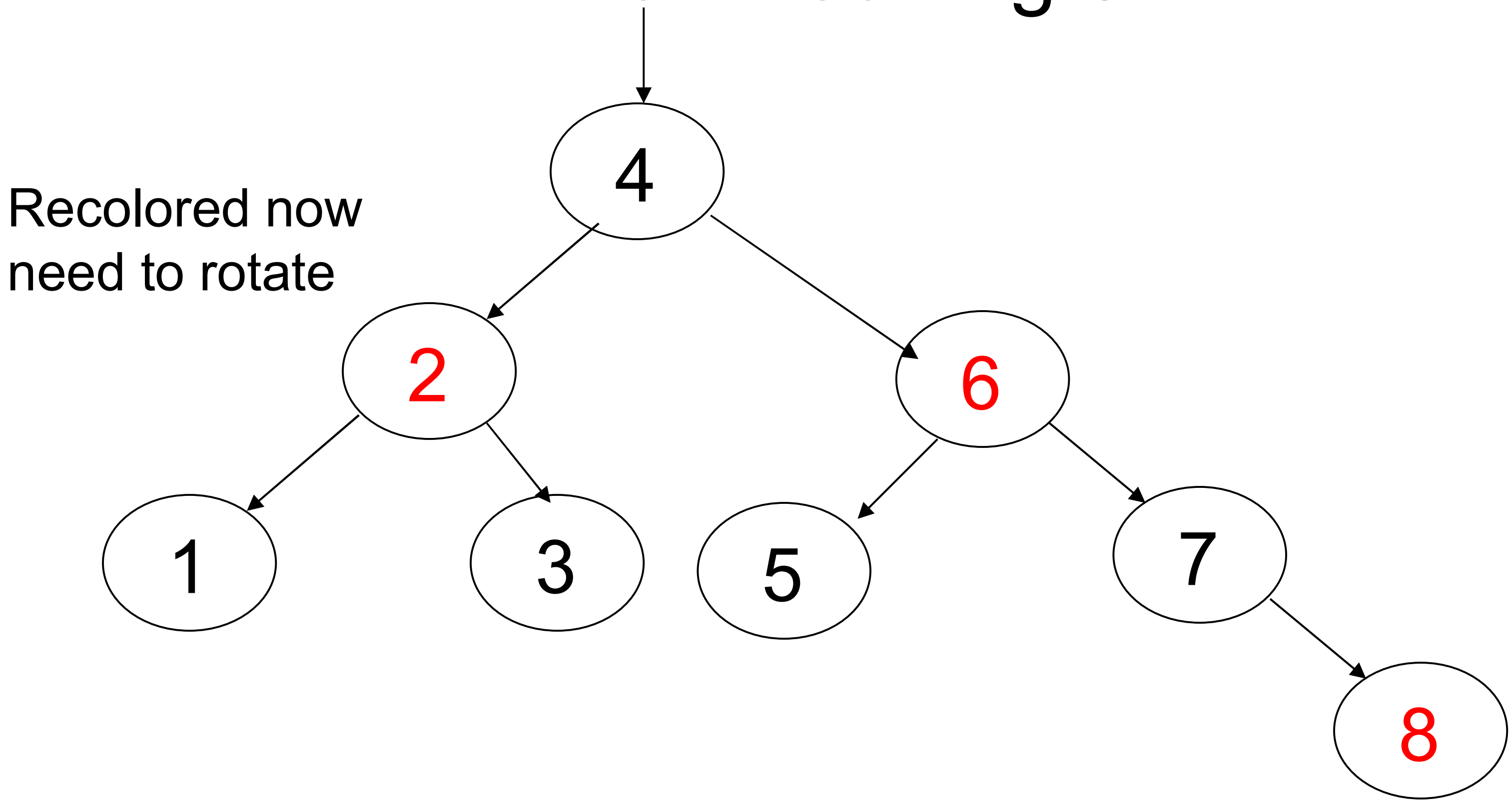
On way down see 6 with 2 red children. Make 6 red and children black. This creates a problem because 6's parent, 4, is also red. Must perform rotation.

Still Inserting 8

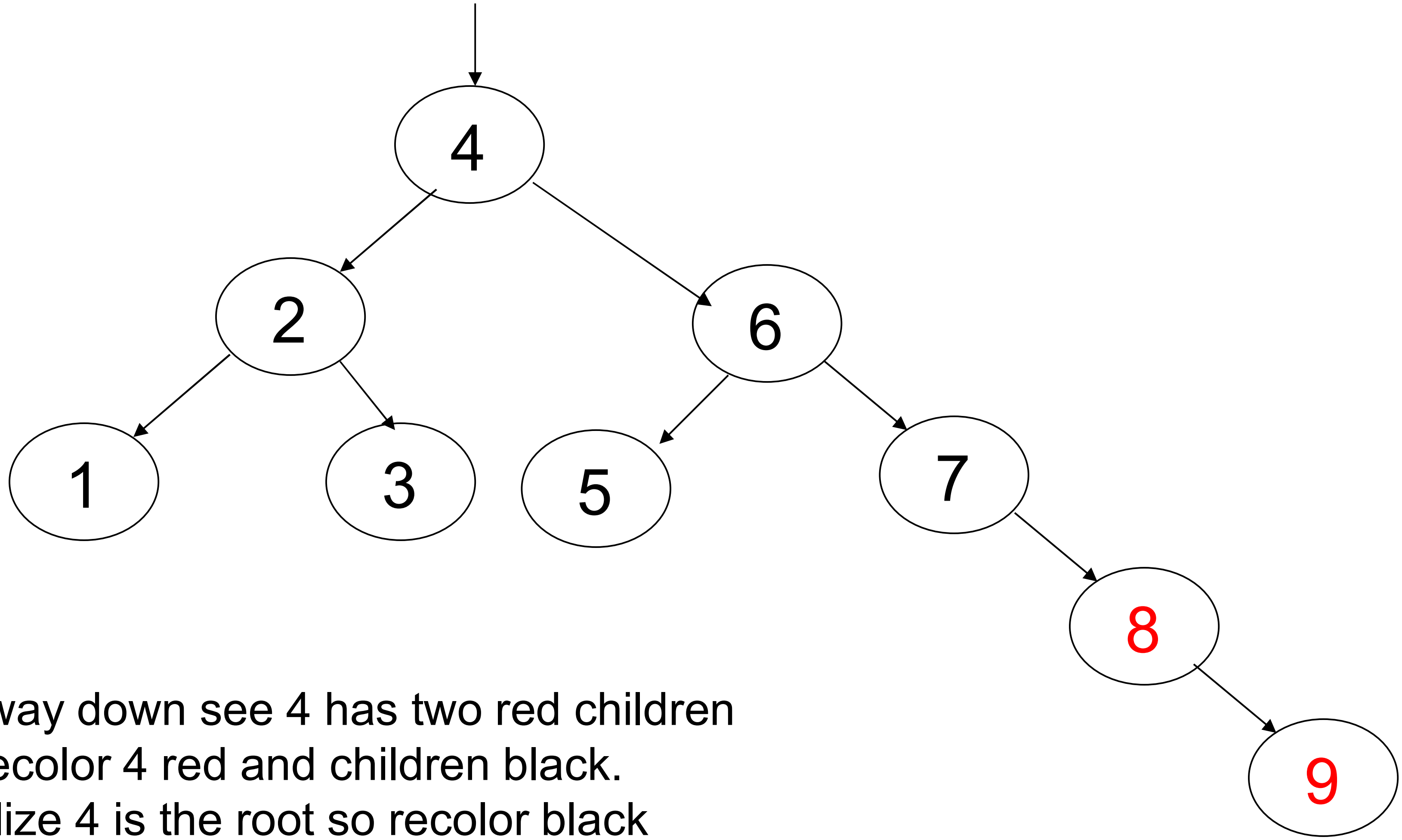
Recolored now
need to rotate



Finish inserting 8

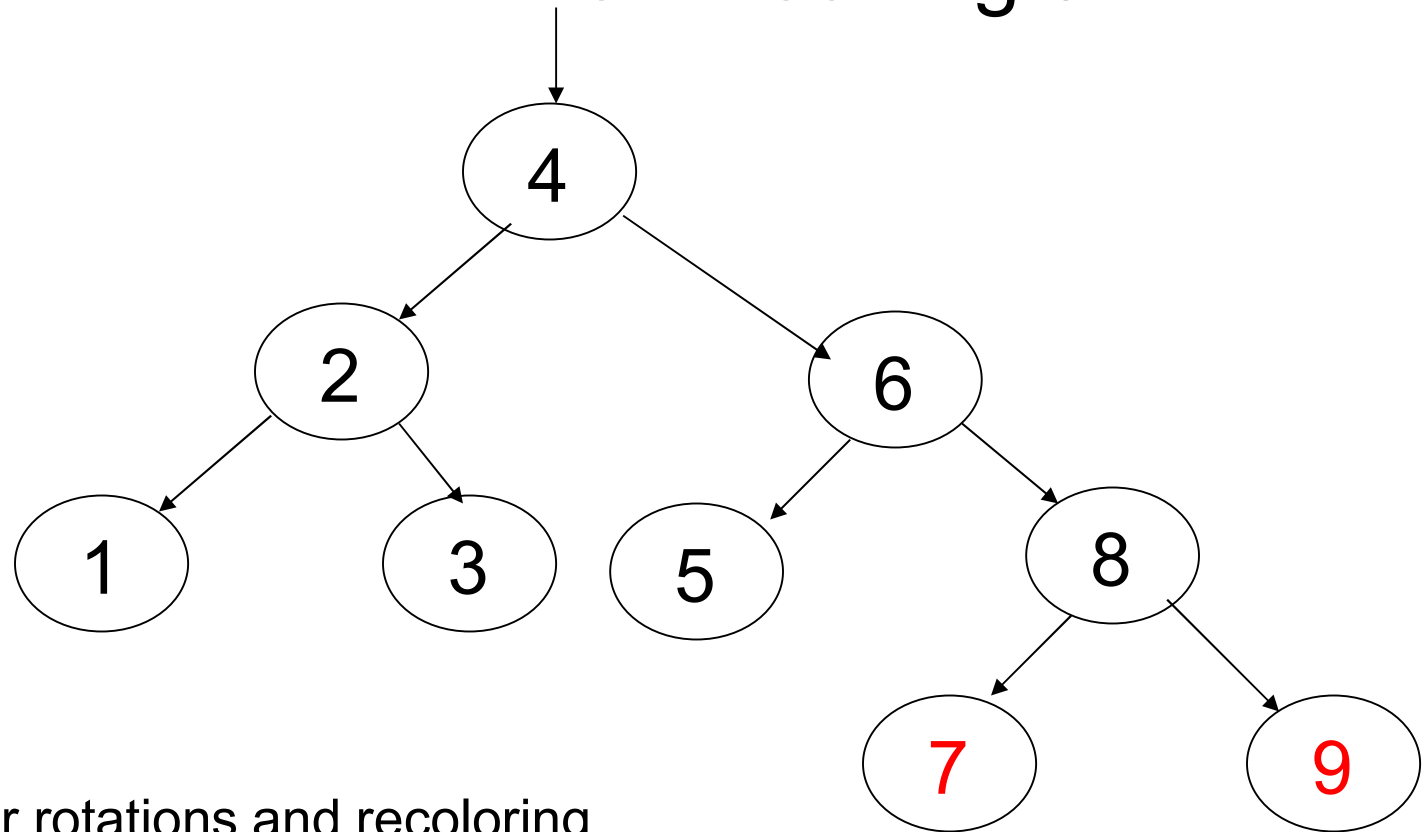


Insert 9



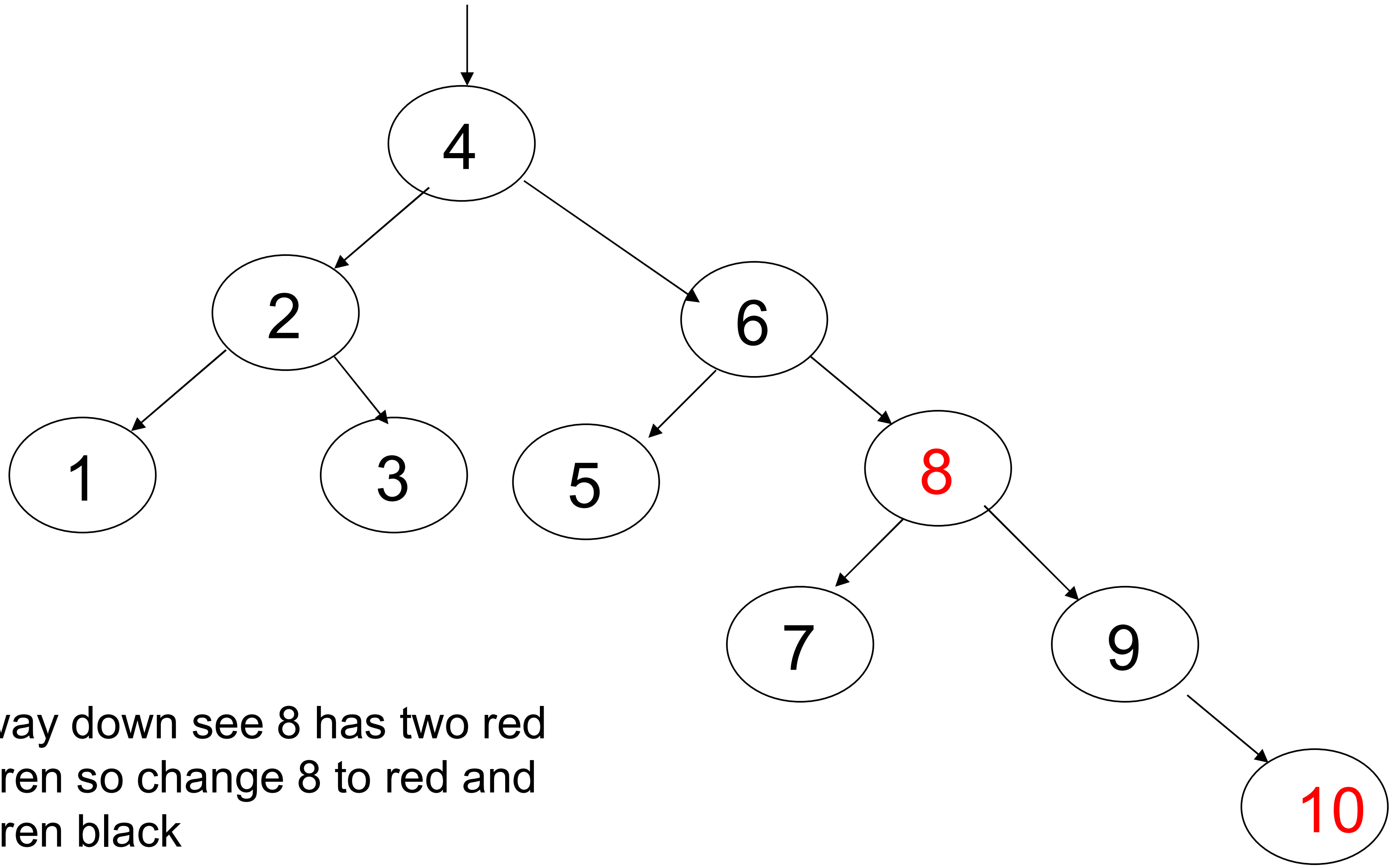
On way down see 4 has two red children
so recolor 4 red and children black.
Realize 4 is the root so recolor black

Finish Inserting 9



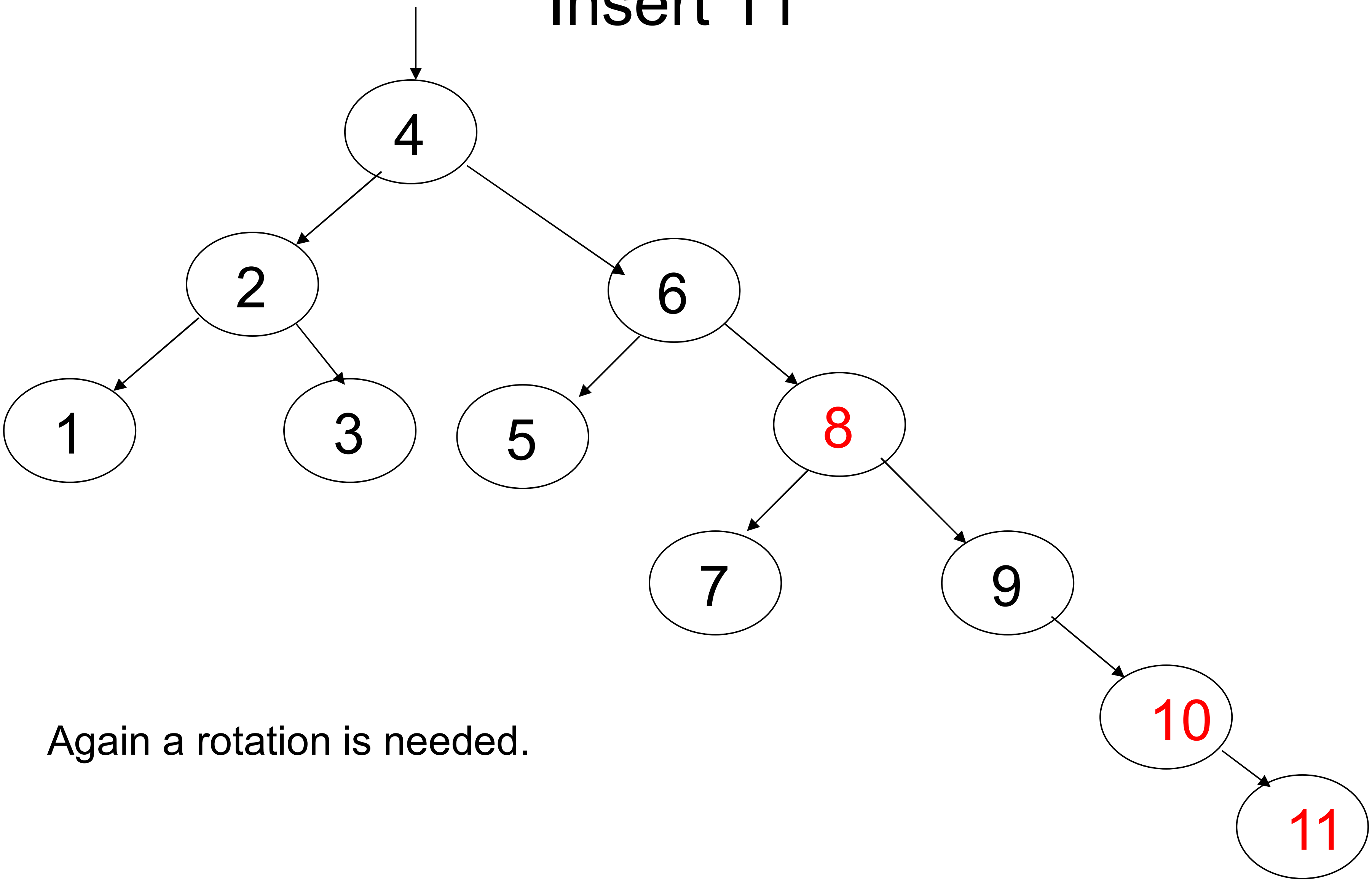
After rotations and recoloring

Insert 10



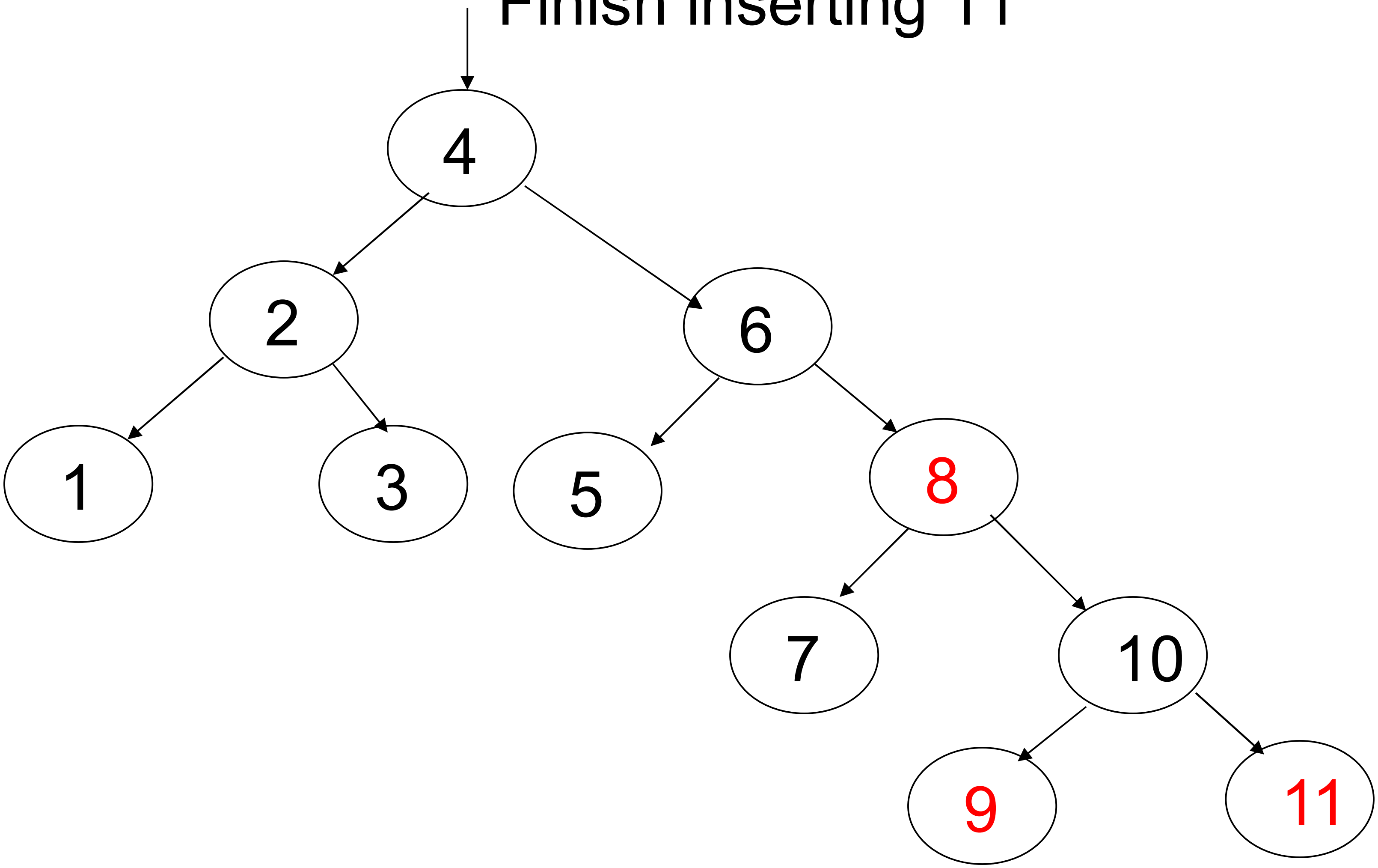
On way down see 8 has two red children so change 8 to red and children black

Insert 11



Again a rotation is needed.

Finish inserting 11



Analysis of Insertion

- **A** red-black tree has $O(\log n)$ height
- Search for insertion location takes $O(\log n)$ time because we visit $O(\log n)$ nodes
- Addition to the node takes $O(1)$ time
- Rotation or recoloring takes $O(\log n)$ time because we perform

* $O(\log n)$ recoloring, each taking $O(1)$ time, and
* at most one rotation taking $O(1)$ time

- Thus, an insertion in a red-black tree takes $O(\log n)$ time

Red-Black Trees

In this topic, we have covered red-black trees

- simple rules govern how nodes must be distributed based on giving each node a colour of either red or black
- insertions and deletions may be performed without recursing back to the root
- only one bit is required for the "colour"
- this makes them, under some circumstances, more suited than AVL trees