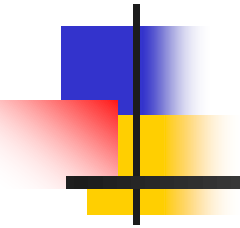
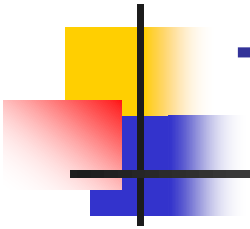
Algorithms and Data Structures
Lecture VIII



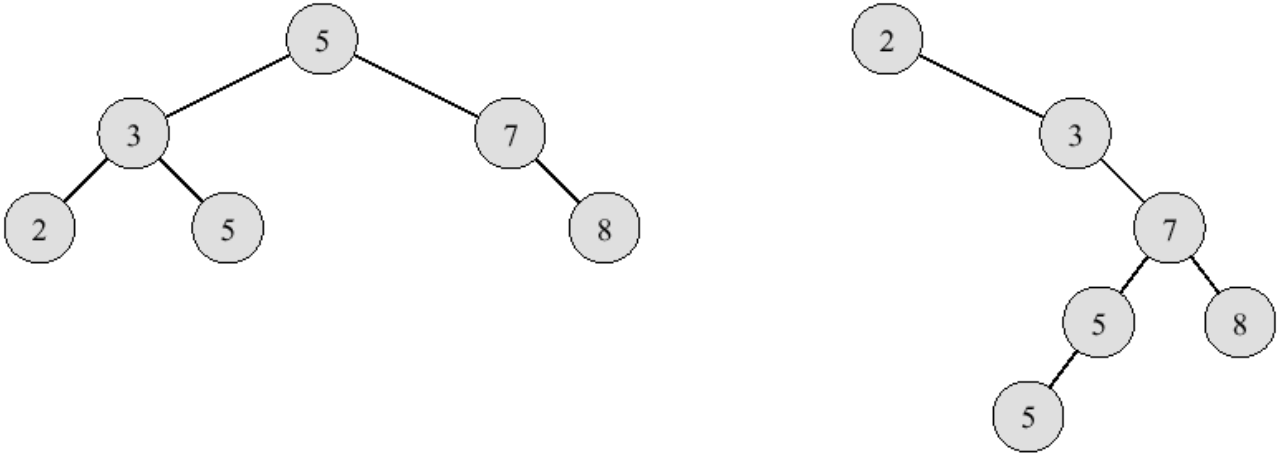


This Lecture

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion

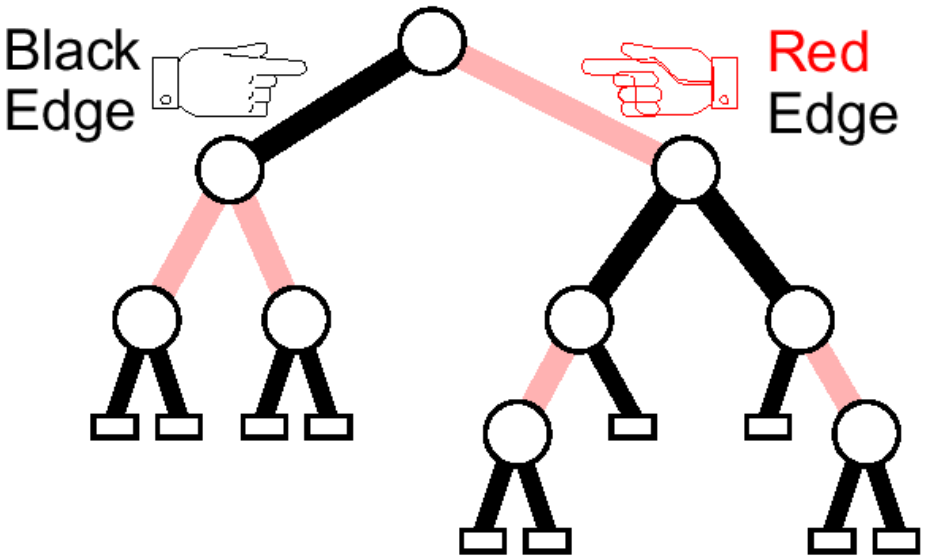
Balanced Binary Search Trees

- Problem: worst-case execution time for dynamic set operations is $\Theta(n)$
- Solution: balanced search trees guarantee small (logarithmic) height!



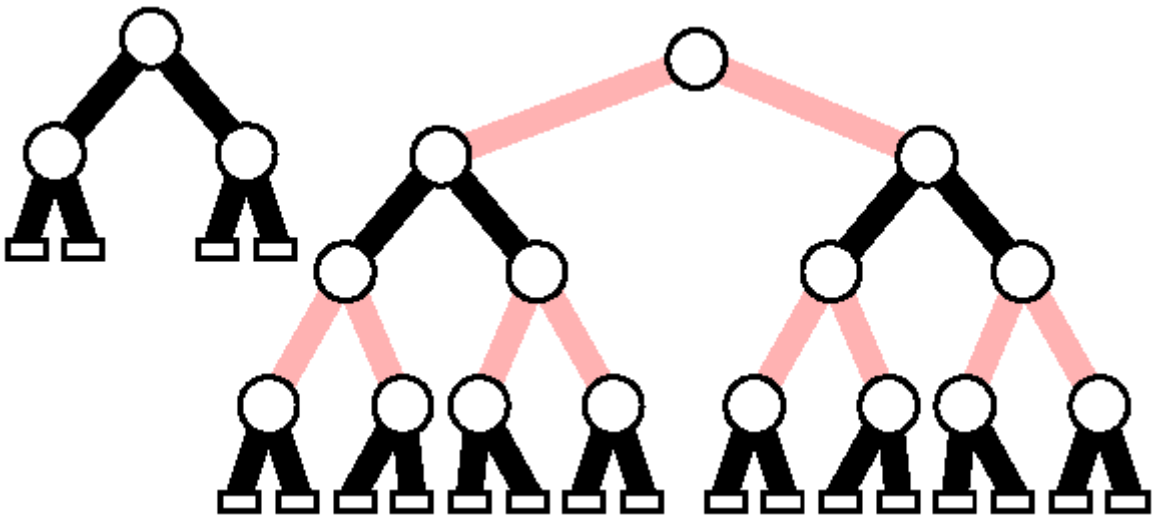
Red/Black Trees

- **A red-black** tree is a binary search tree with the following properties:
  - edges (nodes) are colored **red** or **black**
  - **edges connecting leaves are black**
  - **no two consecutive red edges** on any root-leaf path
  - **same number of black edges** on any root-leaf path (= **black height** of the tree)



More RB-Tree Poperties

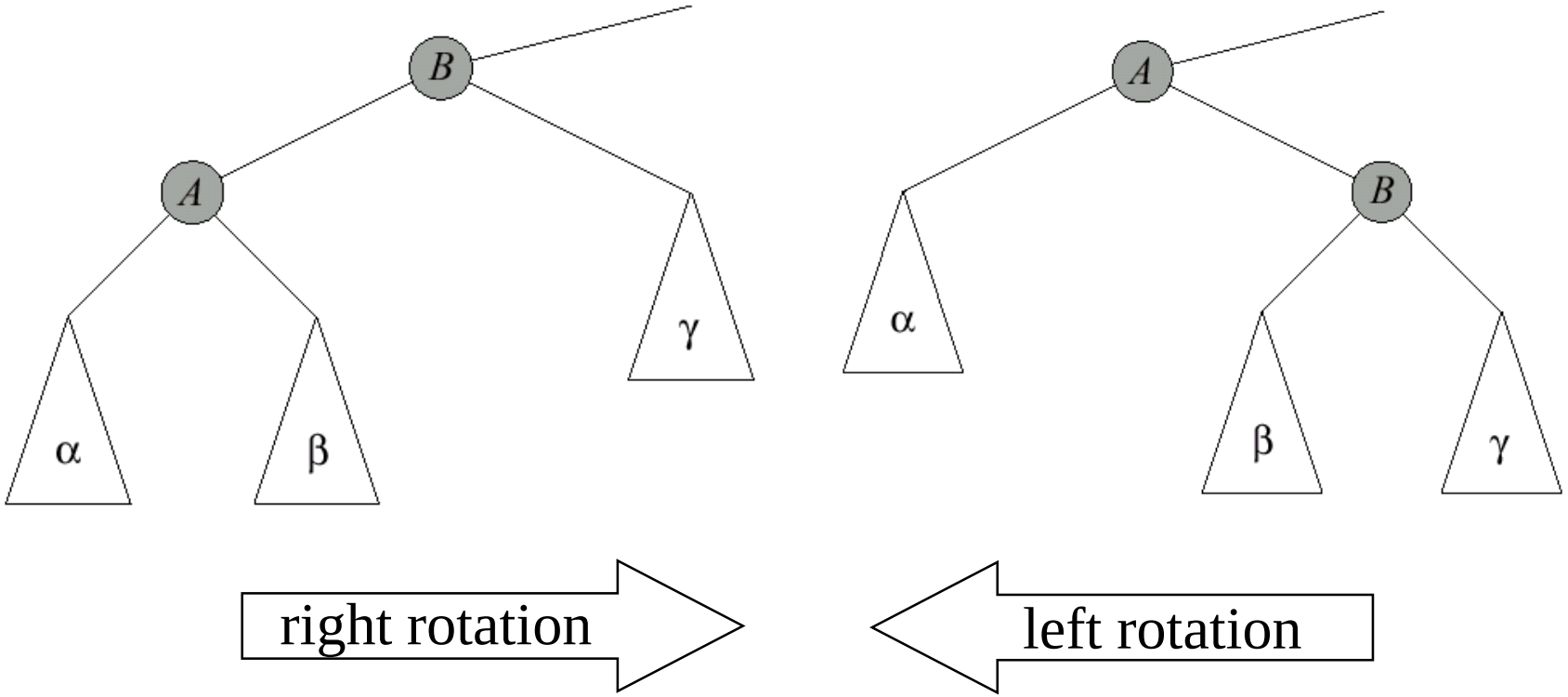
- Some measures
  - n – # of internal nodes
  - l – # leaves $(= n + 1)$
  - h – height
  - bh – black height
- Property 1: $2^{bh} \leq n+1 \leq 4^{bh}$



More RB-Tree Poperties (2)

- Property 2: $\frac{1}{2}\lg(n+1) \le bh \le \lg(n+1)$
- Property 3: $\lg(n+1) \le h \le 2\lg(n+1)$
- Operations in the binary-search tree (search, insert, delete, ...) can be accomplished in $O(h)$ time
- Consequently, the RB-tree is a binary search tree, whose height is bound by 2 log(n+1), the operations run in O(log n)

Rotation



Rotation (2)

- The basic operation for maintaining balanced trees
- Maintains inorder key ordering

$$\forall a \in \alpha, b \in \beta, c \in \gamma$$

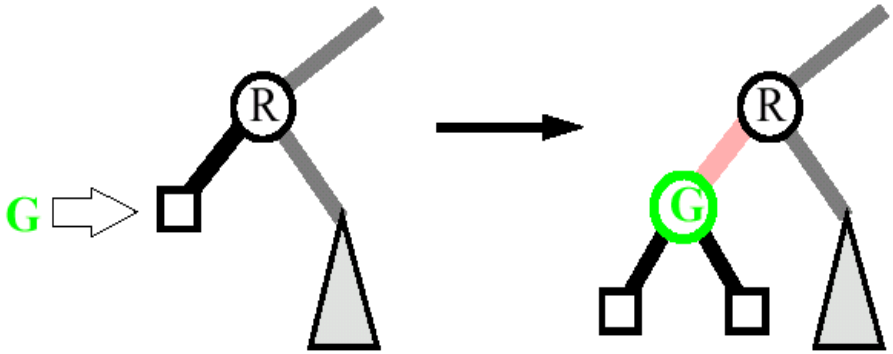
we can state that

$$a \le A \le b \le B \le c$$

- After left rotation (right rotation has the opposite effect)
  - Depth(α) decreases by 1
  - Depth(β) stays the same
  - Depth(γ) increases by 1
- Rotation takes $O(1)$ time

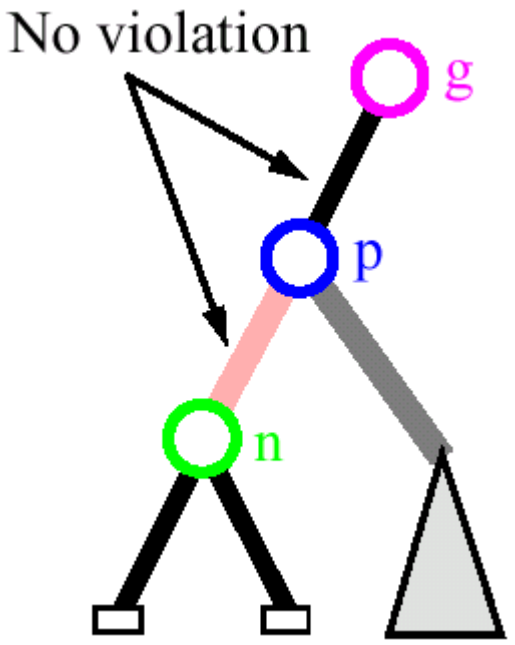
Insertion in the RB-Trees

- Perform a standard search to find the leaf
- Replace the leaf with an internal node with the new key. The new node has two children (leaves) with black incoming edges
- Color the incoming edge of the new node red
- If the parent has an incoming red edge, we have two consecutive red edges! We must **re-organize** a tree to remove that violation.



Insertion, Plain and Simple

- Let
  - n … the new node
  - p … its parent
  - g … its grandparent
- **Case 0**: Incoming edge of **p** is black

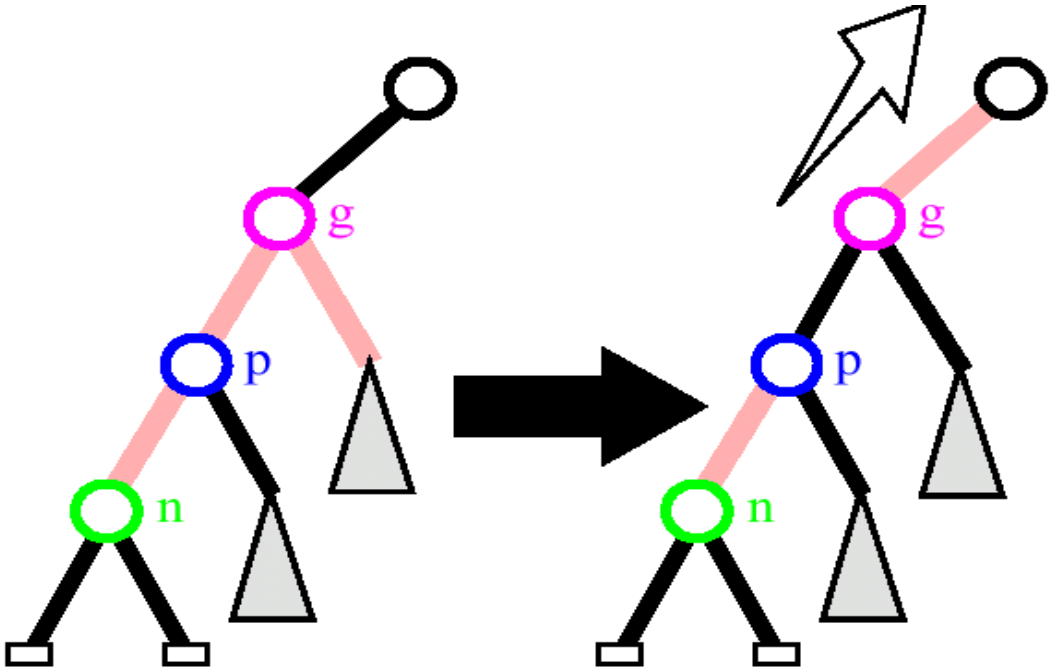


Insertion algorithm stops here.

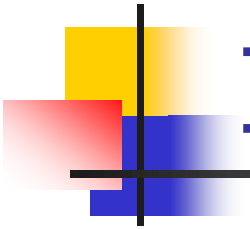
No further manipulation of the tree necessary!

Insertion: Case 1

- Case 1: Incoming edge of **p** is red and its sibling (uncle of n) has a red incoming edge
  - a tree rooted at g is balanced enough



Insertion: Case 1 (2)



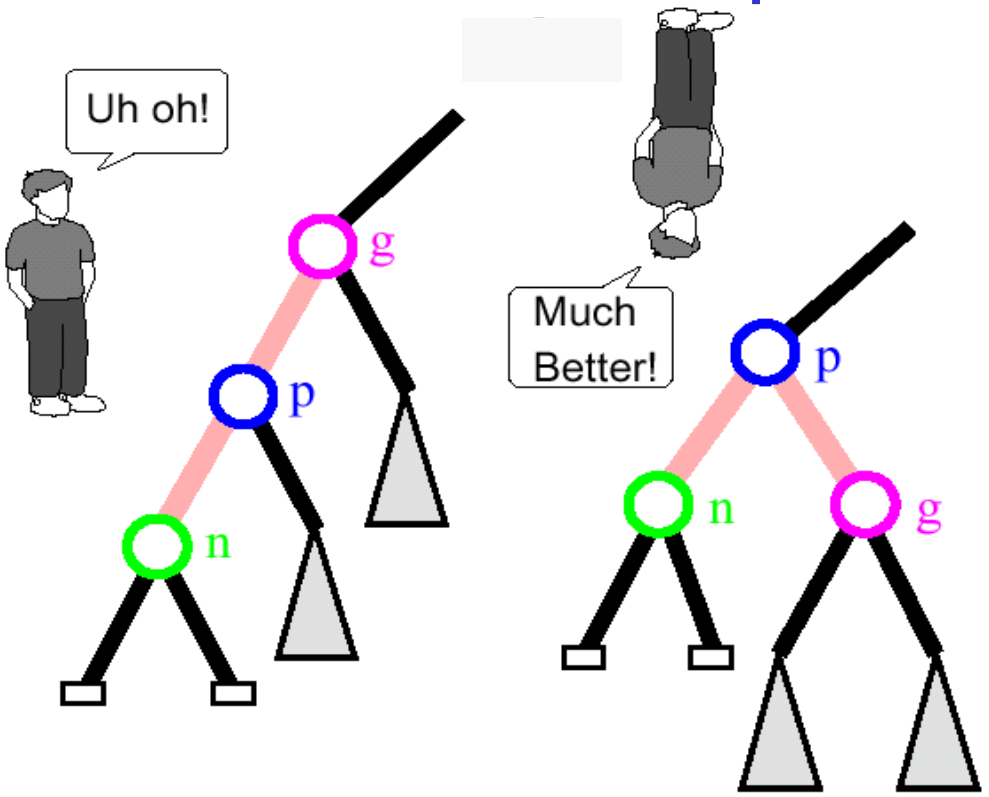
- We call this a **promotion**
- Note how the black depth remains unchanged for all of the descendants of **g**
- This process will continue upward beyond **g** if necessary; rename **g** as **n** and repeat

Insertion: Case 3

- Case 3: Incoming edge of **p** is red, its sibling is black and n is left child of p

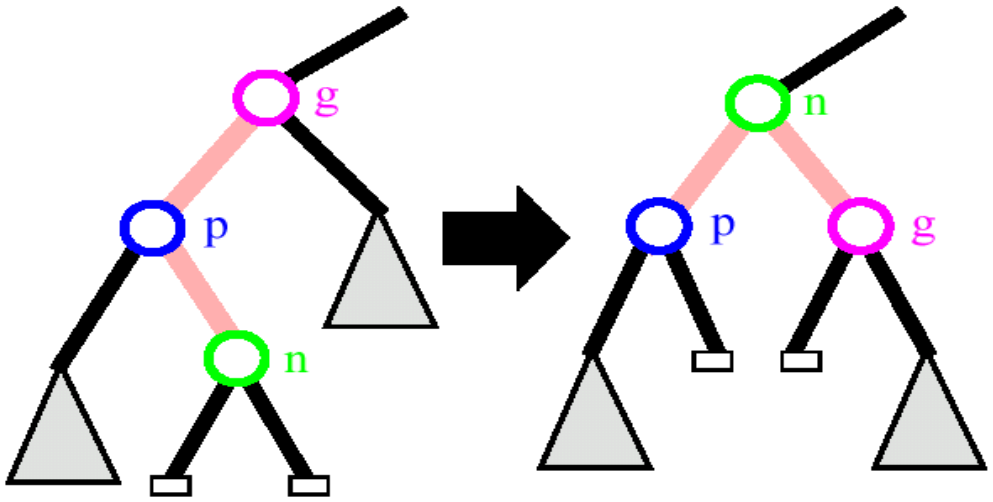
We do a "**right rotation**"

- No further work on tree is necessary
- Inorder remains unchanged
- Tree becomes more balanced
- No two consecutive red edges!



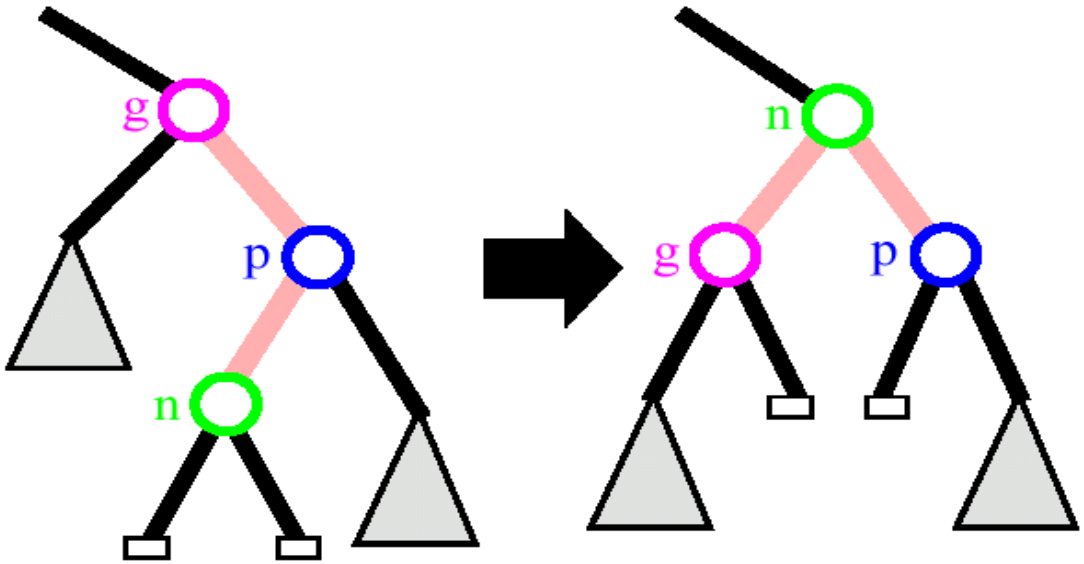
Insertion: Cases 2

- Case 2: Incoming edge of **p** is red, its sibling is black and n is right child of p
  - **We must perform a "double rotation" –** left rotation around p (we end in case 3) followed by right rotation around n



Insertion: Mirror cases

- All three cases are handled analogously if p is a right child
  - For example, case 2 and 3 – **right-left double rotation**



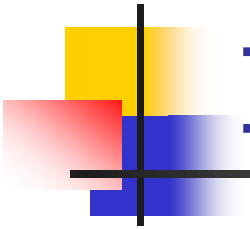
Insertion: Pseudo Code

```
RB-INSERT(T, x)
 1  TREE-INSERT(T, x)
 2  color[x] ← RED
 3  while x ≠ root[T] and color[p[x]] = RED
 4        do if p[x] = left[p[p[x]]]
 5              then y ← right[p[p[x]]]
 6                   if color[y] = RED
 7                      then color[p[x]] ← BLACK
 8                           color[y] ← BLACK
 9                           color[p[p[x]]] ← RED
10                           x ← p[p[x]]
11                      else  if x = right[p[x]]
12                               then x ← p[x]
13                                    LEFT-ROTATE(T, x)
14                            color[p[x]] ← BLACK
15                            color[p[p[x]] ← RED
16                            RIGHT-ROTATE(T, p[p[x]])
17              else  (same as then clause [line 5]
                       with "right" and "left" exchanged)
18 color[root[T]] ← BLACK
```

Case 1

Case 2

Case 3

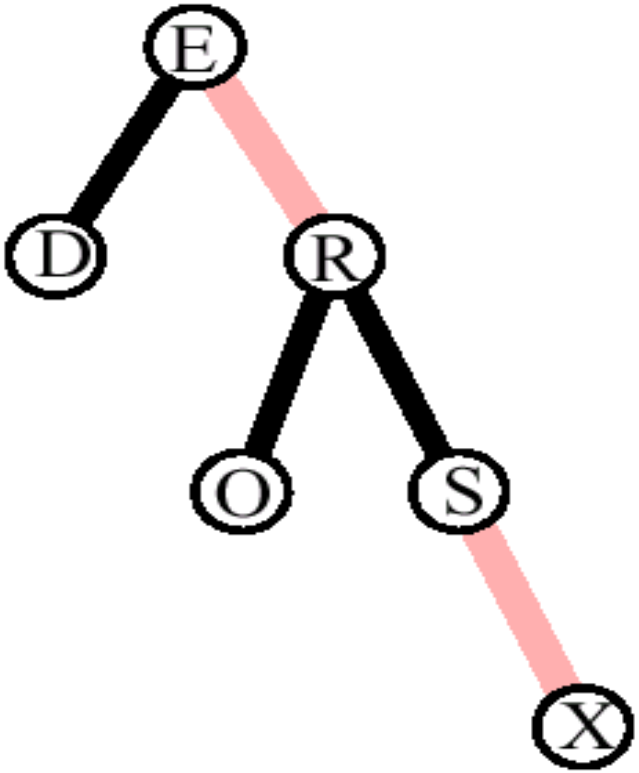


Insertion Summary

- If **two red edges** are present, we do either
  - **a restructuring** (with a simple or double rotation) and **stop** (cases 2 and 3), or
  - **a promotion** and **continue** (case 1)
- A **restructuring** takes constant time and is performed at most once. It reorganizes an off-balanced section of the tree
- **Promotions** may continue up the tree and are executed $O(\log n)$ times (height of the tree)
- The **running time** of an insertion is $\boldsymbol{O(\log n)}$

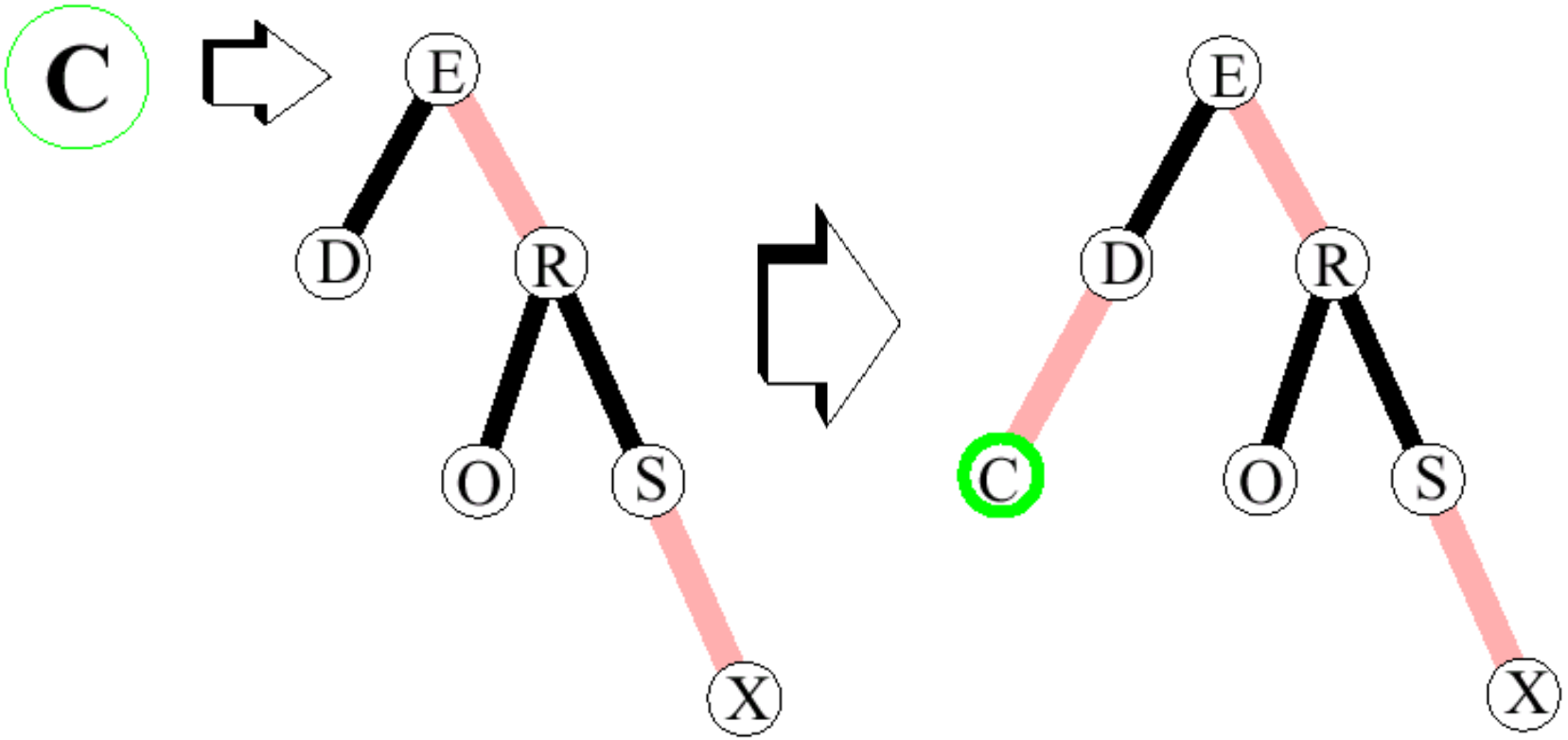
An Insertion Example

- Inserting "REDSOX" into an empty tree

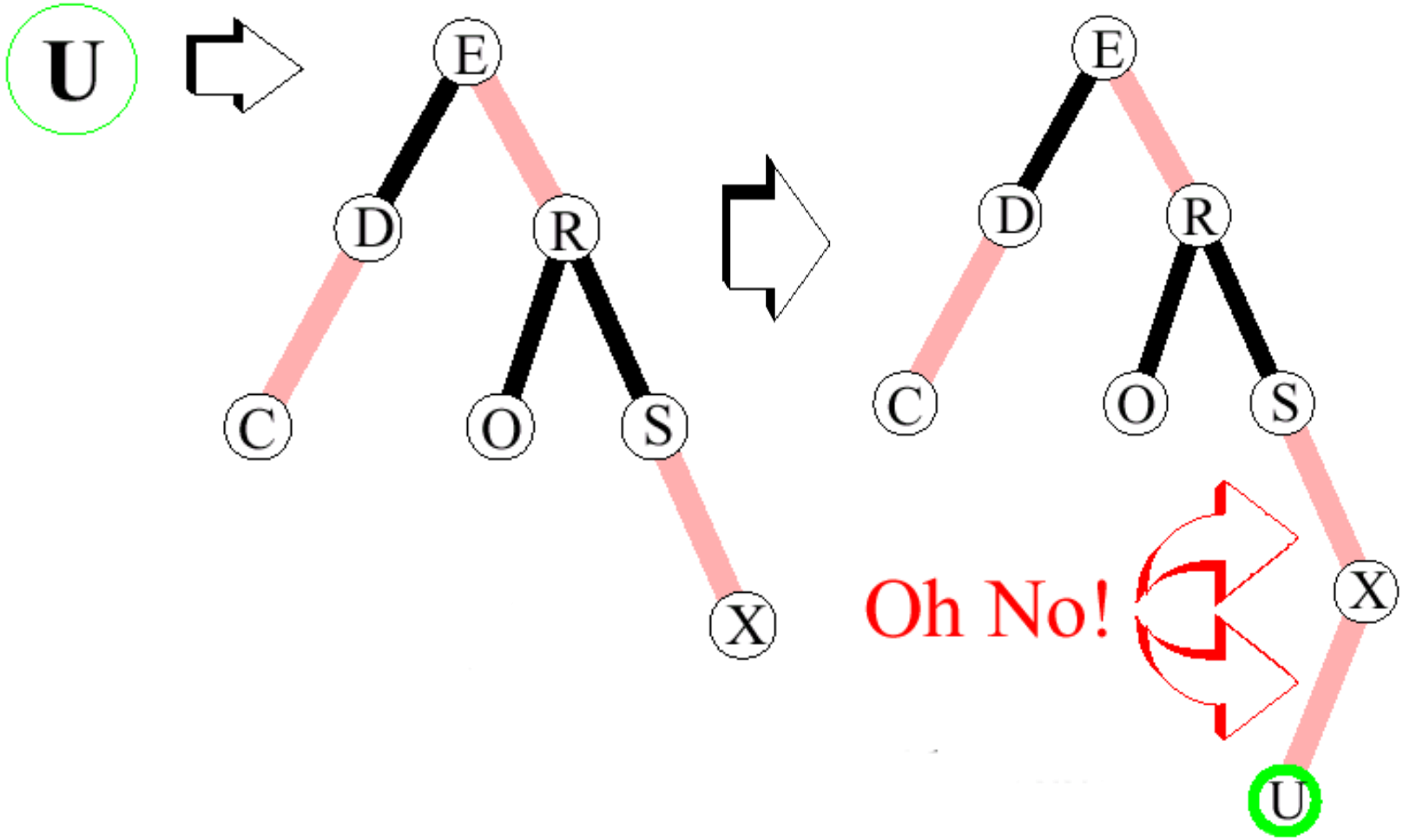


- Now, let us insert "CUBS"

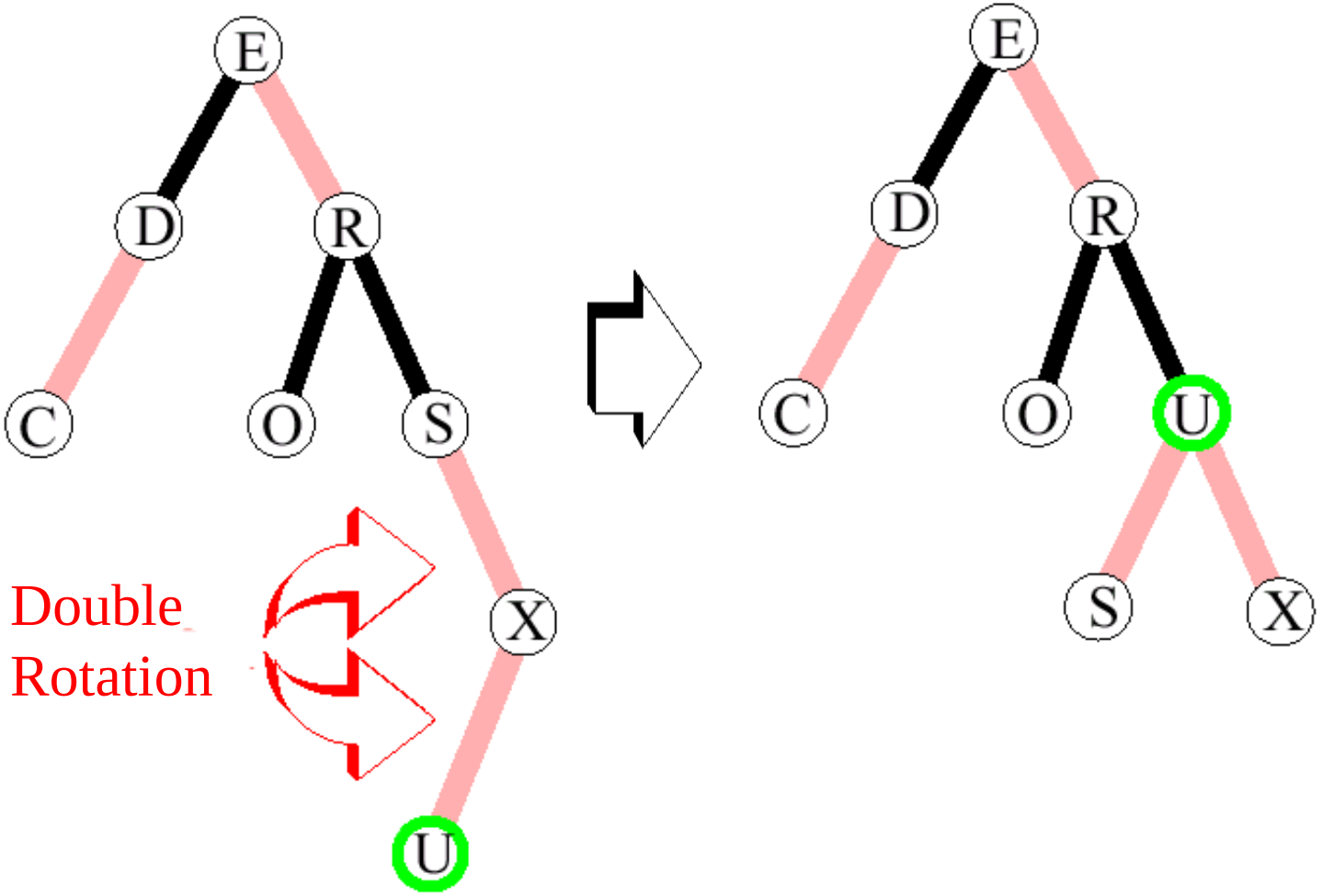
Example C



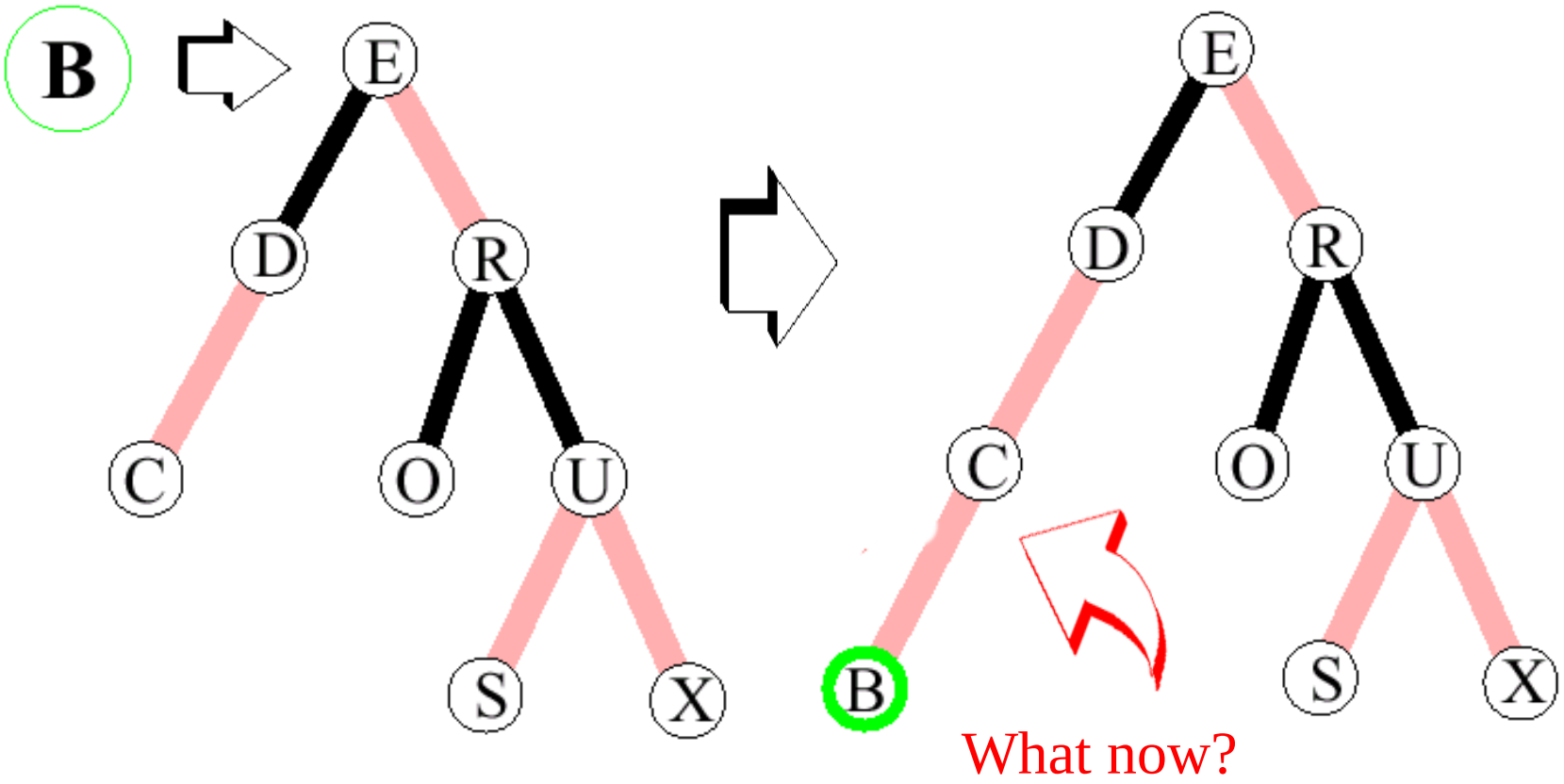
Example U



Example U (2)

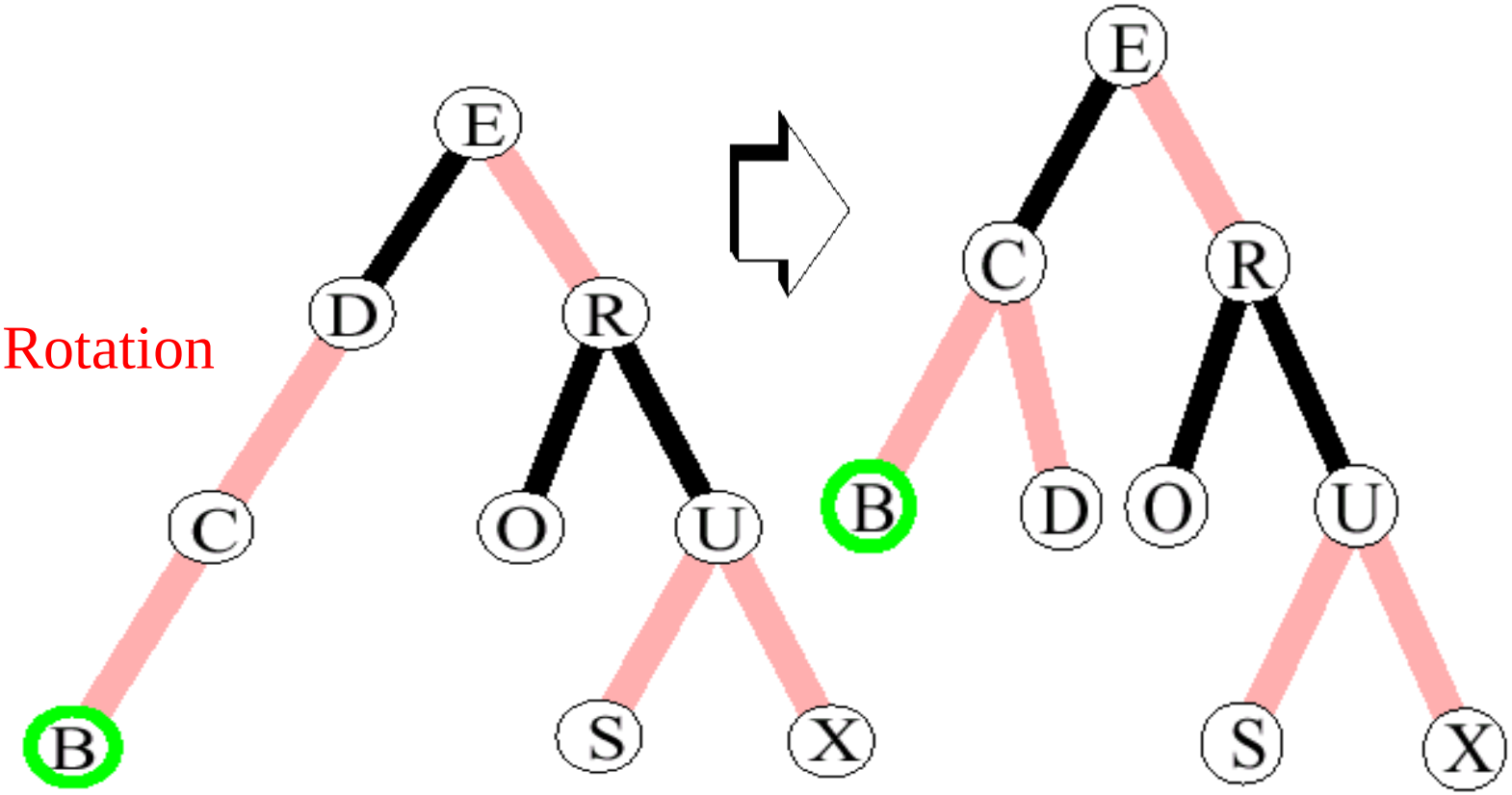


Example B

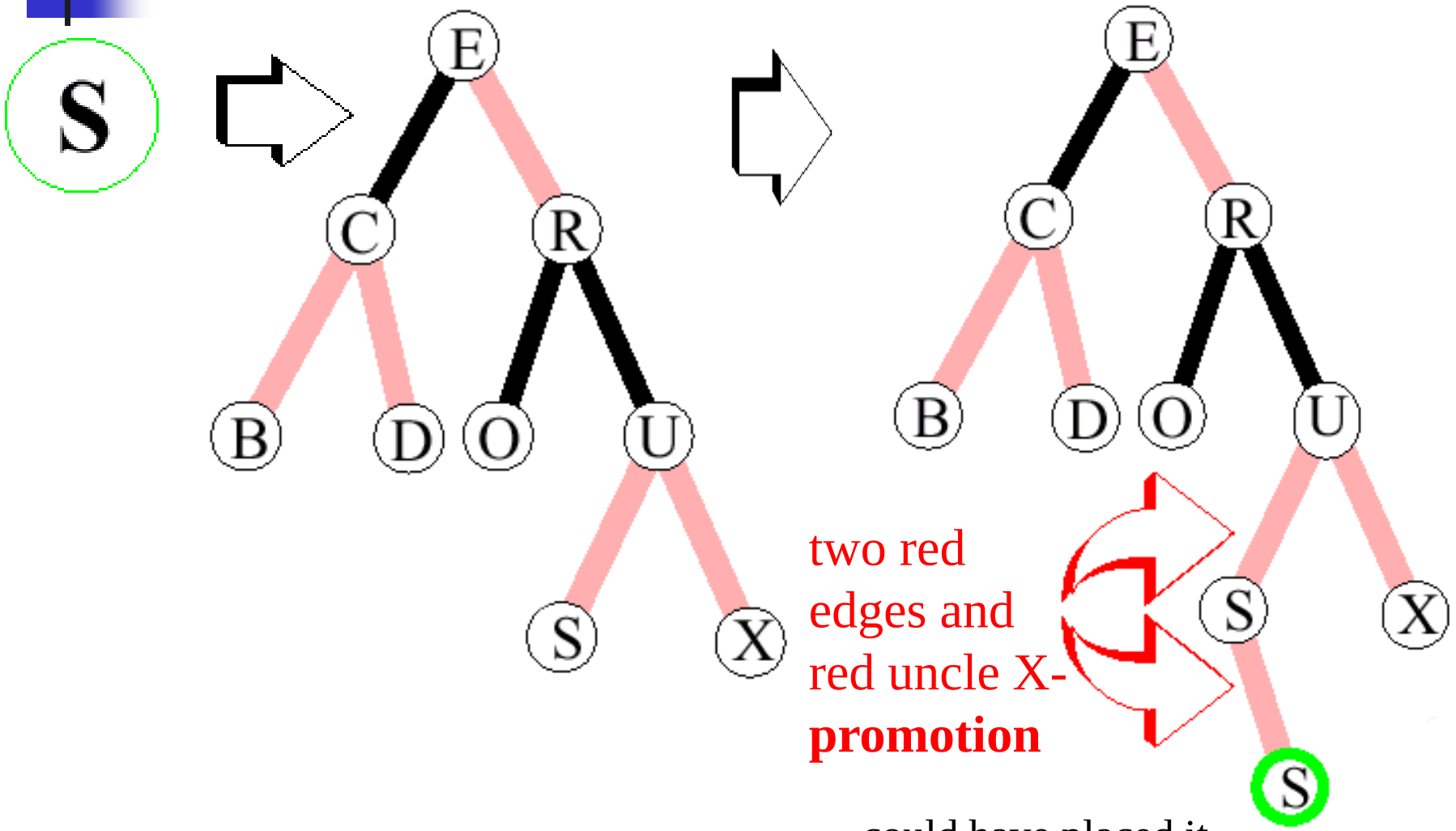


Example B (2)

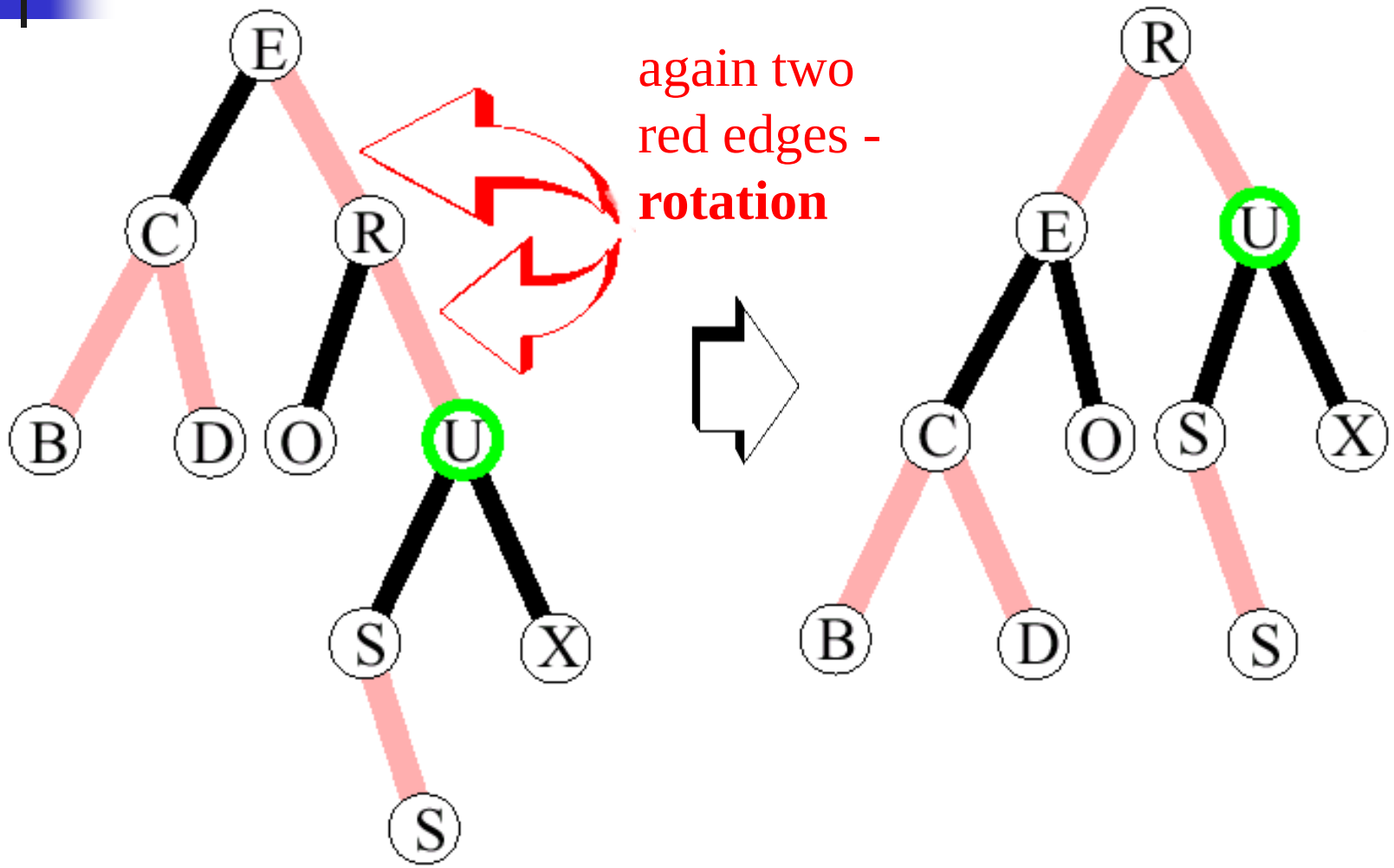
Right Rotation on D



Example S

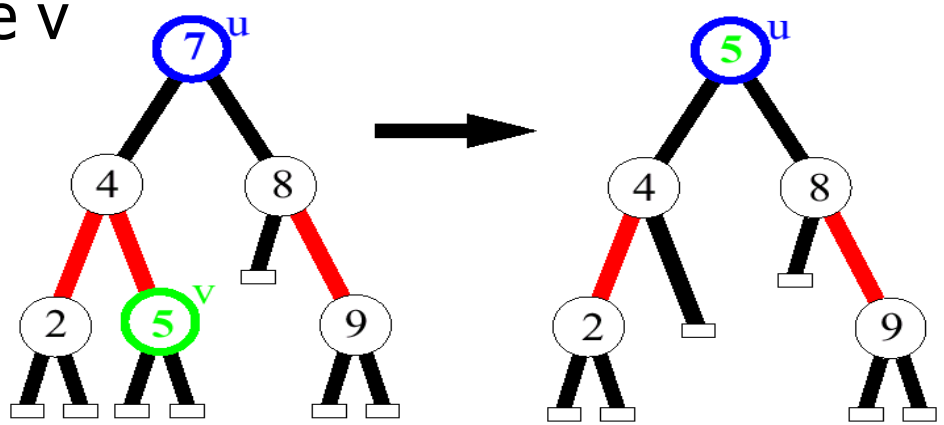

S
E
C
R
B
D
O
U
S
X
E
C
R
B
D
O
U
S
X
S
two red
edges and
red uncle X-
promotion
could have placed it
on either side

Example S (2)



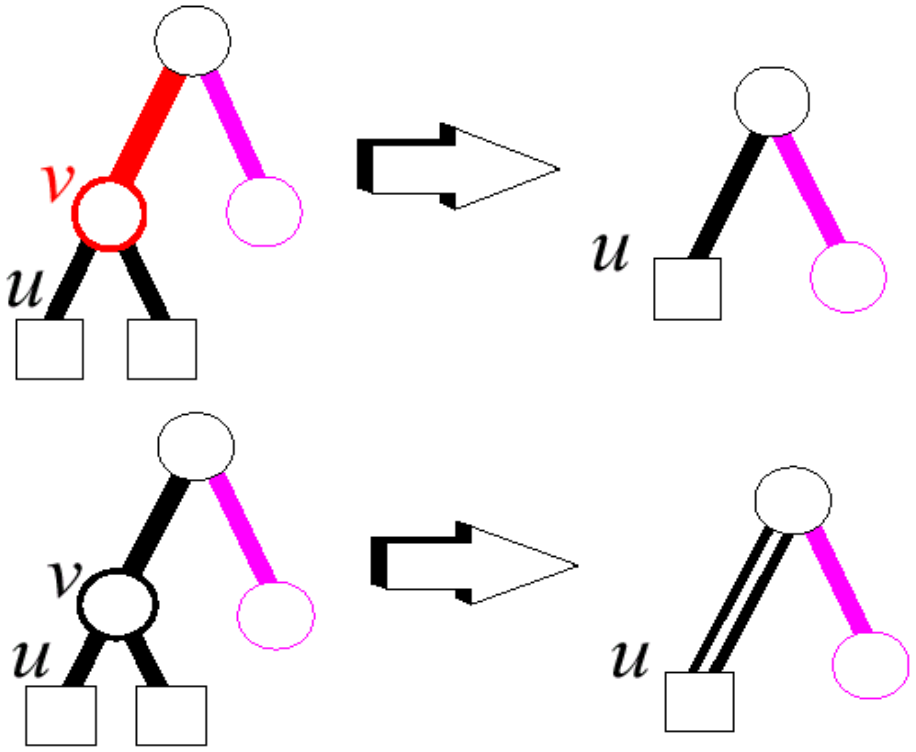
Deletion

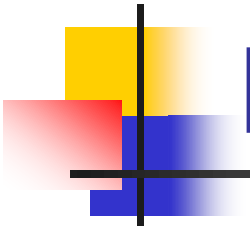
- As with binary search trees, we can always delete a node that has at least one external child
- If the key to be deleted is stored at a node that has no external children, we move there the key of its inorder predecessor (or successor), and delete that node instead
  - **Example:** to delete key 7, we move key 5 to node u, and delete node v



Deletion Algorithm

1. Remove ***v***
2. If ***v*** was red, we are done. Else, color ***u*** **double black**.



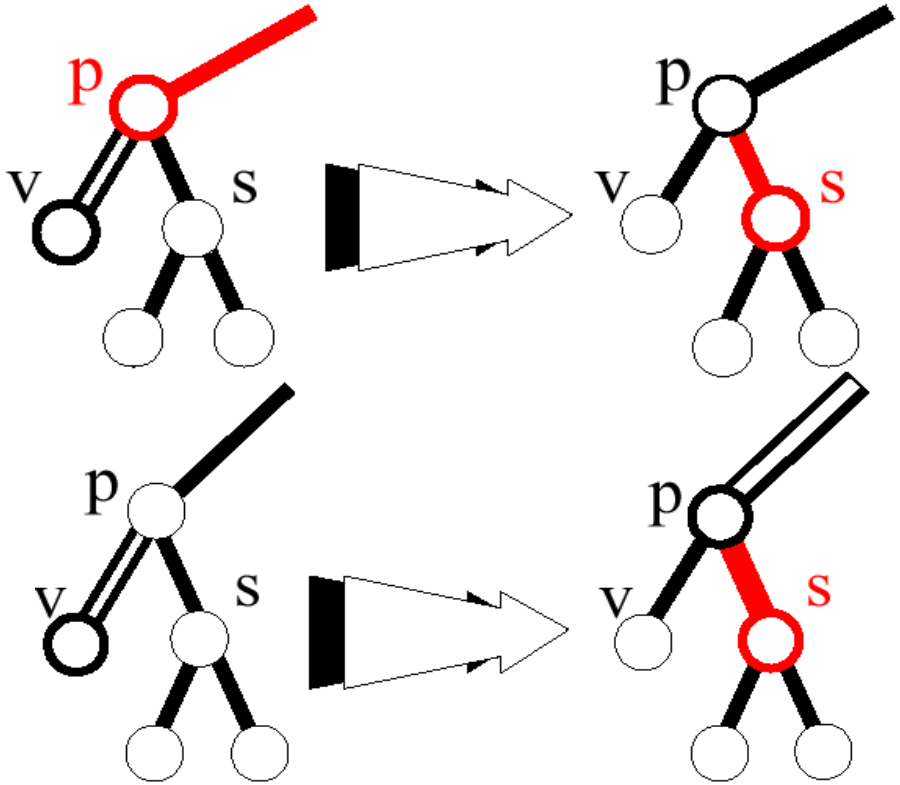


Deletion Algorithm (2)

- How to eliminate double black edges?
  - The intuitive idea is to perform a **"color compensation"**
    - Find a red edge nearby, and change the pair (**red**, **double black**) into (**black, black**)
  - As for insertion, we have two cases:
    - **restructuring**, and
    - **recoloring**
  - Restructuring resolves the problem locally, while recoloring may propagate it two levels up
  - Slightly more complicated than insertion

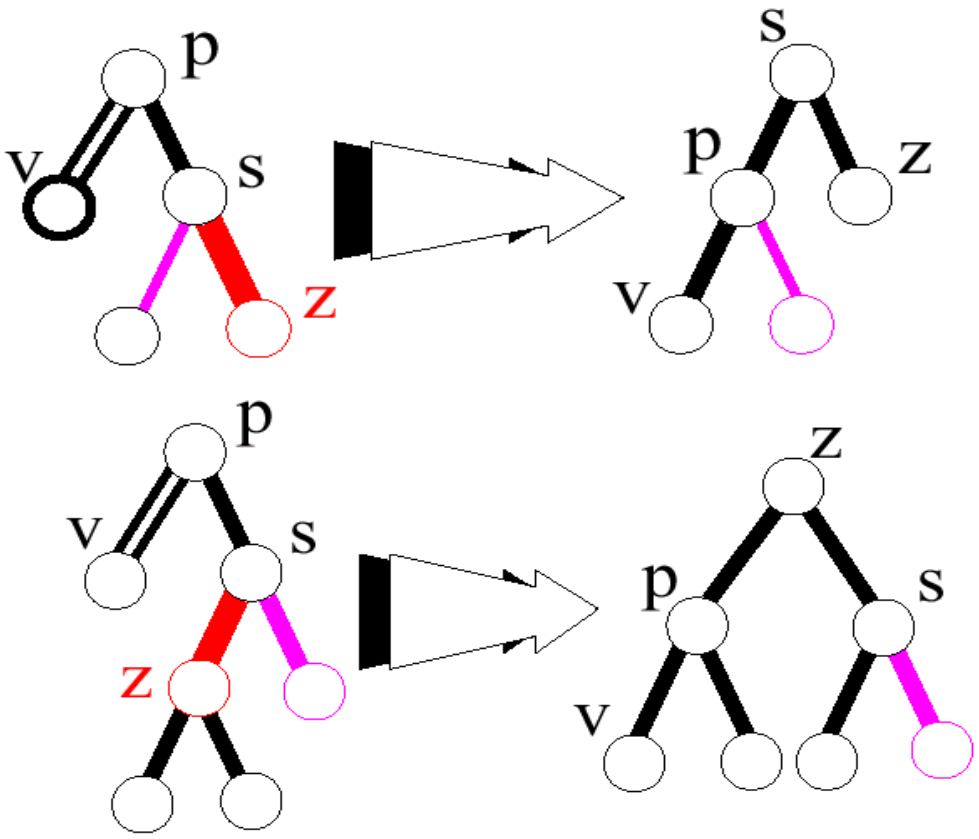
Deletion Case 2

- If sibling and its children are black, perform a **recoloring**
- If parent becomes **double black**, **continue** upward



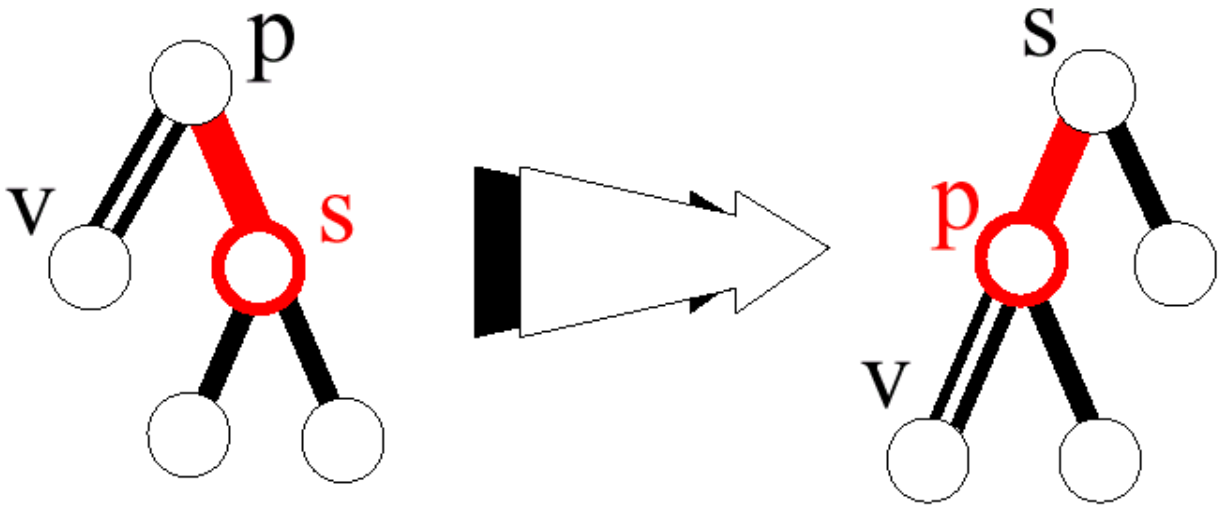
Deletion: Cases 3 and 4

- If sibling is black and one of its children is red, perform a **restructuring**



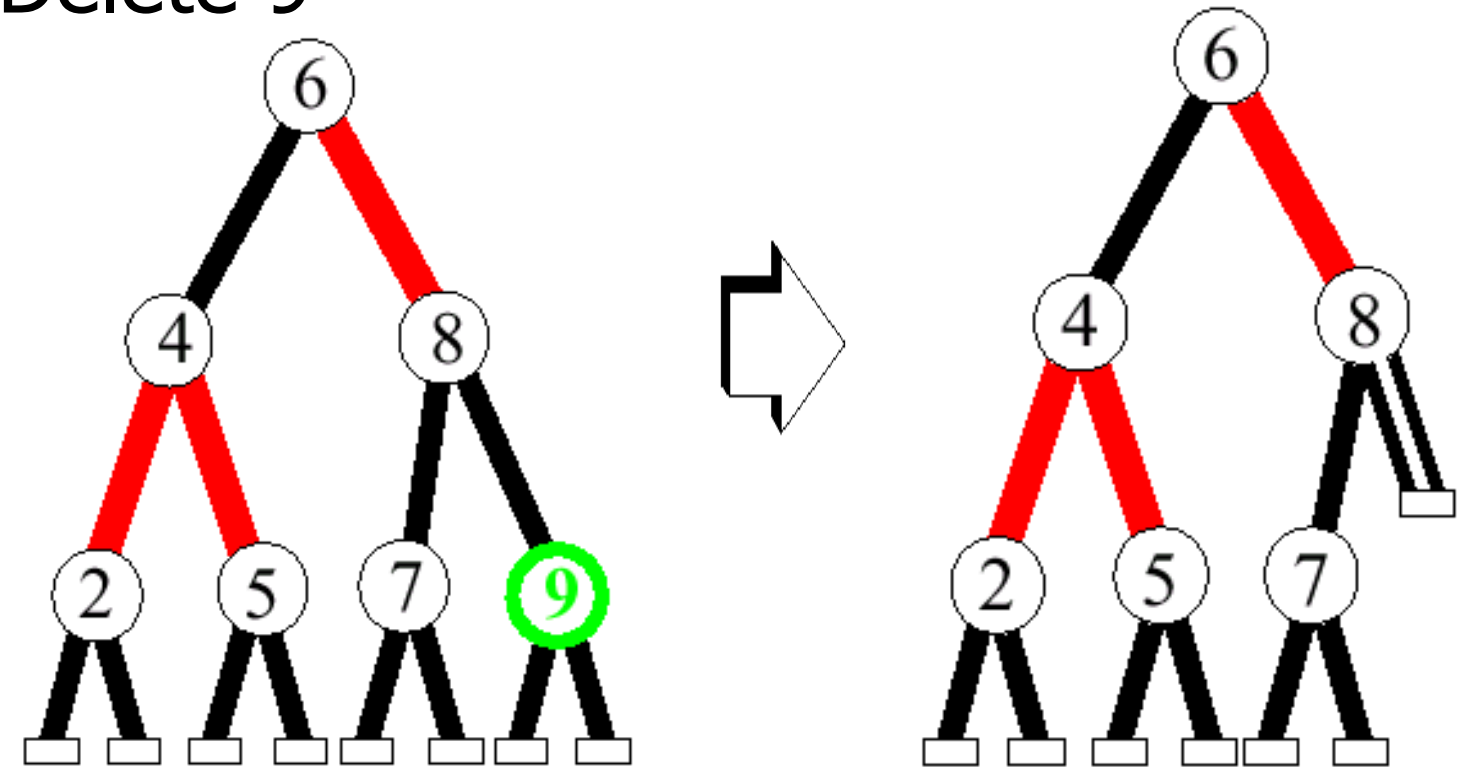
Deletion Case 1

- If sibling is red, perform a rotation to get into one of cases 2, 3, and 4
- If the next case is 2 (recoloring), there is no propagation upward (parent is now red)



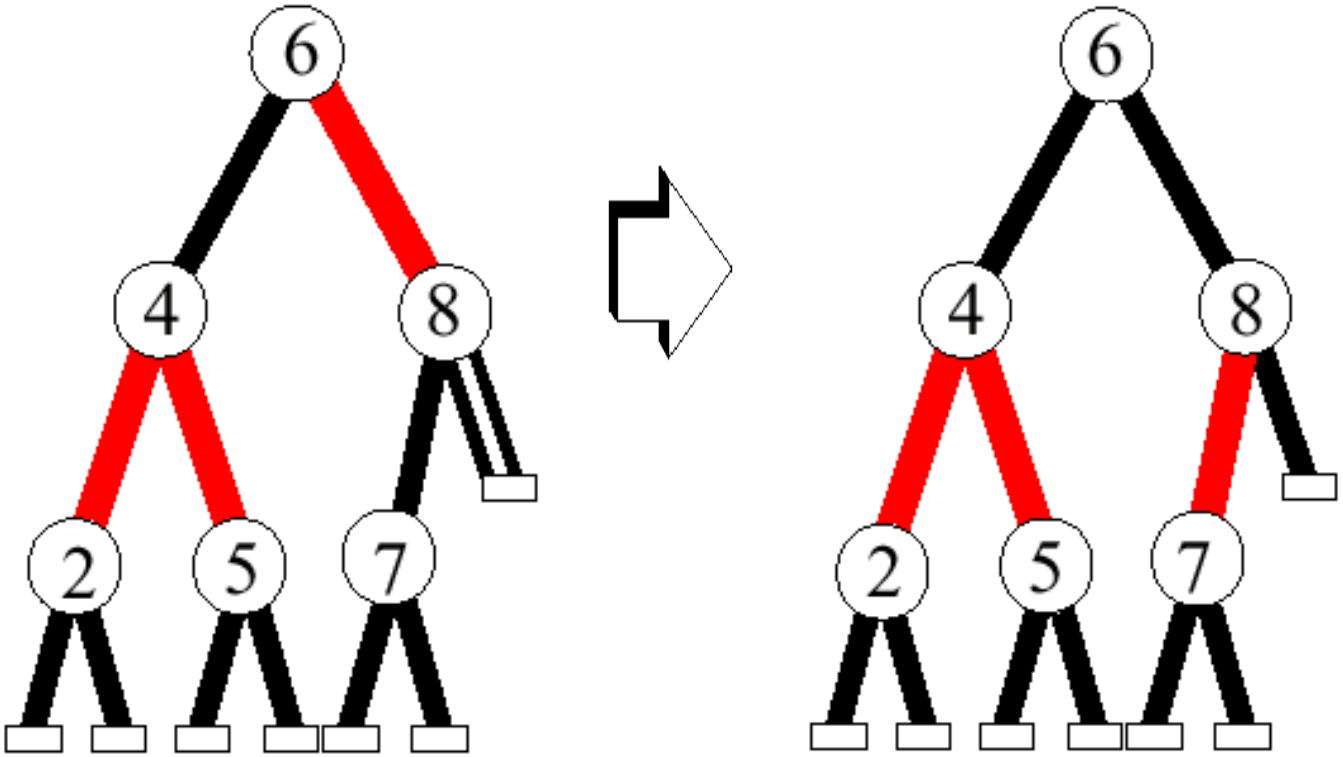
A Deletion Example

- Delete 9



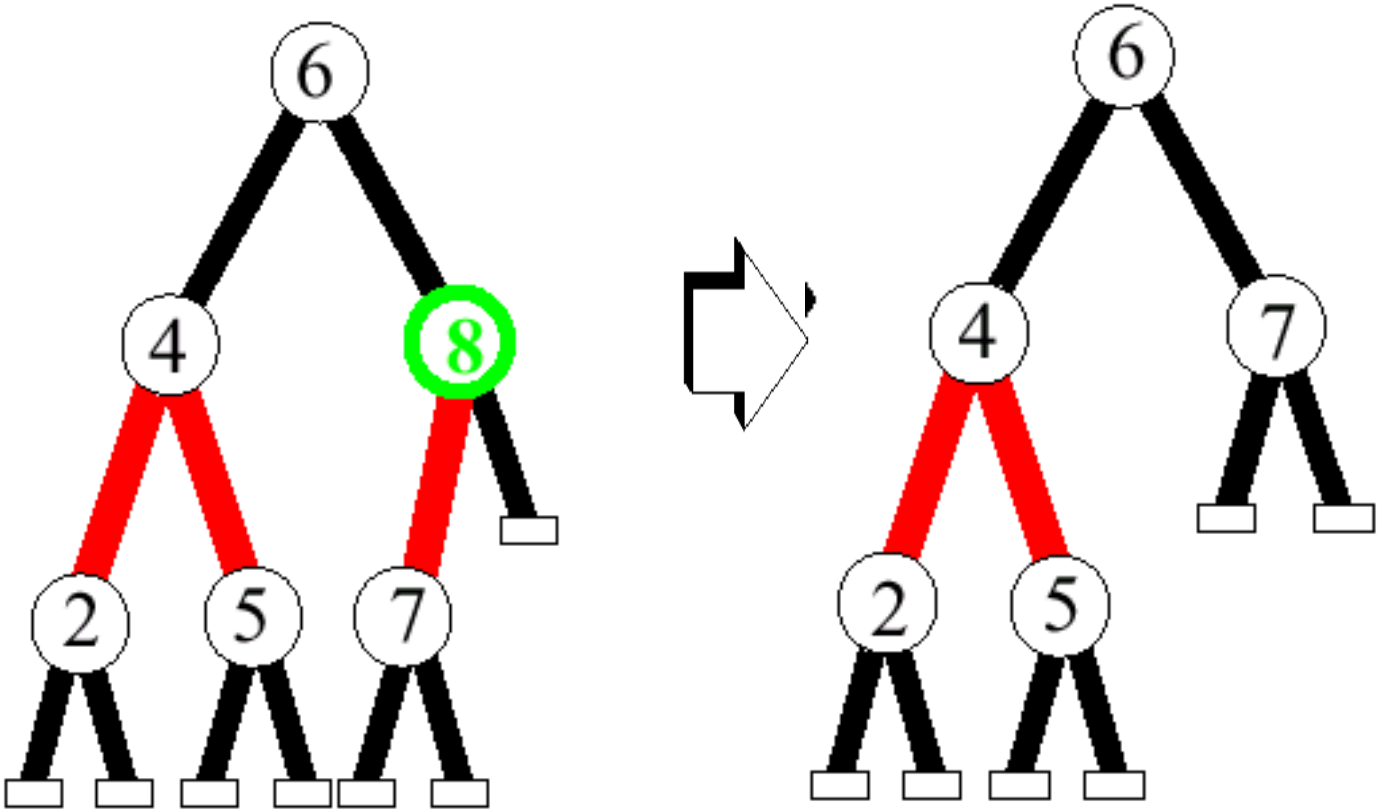
A Deletion Example (2)

- Case 2 (sibling is black with black children) – recoloring



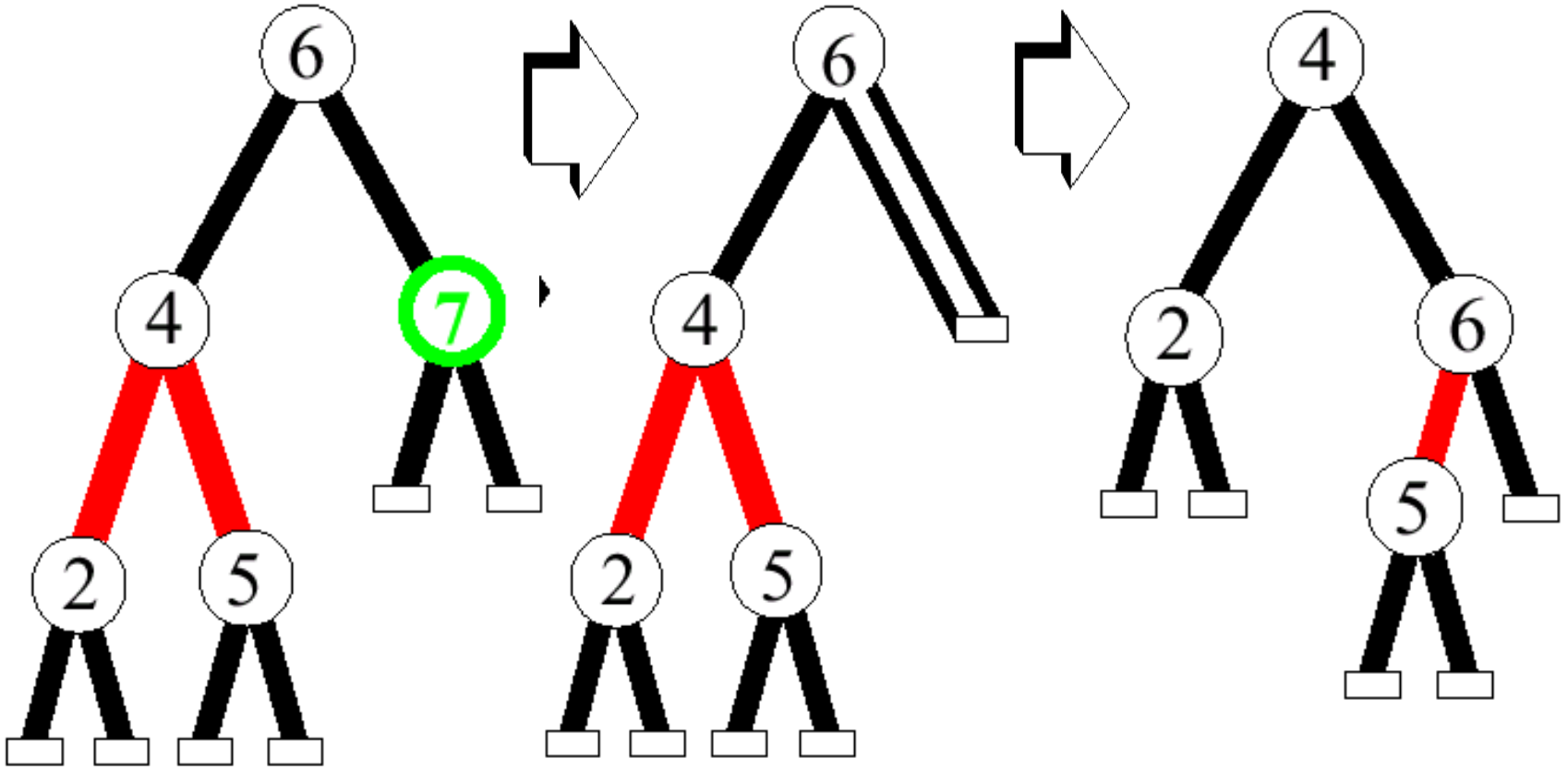
A Deletion Example (3)

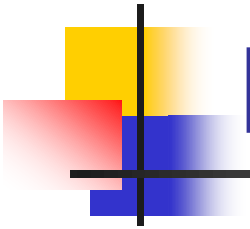
- Delete 8: no double black



A Deletion Example (4)

- Delete 7: restructuring



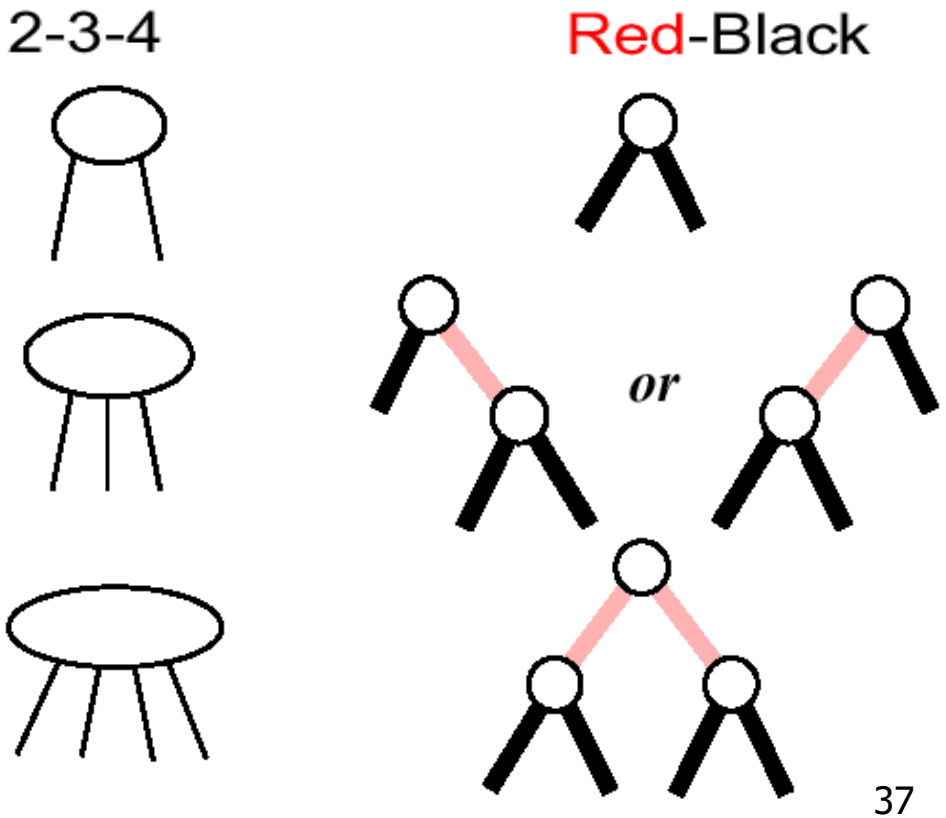


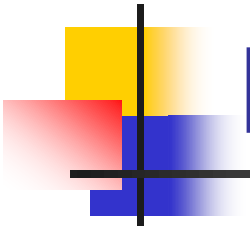
How long does it take?

- Deletion in a RB-tree takes $O(\log n)$
  - Maximum three rotations and $O(\log n)$ recolorings

Other balanced trees

- Red-Black trees are related to 2-3-4 trees (non-binary trees)
- AVL-trees have simpler algorithms, but may perform a lot of rotations





Next lecture

- External storage search trees:
  - B-Trees