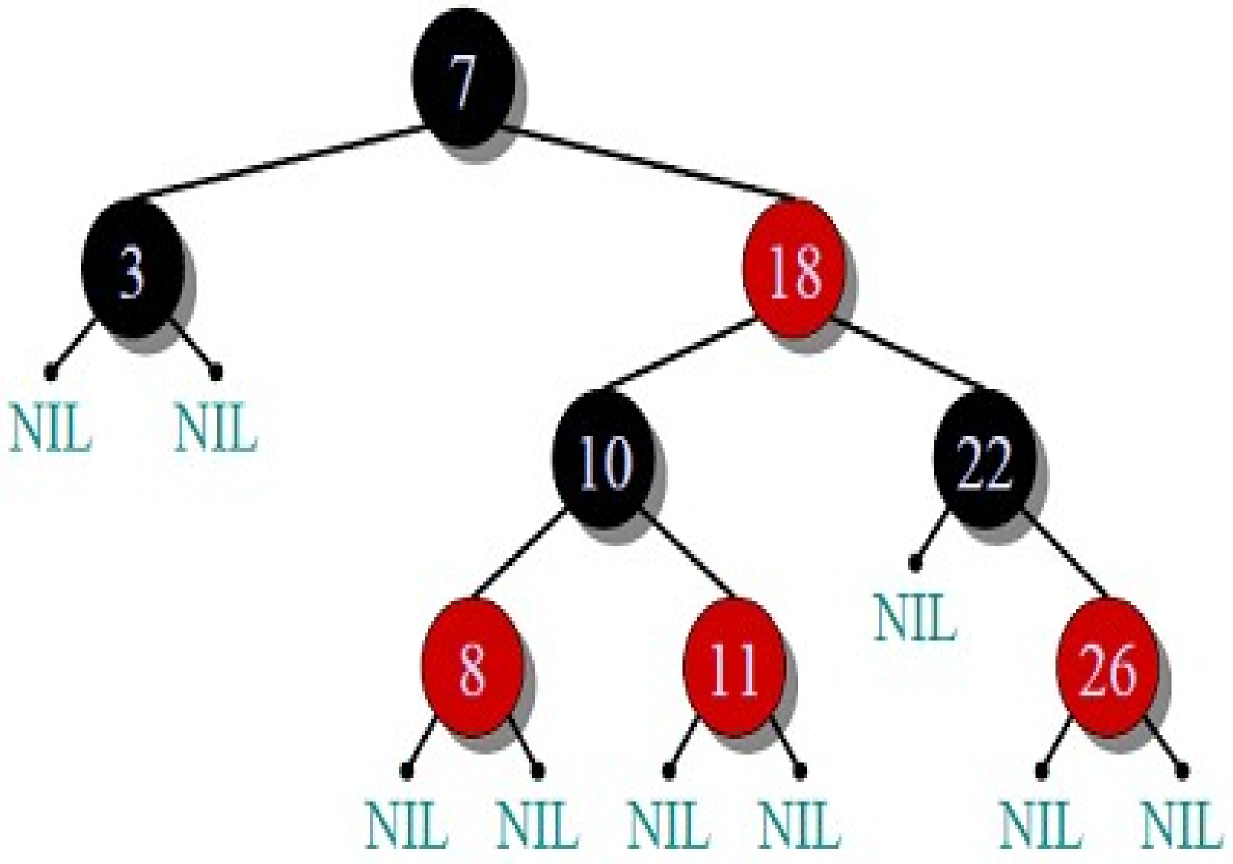
Red Black Trees

Kumkum Saxena

Red-Black Tree

- Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.
  - **1)** Every node has a color either red or black.
  - **2)** Root of tree is always black.
  - **3)** There are no two adjacent red nodes (A red node cannot have a red parent or red child).
  - **4)** Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes.


7
3
18
NIL
NIL
10
22
NIL
8
11
26
NIL
NIL
NIL
NIL
NIL
NIL

Why Red-Black Trees?

- Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST.
- The cost of these operations may become O(n) for a skewed Binary tree.
- If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of a Red-Black tree is always O(Logn) where n is the number of nodes in the tree.

Comparison with AVL Tree

- The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion.
- So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred.
- And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

How does a Red-Black Tree ensure balance?

- A simple example to understand balancing is, a chain of 3 nodes is not possible in the Red-Black tree. We can try any combination of colours and see all of them violate Red-Black tree property.

```
A chain of 3 nodes is nodes is not possible in Red-Black Trees.
Following are NOT Red-Black Trees
        30             30              30
       / \            /  \            /  \
     20  NIL        20   NIL        20   NIL
    / \            / \             /  \
  10  NIL        10  NIL         10  NIL
Violates         Violates        Violates
Property 4.     Property 4      Property 3

Following are different possible Red-Black Trees with above 3 keys
         20                           20
       /   \                        /   \
     10     30                    10     30
    /  \   /  \                  /  \   /  \
 NIL  NIL NIL NIL             NIL  NIL NIL NIL
```

Black Height of a Red-Black Tree

- *Black height is number of black nodes on a path from root to a leaf.*
- *Leaf nodes are also counted black nodes.*
- *From above properties 3 and 4, we can derive, **a Red-Black Tree of height h has black-height >= h/2**.*
- ***Every Red Black Tree with n nodes has height <=** $2\text{Log}_2(n+1)$*

- This can be proved using following facts:
  - For a general Binary Tree, let **k** be the minimum number of nodes on all root to NULL paths, then $n >= 2^k - 1$ (Ex. If k is 3, then n is atleast 7). This expression can also be written as $k <= Log_2(n+1)$
  - From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most $Log_2(n+1)$ black nodes.
  - From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least $\lfloor n/2 \rfloor$ where n is the total number of nodes.
- From above 2 points, we can conclude the fact that Red Black Tree with **n** nodes has height $<= 2Log_2(n+1)$

Red-Black Tree(Insert)

- In AVL tree insertion, we used rotation as a tool to do balancing after insertion caused imbalance. In Red-Black tree, we use two tools to do balancing.
  - Recoloring
  - Rotation

  We try recoloring first, if recoloring doesn’t work, then we go for rotation

- The algorithms has mainly two cases depending upon the color of uncle.
- If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.
- Color of a NULL node is considered as BLACK.

Let x be the newly inserted node.

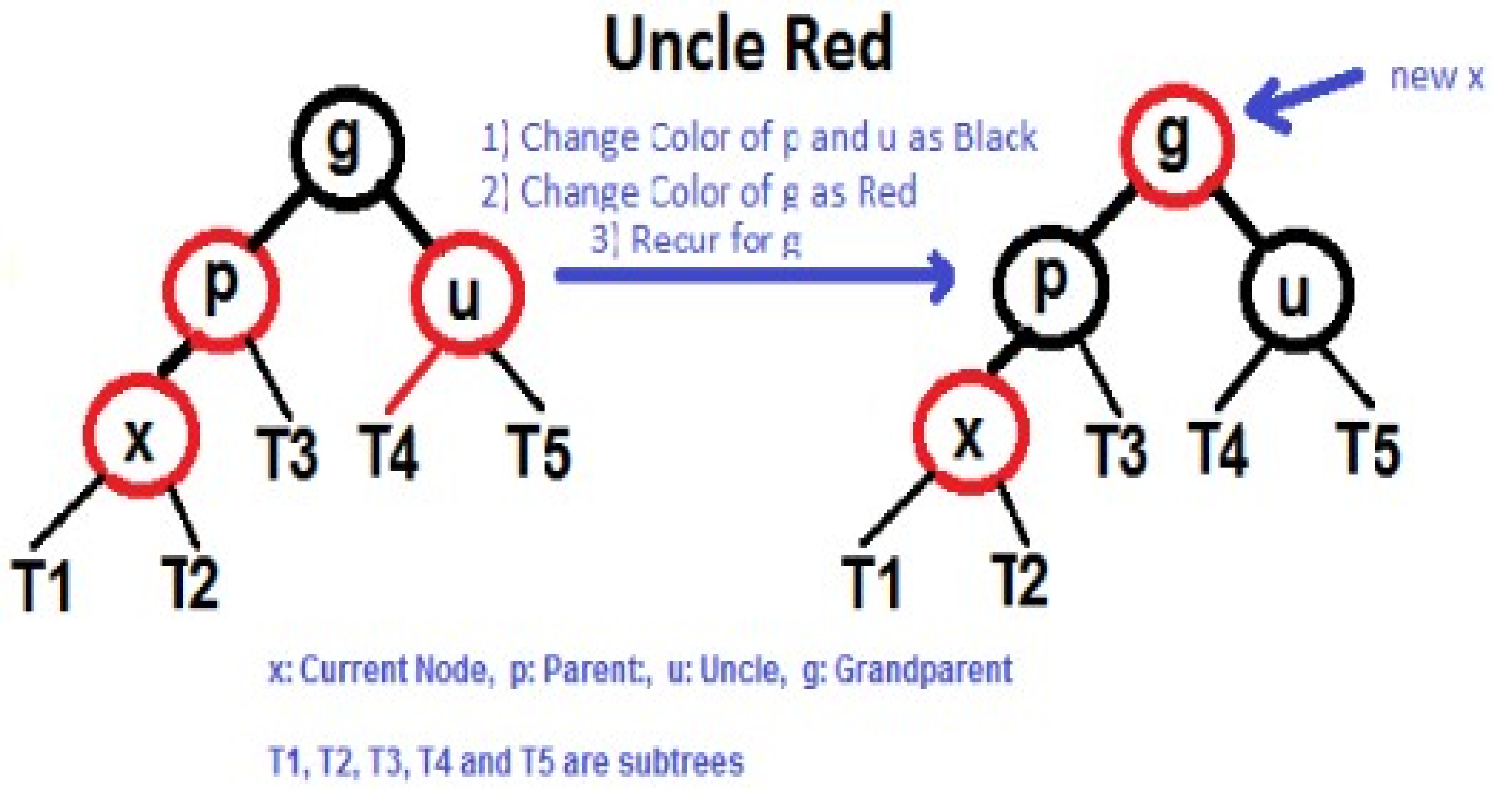
1. Perform standard BST insertion and make the color of newly inserted nodes as RED.
2. If x is root, change color of x as BLACK (Black height of complete tree increases by 1).
3. Do following if color of x's parent is not BLACK **and** x is not root.
    1. **a) If x's uncle is RED** (Grand parent must have been black from property 4)
    2. .**b) If x's uncle is BLACK**,

a) If x’s uncle is RED (Grand parent must have been black from property 4)

(i) Change color of parent and uncle as BLACK.

(ii) color of grand parent as RED.

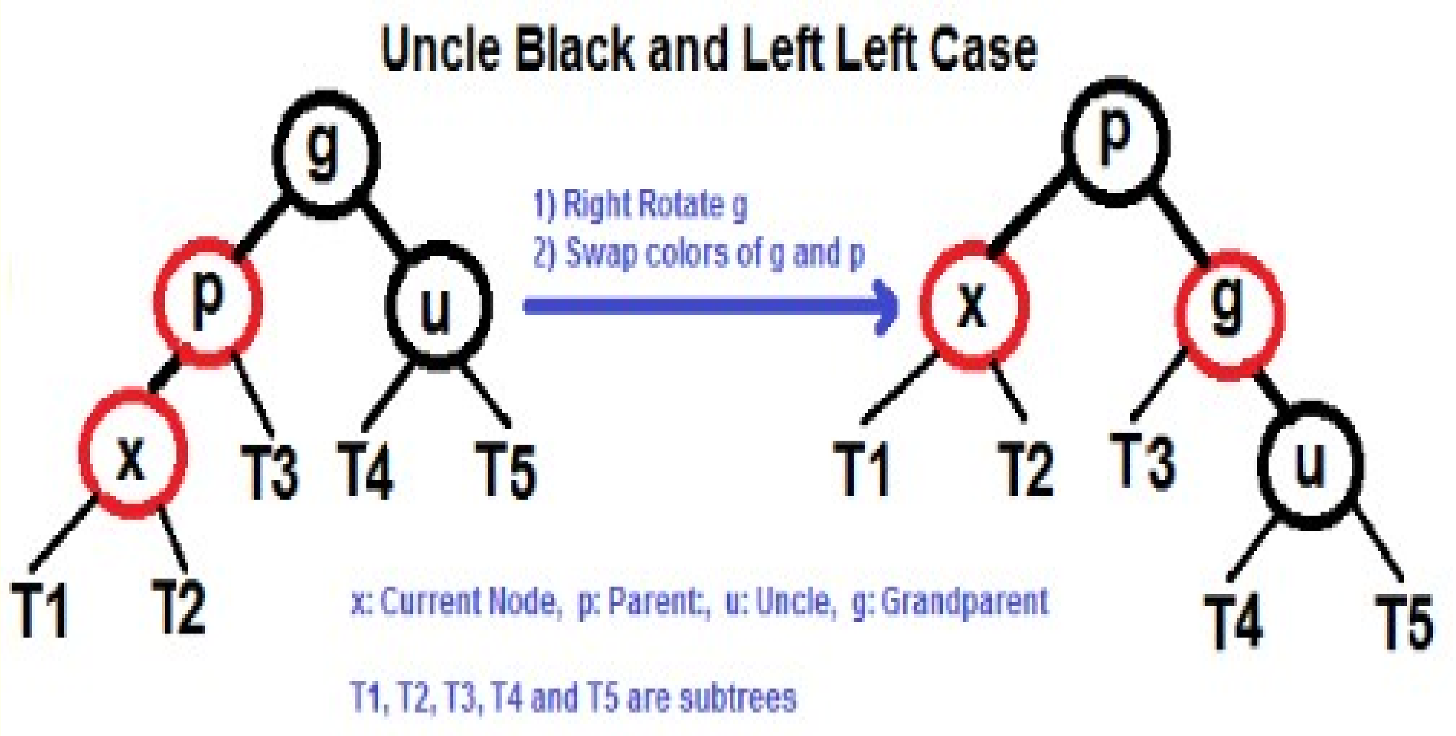
(iii) Change x = x’s grandparent, repeat steps 2 and 3 for new x.



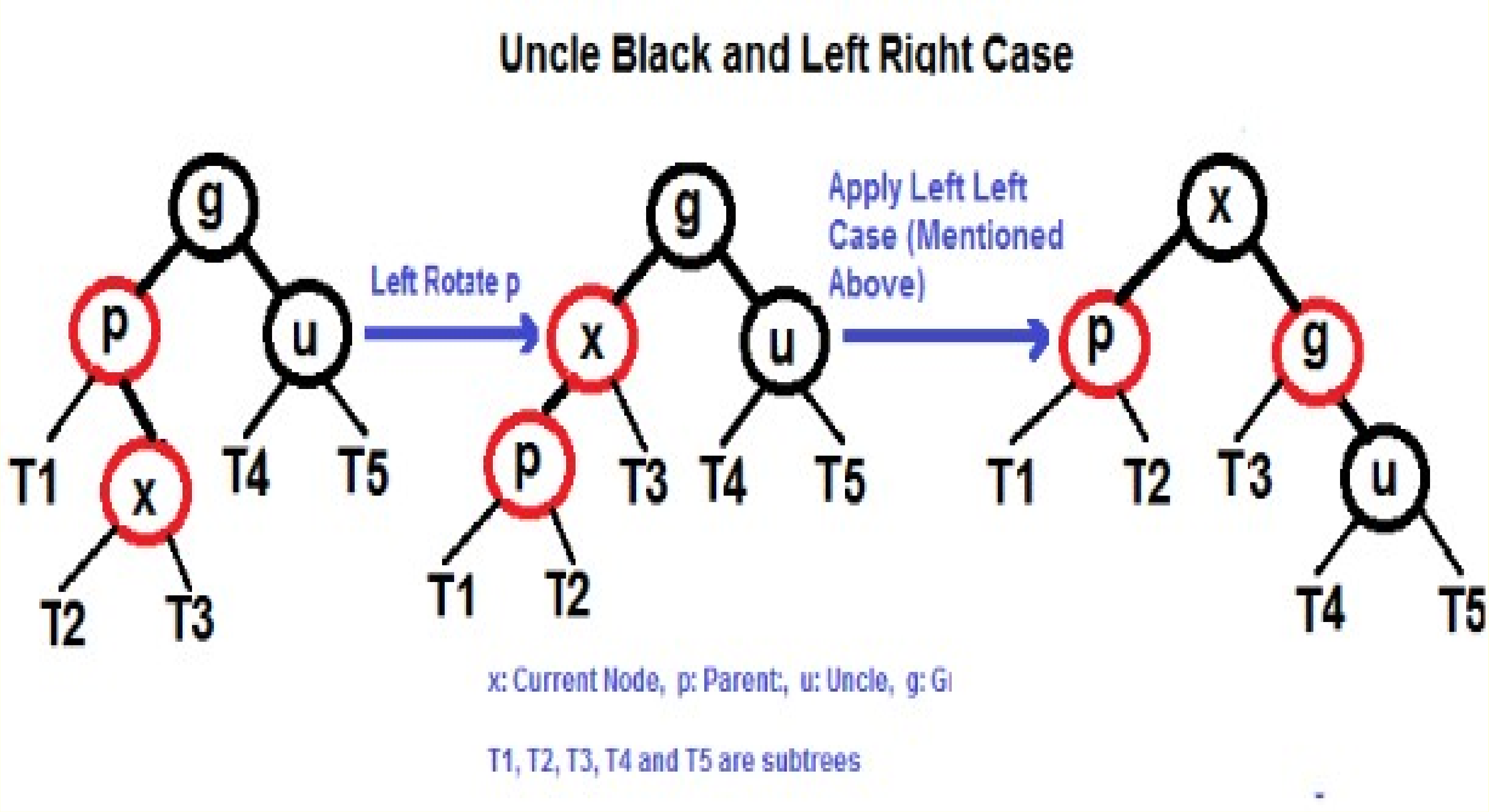
Uncle Red
new x
1) Change Color of p and u as Black
2) Change Color of g as Red
3) Recur for g
g
p
u
x
T1
T2
T3
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

- **b) If x's uncle is BLACK**, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**) (This is similar to AVL Tree)
  i) Left Left Case (p is left child of g and x is left child of p)
  ii) Left Right Case (p is left child of g and x is right child of p)
  iii) Right Right Case (Mirror of case i)
  iv) Right Left Case (Mirror of case ii)

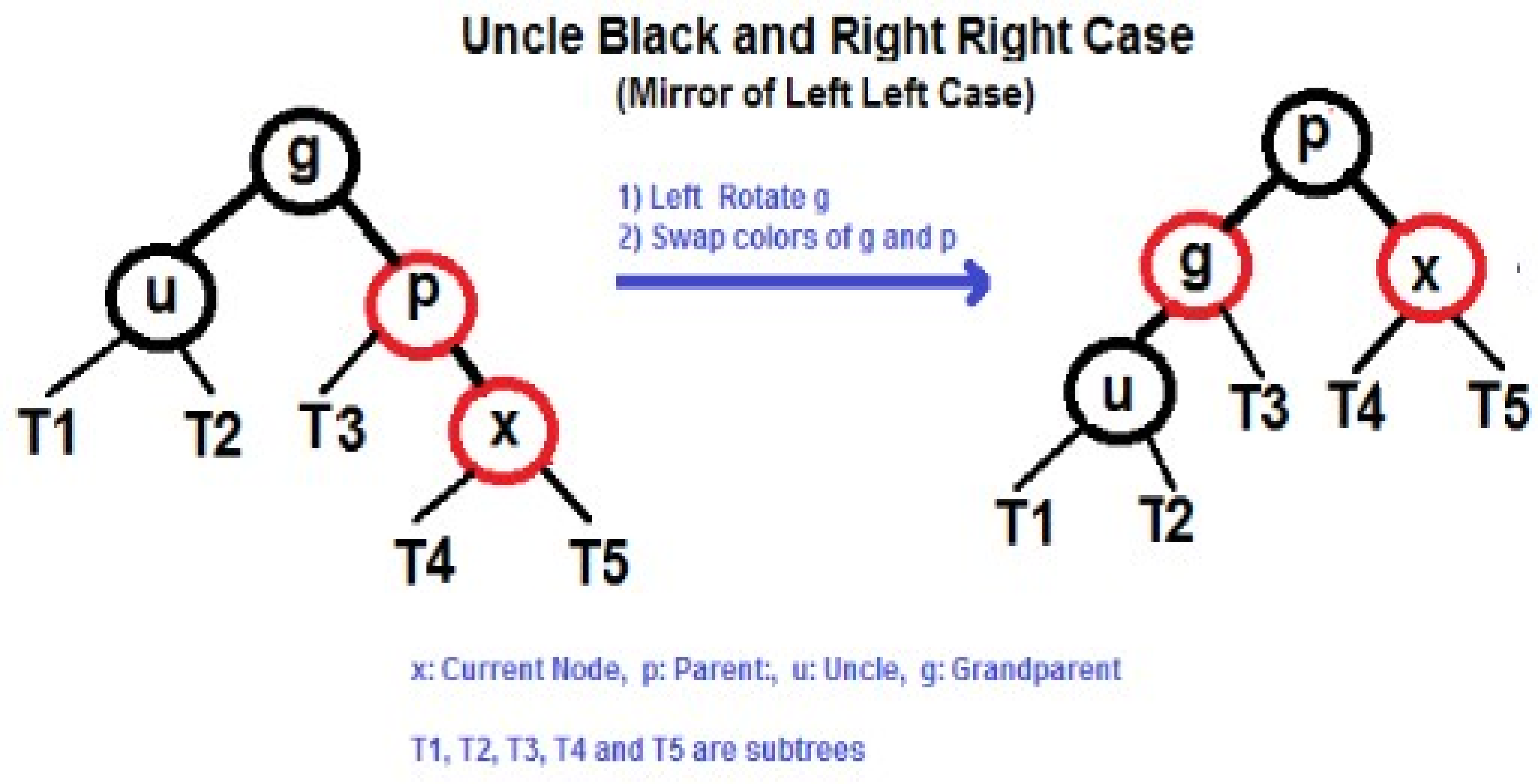
Left Left Case (See g, p and x)



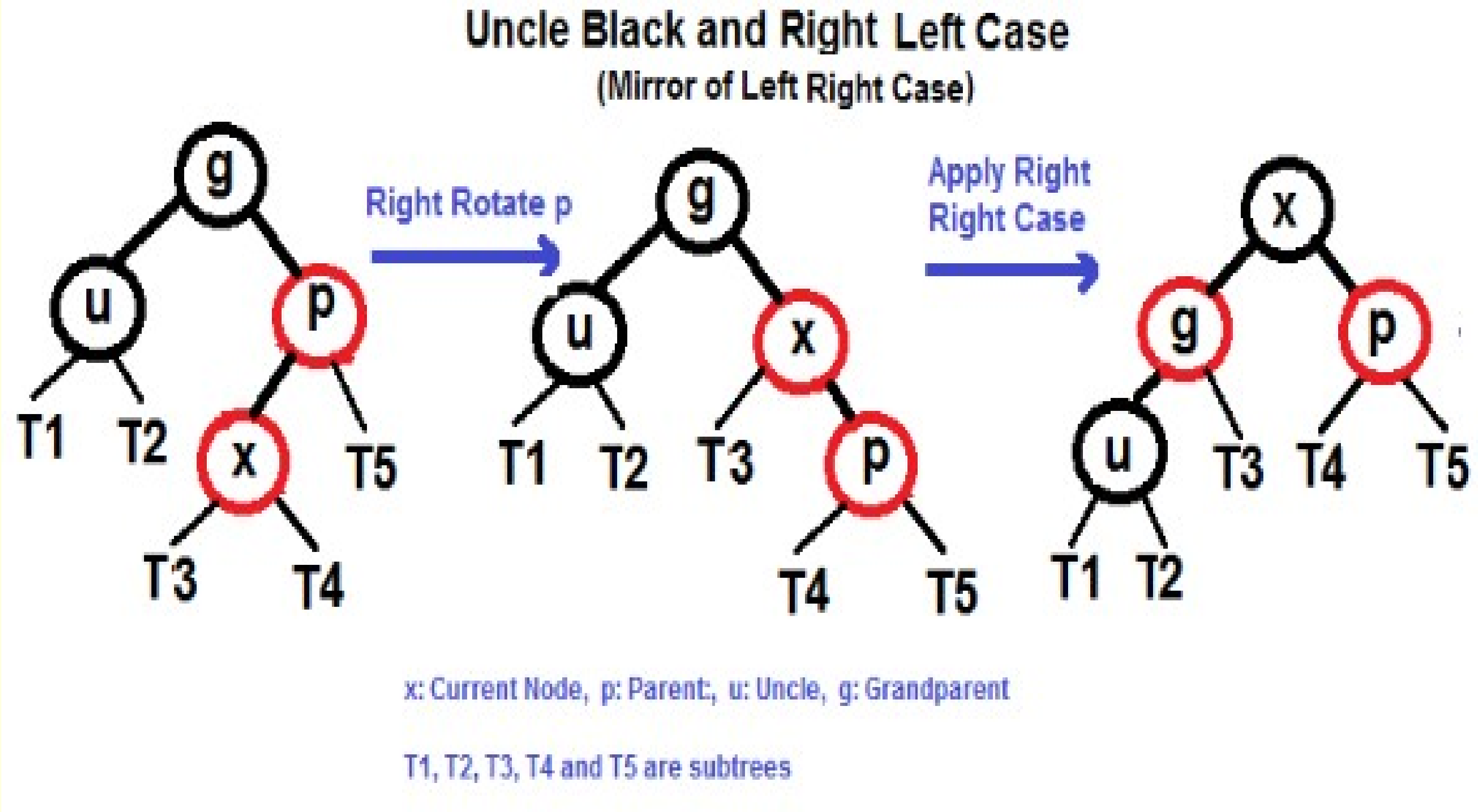
Left Right Case (See g, p and x)



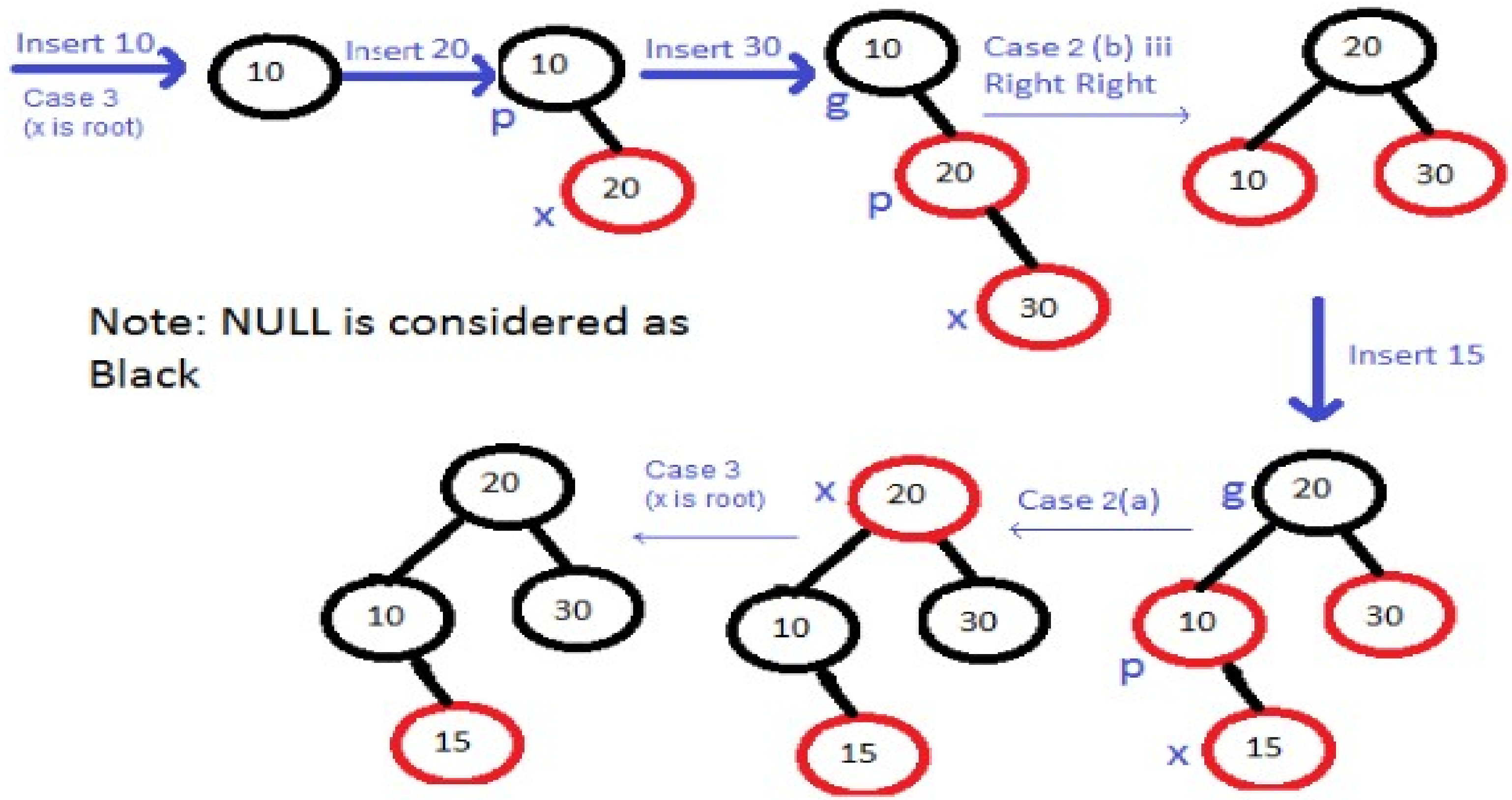
Right Right Case (See g, p and x)



Right Left Case (See g, p and x)



Insert 10, 20, 30 and 15 in an empty tree



Note: NULL is considered as Black

Algorithm Analysis

- $O(\lg n)$ time to get through RB-Insert up to the call of RB-Insert-Fixup.
- Within RB-Insert-Fixup:
  - Each iteration takes $O(1)$ time.
  - Each iteration but the last moves z up 2 levels.
  - $O(\lg n)$ levels $\Rightarrow O(\lg n)$ time.
  - Thus, insertion in a red-black tree takes $O(\lg n)$ time.
  - Note: there are at most 2 rotations overall.

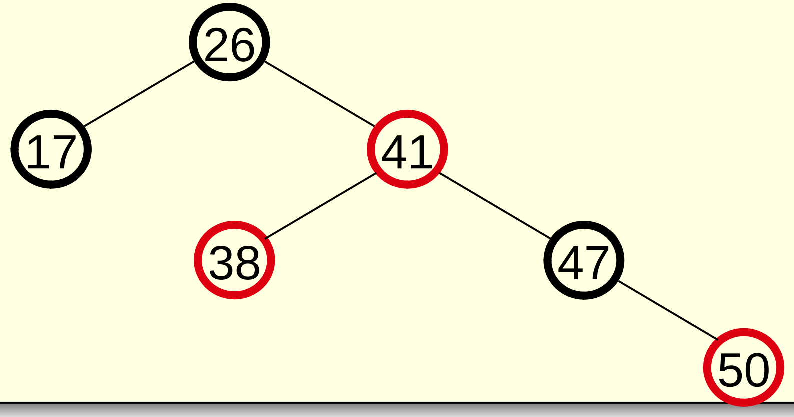
RB Properties Affected by Insert

1. Every node is either **red** or **black** OK!
2. The root is **black** If z is the root ⇒ not OK
3. Every leaf (NIL) is **black** OK!
4. If a node is red, then both its children are black

If p(z) is red ⇒ not OK
z and p(z) are both red

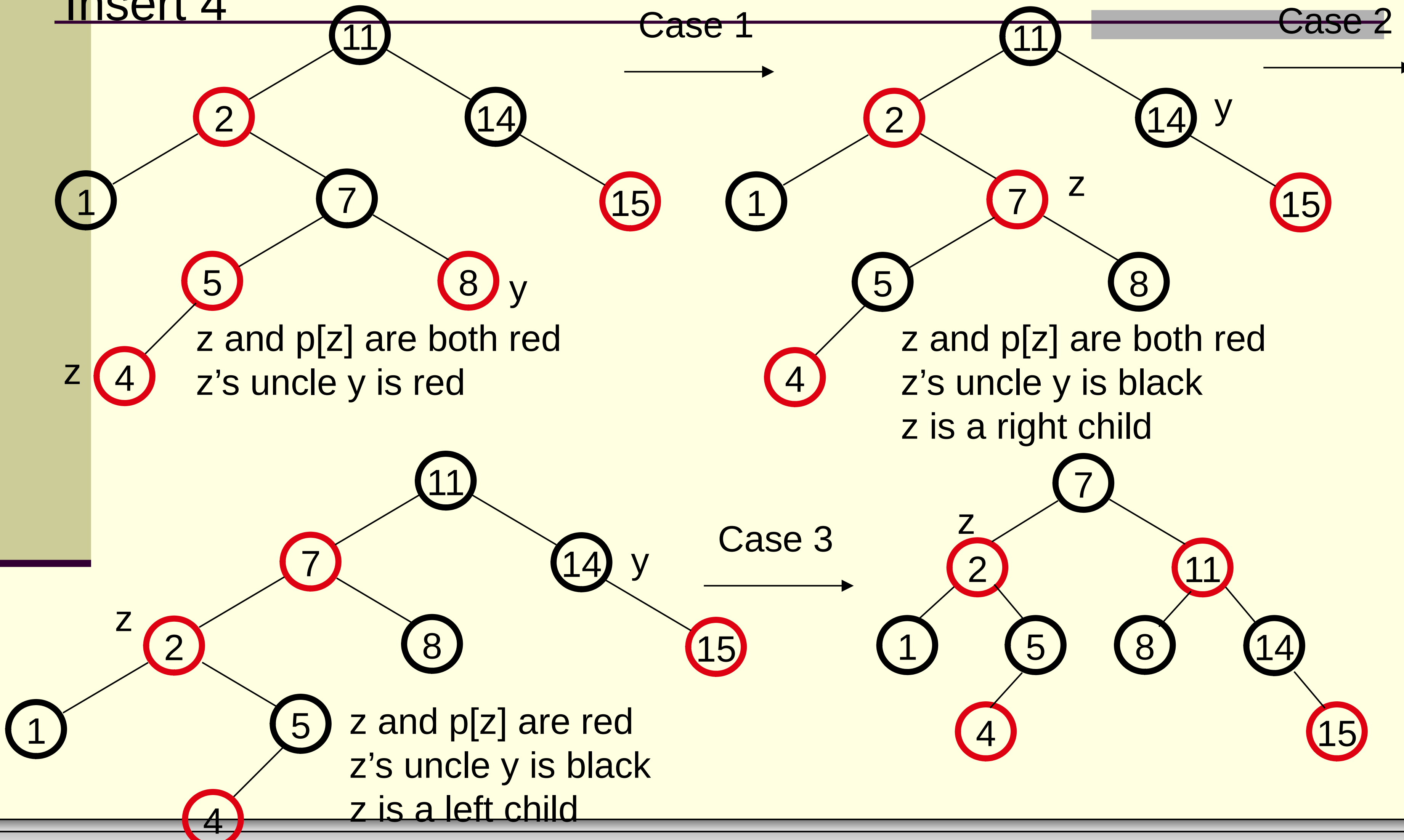
OK!

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes



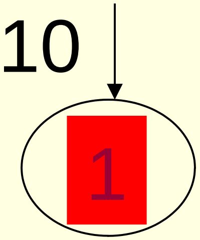
Example

Insert 4

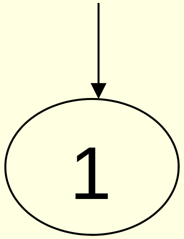


Example of Inserting Sorted Numbers

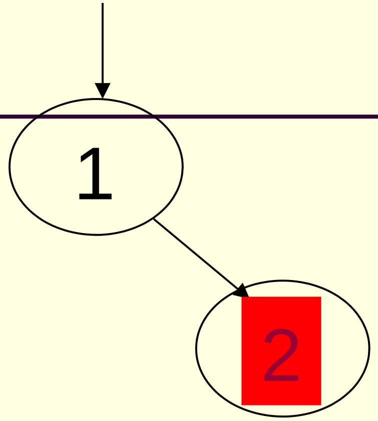
- 1 2 3 4 5 6 7 8 9 10



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



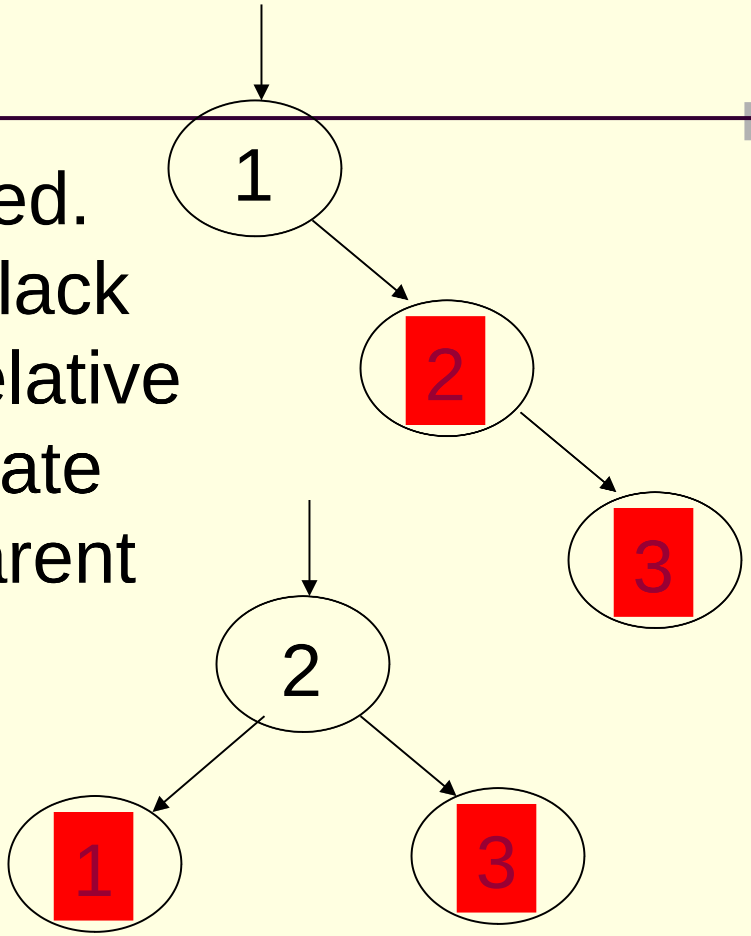
Insert 2



make 2 red. Parent is black so done.

Insert 3

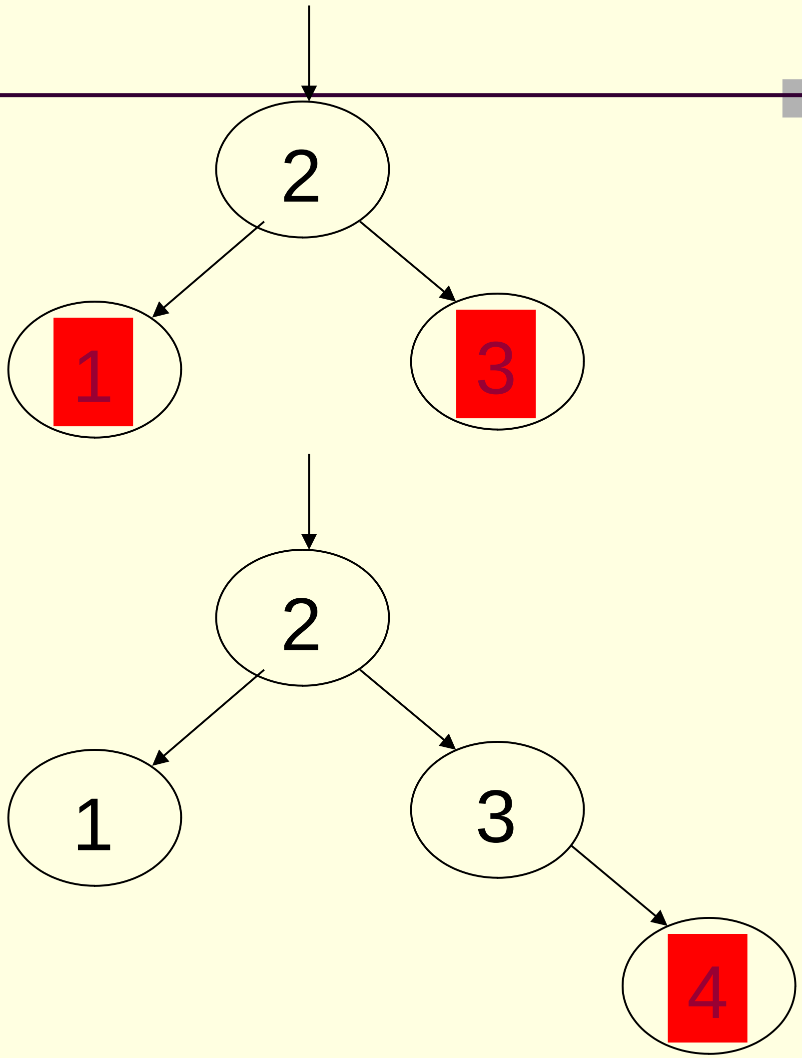
Insert 3. Parent is red. Parent's sibling is black (null) 3 is outside relative to grandparent. Rotate parent and grandparent



Insert 4

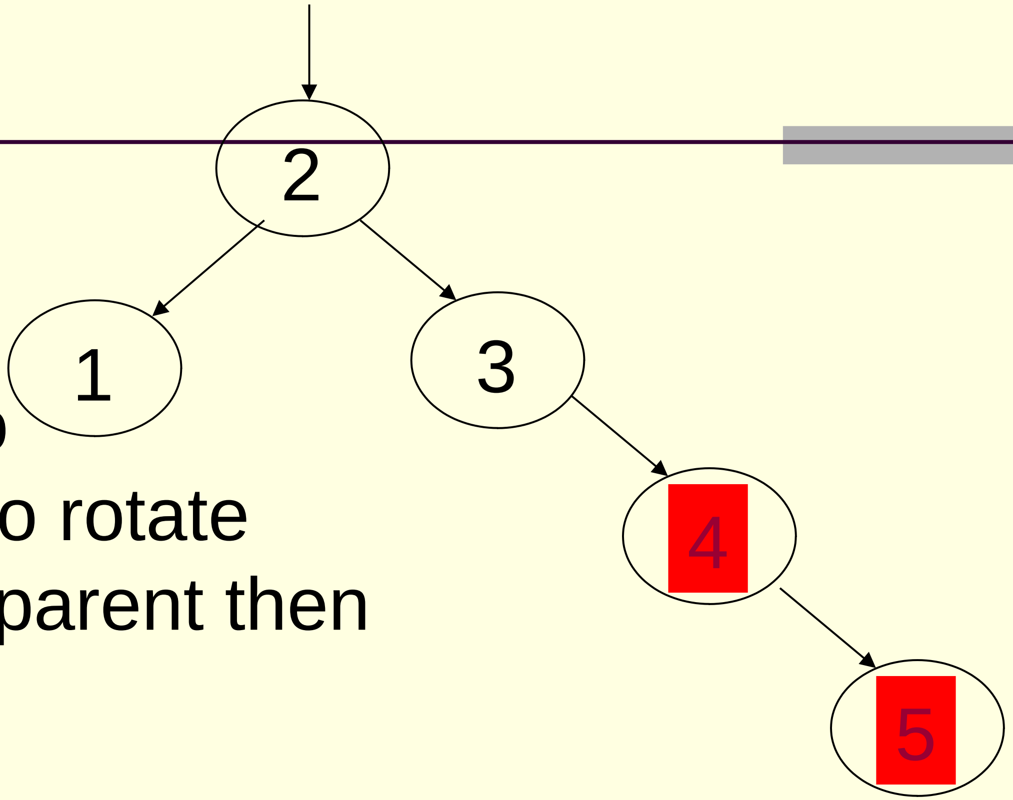
On way down see 2 with 2 red children.
Recolor 2 red and children black.
Realize 2 is root so color back to black

When adding 4 parent is black so done.

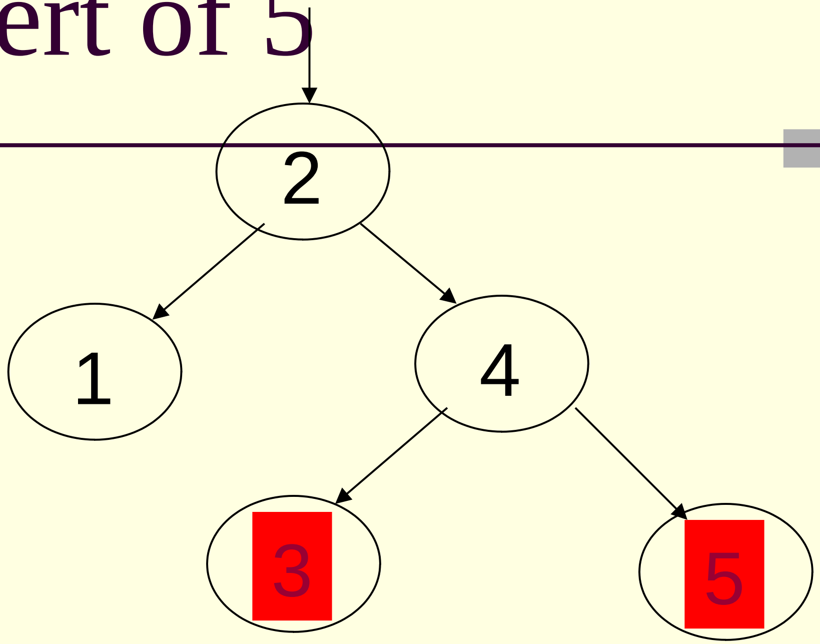


Insert 5

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

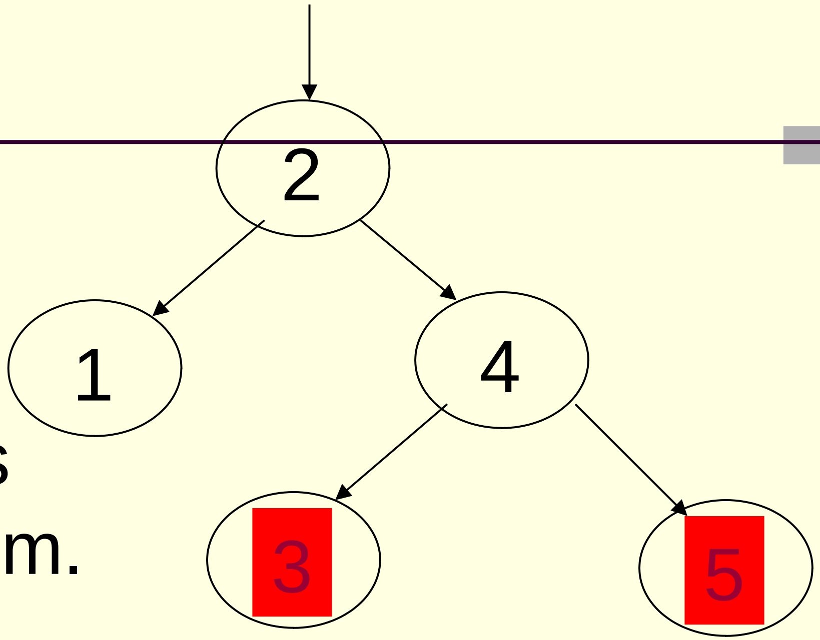


Finish insert of 5



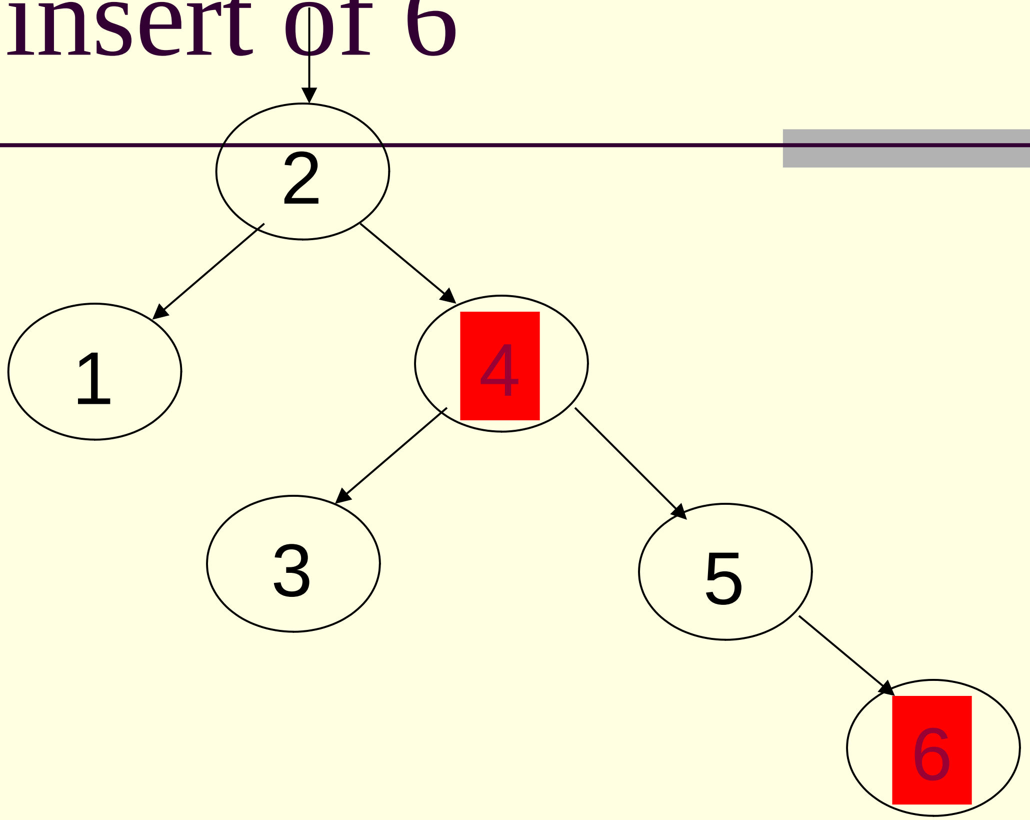
Insert 6

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



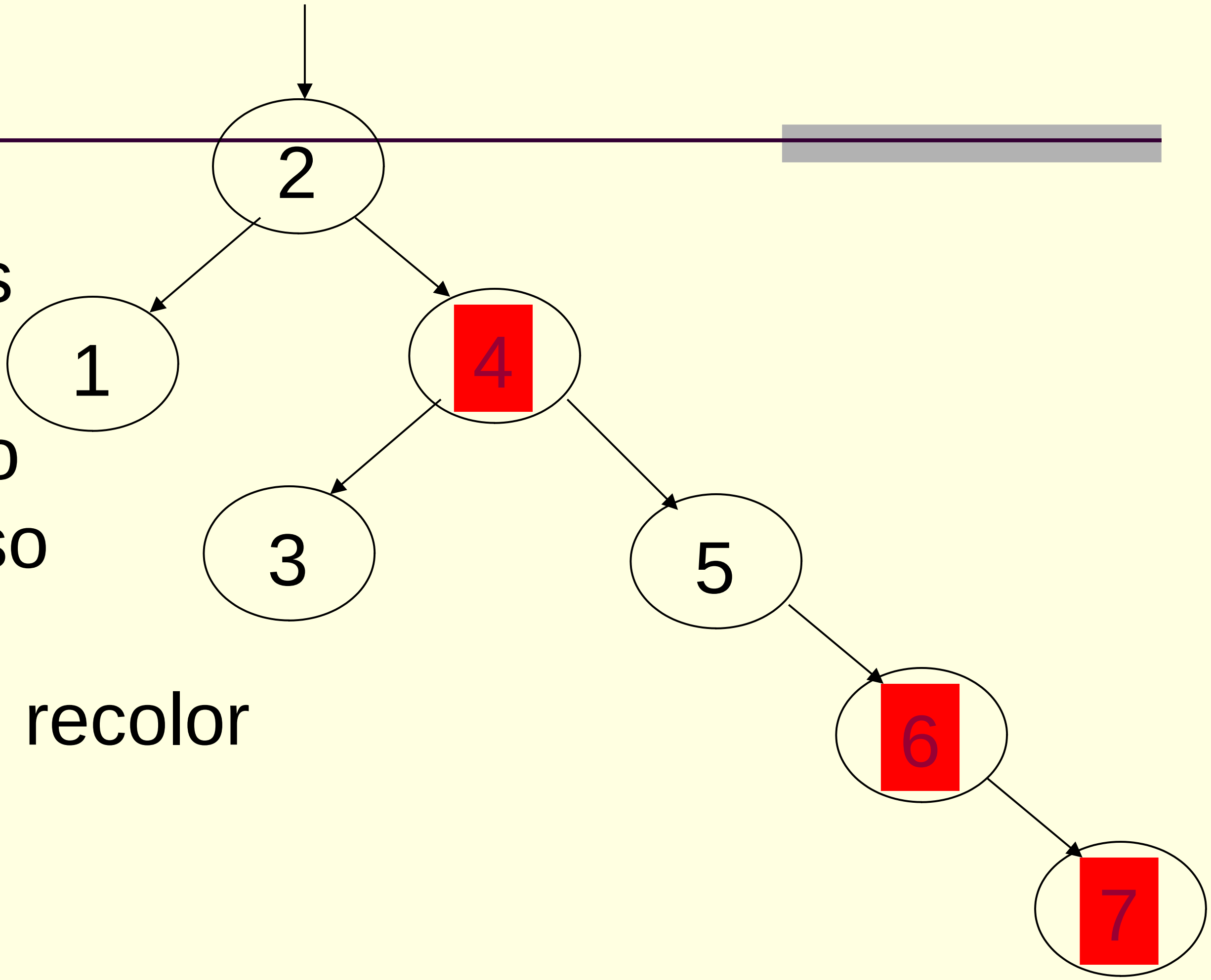
Finishing insert of 6

6's parent is black so done.

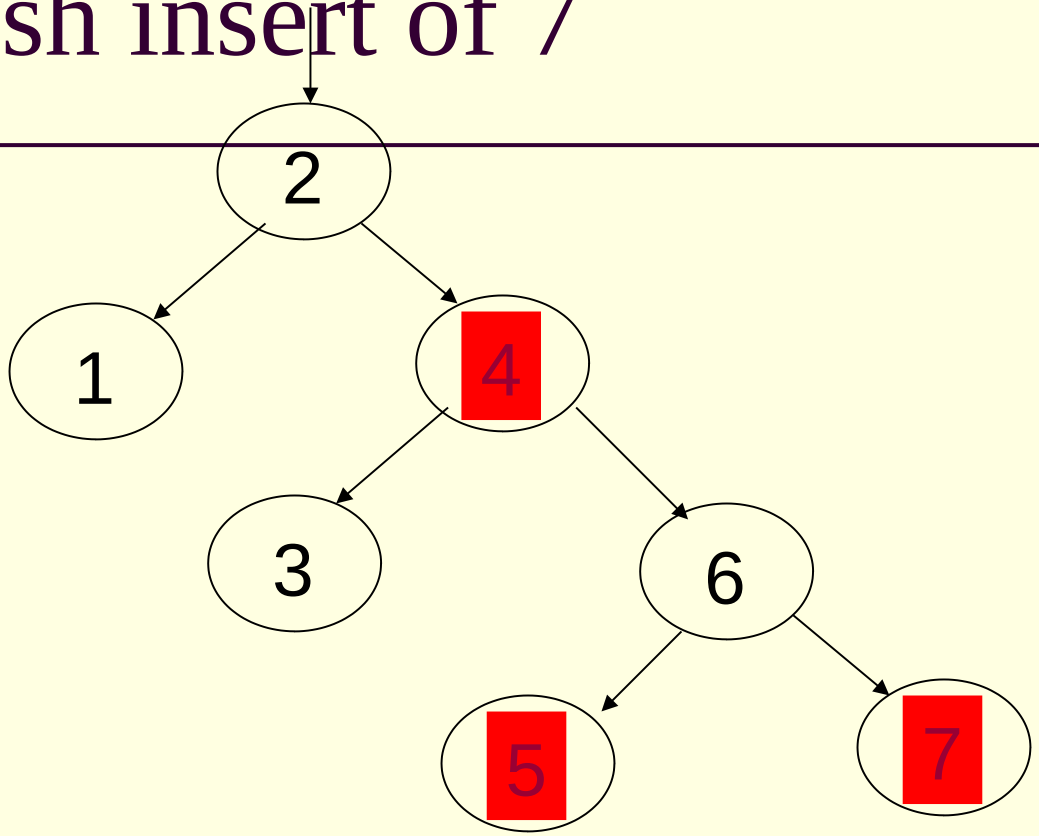


Insert 7

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

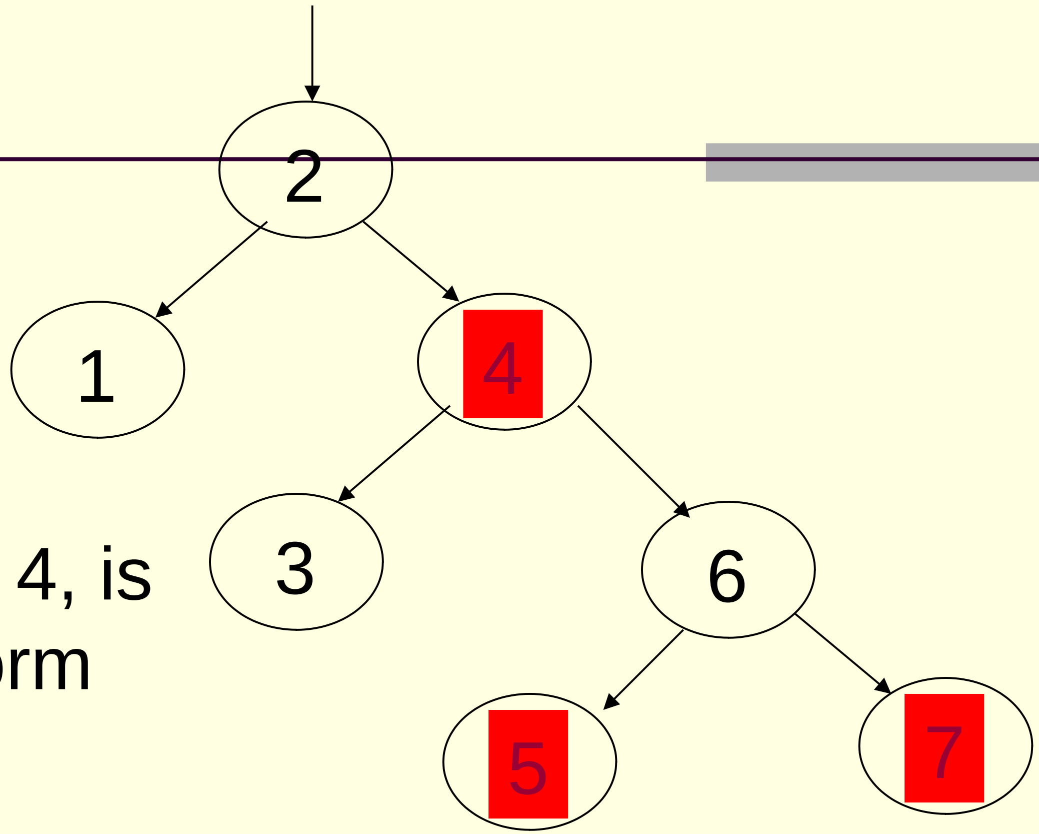


Finish insert of 7



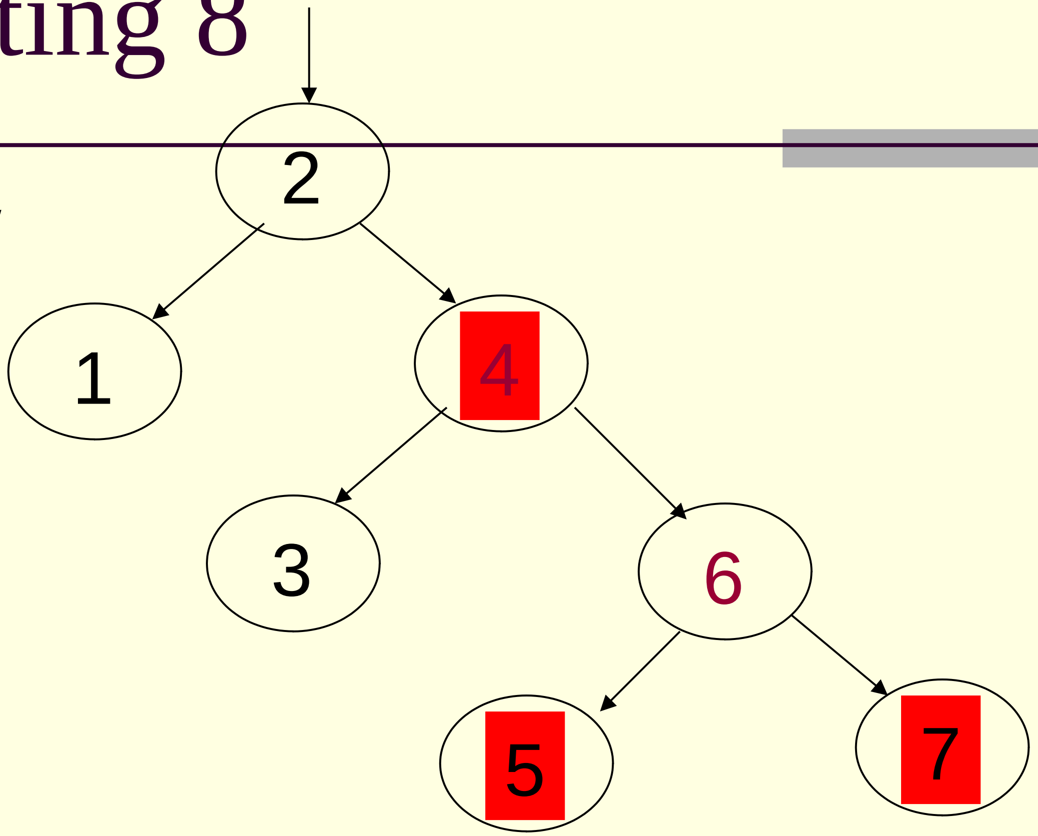
Insert 8

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.



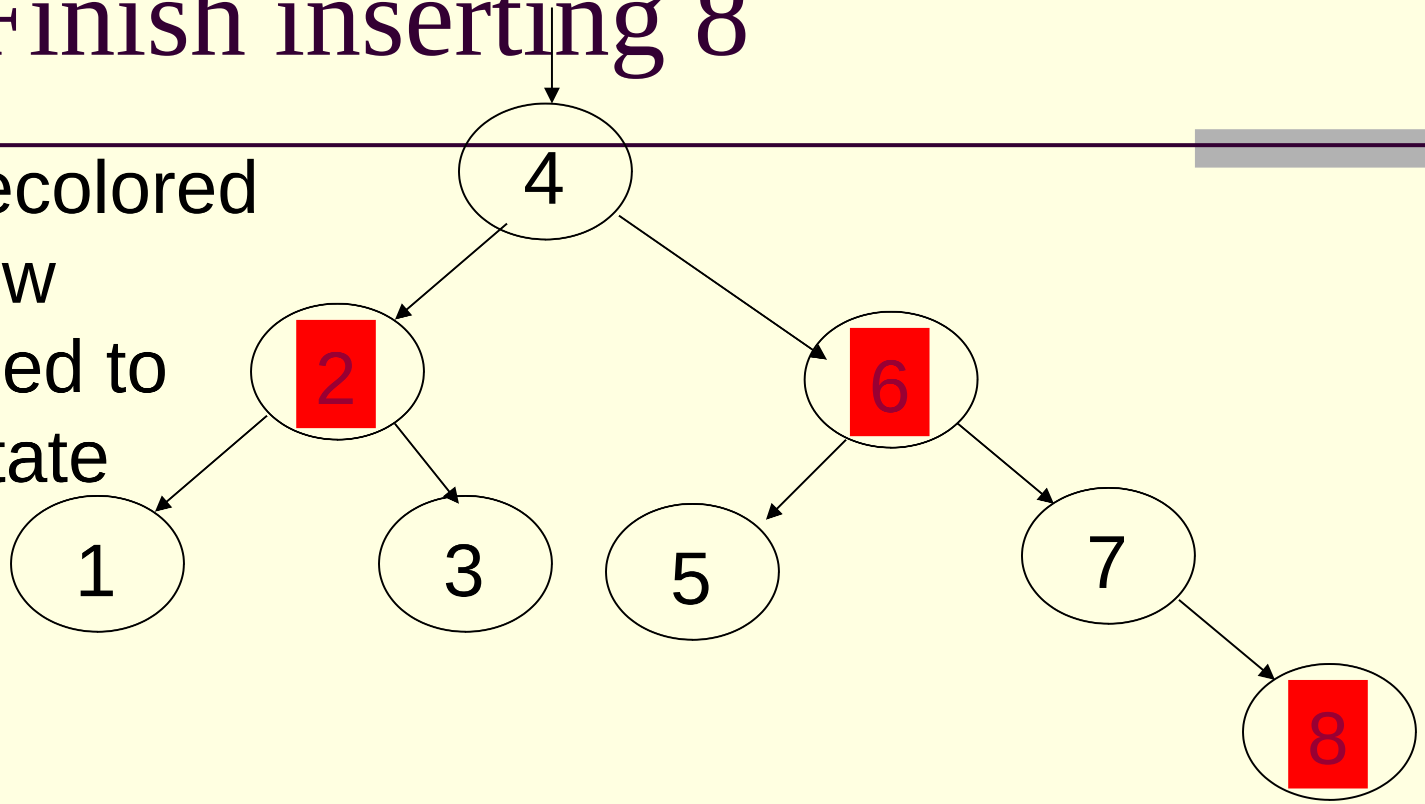
Still Inserting 8

Recolored now need to rotate

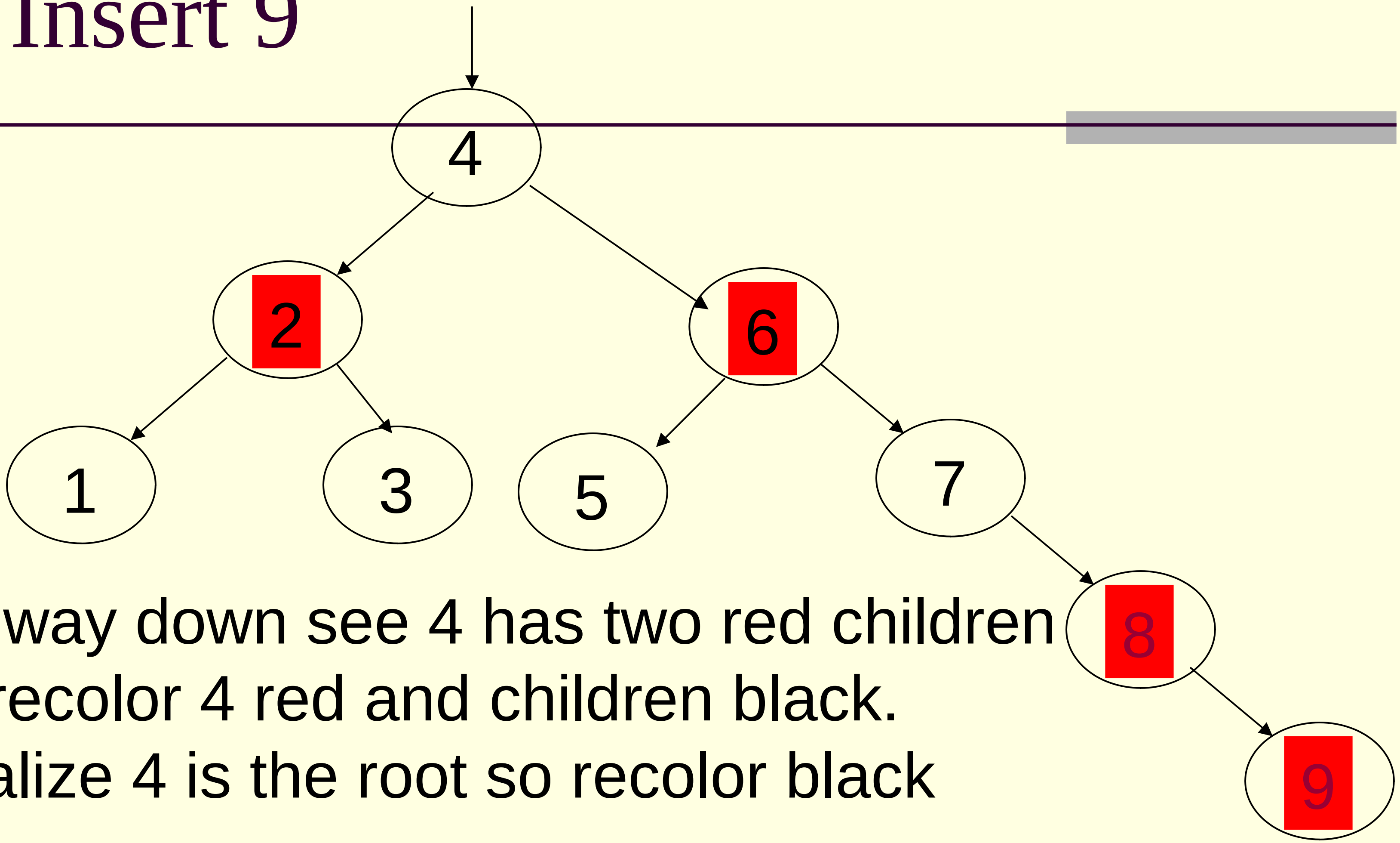


Finish inserting 8

Recolored now need to rotate

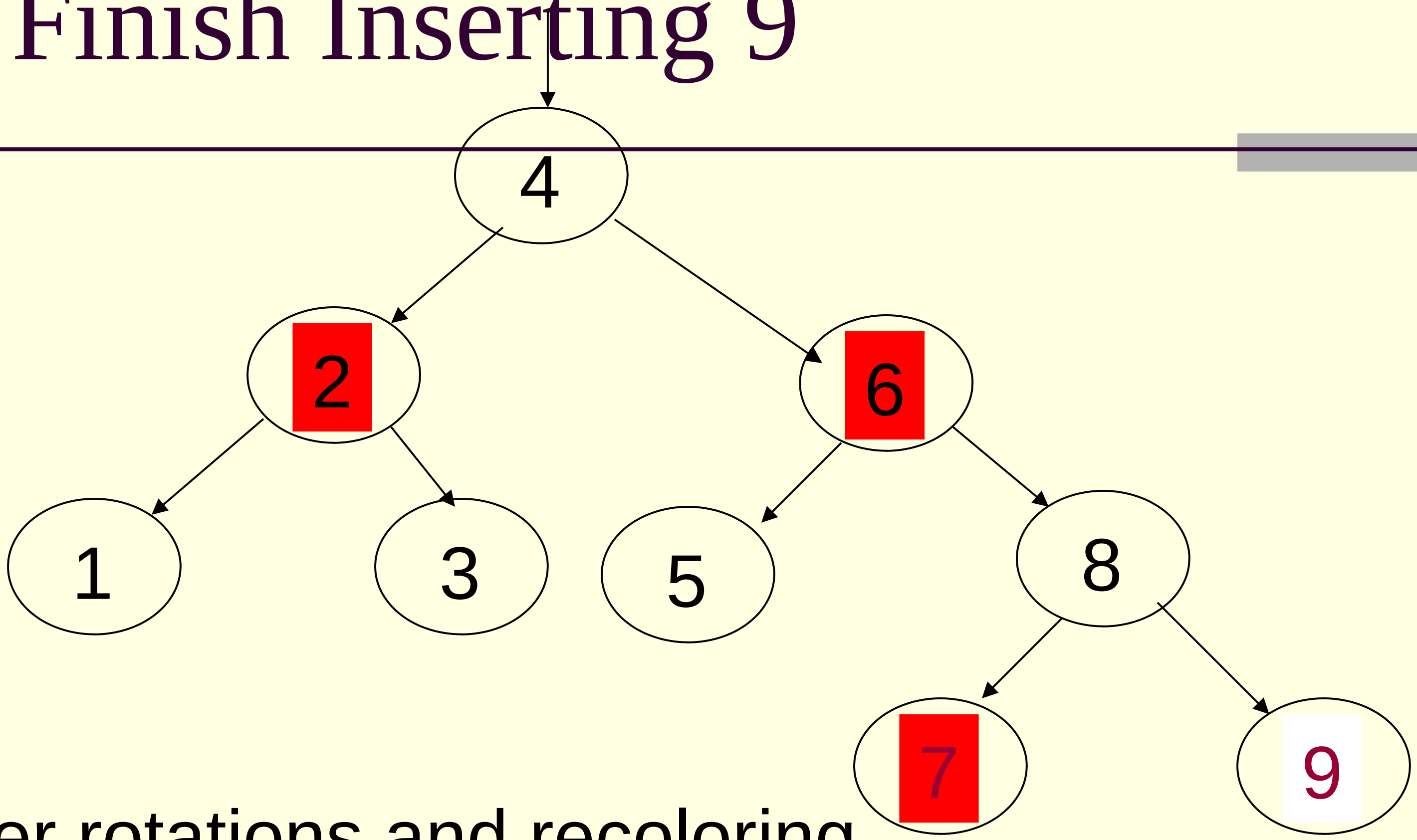


Insert 9



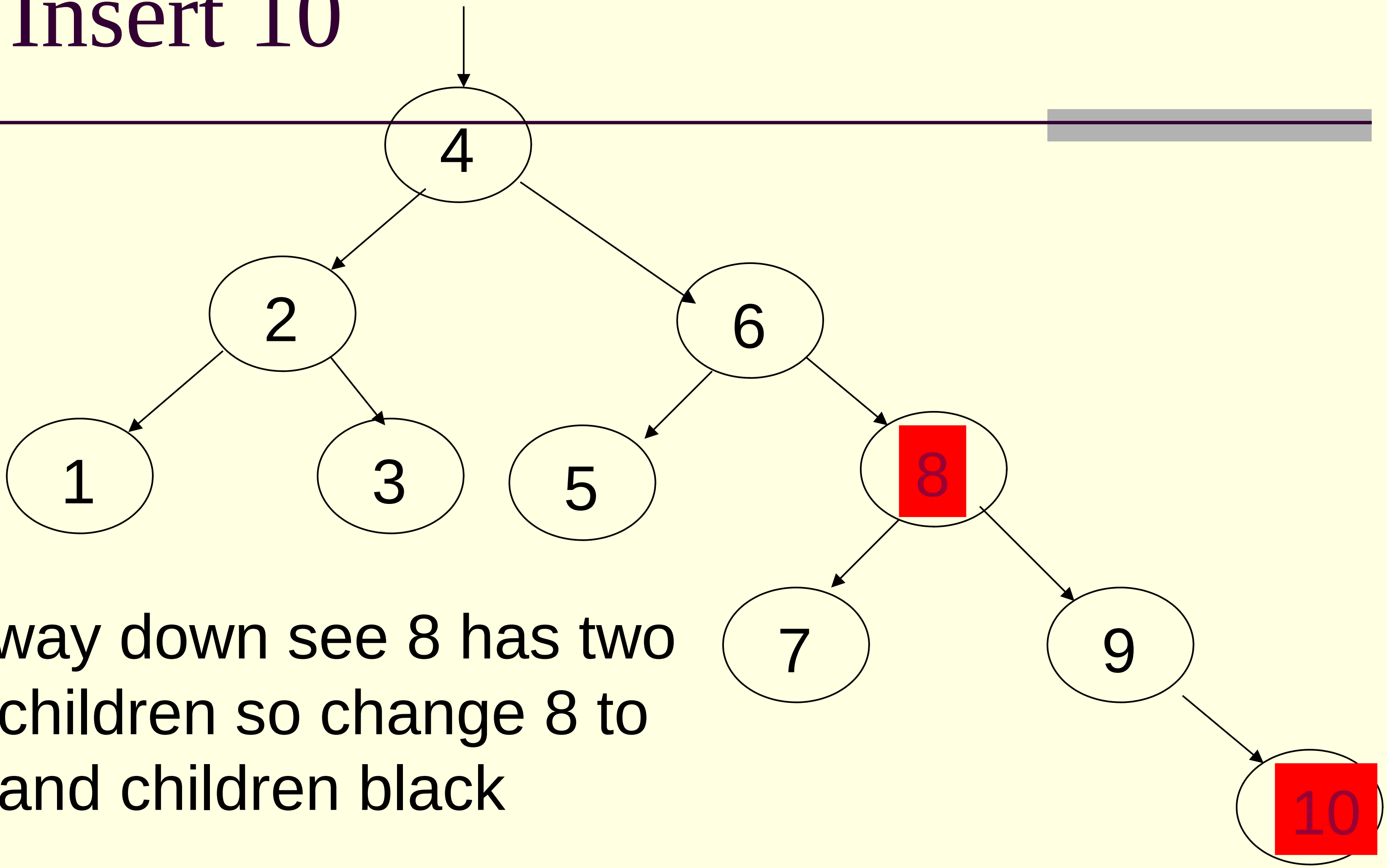
On way down see 4 has two red children so recolor 4 red and children black.
Realize 4 is the root so recolor black

Finish Inserting 9



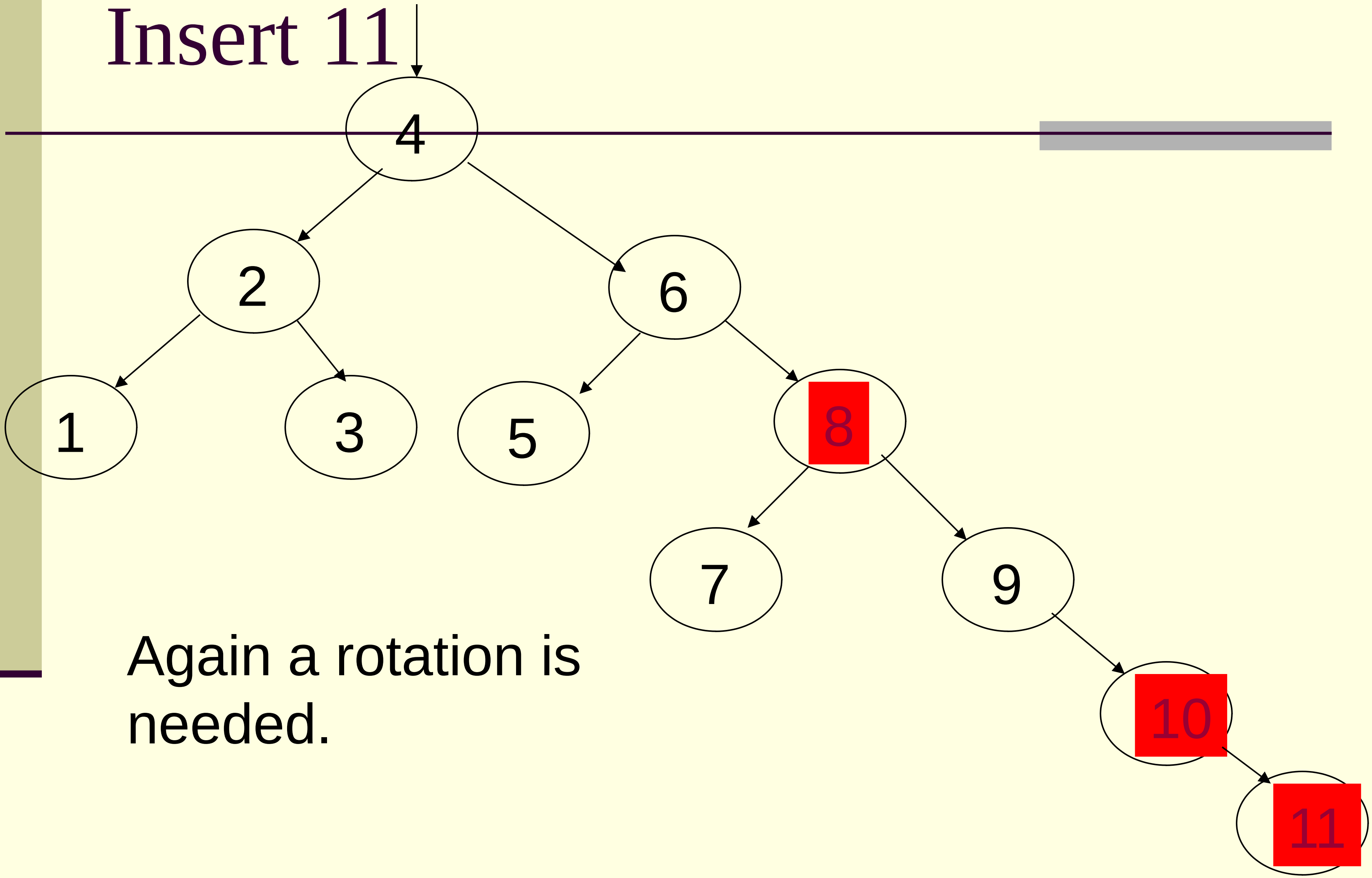
After rotations and recoloring

Insert 10



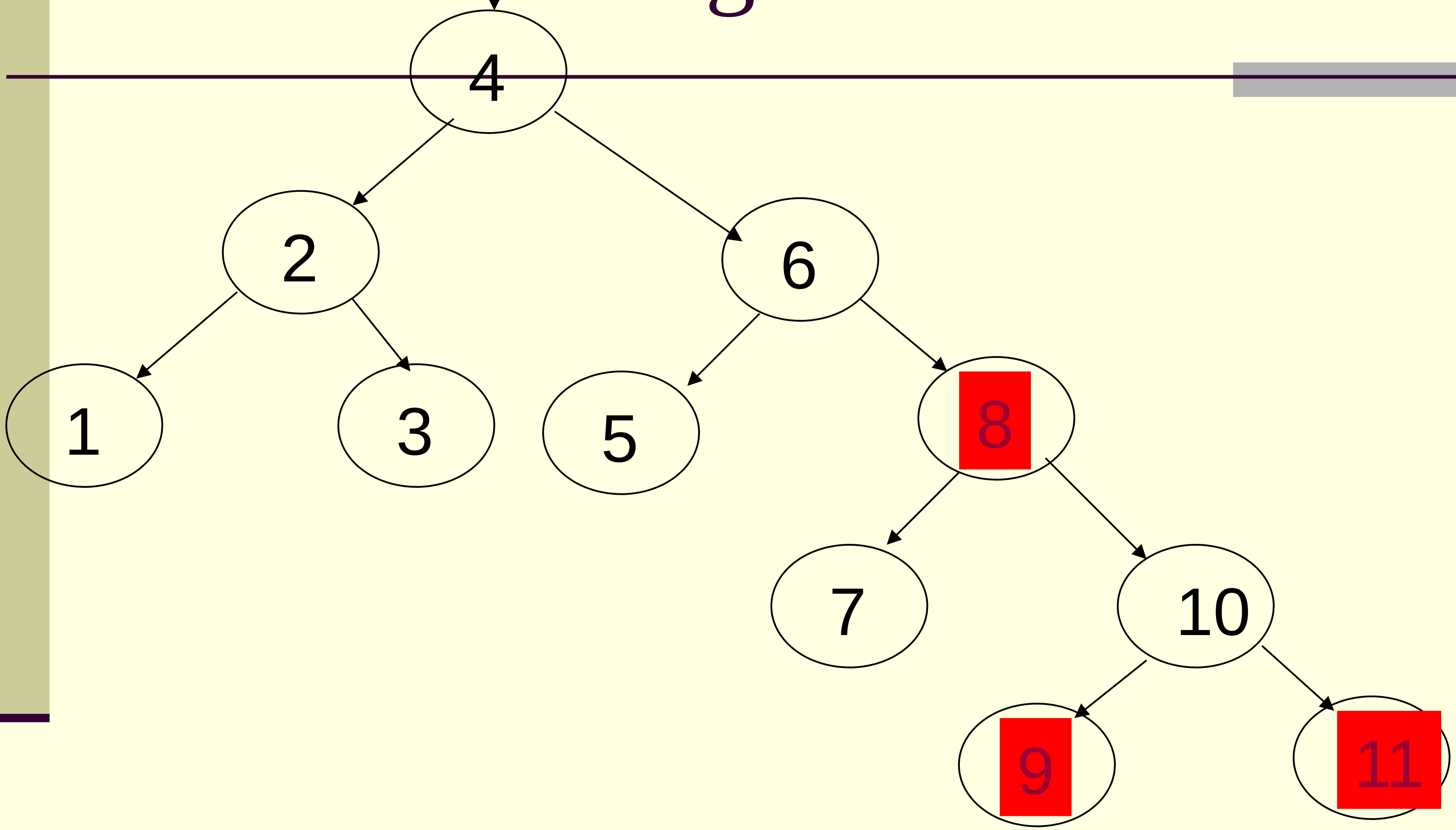
On way down see 8 has two red children so change 8 to red and children black

Insert 11



Again a rotation is needed.

Finish inserting 11



Red-Black Tree (Delete)

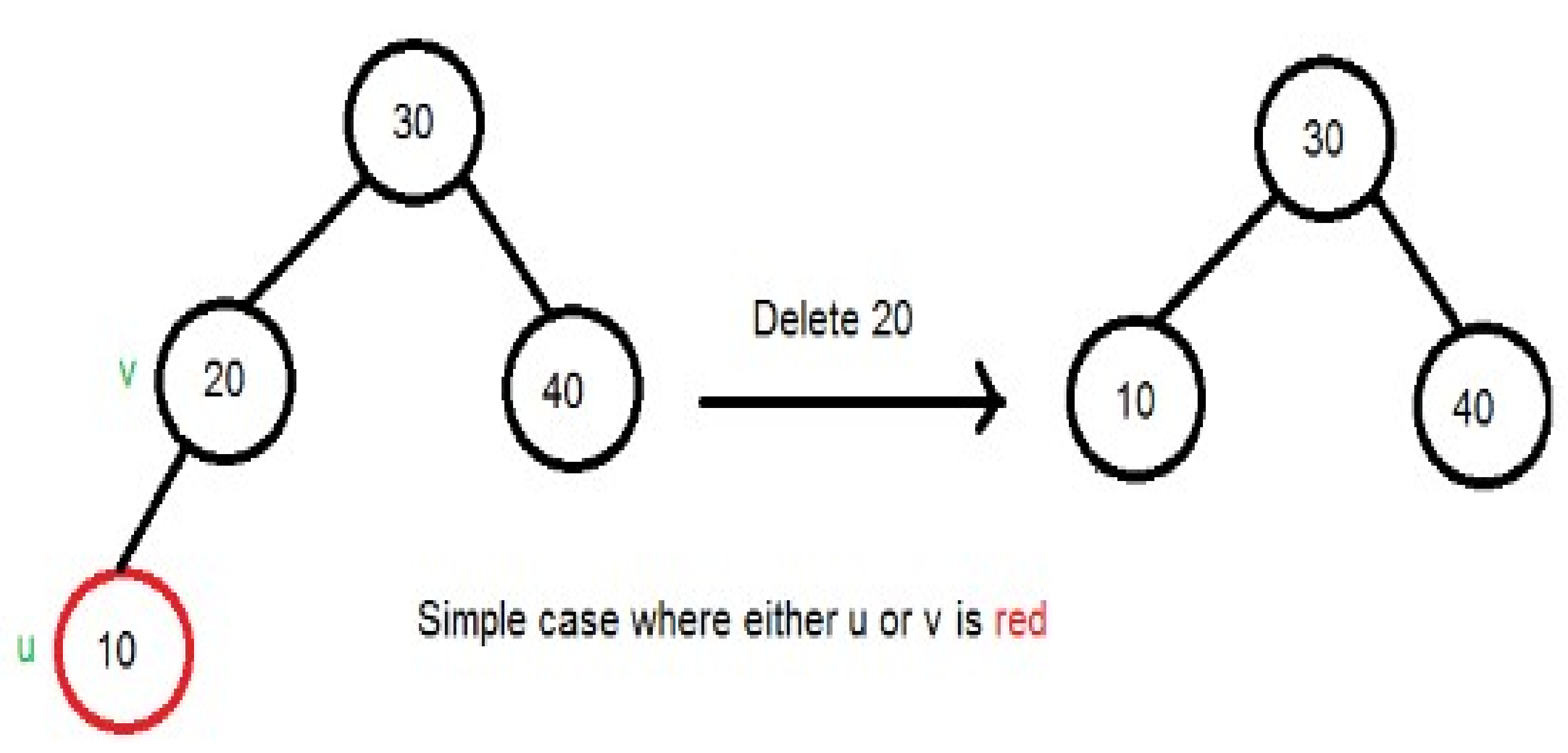
- Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.
- In insert operation, we check color of uncle to decide the appropriate case. In delete operation, ***we check color of sibling*** to decide the appropriate case.

- The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.
- Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as ***double black***. The main task now becomes to convert this double black to single black.

Deletion Steps

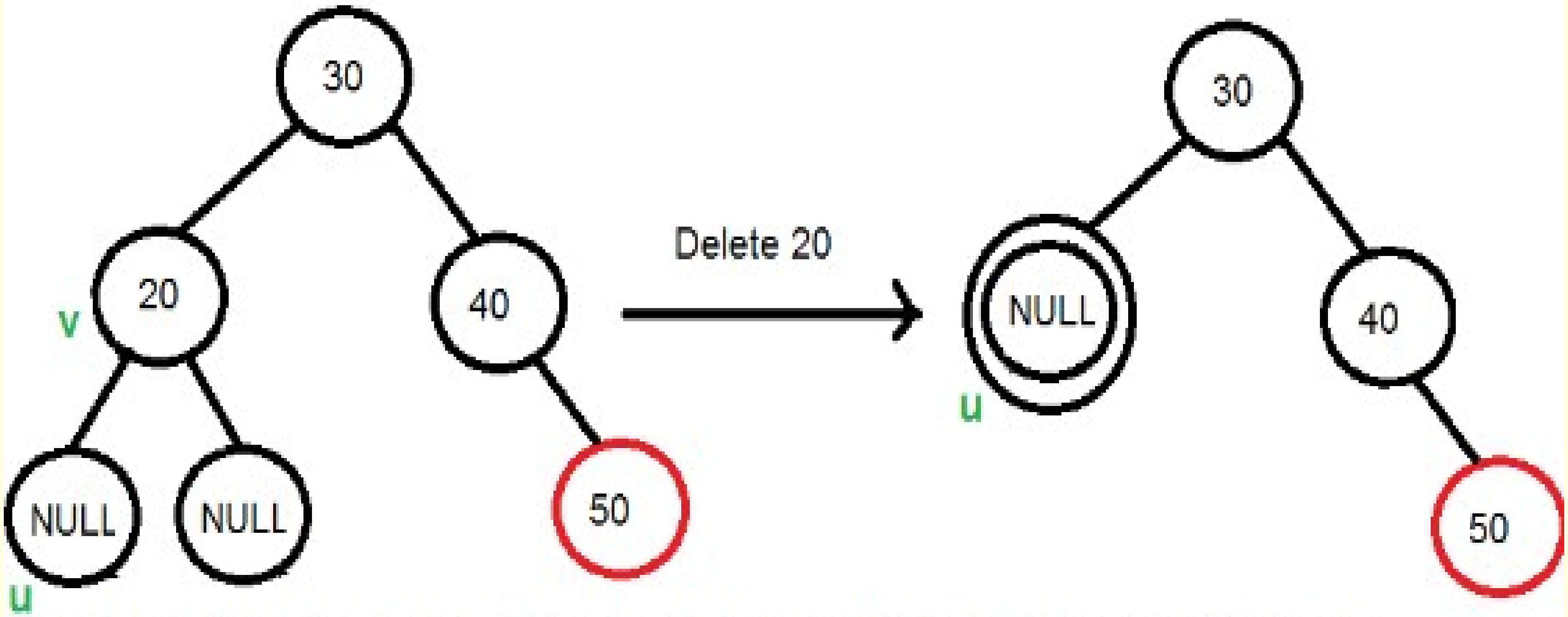
- **1)**Perform standard BST delete.
- When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

- **2) Simple Case: If either u or v is red,** we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.


30
v
20
40
u
10
Delete 20
30
10
40
Simple case where either u or v is red

3) If Both u and v are Black.

- **3.1)** Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.

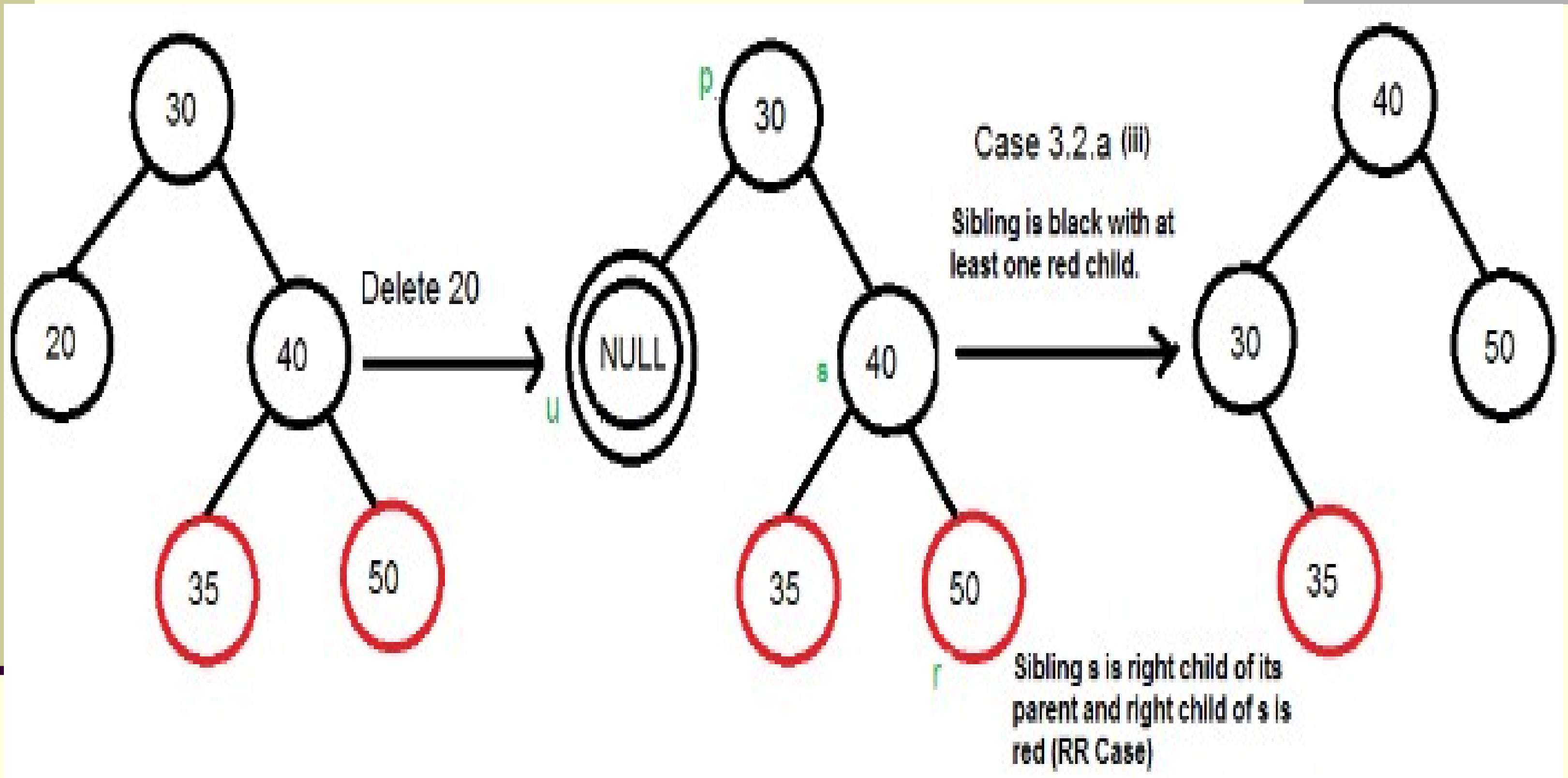


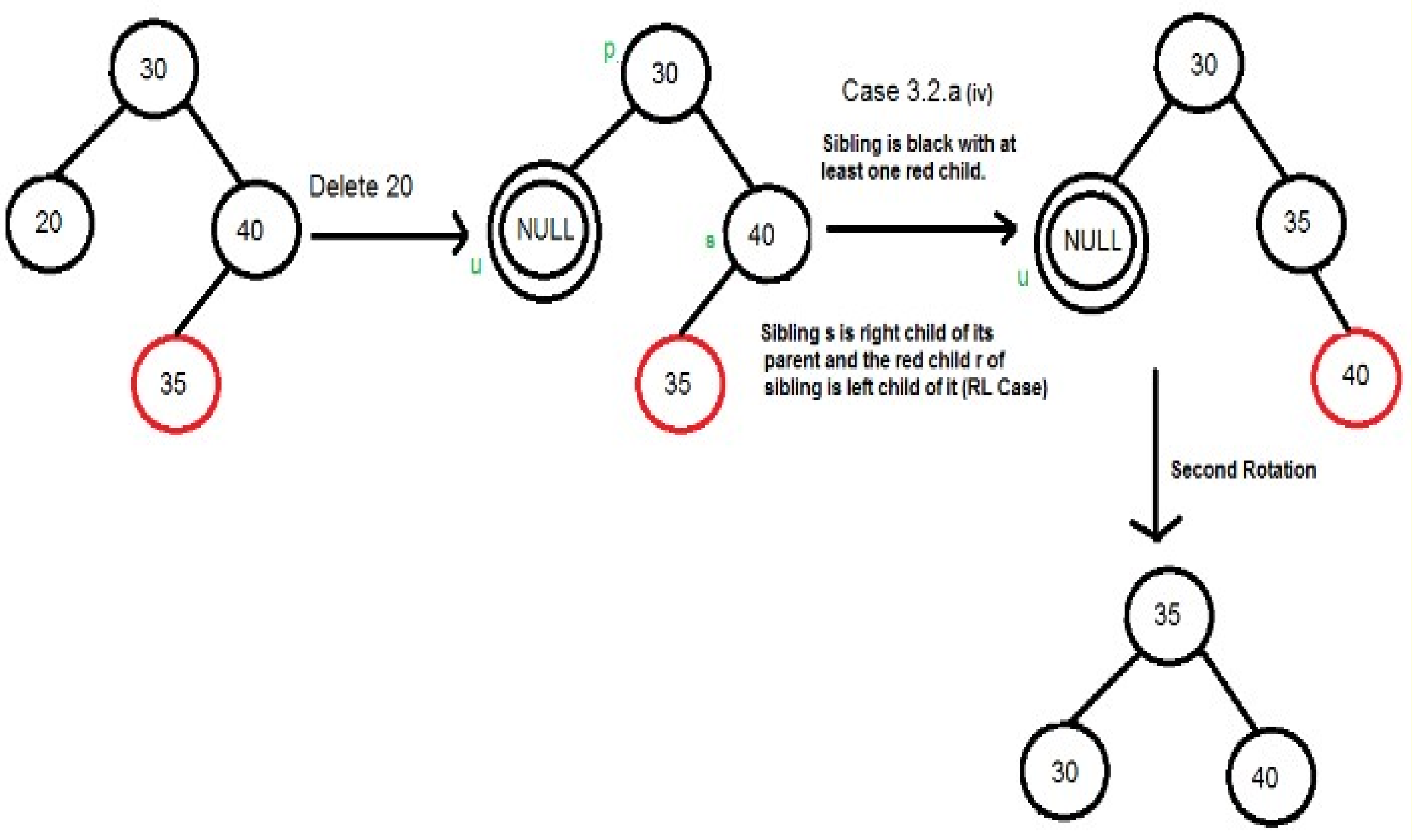
When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.

Note that deletion is not done yet, this double black must become single black

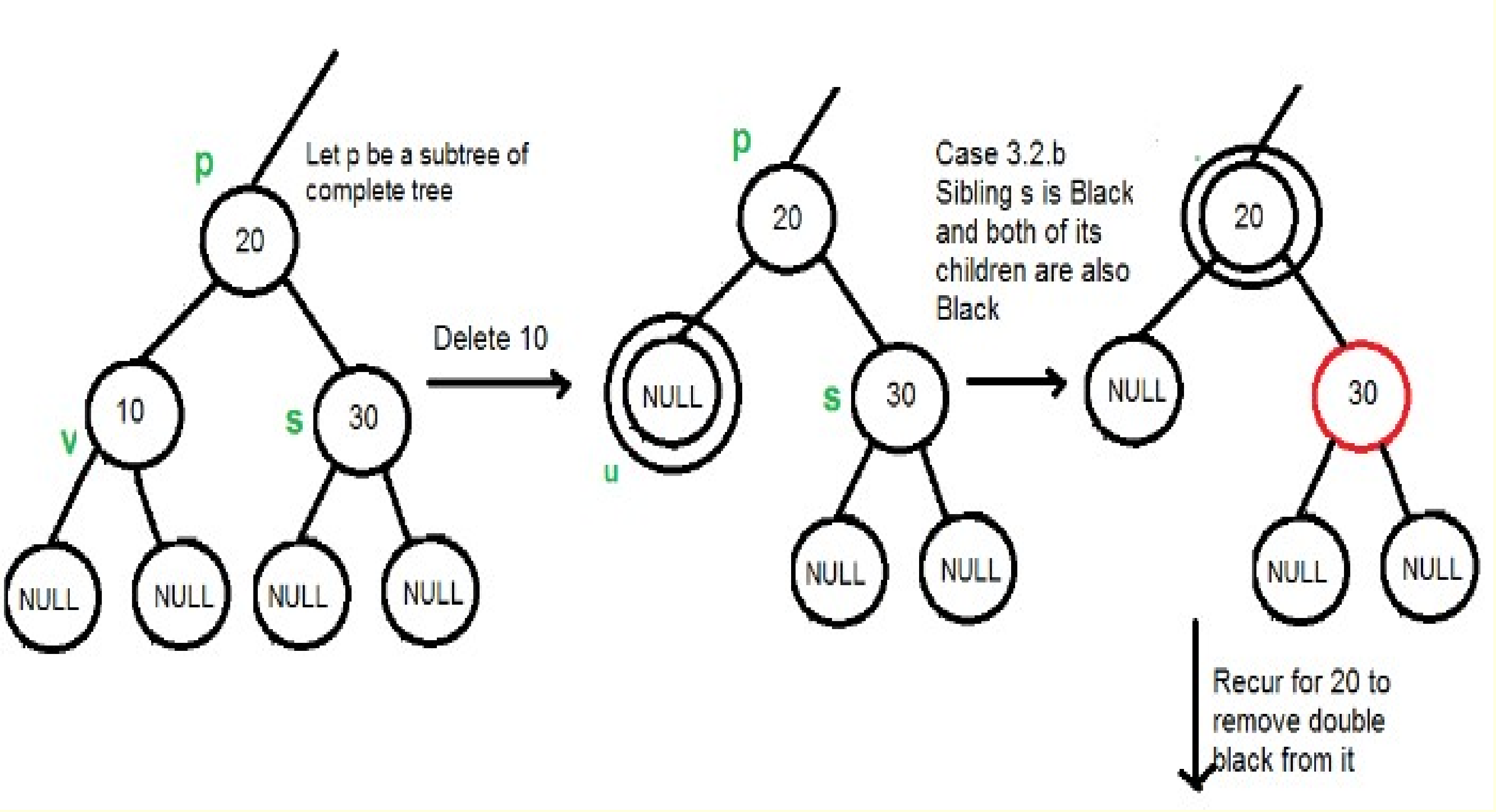
- **3.2)** Do following while the current node u is double black and it is not root. Let sibling of node be **s**.

- **(a): If sibling s is black and at least one of sibling's children is red**, perform rotation(s). Let the red child of s be **r**. This case can be divided in four subcases depending upon positions of s and r.
- **(i)** Left Left Case (s is left child of its parent and r is left child of s or both children of s are red).
- **(ii)** Left Right Case (s is left child of its parent and r is right child).
- **(iii)** Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)
- .**(iv)** Right Left Case (s is right child of its parent and r is left child of s)


30
20
40
35
50
Delete 20
p
30
u
NULL
s
40
35
50
r
Case 3.2.a (iii)
Sibling is black with at least one red child.
40
30
50
35
Sibling s is right child of its parent and right child of s is red (RR Case)

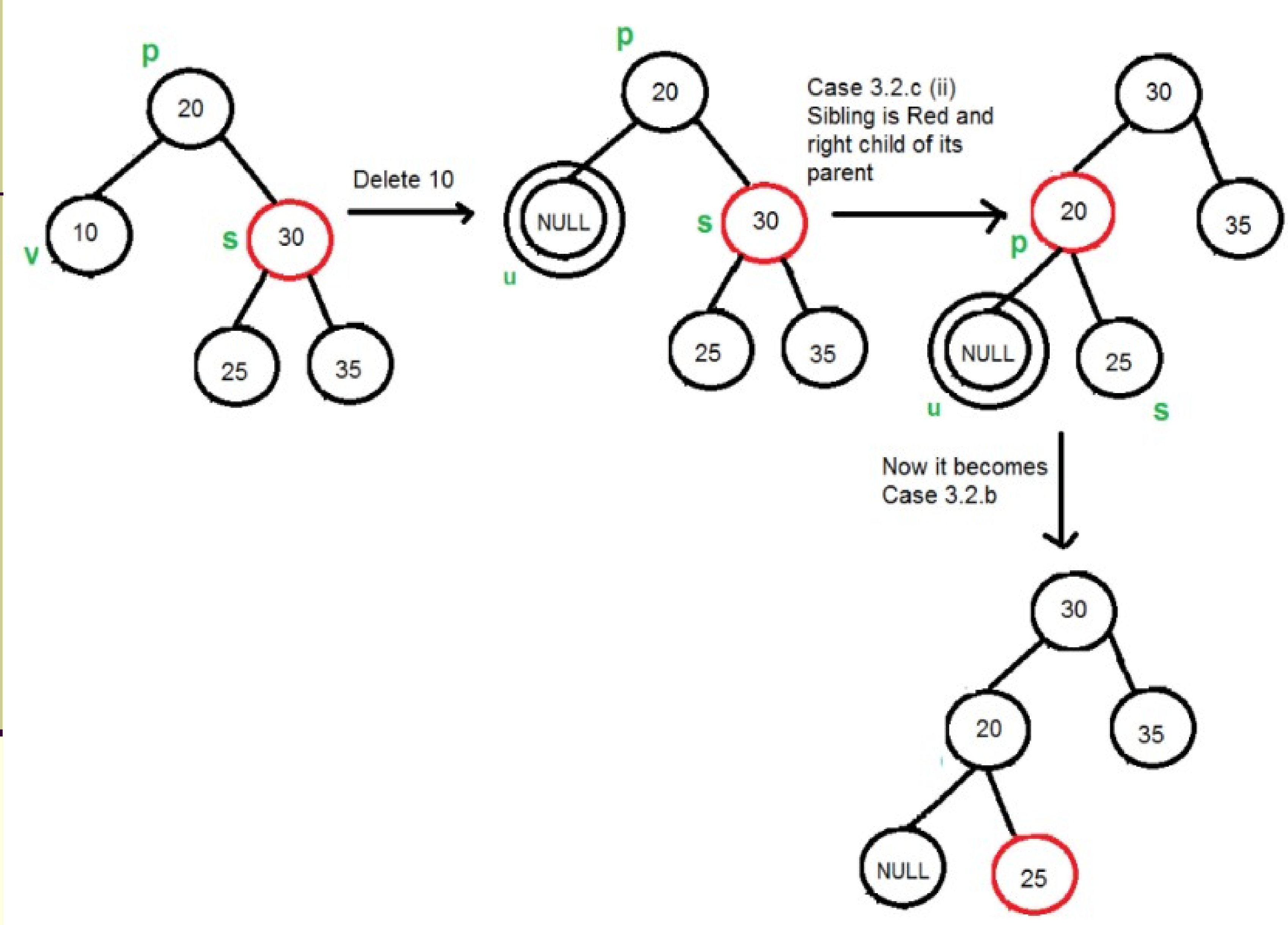

30
20
40
35
Delete 20
p
30
u
NULL
s
40
35
Case 3.2.a (iv)
Sibling is black with at least one red child.
Sibling s is right child of its parent and the red child r of sibling is left child of it (RL Case)
30
u
NULL
35
40
Second Rotation
35
30
40

- **(b): If sibling is black and its both children are black**, perform recoloring, and recur for the parent if parent is black.
- In this case, if parent was red, then we didn’t need to recur for prent, we can simply make it black (red + double black = single black)


p
Let p be a subtree of complete tree
20
10
v
s
30
NULL
NULL
NULL
NULL
Delete 10
p
20
NULL
u
s
30
NULL
NULL
Case 3.2.b
Sibling s is Black and both of its children are also Black
20
NULL
30
NULL
NULL
Recur for 20 to remove double black from it

- **(c): If sibling is red**, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black . This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b).
- This case can be divided in two subcases. **(i)** Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent

(iii) Right Case (s is right child of its parent). We left rotate the parent p.


p
20
10
v
s
30
25
35
Delete 10
p
20
NULL
u
s
30
25
35
Case 3.2.c (ii)
Sibling is Red and
right child of its
parent
30
20
p
35
NULL
u
25
s
Now it becomes
Case 3.2.b
30
20
35
NULL
25

- **3.3)** If u is root, make it single black and return (Black height of complete tree reduces by 1).

Red-black Tree Deletion

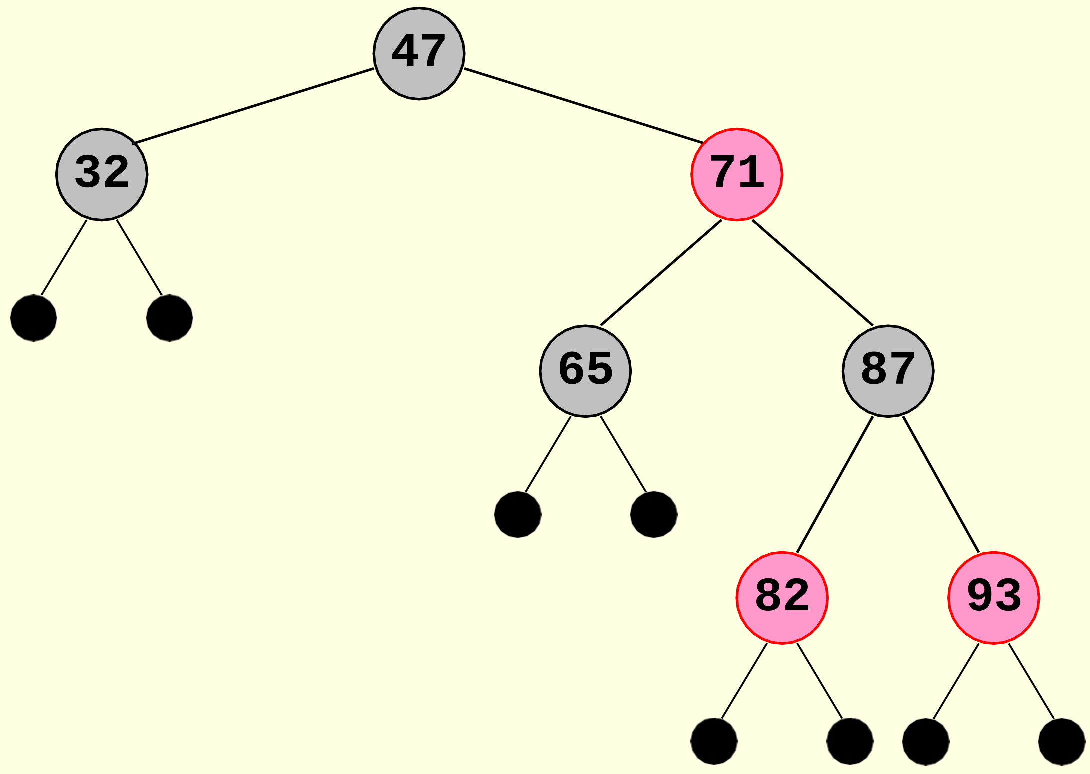
- First use the standard BST tree deletion algorithm
  - If the node to be deleted is replaced by its successor/predecessor (if it has two non-null children), consider the deleted node's data as being replaced by it's successor/predecessor's, and its color remaining the same
    - The successor/predecessor node is then removed
- Let y be the node to be removed
- If the removed node was red, no property could get violated, so just remove it.
- Otherwise, remove it and call the tree-fix algorithm on y's child x (the node which replaced the position of y)
  - Remember, the removed node can have at most one real (non-null) child
  - If it has one real child, call the tree-fix algorithm on it
  - If it has no real children (both children are null), Note that this child may be a (black) pretend (null) child

Fixing a red-black Tree

- The tree-fix algorithm considers the parameter (**x**) as having an *“extra” black token*
  - This corrects the violation of property 4 caused by removing a black node
- If **x** is red, just color it black
- But if **x** is black then it becomes “doubly black”
  - This is a violation of property 1
  - The extra black token is pushed up the tree until
    - a red node is reached, when it is made black
    - the root node is reached or
    - it can be removed by rotating and recoloring

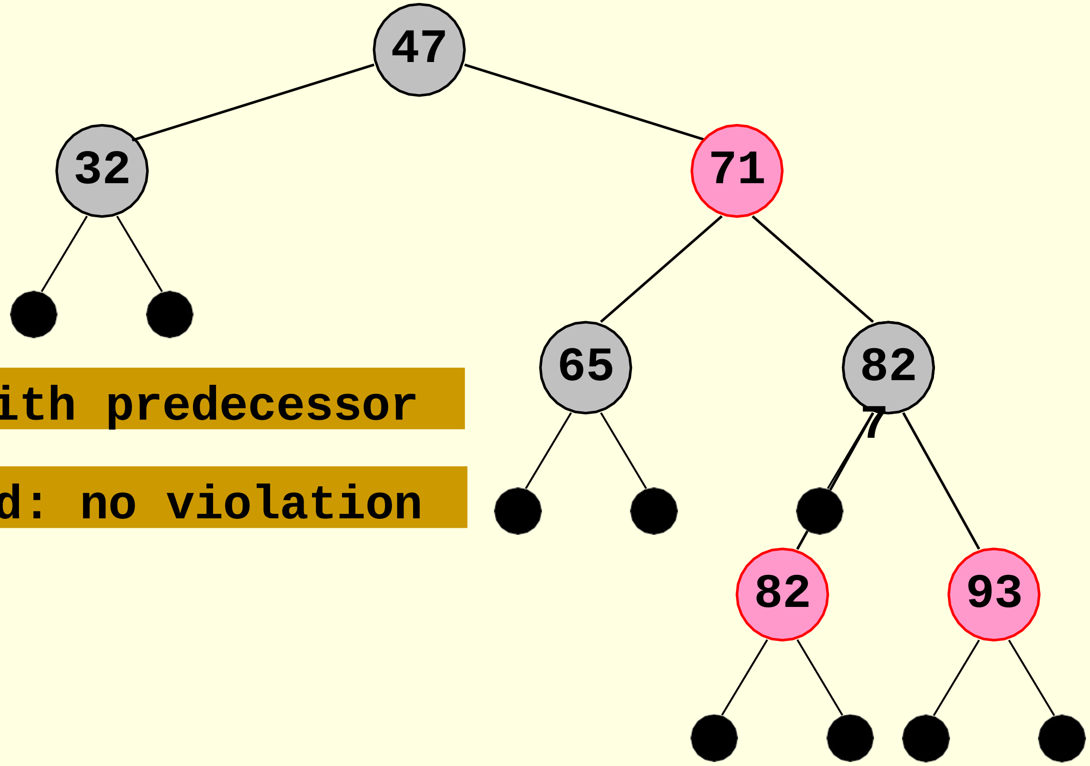
Deletion Example 1

Delete 87



Deletion Example 1

Delete 87

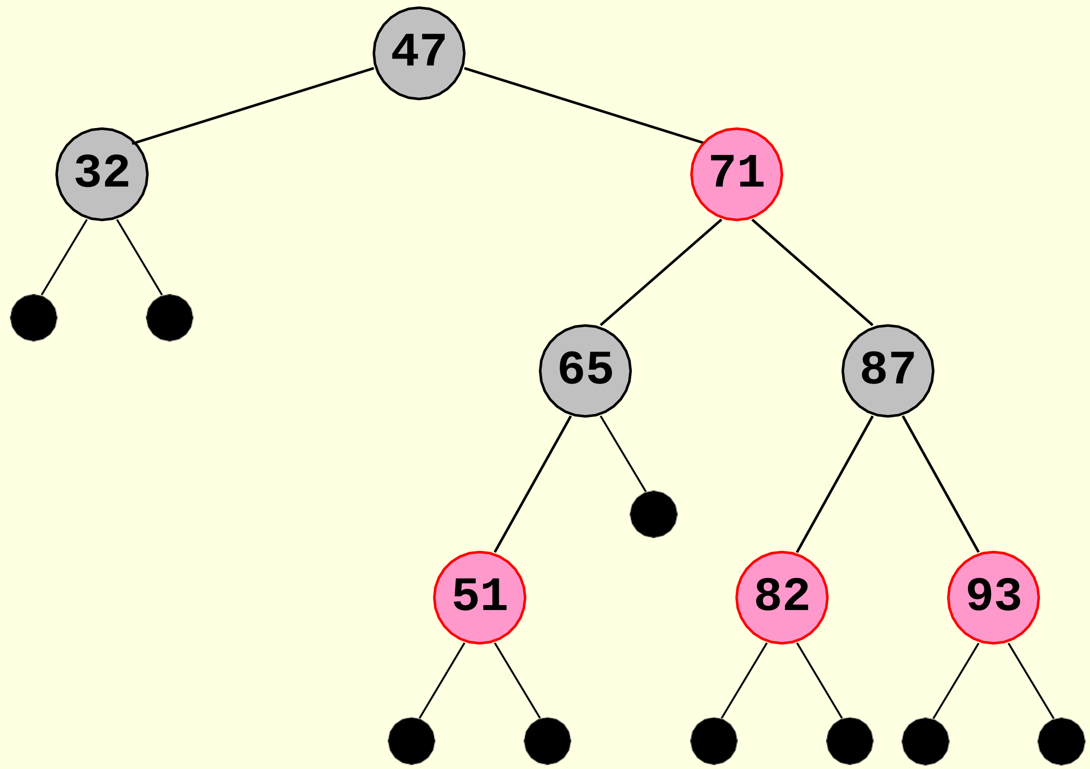


Replace data with predecessor

Predecessor red: no violation

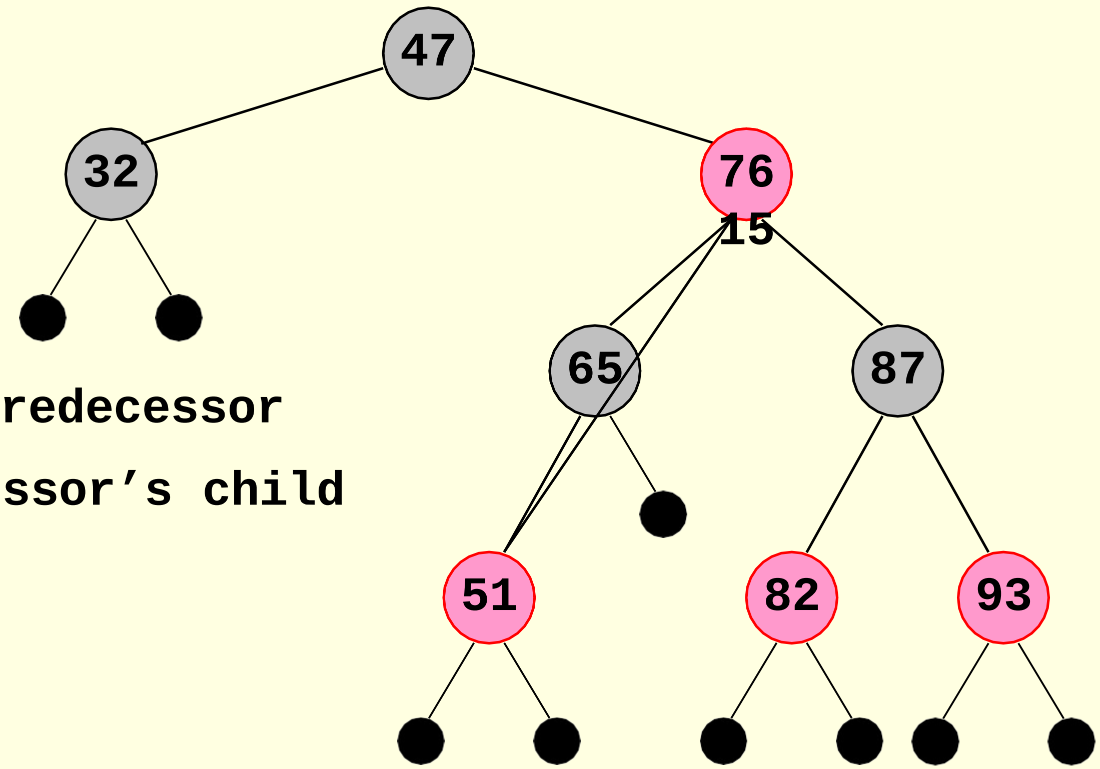
Deletion Example 2

Delete 71



Deletion Example 2

Delete 71

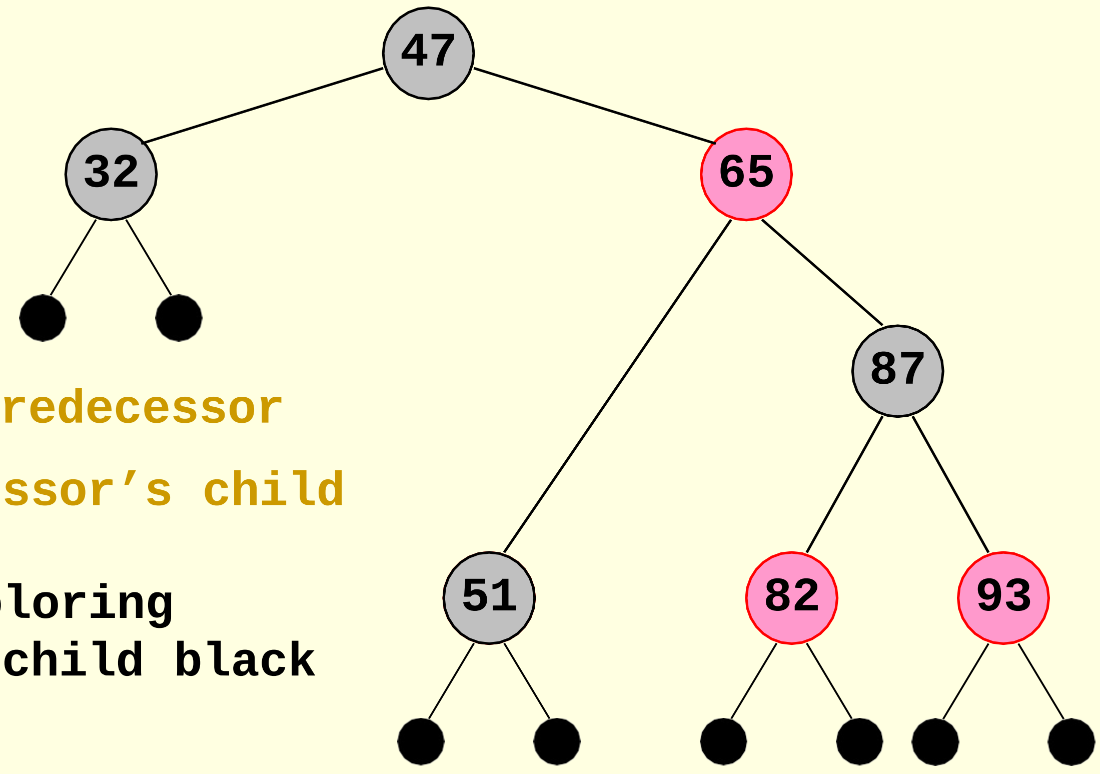


Replace with predecessor

Attach predecessor's child

Deletion Example 2

Delete 71



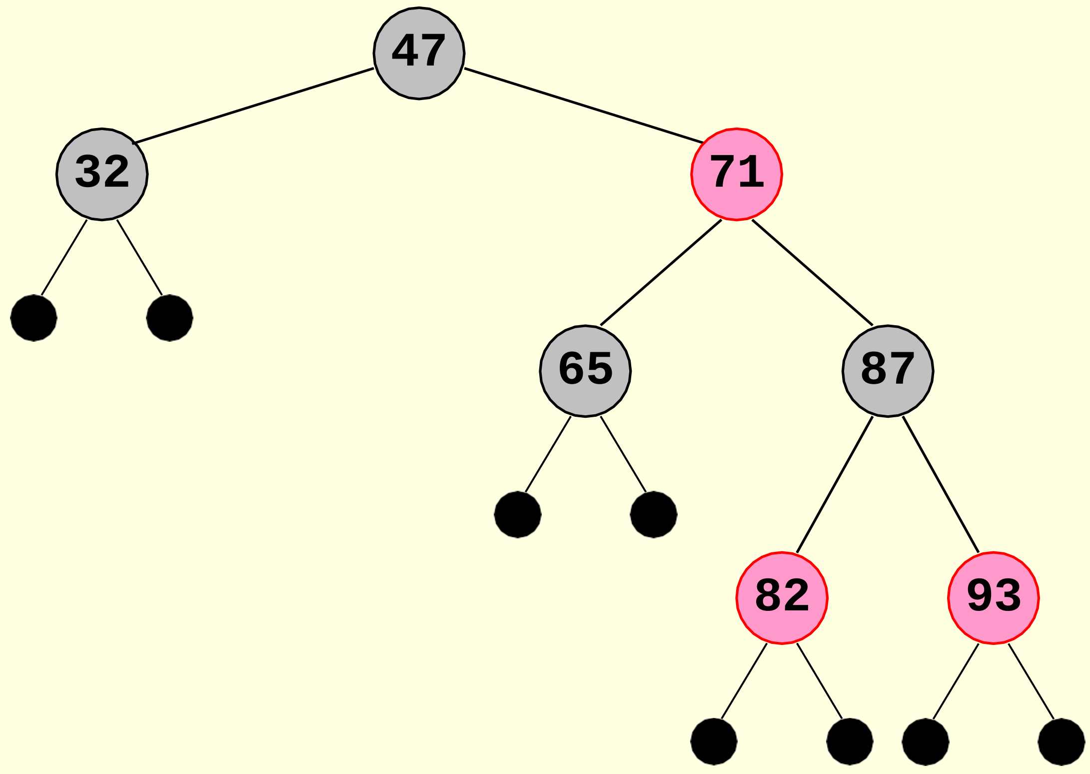
Replace with predecessor

Attach predecessor's child

Fix tree by coloring predecessor's child black

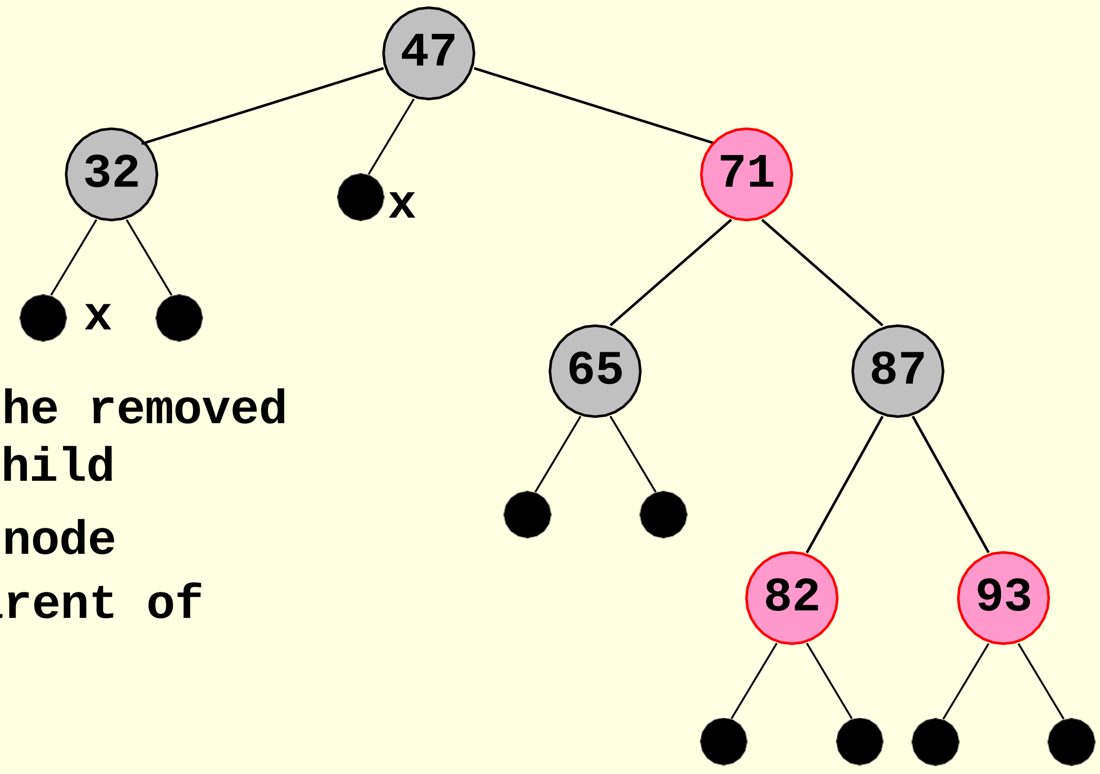
Deletion Example 3

Delete 32



Deletion Example 3

Delete 32



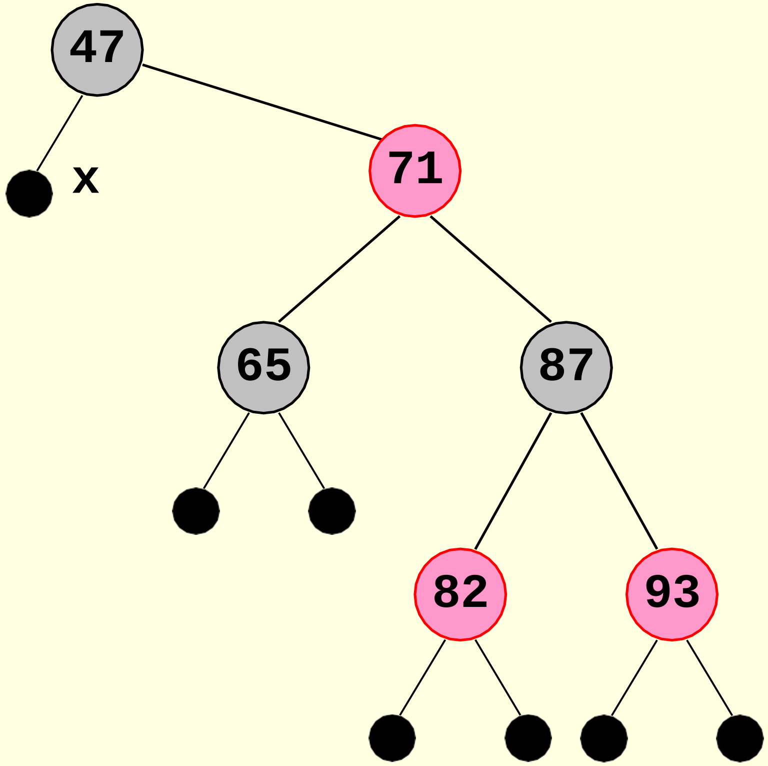
Identify x – the removed node's left child

Remove target node

Attach x to parent of target

Deletion Example 3

Delete 32



Identify x – the removed node's left child

Remove target node

Attach x to parent of target

Call rbTreeFix on x

RB Tree Deletion Algorithm

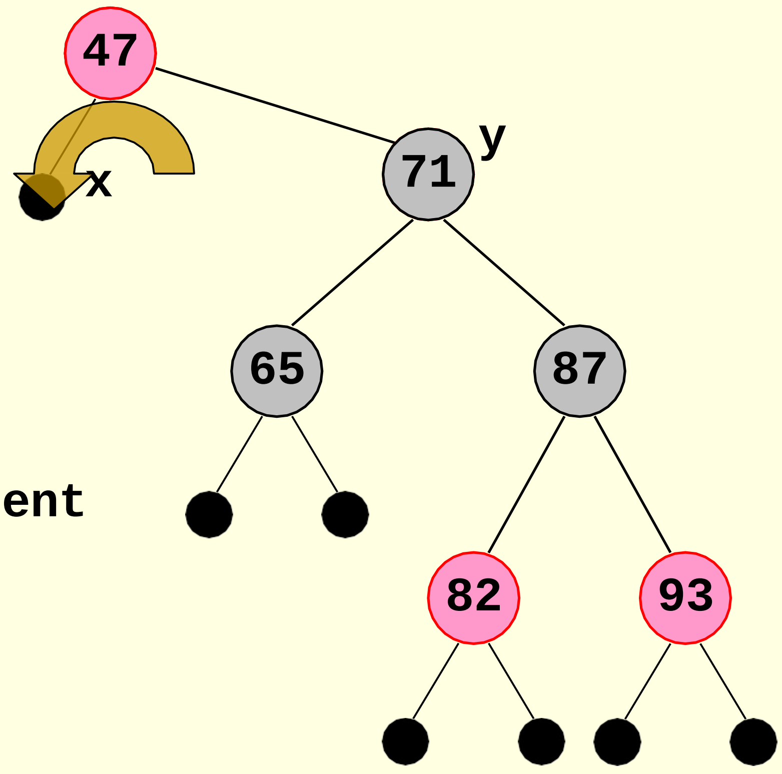
```
TreeNode<T> rbDelete(TreeNode<T> root,TreeNode<T> z)
//return new root, z contains item to be deleted
{
  TreeNode<T> x,y;
  // find node y, which is going to be removed
  if (z.getLeft() == null || z.getRight() == null)
     y = z;
else {
  y = successor(z); // or predecessor
    z.setItem(y.getItem); // move data from y to z
}

// find child x of y
  if (y.getRight() != null)
     x = y.getRight();
else
  x = y.getLeft();

// Note x might be null;
  create a pretend node
if (x == null) {
  x = new TreeNode<T>(null);
    x.setColor(black);
```

RB Tree Deletion Algorithm

```
  x.setParent(y.getParent()); // detach x from y
  if (y.getParent() == null)
    // if y was the root, x is a new root
     root = x;
  else
    // Atttach x to y's parent
    if (y == y.getParent().getLeft()) // left child
       y.getParent().setLeft(x);
  else
    y.getParent().setRight(x);

if (y.getColor() == black)
  root=rbTreeFix(root,x);

if (x.getItem() == null) // x is a pretend node
    if (x==x.getParent().getLeft())
       x.getParent().setLeft(null);
  else
    x.getParent().setRight(null);

return root;
}
```

Deletion Example 3 (continued)

After deleting 32, x is a node with black token



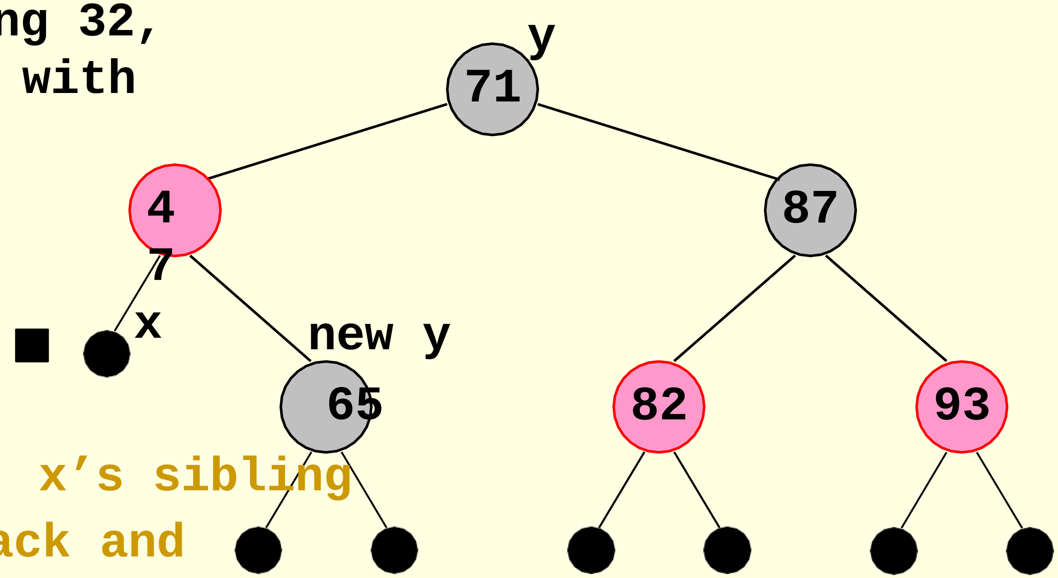
Identify y, x’s sibling

Make y black and y’s parent red

Left rotate x’s parent

Deletion Example 3

After deleting 32, x is a node with black token



Identify y, x’s sibling

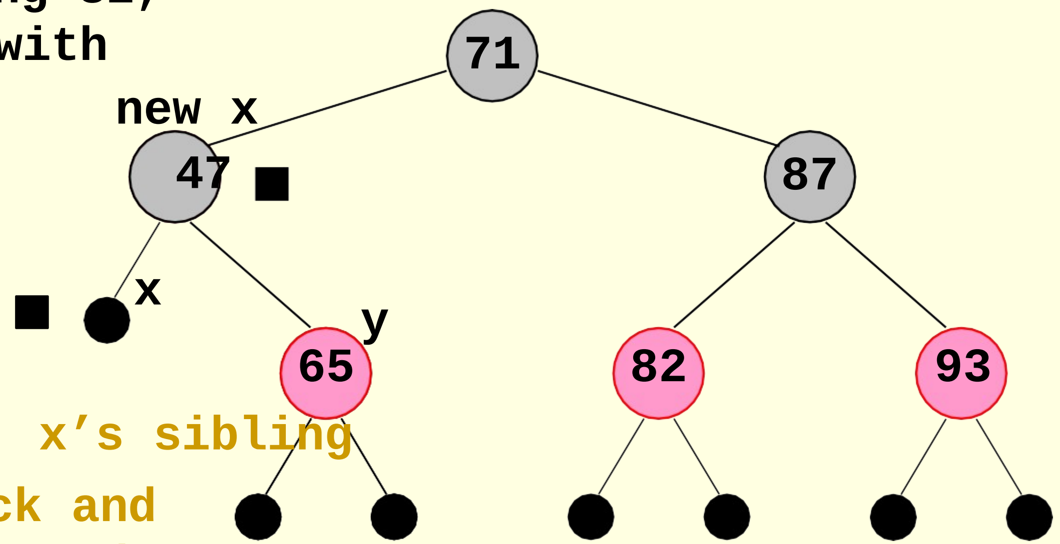
Make y black and y’s parent red

Left rotate x’s parent

Identify y – x’s new sibling

Deletion Example 3

After deleting 32,
x is a node with
black token



Identify y, x’s sibling

Make y black and
y’s parent red

Left rotate x’s parent

Identify y – x’s new sibling

Color y red

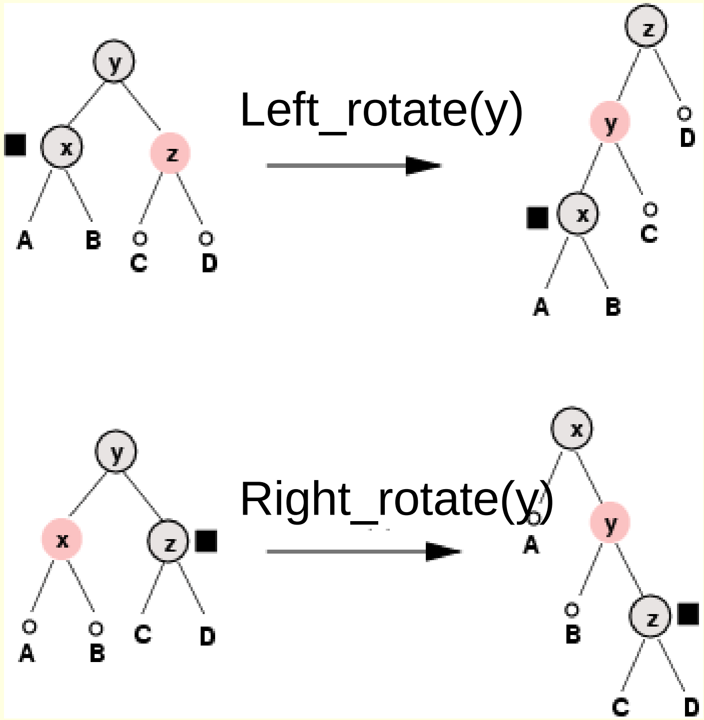
Assign x it’s parent, and color it black

Tree Fix algorithm cases: case (1) x is red

- The simplest case
- x has a black token and is colored red, so just color it black and remove token a we are done!

- In the remaining cases, assume x is black (and has the black token, i.e., it's double black)

Tree Fix algorithm cases: case (2) x's sibling is red



Colors of y and z were swapped

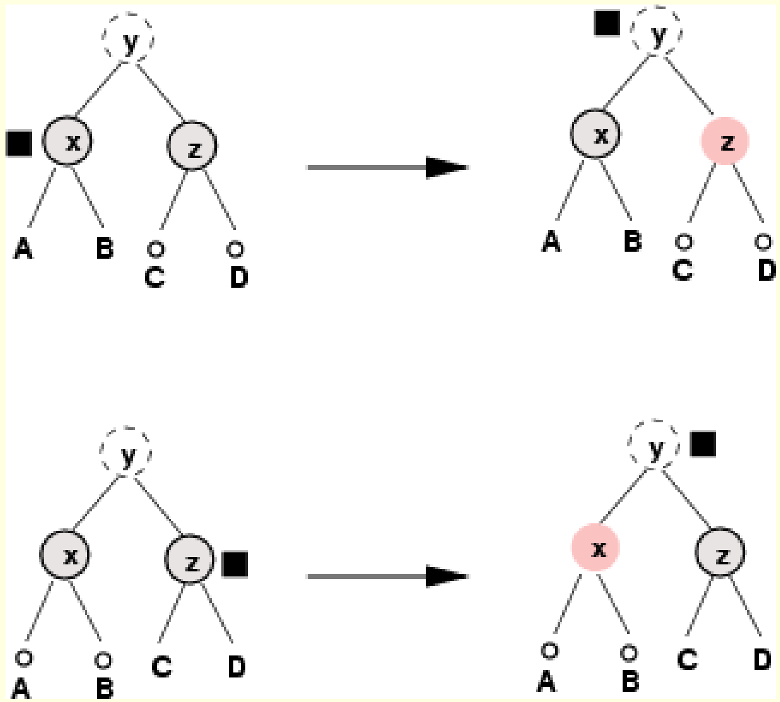
Colors of x and y were swapped

Remarks:

- the roots of subtrees C and D are black
- the second is the symmetric case, when x is the right child
- in the next step (case (3) or (4)) the algorithm will finish!

Tree Fix algorithm cases: case (3)

x's sibling is black and both nephews are black

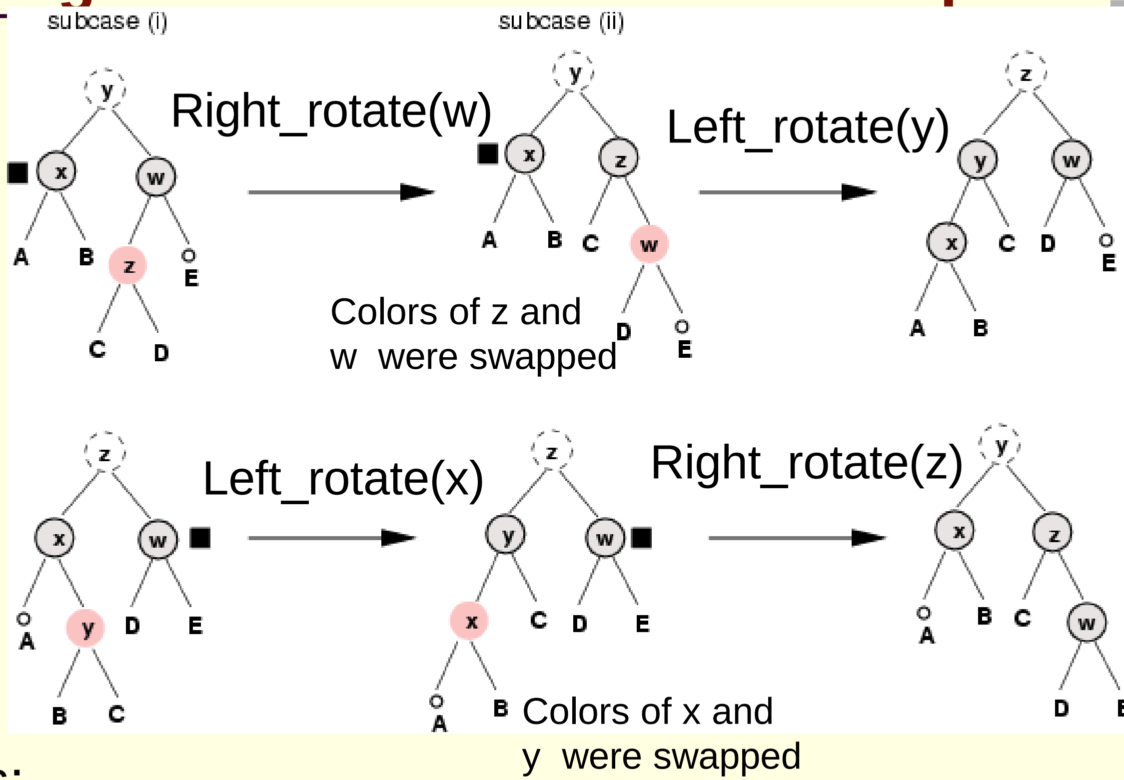


Remarks:

- nephews are roots of subtrees C and D

Tree Fix algorithm cases: case (4)

x’s sibling is black and at least one nephew is red



Colors of y and z were swapped. Far nephew is colored black and black token is removed.

Colors of z and y were swapped. Far nephew is colored black and black token is removed.

Remarks:

- in this case, the black token is removed completely
- if the “far” nephew is black (subcase (i)), rotate its parent, so that

RB Trees efficiency

- All operations work in time O(height)
- and we have proved that heigh is O(log n)

- hence, all operations work in time O(log n)! – much more efficient than linked list or arrays implementation of sorted list!

Sorted List	Search	Insertion	Deletion
with arrays	O(log n)	O(n)	O(n)
with linked list	O(n)	O(n)	O(n)
with RB trees	O(log n)	O(log n)	O(log n)