ILLINOIS INSTITUTE OF TECHNOLOGY

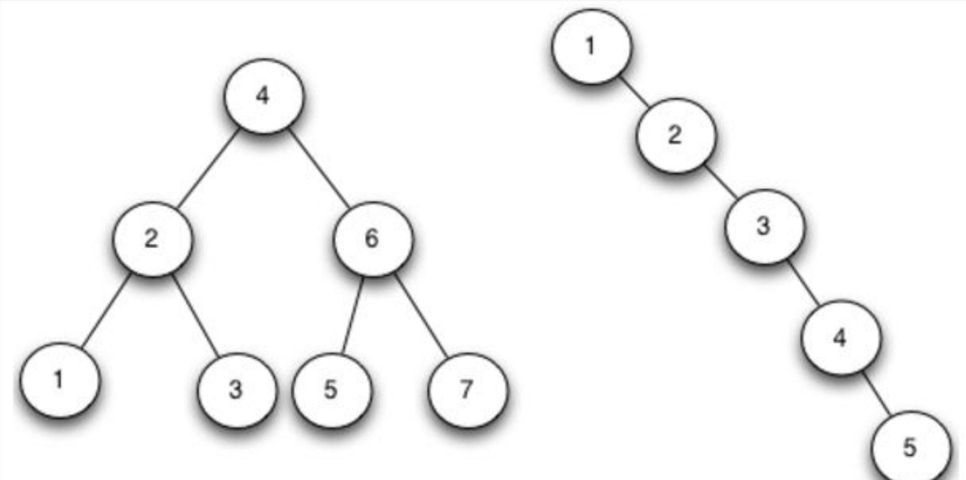
Transforming Lives. Inventing the Future. www.iit.edu

CS430
Introduction to Algorithms

L13-L14

Prelecture Questions

- If the ordered array is 1,2,3,4,5, 6,7, what is the best case to BST? Worst?



O(lgn) **O(n)**

▪ Balance BST

- A node in a tree is height-balanced if the heights of its subtrees differ by no more than 1. (That is, if the subtrees have heights h1 and h2, then $|h1 - h2| \leq 1$.)
- A tree is height-balanced if all of its nodes are height-balanced. (An empty tree is height-balanced by definition.)

```
Input:
        4
       /
      3
     /
    2
   /
  1
Output:
```

Ex:

```
        20
       /  \
      /    \
    10      30
           /  \
          /    \
        25      40
               /
              /
            35
```

```
        20
       /  \
      /    \
    10      30
           /  \
          /    \
        25      40
          \
           \
            28
```

HOW?

Outlines

Red-Black BST

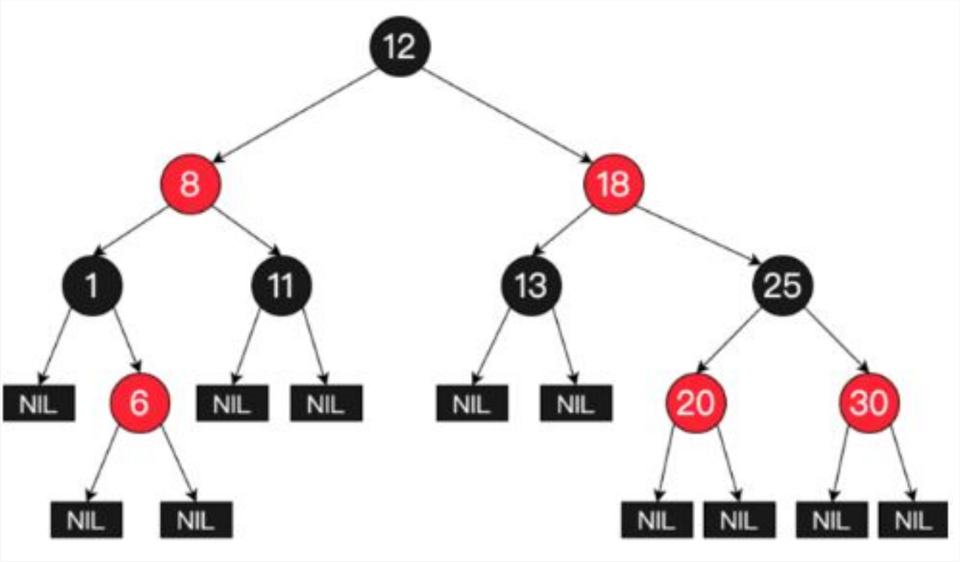
- Definition
- Properties
- O Proof
- Insert

Balanced Binary Search Trees - Red-Black Trees*

- Maintain approximate balance of BST by use of additional color bit per node (red or black)
- Constrain the ways nodes can be colored on any path from root to leaf.
- Height of subtrees of any node are at most twice as big as each other (this is good enough to show O(lg n) height)

R-B BST

Ex:



12
8
18
1
11
13
25
NIL
6
NIL
NIL
NIL
NIL
20
30
NIL
NIL
NIL
NIL
NIL
NIL

Red-Black Properties*

1. Every node is colored either red or black
2. The root is black
3. Every null pointer descending from a leaf is considered to be a null black leaf node
4. If a node is red, then both its children are black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes (black height)

bh(x) = the number of black nodes on any path from, but not including, a node "x" down to a null black leaf node

Height of a R-B BST

What we expect:

- The height of a R-B BST is O(lgn)
- Lemma - A red-black

with n internal nodes

(n key values) has

height at most 2lg(n+1)

(proof using induction a

Properties)

$$h \le 2\lg(n+1) \Rightarrow$$

$$\frac{h}{2} \le \lg(n+1) \Rightarrow$$

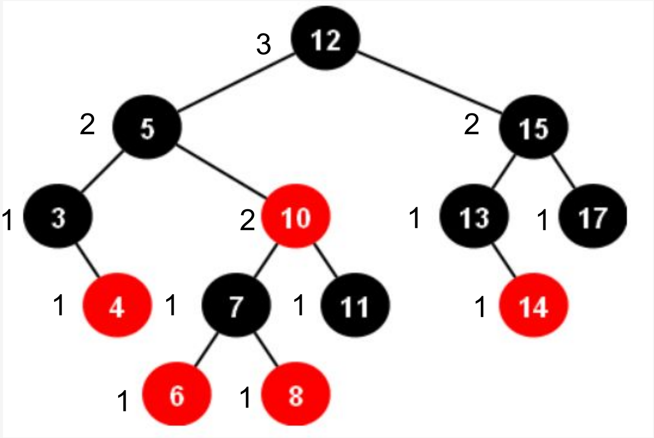
$$2^{\frac{h}{2}} \le 2^{\lg(n+1)} = n+1$$

$$\Rightarrow 2^{\frac{h}{2}} \le n+1 \Rightarrow$$

$$n \ge 2^{\frac{h}{2}} - 1$$

Proof: height at most 2lg(n+1)

Part A - First show the sub-tree rooted at node "x " has at least $2^{bh(x)}$-1 internal nodes



Proof by Induction

Base case - bh(x)=0

x is a nil root node; no key

$2^{bh(x)}-1 = 2^0-1 = 0$ internal nodes minimum

Base Case - bh(x)=1

```
      B              B                 B
   /     \         /   \             /   \
 nil     nil     nil    R           R      R
                       /  \        / \    / \
                     nil  nil    nil nil nil nil
1 internal      2 internal      3 internal
```

$2^{bh(x)}-1 = 2^1-1 = 1$ internal nodes minimum

Now consider node x with bh(x)=k

All paths to leafs have k black nodes
What about the children of node x?
If y=child(x) is red, then bh(y) =bh(x)= k
If y=child(x) is black, then bh(y) =bh(x)-1= k-1

Assume it is true for y (a child of x), that it has at least $2^{bh(y)}$-1 internal nodes, 2^{k-1}-1 at least (if the child is red it has more as 2^k-1)

So if we build a tree one level up by adding the “at least” internal nodes for both children of x and x, we get . . .

Internal nodes of x

$$>= (2^{k-1}-1) + (2^{k-1}-1) + 1$$

$$>= 2 * 2^{k-1} - 2 + 1$$

Internal nodes of n $>= 2^k - 1$

where bh(x)=k.

QED.

Proof: height at most 2lg(n+1)

Part B:

proved: n > = $2^{bh(x)}$ - 1

by property #4: at least half the nodes on path from root to leaf are black $\Rightarrow$

bh(x) >= h/2

$$h \le 2\lg(n+1) \Rightarrow$$

$$\frac{h}{2} \le \lg(n+1) \Rightarrow$$

$$2^{\frac{h}{2}} \le 2^{\lg(n+1)} = n+1$$

$$\Rightarrow 2^{\frac{h}{2}} \le n+1 \Rightarrow$$

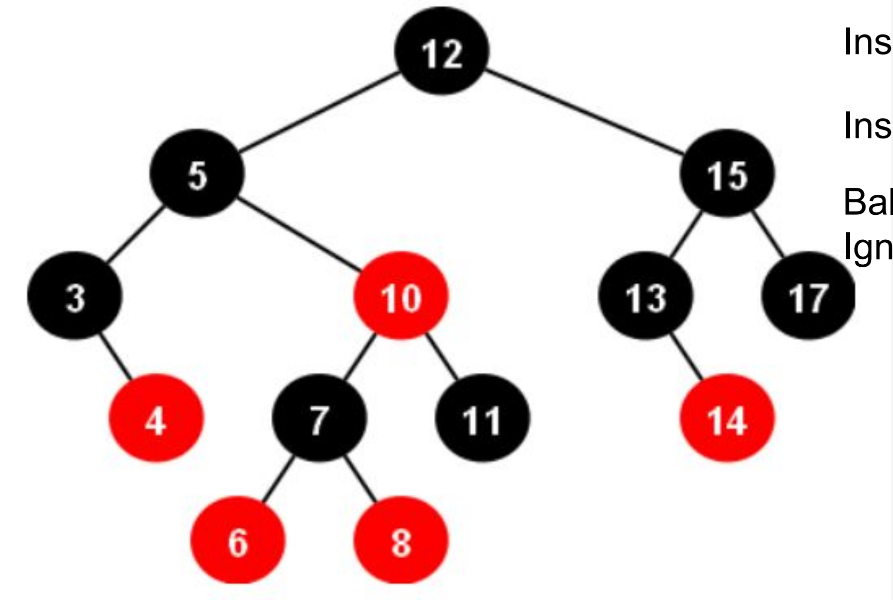
$$n \ge 2^{\boxed{\frac{h}{2}}} - 1$$

h <= 2log(n+1)   h=O(lg n)

Red-Black Tree Operations*

- Red-Black Tree search, predecessor, successor, minimum, maximum operations identical to BST (ignore the color of a node)
- Red-Black Tree insert and delete, since they modify the tree, require changing the colors of some nodes <u>and</u> **rotations** in the tree to maintain the Red-Black Properties and the approximate balanced property of the tree.

Red-Black Tree Insert



Insert 11.5?

Insert 14.5?

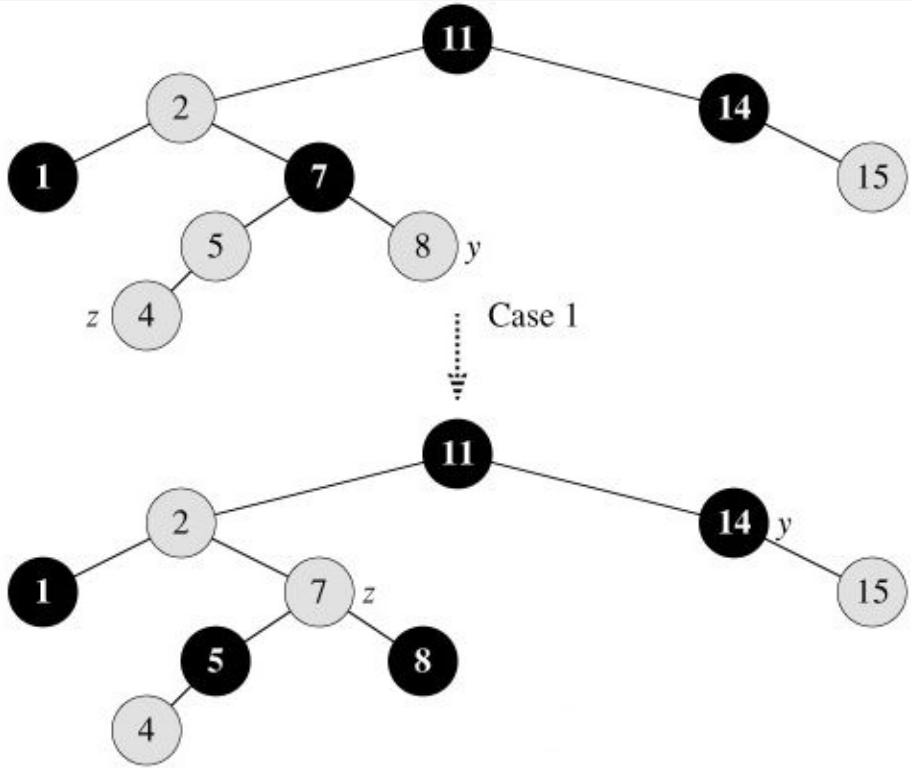
Balanced?
Ignore it!

Red-Black Tree Insert

1. locate leaf position to insert new node
2. then color new node red and create 2 new black nil leafs below newly inserted red node
3. and possible procedure to recolor nodes and perform rotations to maintain red-black properties (also color root black). If the parent of new insert is black, then DONE. Otherwise......

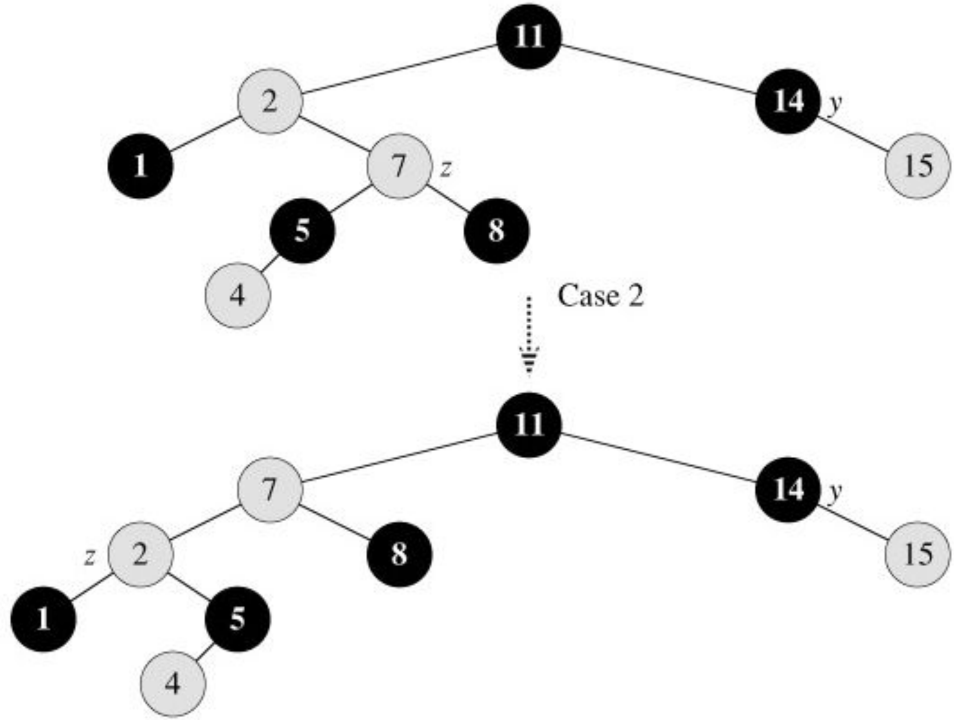
R-B Property #4 broken when insert a red node (or change color to red) and its parent is also red -- 3 Cases

1. Node "Z" (red) is a left or right child and its parent is red and its uncle is red
   - Change Z’s parent and uncle to black
   - Change Z’s grandparent to red
   - No effect on black height on any node
   - Z’s grandparent is now Z and recursion to top if #4 broken


11
2
14
1
7
15
5
8
y
z
4
Case 1
11
2
14
y
1
7
z
15
5
8
4

R-B Property #4 broken when insert a red node (or change color to red) and its parent is also red–3 Cases

2. Node "Z" is a right child and its parent is red and its uncle is NOT red
   - Rotate left on parent of Z
   - Re-label old parent of Z as Z and continue to case #3

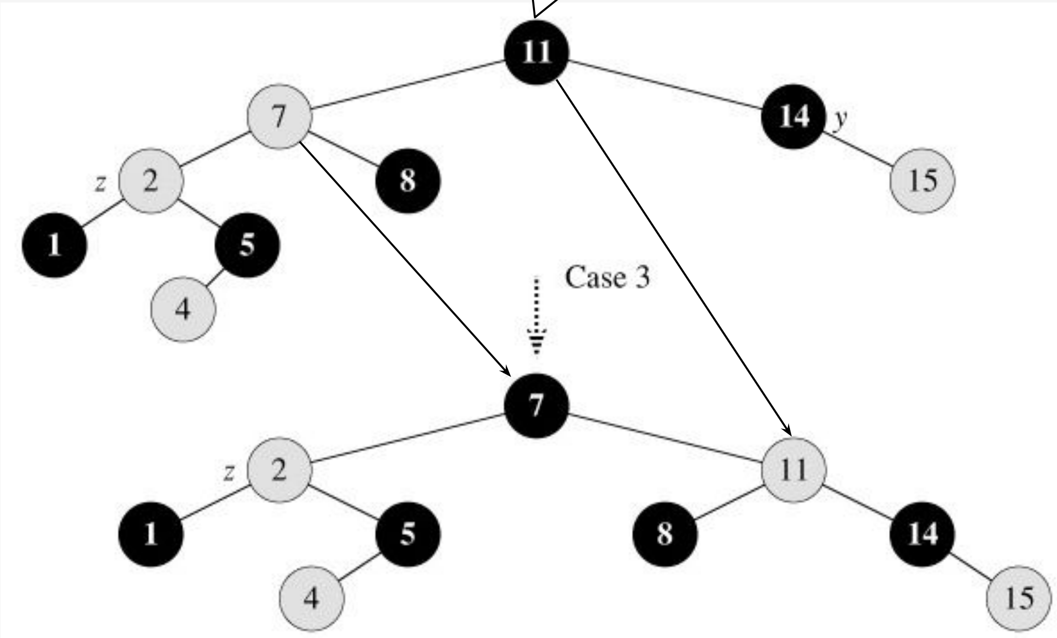


11
2
14
y
1
7
z
15
5
8
4
Case 2
11
7
14
y
z
2
8
15
1
5
4

R-B Property #4 broken when insert a red node (or change color to red) and its parent is also red. 3 Cases

3. Node "Z" is a left child and its parent is red and its uncle is NOT red
   - Rotate right on grandparent of Z
   - Color old parent of Z black
   - Color old grandparent of Z red

black for sure



Red-Black Tree Delete

- Find node to delete
- Delete node as in a regular BST (Node actually removed will have at most one child) Why?
- If we delete a Red node, STOP, tree still is a Red-Black tree
- If we delete a black node. Let x be the child of deleted node (if exists) We have to adjust the tree. Why?
  - If x is red, color it black, STOP
  - If x is black, mark it double black and perform a series of confusing, O(height of tree) operations to walk the double black up the tree and eliminate it

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

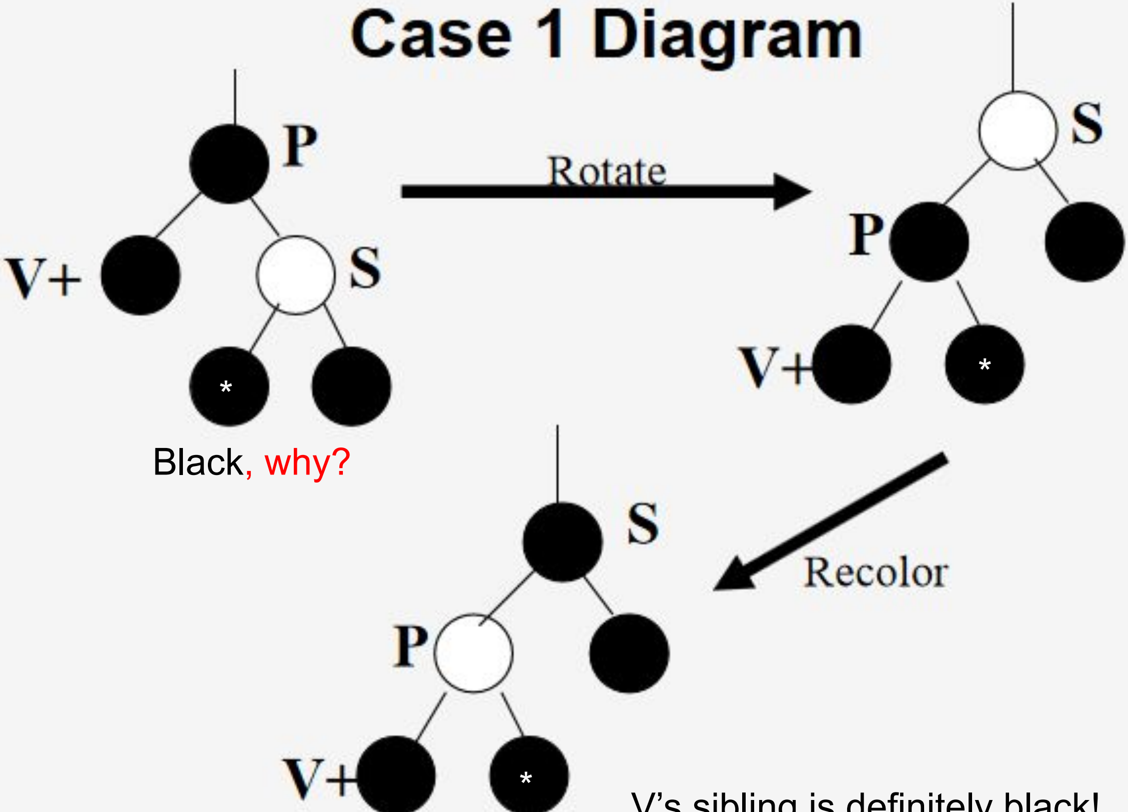
Red or Black and don't care

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



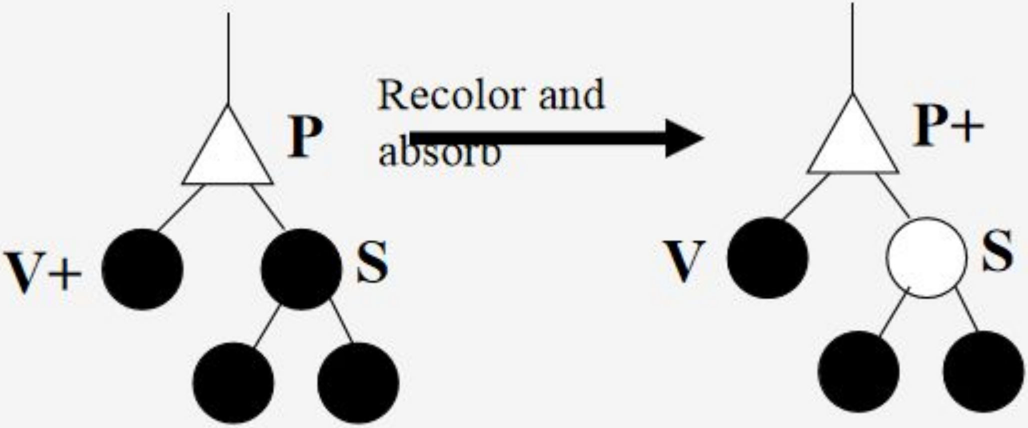
Black, why?

V's sibling is definitely black!

Bottom-Up Deletion
Case 2

- V's sibling, S, is black and has two black children.
  - Recolor S to be Red
  - P absorbs V's extra blackness
    - If P was Red, make it black, we're done
    - If P was Black, it now has extra blackness and problem has been propagated up the tree

Case 2 diagram



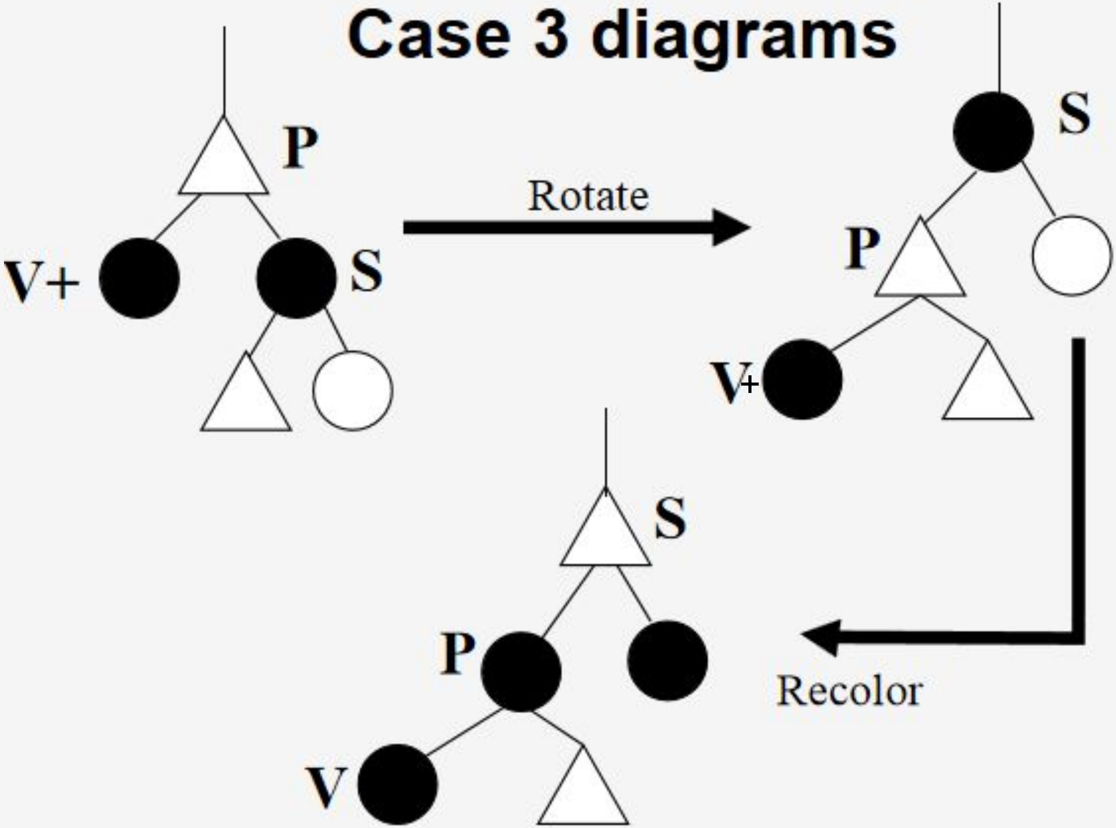
Either extra black absorbed by P or

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

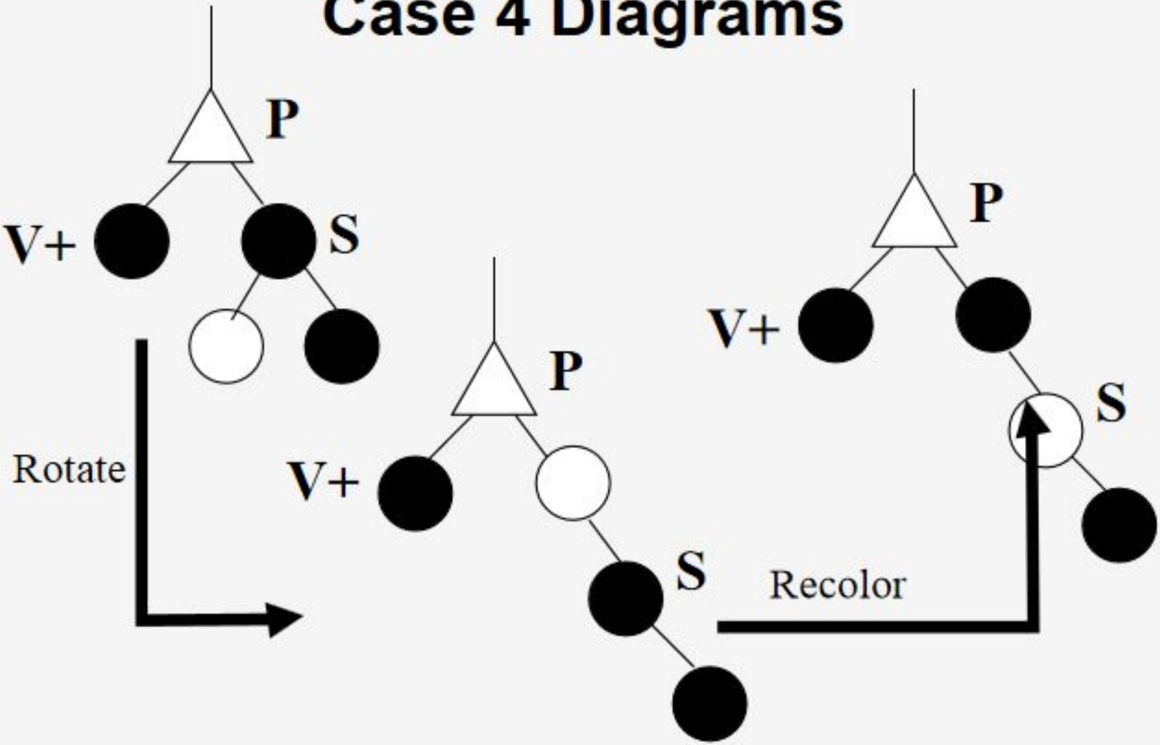
Case 3 diagrams



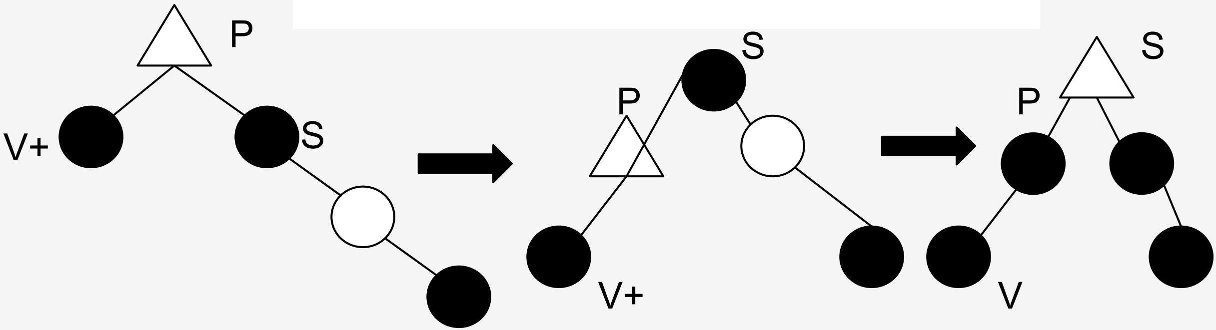
Bottom-Up Deletion
Case 4

- 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



Case 4 turns to 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

AVL Trees*

- An AVL tree is a special type of binary tree that is always "partially" balanced. The criteria that is used to determine the "level" of "balanced-ness" is the difference between the heights of sub-trees of every node in the tree. The "height" of tree is the "number of levels" in the tree.
- An AVL tree is a binary tree in which the difference between the height of the right and left sub-trees (of any node) is never more than one.

AVL Trees – Maintaining Balance*

- The idea behind maintaining the "AVL-ness" of an AVL tree is that whenever we insert or delete an item, if we have "violated" the "AVL-ness" of the tree in anyway, we must then restore it by performing a set of manipulations (called "rotations") on the tree. These rotations come in two flavors: single rotations and double rotations (and each flavor has its corresponding "left" and "right" versions).

AVL Trees – Single Rotations

left sub-tree has a height of 2 but the right sub-tree has a height of 0

Perform single right rotation at “c” (R-rotation)

Similar idea for single left rotation (L-Rotation)

AVL Trees – Double Rotations

right sub-tree has a height of 2 but the left sub-tree has a height of 0

Perform right rotation at “c” then left rotation at “a” (RL-rotation)

Similar idea for left rotation then right rotation (LR-Rotation)

AVL – Detecting Balance

- To detect when a "violation" of the AVL criteria occurs, each node must keep track of the difference in height between its right and left sub-trees. We call this "difference" the "balance" factor and define it to be the height of the right sub-tree minus the height of the left sub-tree of a node.
- As long as the "balance" factor of each node is never >1 or <-1 we have an AVL tree. As soon as the balance factor of a node becomes 2 (or -2) we need to perform one or more rotations to ensure that the resultant tree satisfies the AVL criteria.

Classwork on Desmos:

https://student.desmos.com/?prepopulateCode=b33pqa