Red-Black Trees

Based on materials by Dennis Frey, Yun Peng, Jian Chen, and Daniel Hood

Advanced Data Structures

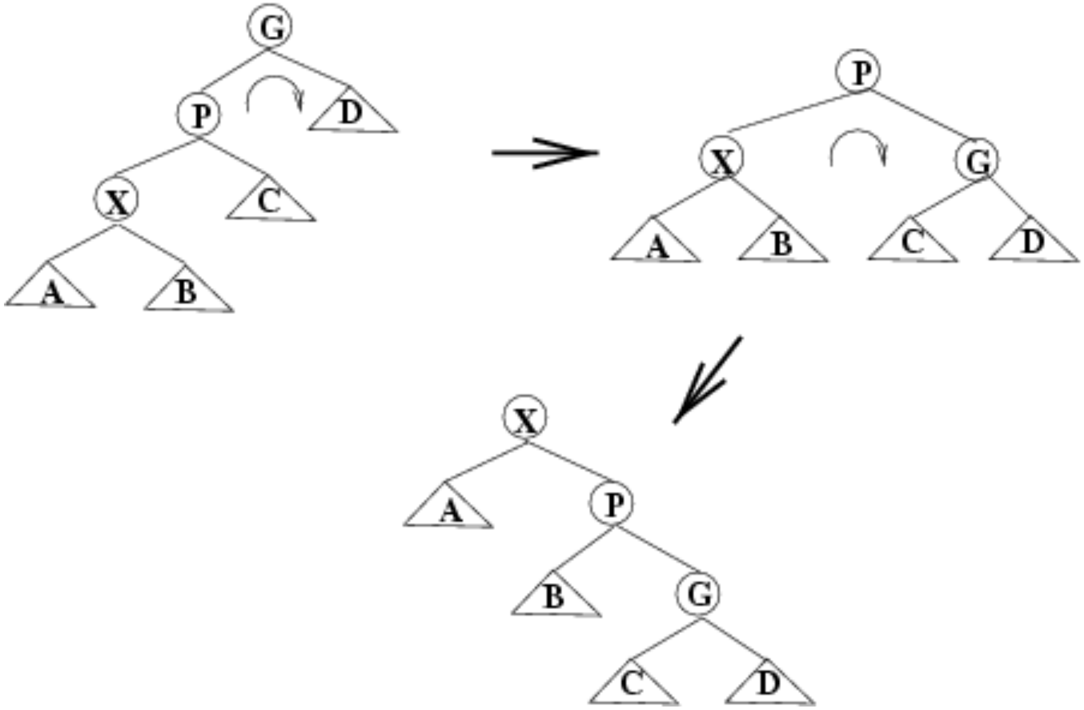
- CS 206 covered basic data structures
  - Lists, binary search trees, heaps, hash tables
- CS 246 will introduce you to some advanced data structures and their use in applications
  - Red-Black Trees: a type of self-balancing BST
  - KD-Trees: a type of space partitioning tree
  - Graphs: represents a set of entities and relations

- Over the next few weeks, we will discuss these data structures, starting today with Red-Black Trees

Quick Review of Binary Search Trees

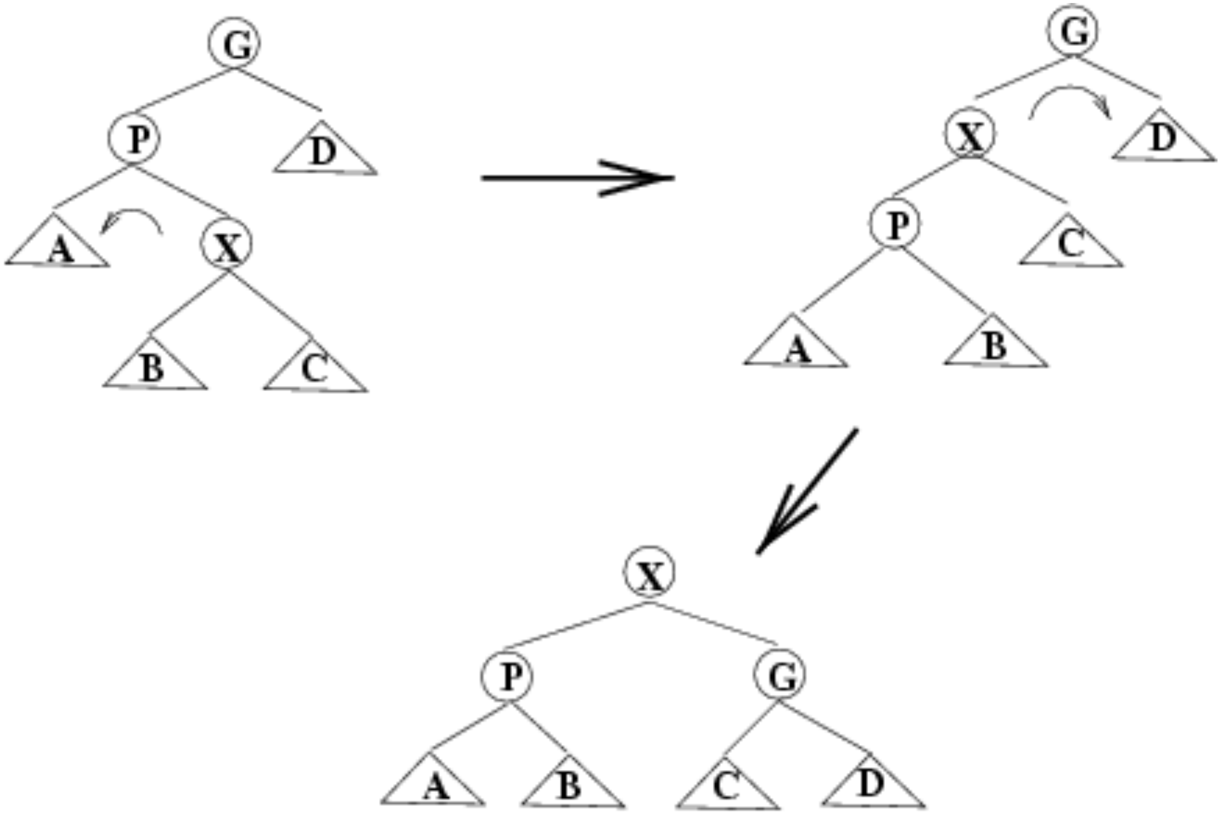
- Given a node n...
  - All elements of n’s left subtree are less than n.data
  - All elements of n’s right subtree are greater than n.data
- We are prohibiting duplicate values
- Insert/Find/Remove are O(height) (why?)
- The tree’s height varies between lg N and N
  - A balanced tree has height lg N

Review of Tree Rotations: Zig-Zig (Node and Parent are Same Side)



Rotate P around G, then X around P

Review of Tree Rotations: Zig-Zag (Node and Parent are Different Sides)

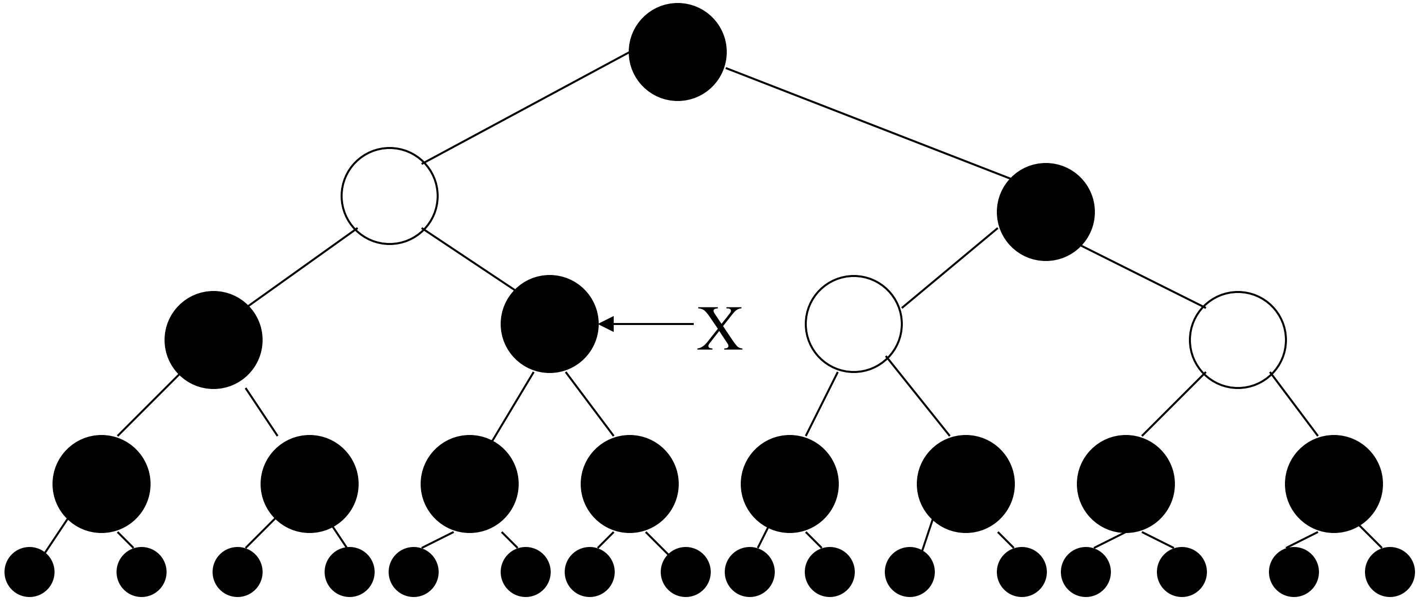


Rotate X around P, then X around G

DEFINITIONS

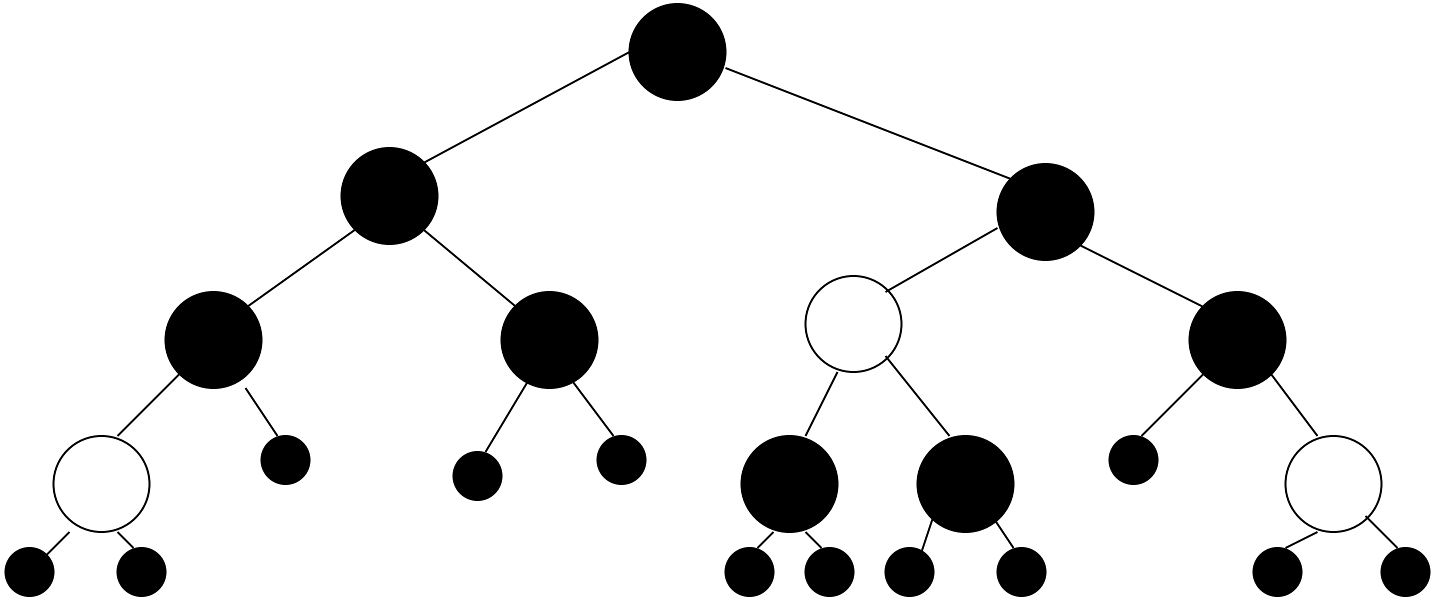
Red-Black Trees

- Definition: A red-black tree is a binary search tree in which:
  - Every node is colored either Red or Black.
  - Each NULL pointer is considered to be a Black "node".
  - If a node is Red, then both of its children are Black.
  - Every path from a node to a NULL contains the same number of Black nodes.
  - By convention, the root is Black
- Definition: The black-height of a node X in a red-black tree is the number of Black nodes on any path to a NULL, not counting X.



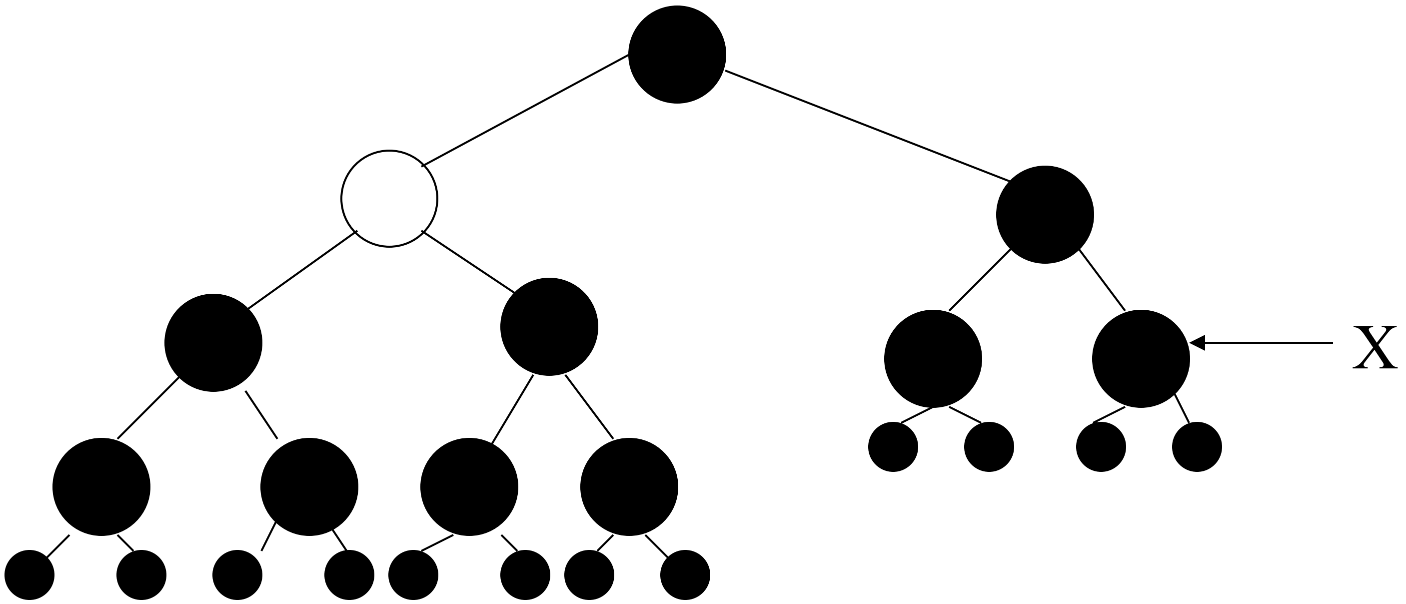
A Red-Black Tree with NULLs shown

Black-Height of the tree (the root) = 3
Black-Height of node “X” = 2



A Red-Black Tree with

Black-Height = 3



Black Height of the tree?

Black Height of X?

Theorem 1 – Any red-black tree with root ***x***, has **$n \geq 2^{bh(x)} - 1$** nodes, where bh(x) is the black height of node x.

Proof: by induction on height of x.

Theorem 2 – In a red-black tree, at least half the nodes on any path from the root to a NULL must be Black.

Proof – If there is a Red node on the path, there must be a corresponding Black node.

Algebraically this theorem means

$$bh(x) \geq h/2$$

Theorem 3 – In a red-black tree, no path from any node, X, to a NULL is more than twice as long as any other path from X to any other NULL.

Proof: By definition, every path from a node to any NULL contains the same number of Black nodes. By Theorem 2, a least ½ the nodes on any such path are Black. Therefore, there can no more than twice as many nodes on any path from X to a NULL as on any other path. Therefore the length of every path is no more than twice as long as any other path.

Theorem 4 –

A red-black tree with ***n*** nodes has height

$$h \leq 2 \lg(\boldsymbol{n} + 1).$$

Proof:

Let h be the height of the red-black tree with root x. By Theorem 2,

$$bh(x) \geq h/2$$

From Theorem 1, $\boldsymbol{n} \geq 2^{bh(x)} - 1$

Therefore $n \geq 2^{h/2} - 1$

$$n + 1 \geq 2^{h/2}$$

$$\lg(n + 1) \geq h/2$$

$$2\lg(n + 1) \geq h$$

BOTTOM-UP INSERTION

Bottom –Up Insertion

- Insert node as usual in BST
- Color the node Red
- What Red-Black property <u>may</u> be violated?
    - Every node is Red or Black?
    - NULLs are Black?
    - If node is Red, both children must be Black?
    - Every path from node to descendant NULL must contain the same number of Blacks?

Bottom Up Insertion

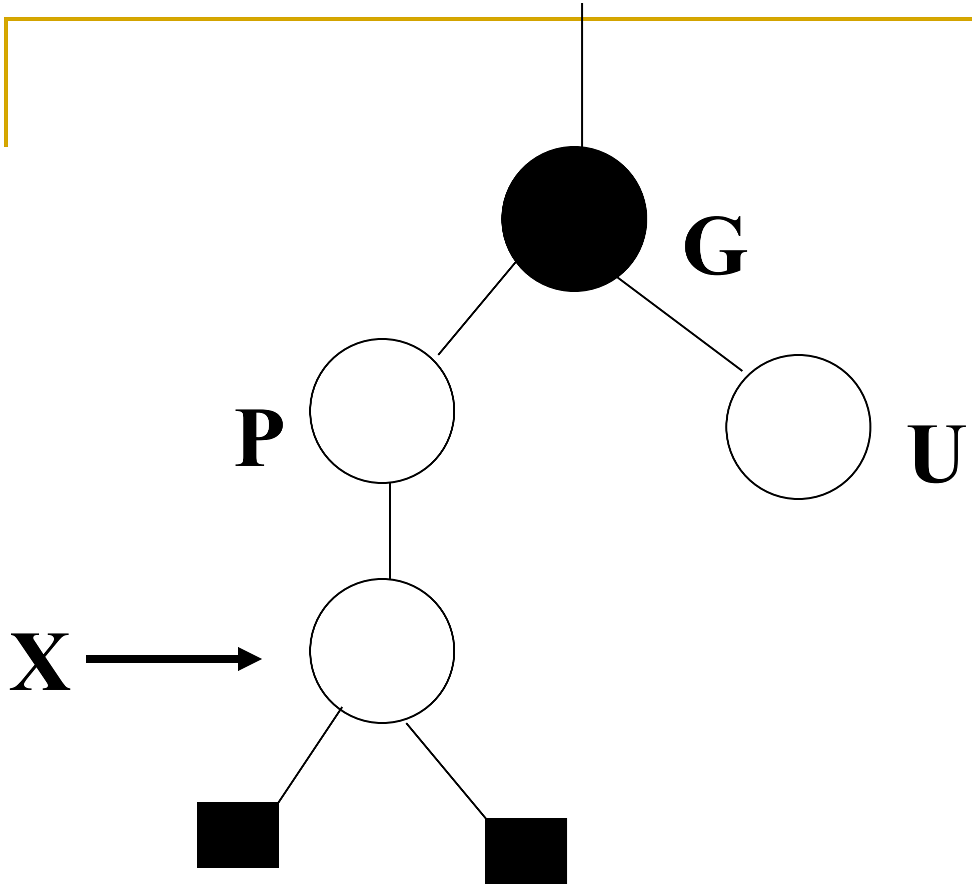
- Insert node; Color it Red; X is pointer to it
- Cases

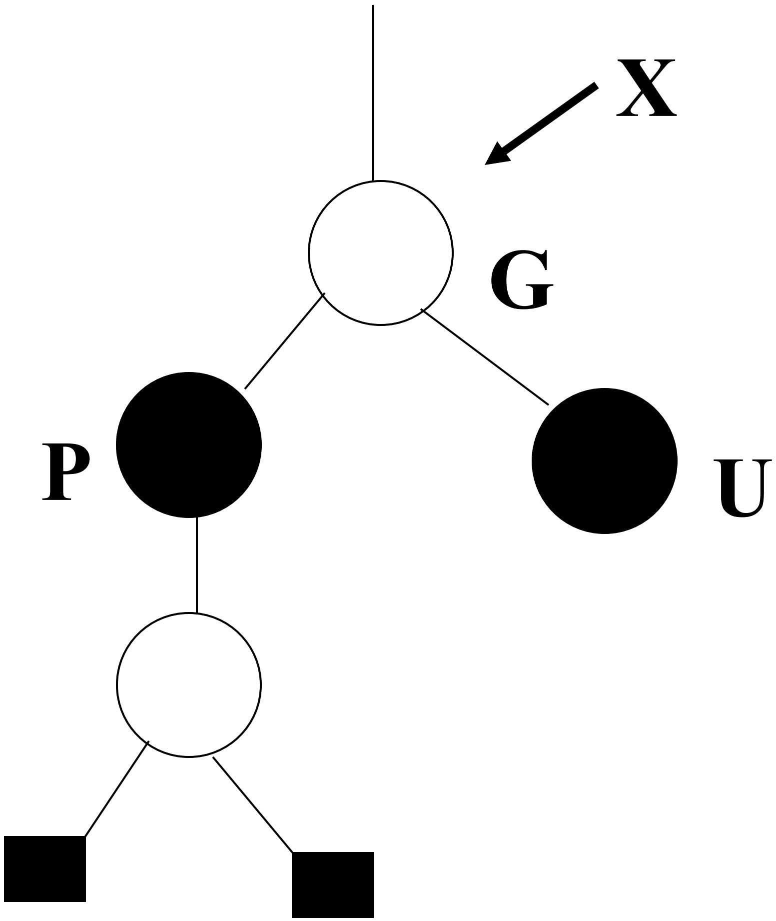
0: X is the root -- color it Black

1: Both parent and uncle are Red -- color parent and uncle Black, color grandparent Red. Point X to grandparent and check new situation.

2 (zig-zag): Parent is Red, but uncle is Black. X and its parent are opposite type children -- color grandparent Red, color X Black, rotate left(right) on parent, rotate right(left) on grandparent

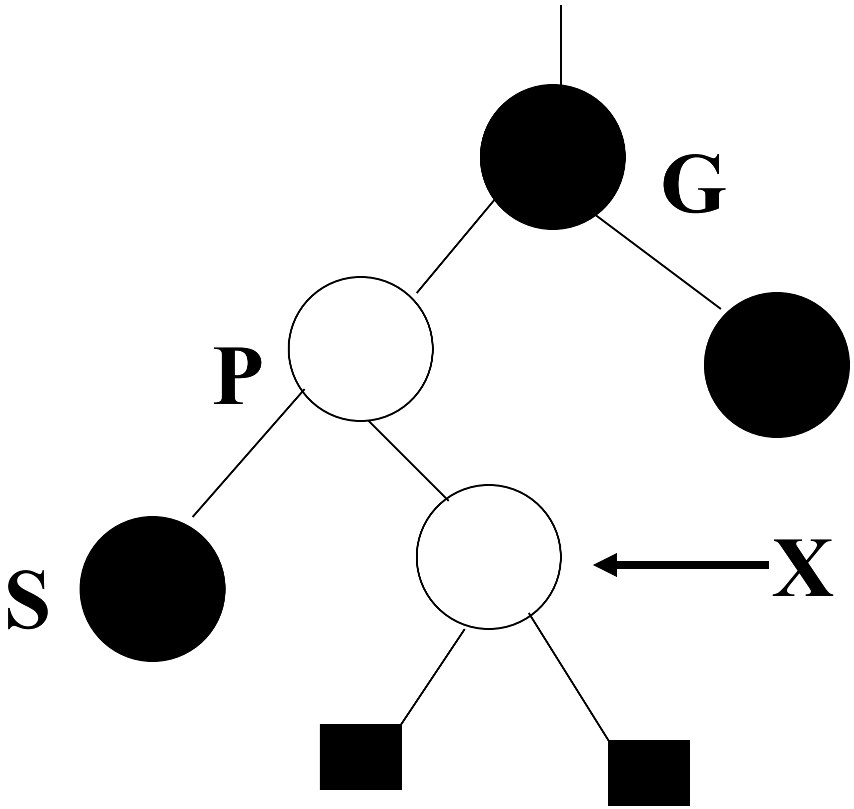
3 (zig-zig): Parent is Red, but uncle is Black. X and its parent are both left (right) children -- color parent Black, color grandparent Red, rotate right(left) on grandparent





Case 1 – U is Red

Just Recolor and move up

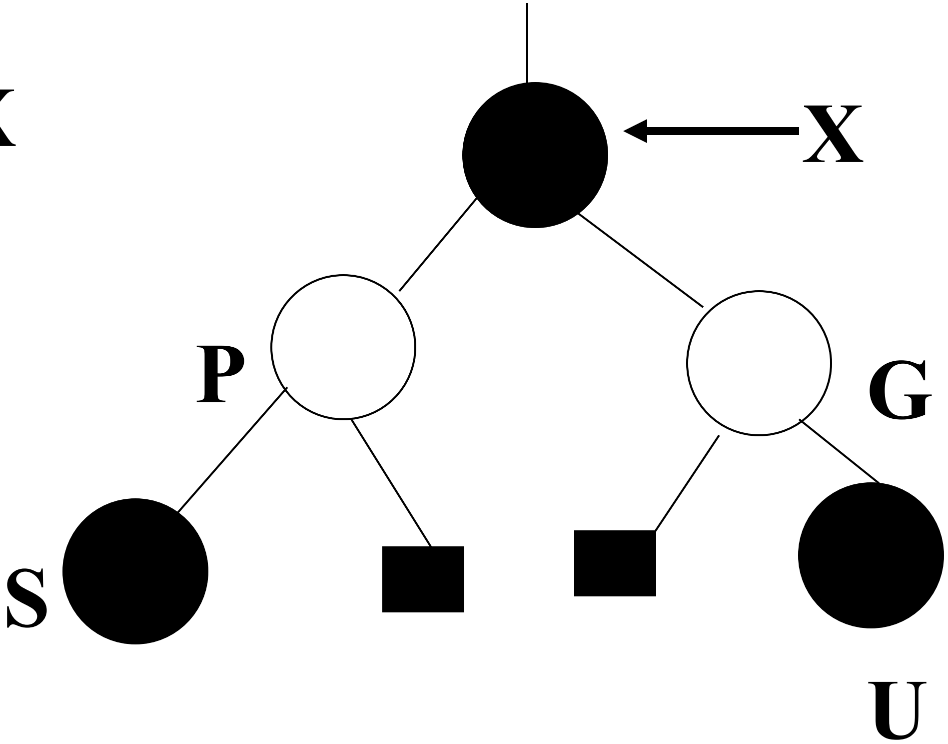


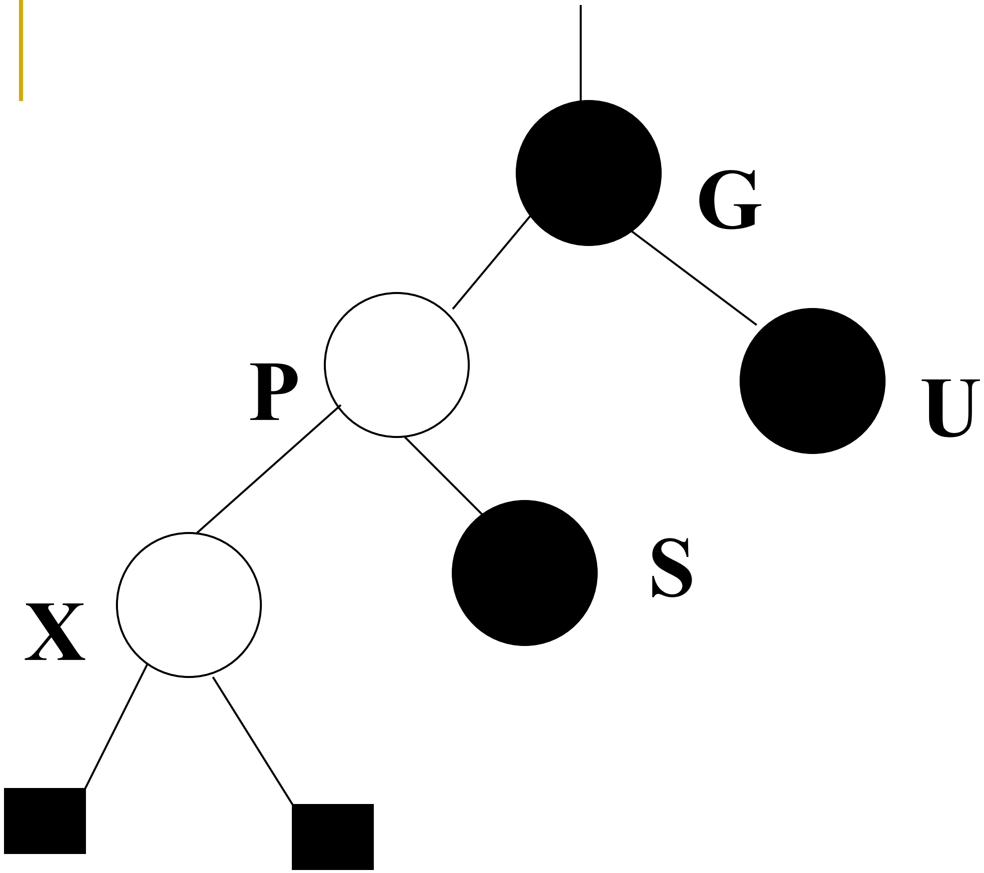
Case 2 – Zig-Zag

Double Rotate

X around P; X around G

Recolor G and X

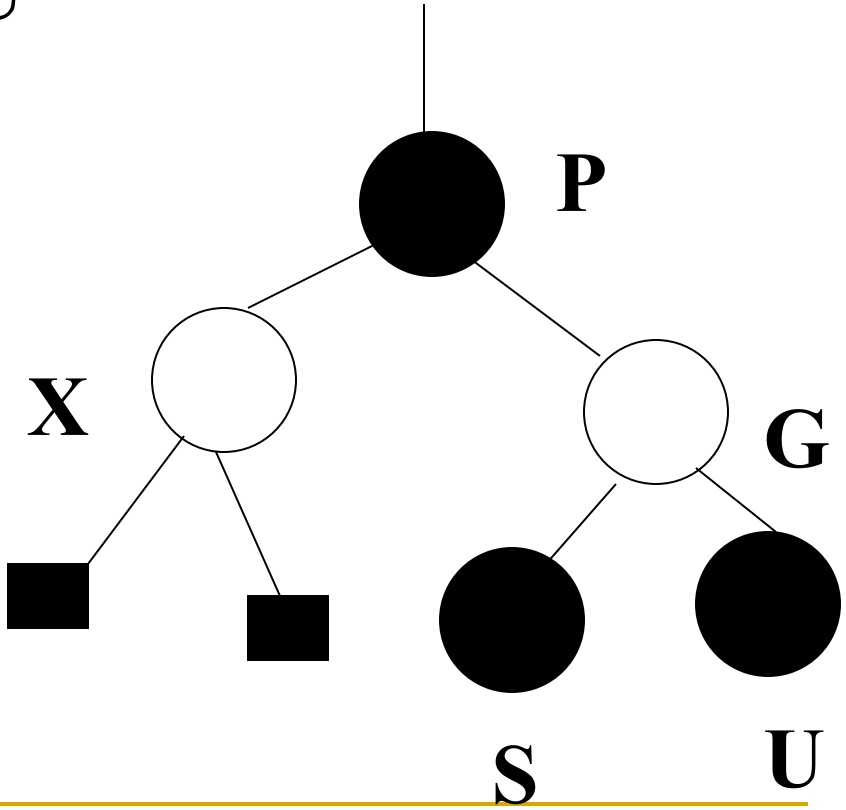




Case 3 – Zig-Zig

Single Rotate P around G

Recolor P and G

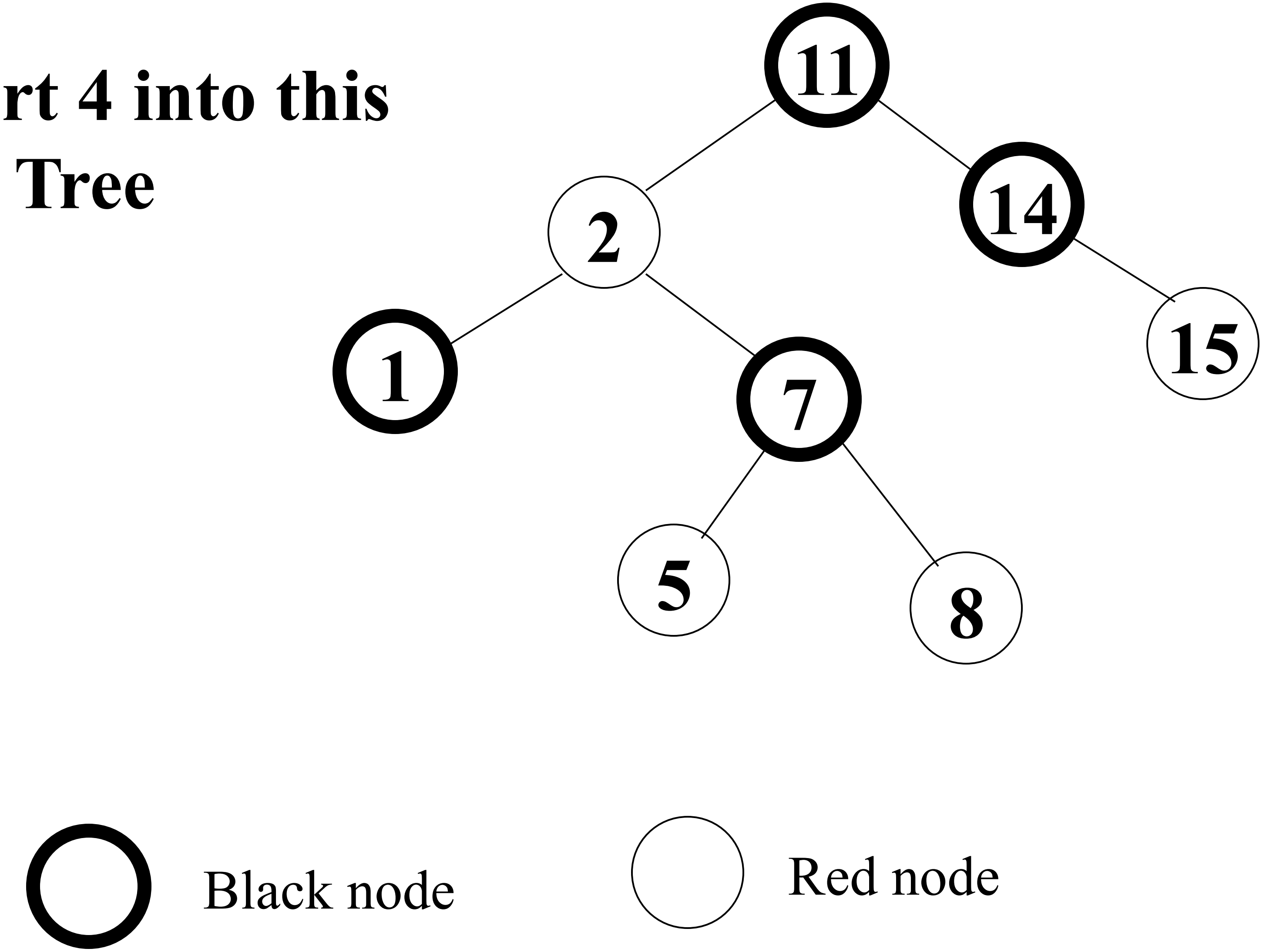


Asymptotic Cost of Insertion

- *O*(lg n) to descend to insertion point
- *O*(1) to do insertion
- *O*(lg n) to ascend and readjust == worst case only for case 1

- Total: *O*(lg n)

Insert 4 into this R-B Tree



Insertion Practice

Insert the values 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty Red-Black Tree

Top-Down Insertion

An alternative to this “bottom-up” insertion is “top-down” insertion.

Top-down is iterative. It moves down the tree, “fixing” things as it goes.

What is the objective of top-down’s “fixes”?

BOTTOM-UP DELETION

Recall "ordinary" BST Delete

1. If node to be deleted is a leaf, just delete it.
2. If node to be deleted has just one child, replace it with that child (splice)
3. If node to be deleted has two children, replace the **value** in the node by its in-order predecessor/successor's value then delete the in-order predecessor/successor (a recursive step)

Bottom-Up Deletion

1. Do ordinary BST deletion. Eventually a "case 1" or "case 2" deletion will be done (leaf or just one child).
   -- If deleted node, U, is a leaf, think of deletion as replacing U with the NULL pointer, V.
   -- If U had one child, V, think of deletion as replacing U with V.
2. What can go wrong??

Which RB Property may be violated after deletion?

1. If U is Red?

   Not a problem – no RB properties violated

2. If U is Black?

   If U is not the root, deleting it will change the black-height along some path

Fixing the problem

- Think of V as having an “extra” unit of blackness. This extra blackness must be absorbed into the tree (by a red node), or propagated up to the root and out of the tree.
- There are four cases – our examples and “rules” assume that V is a left child. There are symmetric cases for V as a right child.

Terminology

- The node just deleted was U
- The node that replaces it is V, which has an extra unit of blackness
- The parent of V is P
- The sibling of V is S

● Black Node

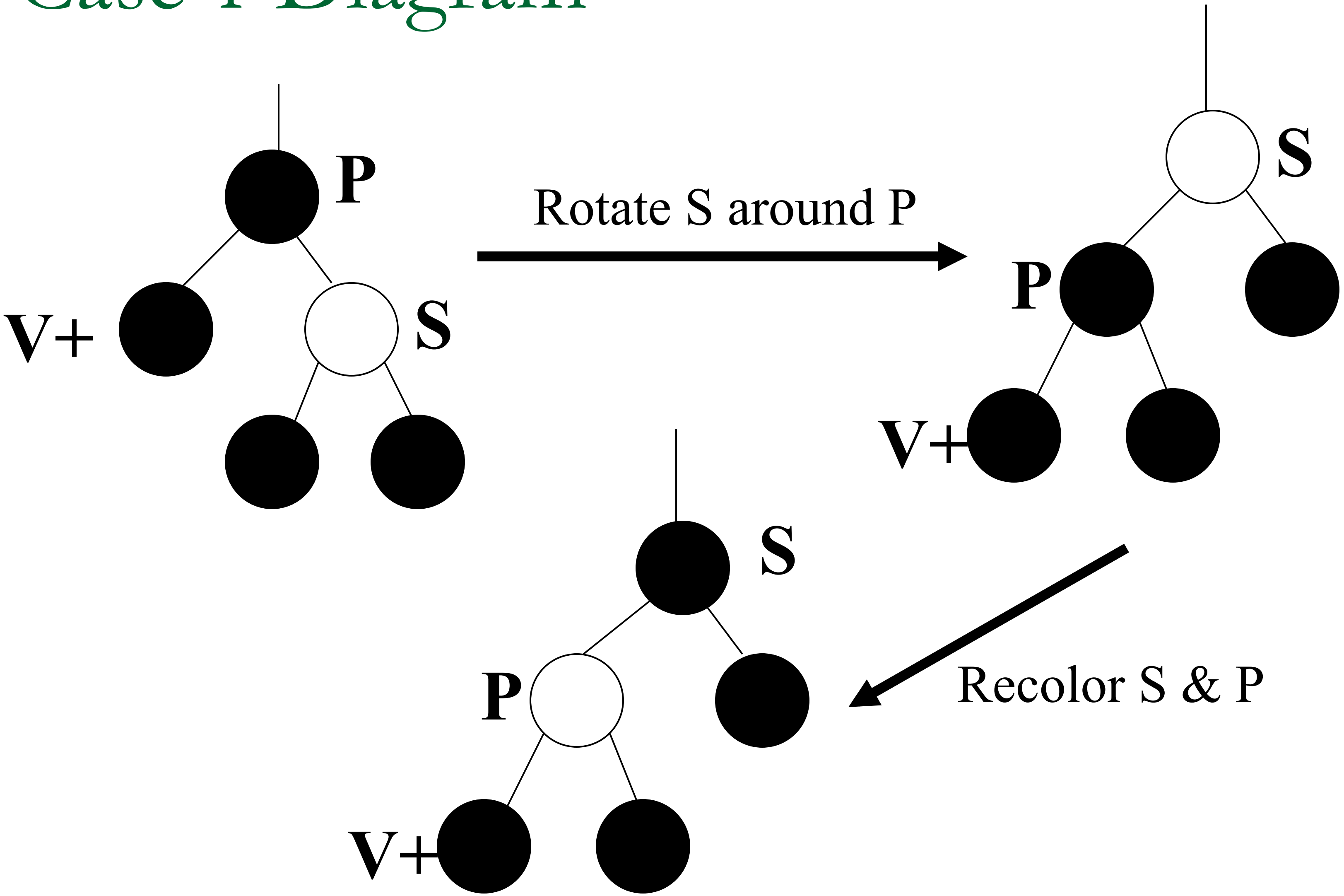
△ Red or Black and don’t care

○ Red Node

Bottom-Up Deletion
Case 1

- V's sibling, S, is Red
  - Rotate S around P and recolor S & P
- NOT a terminal case – One of the other cases will now apply
- All other cases apply when S is Black

Case 1 Diagram



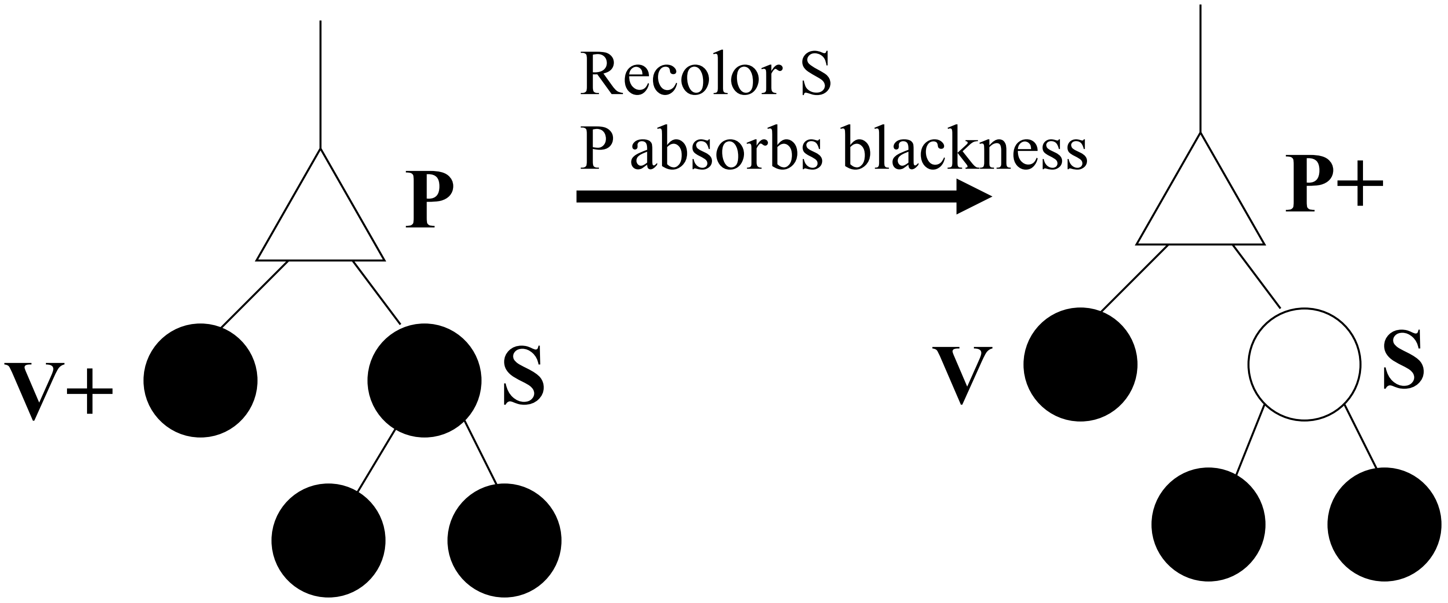
Rotate S around P

Recolor S & P

Bottom-Up Deletion
Case 2

- V’s sibling, S, is Black and has <u>two Black children</u>.
  - Recolor S to be Red
  - P absorbs V’s extra blackness
    - If P is Red, we’re done (it absorbed the blackness)
    - If P is Black, it now has extra blackness and problem has been propagated up the tree

Case 2 diagram



Either extra Black absorbed by P

or

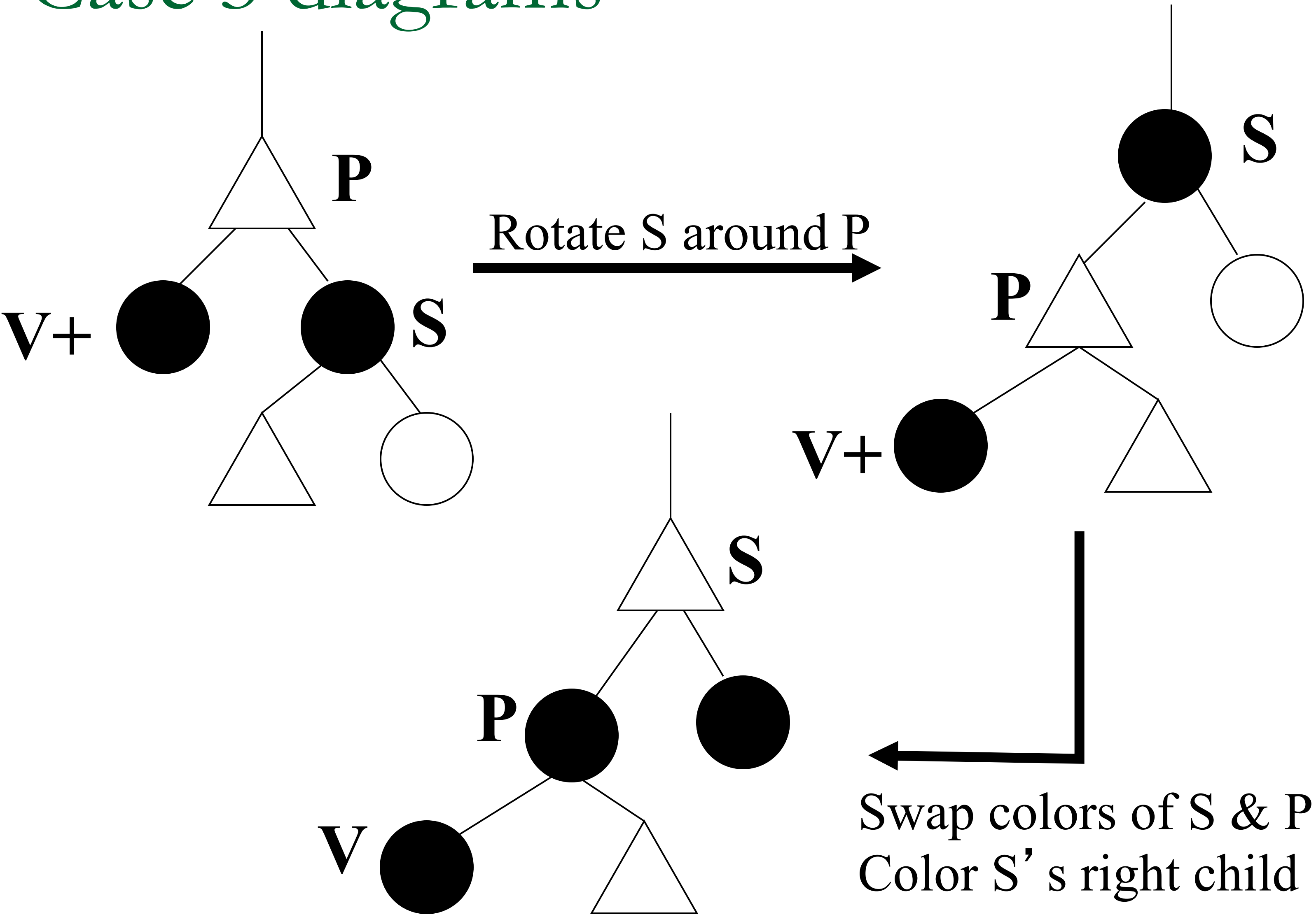
P now has extra blackness

Bottom-Up Deletion
Case 3

- S is Black
- S's right child is RED (Left child either color)
    - Rotate S around P
    - Swap colors of S and P, and color S's right child Black

- This is the terminal case – we're done

Case 3 diagrams



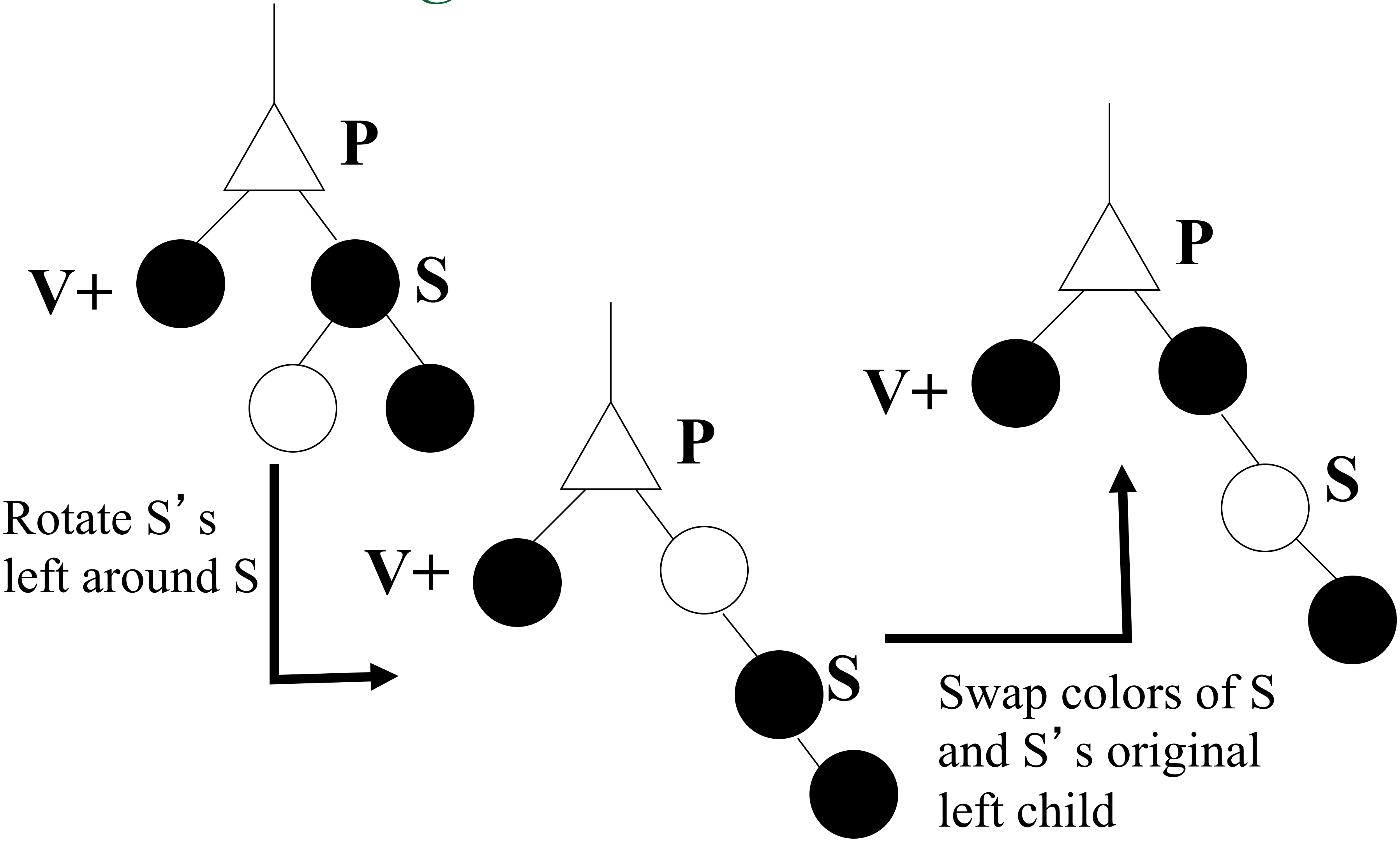
Rotate S around P

Swap colors of S & P
Color S’s right child Black

Bottom-Up Deletion
Case 4

- S is Black, S's right child is Black and S's left child is Red
  - Rotate S's left child around S
  - Swap color of S and S's left child
  - Now in case 3

Case 4 Diagrams



Rotate S's left around S

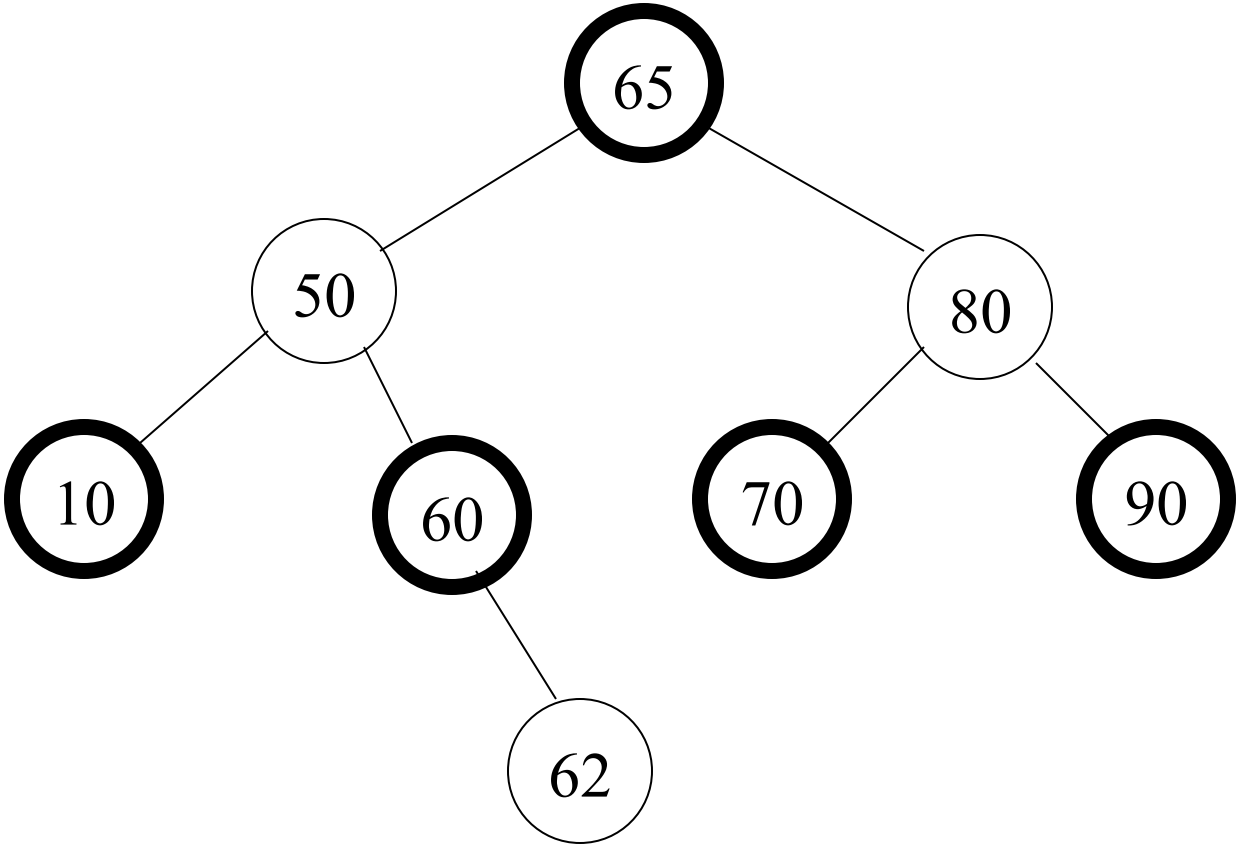
Swap colors of S and S's original left child

Top-Down Deletion

An alternative to the recursive "bottom-up" deletion is "top-down" deletion.

This method is iterative. It moves down the tree only, "fixing" things as it goes.

What is the goal of top-down deletion?



Perform the following deletions, in the order specified
Delete 90, Delete 80, Delete 70