# Data Structures: Red Black Trees

R. K. Ghosh

IIT Bhilai

Red Black Trees

## Important characteristics

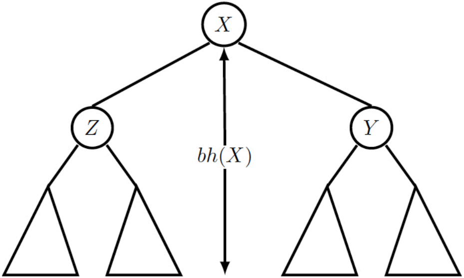
- Preserves ordering property of a BST - Order Invariant.
- Nodes are colored either as red or as black.
- No two consecutive nodes can be colored red - Color Invariant
- All leaf nodes are colored black.
- Root is colored black.
- Number of black nodes on a path from root to a leaf node is the same - Height Invariant.

# Red Black Trees

## Important characteristics

- Search is easy due to BST property.
- Newly inserted node is colored red which may violate color invariant - Two consecutive node color being red.
- Insertion requires color flipping and restructuring to restores black height and color invariance property.

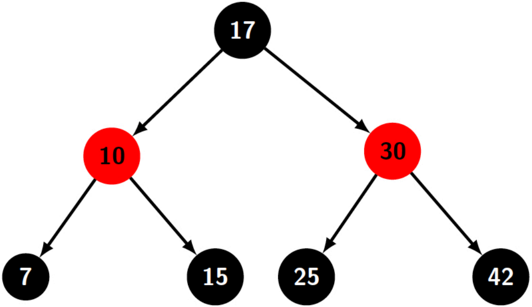
# Black Height




**Black height**

Black height of a node $n$ is equal to the number black node on the path to farthest leaf node excluding $n$

# Occurrences of Red Nodes

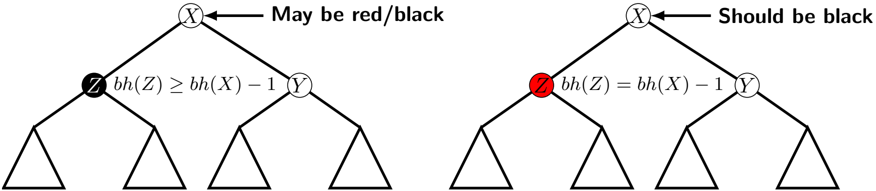



- No two consecutive nodes can be red.

# Black Height and Number of Internal Nodes

**Property I**

Let the black height of a RBT $T$ with root $X$ be $bh(X)$. Then $T$ consists of at least $2^{bh(X)} - 1$ internal nodes.




- Implies $bh(Z) \geq bh(X) - 1$

# Black Height and Number of Internal Nodes

- If $X$ is a leaf node then $bh(X) = 0$, and it consists of $2^0 - 1 = 0$ internal nodes.
- Apply induction to the subtree with root $Z$.
  - It has at least $2^{bh(Z)} - 1$ internal nodes.
  - Since $bh(Z) \geq bh(X) - 1$, it has at least $2^{bh(X)-1} - 1$ internal nodes.
  - So, $X$ has at least $2(2^{bh(X)-1} - 1) + 1 = 2^{bh(X)} - 1$ internal nodes.

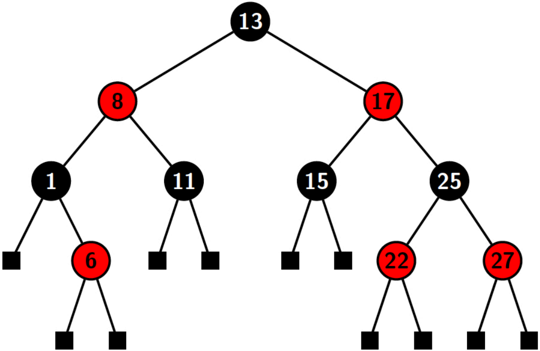
# Height of a Red Black tree.

**Property II**

Height of a red black tree with $n$ nodes is O($\log n$).

- If a red black tree has $n$ nodes then $n \geq 2^{bh(root)} - 1$.
- At least half the nodes on a path from the root to a leaf node are black.
- So, $bh(root) \geq h/2$, implying that $n \geq 2^{h/2} - 1$, where $h$ is the height of the tree.
- Therefore, $h/2 \leq \log(n+1)$, or $h \leq 2\log(n+1)$
- That is $h = O(\log n)$.

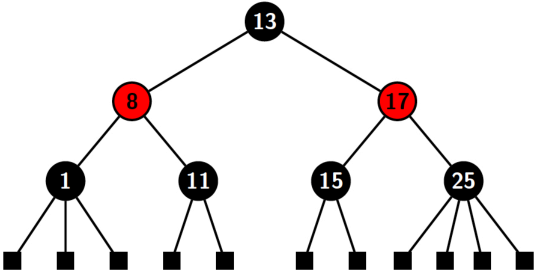
# Collapsing Red Nodes




Initial Red Black Tree

- Collapsing all red nodes into their respect black parents.

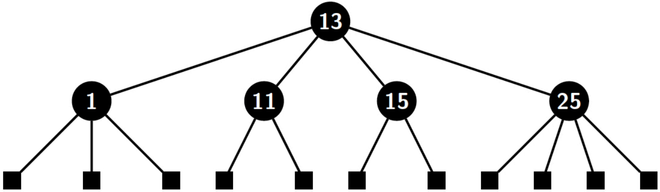
# Collapsing Red Nodes




After collapsing 6, 22, 27

- A node in compact tree may have 2,or 3,or 4 children.

# Collapsing Red Nodes



After collapsing 8 and 17

- The height of collapsed tree is $h' \geq h/2$
- All external node are at same level.
- The number of internal nodes in the tree is $n \geq 2^{h'} - 1 \geq 2^{h/2} - 1$
- Therefore, $h \leq 2log(n + 1)$

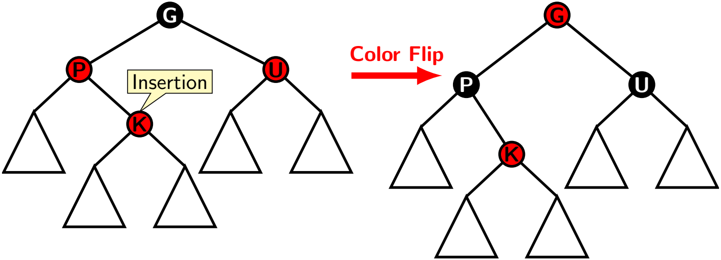
# Summary of Properties

1. Every node is colored either red or black.
2. Root is always colored black.
3. Every leaf is colored black.
4. If a node is red, then both its children are black.
5. All paths from a node to descendant leaves have same number of black nodes.

# Insertion in a RB Tree

- External nodes not shown explicitly in examples.
- An insertion occurs at the place of an external nodes.
- The inserted node is always colored red (why?).
  - Using black color will add black depth problem.
  - Red color preserves the depth invariance.
  - So, properties 1, 3, 5 are satisfied.
- The violation of color invariance, if any, may occur for properties 4.
- The violation of property 2 may show up at a later point of time).

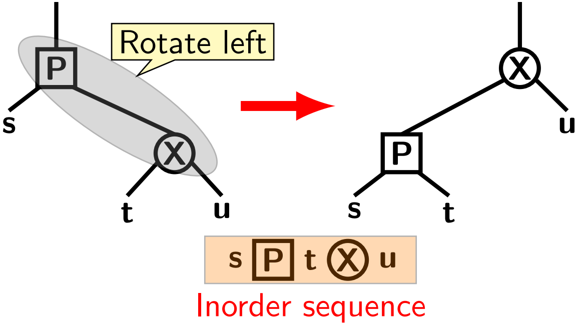
# Color Flipping Rule



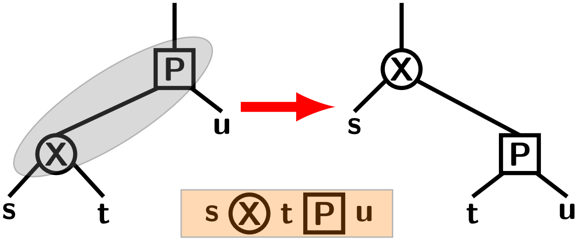

- Let $K$ be new node, its parent $P$ is colored red.
- $P$ is left child of $G$ and uncle $U$ of $K$ is red.
- Transfer (black) color of $G$ to $P$ and $U$ and recolor $G$ red.
- It pushes the problem to $G$ and parent of $G$.

# Restructuring Rules

- Four cases of restructuring may be applicable when the uncle of inserted node is black.
- Restructuring consists of rotations:
  - Right ,Left, Left right, and Right left.
- Rotation mutates the tree without losing the BST property.

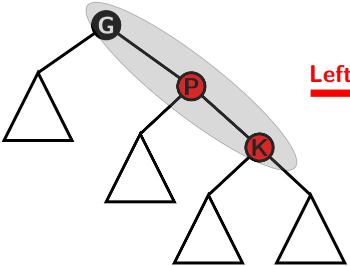

Rotate left
P
X
s
t
u
X
P
u
s
t
s P t X u
Inorder sequence

# Right Rotation

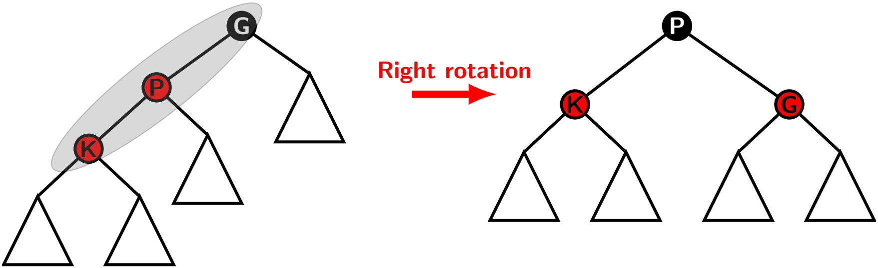



Inorder sequence

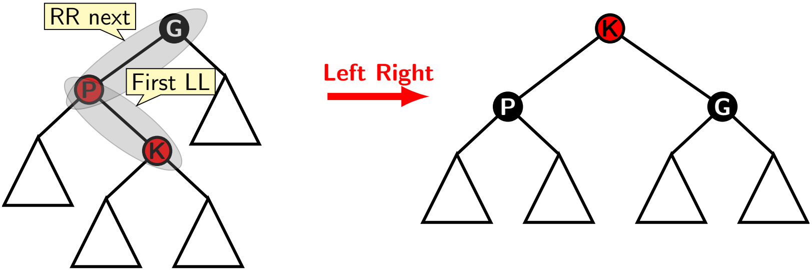
# Case 1: Left Rotation (LL) and Swap Color



# Case 2 is Symmetric to Case 1 (RR)

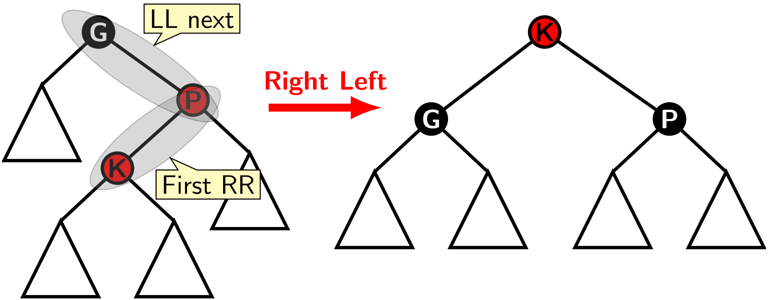


# Case 3: Left Rotate then Right Rotate (LR)



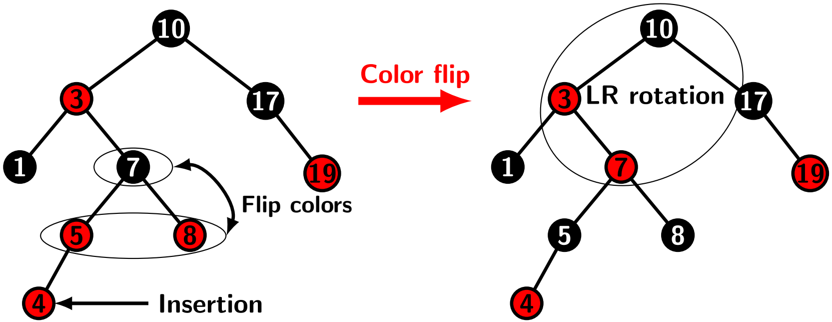

- LL leads to Case 2 which requires RR.

# Case 4 is Symmetric to Case 3 (RL)

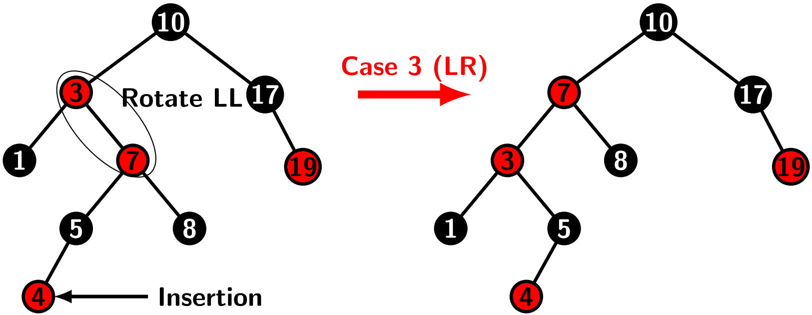



- RR leads to Case 1 which requires LL.

# Insertion of 4: Case 0 Color Flipping

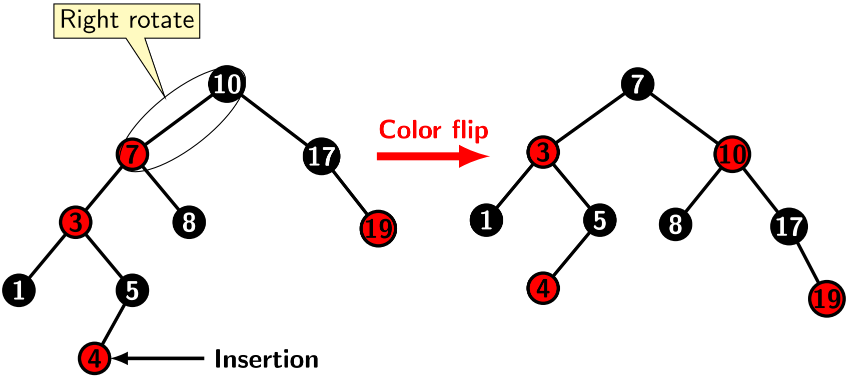



- Pushes color problem up, Case 3 results.

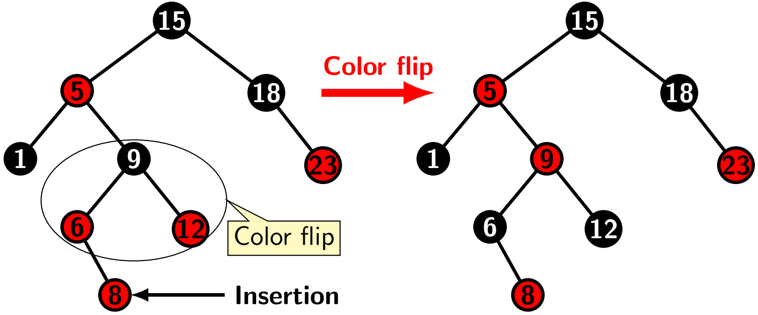



- LL leads to Case 2.

# Insertion of 4: Apply Case 2

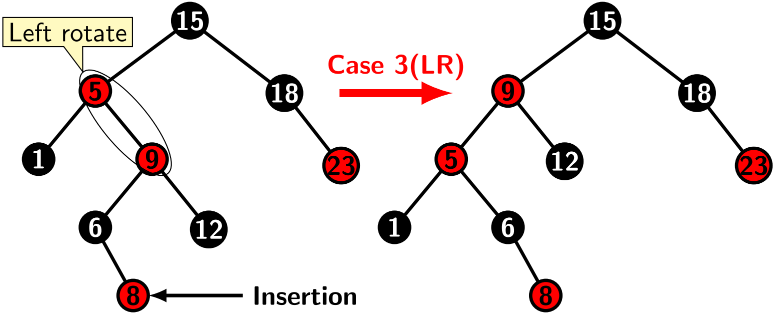


# Insertion of 8: Apply Case 0



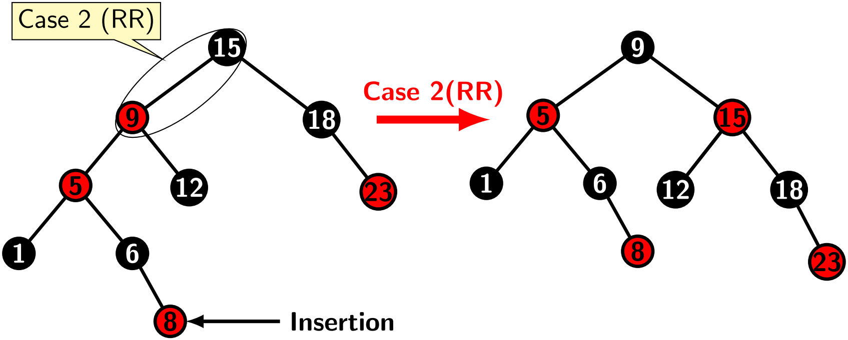

- Insertion violate the color invariants.
- Since uncle of 8 is red, recolor parent and uncle which leads to Case 4 (LR double rotation).

# Insertion of 8: Case 3 (LR)



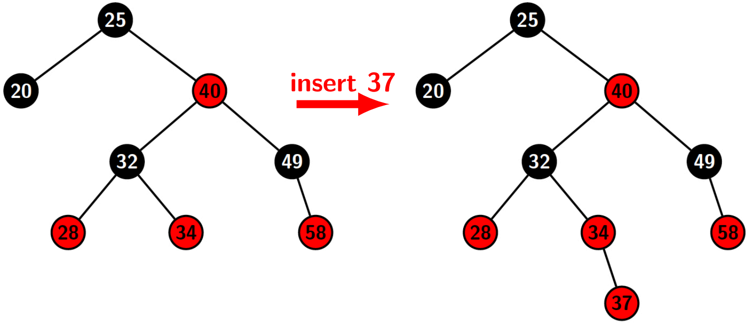

- Left rotation leads to Case 2.

# Insertion of 8: Apply Case 2

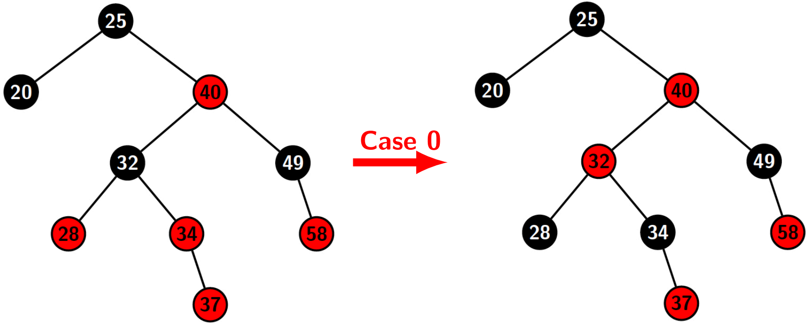



- Right rotate and swap color.
- Color and height invariants are restored.

# Insertion of 37

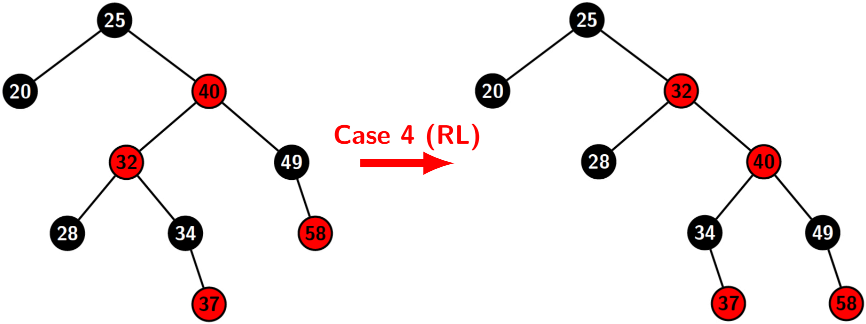


# Insertion of 37: Case 0 (Color Flip)



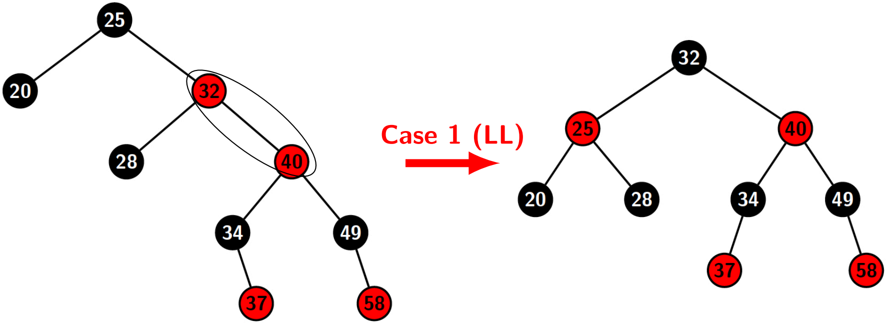

- Leads to Case 4 (LL and Color Swap)

# Insertion of 37: Apply Case 4 (LR)




- Right rotation around 32 leads to Case 1.

# Insertion of 37: Apply Case 1




- Left rotation around 32 and color swap.

# Pseudo Code

```
insertRBtree (T, x) {
    color[x] = red;
    while (x ≠ root[T] && color[p[x]] == red) {
        if (p[x] == left[G[x]]) {
            y = right[G[x]];
            if (color[y] == red) Case 1 operations;
            else if (x == right[p[x]]) {
                Case 2 operations;
                // Case 2 ==> Case 3
                Case 3 operations;
            }
        } else //if clause with left and right
            interchanged ;
    }
    color[root[T]] = black ;
}
```

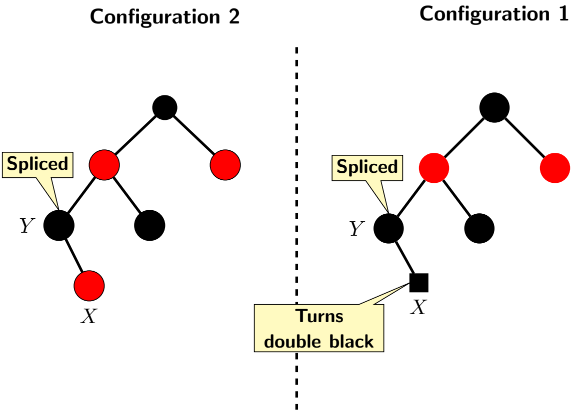
# Time Complexity

- Recoloring takes on O(1) time.
- Restructuring or a rotation involves three nodes. Hence a single rotation also takes O(1) time.
- Fixing the the color invariant (using rotations) may push the violation of property 4 one level at a time.
- In the worst case fixing operation may have to be executed O($h$) time.
- Since $h$ = O($\log n$), the time for fixing a violation of color invariance may take up to O($\log n$) time.

# Deletion in Red Black Tree

- The node to be deleted has either 1 child or has no child.
- So, just tackle the cases of deleting a leaf or a node with 1 child.
  - Let $Y$ be the node to be deleted.
  - Let $X$ be the left child of $Y$.
  - $X$ will be NIL if $Y$ has no child and NIL is considered as black.
  - $X$ will not be NIL if $Y$ has one child.

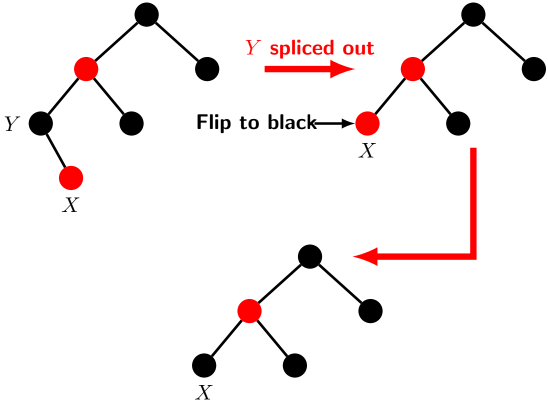
# Configurations for Deletion




- If $Y$ is red, then deleting it does not affect any color invariants. We don't consider it any further.

# Case 0: $X$ is Red

- If $Y$ is black and has a single red child $X$ then recolor $X$ black. This case is treated as **Case 0**.



# Loss of Blackness

- If $Y$ is black and not the root, then deleting it will reduce the black path lengths for all paths passing through $Y$ prior to deletion.
- If $X$ is black, the blackness of $Y$ passed on $X$ before deleting $Y$.
- So, $X$ acquires extra blackness, we say $X$ is double black.
- The approach will be get rid of extra blackness of $X$.

# Main Idea Behind the Approch

- The extra black should be pushed up until.
  - Reaching a red node that can be colored black.
  - Reaching the root where extra black can discarded.
  - Use rotation and recoloring to fix the property that a red node has only black children.

# Four Cases for Excess Blackness

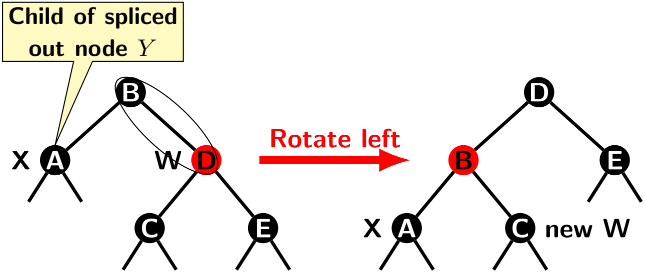
**Sibling $W$ of $X$ is right child**

Let $W$ be sibling of $X$, it is then right child of its parent as $X$ is assumed to be left child.

The four cases are:

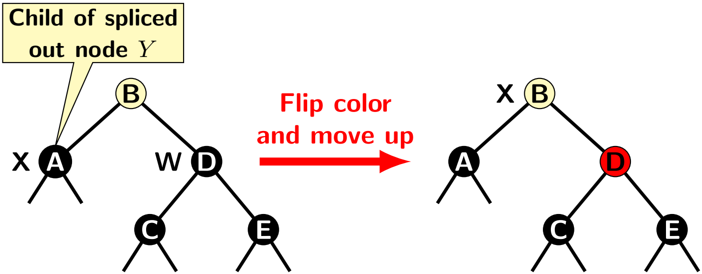
1. $W$ is red.
2. $W$ is black and its both children are black.
3. Right-left case: Let $R$ be the red child of $W$ and it also is the left child of $W$.
4. Right-right: $R$ is right child of $W$.

# Case 1: $W$ is Red




- Left rotate on B, swap colors between B (red) and D (black).
- Case 1 transforms into one of the cases: 2, 3, 4.
- Now one of the cases 2, 3, or 4 may arise as "new $W$" (C) is black.

# Case 2: Both Children of $W$ are Black

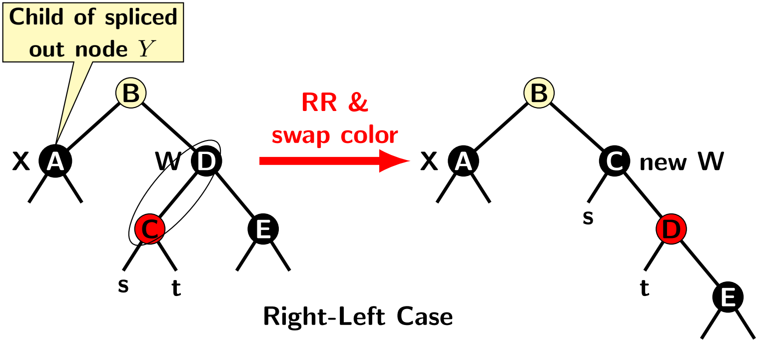



- Flip color of $W$ to red, move $X$ up.
- If "new $X$" (B) is red, flip its color to black. which absorbs excess black.

# Case 2 (contd): Both Children of $W$ are Black

- If new $X$ (B) is black, then double back of $X$ pushed up and it is a non terminating case.
- Note that if Case 2 is reached from Case 1, $X$'s parent should be red.
  - Therefore, after $X$ moves up, i.e., $X$ becomes $Parent[X]$, it will be a terminating case.

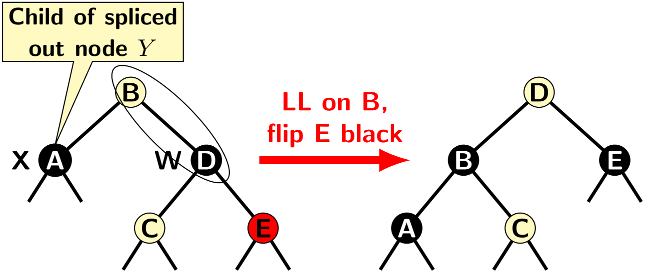
# Case 3 (Right-left): Left Child of $W$ is Red




Right-Left Case

- Rotate right on $W$ swap colors of $C$ and $D$.
- If $C$ becomes new $W$.
- Transforms to case 4.

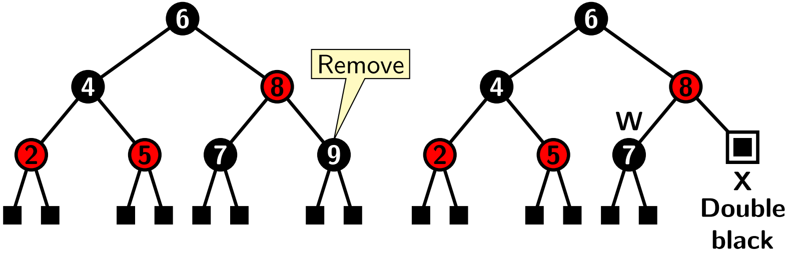
# Case 4 (Right-right): Right Child of $W$ is Red




Right-Right Case

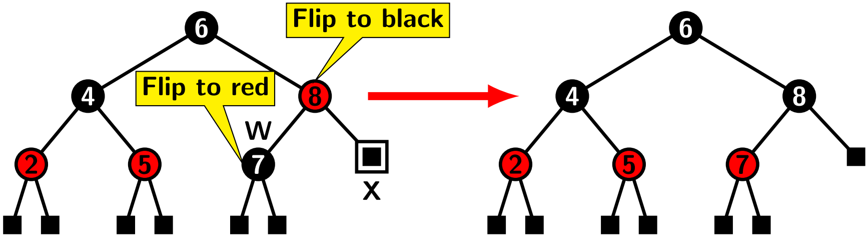
- $W$ is black but $W$'s right child is red.
- Left rotate on B and swap colors of B and D.
- Flip color of E to black.

# Example: Delete 9



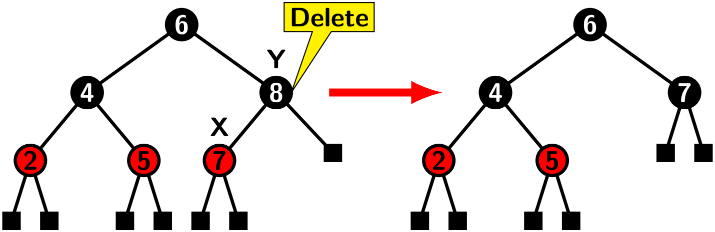

Leads to case 2 as sibling W's children are black

# Example: Delete 9 (contd)




- Double black of $X$ absorbed by its parent.
- Flipping $W$'s color decrements black path length from $W$
- But, flipping color of $W$'s parent restores the black path length from the parent.

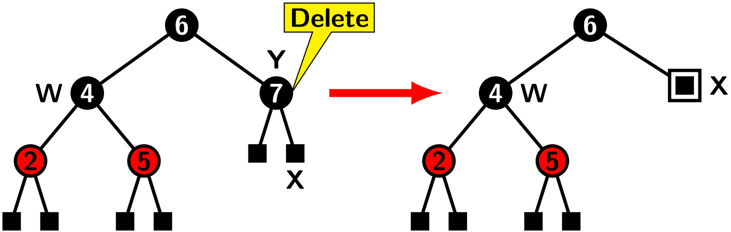
# Example: Delete 8




It is case 0 as X is red, so recolor X black

- Since, $X$ takes position of $Y$ flipping its color to red will restore black length.
- No excess blackness problem arises.

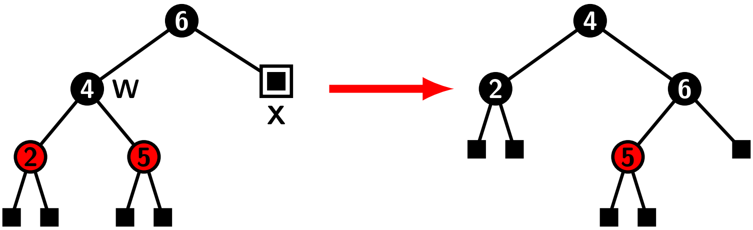
# Example: Delete 7




**Symmetric Case 4 (Left-Left)**

- Consider is symmetric Case 4 because $W$ is the left child of its parent and its left child is red.

# Example: Delete 7 (contd)




**Rotate left about 6, W's recolor 2 black**

- Rotate right about 6 and 4, swap colors of 6 and 4.
- Both 6 and 4 are black, so color swap does not matter.
- Flipping coloring of 2 to black absorbs the excess black from $X$.

# Complexity

- Red black tree is another interesting way of keeping a BST balanced.
- It uses rotations like AVL tree, but much more sparingly.
- It requires an additional information field for keeping color information. However, the information is just 1 bit.
- It does not require height recomputation as it was required in AVL tree each time an insertion or deletion happen.
- The asymptotic time complexity remains O($h$) where $h$ is the height of the tree.