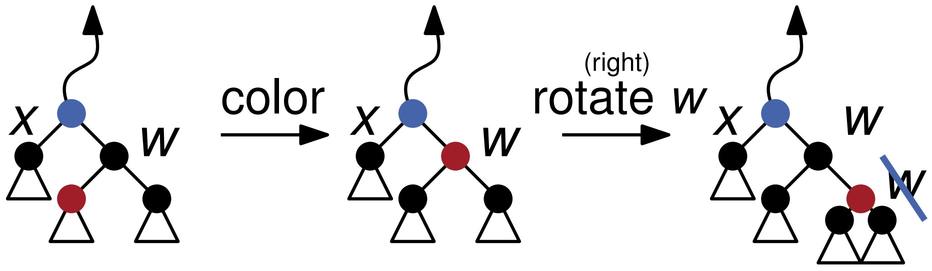
COSC 302, Spring 2018
Lecture 6.1: Red-black trees

Prof. Darren Strash

Department of Computer Science
Colgate University



Binary Search Trees

Dynamically store items in a way that maintains order

Why dynamic?:

Use an array otherwise

- Can represent all binary searches as a tree.

1	2	4	5	7	9	10	11	12	14	16	17	18	20

Decision tree lower bounds for search

Recall: decision trees model any algorithm that performs comparisons.

Comparison-based search:

- Search by comparing elements, but unable to inspect or use the values of those elements in the algorithm.

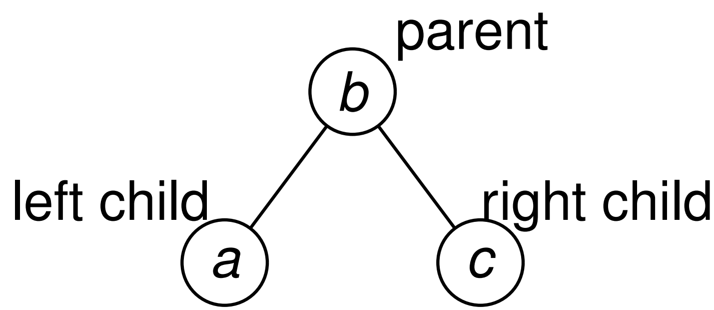
Model as a decision tree!

\# leaves $\leq 2^h$

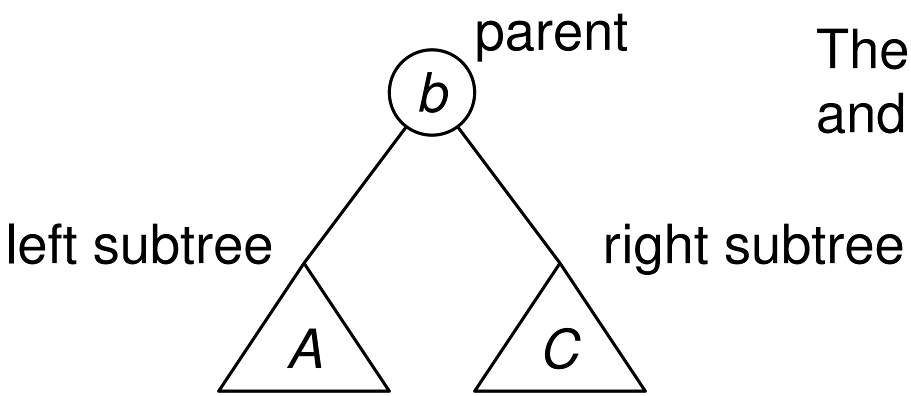
$\lg n \leq h$

Binary search trees (BST)

Consist of n nodes, containing search *keys*



Binary search tree property: Let b be a node, with left subtree A and right subtree C.



Then for $a \in A$, $key[a] \leq key[b]$,
and for $c \in C$, $key[c] \geq key[b]$.

Red-black trees

A *binary search tree* with height matching the **search lower bound**.

All operations take time proportional to height $= O(\lg n)$.

In red-black trees:

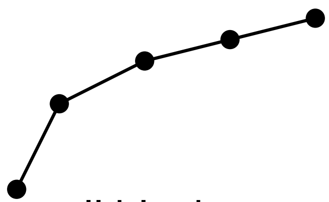
- Every node is colored red or black
- The root is black
- Every leaf is black
- Every *red* node has only black children.
- For each node, all paths from that node to a leaf have the same number of black nodes.

Critical!

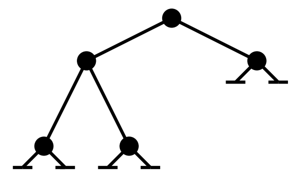
Balance in red-black trees

Why properties? What if we try to keep perfect balance?

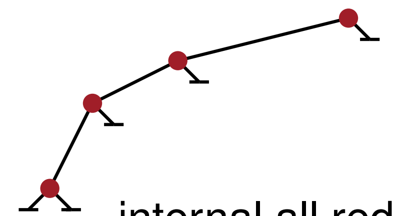
$\rightarrow$ Maintain balance (What's wrong with these trees?)



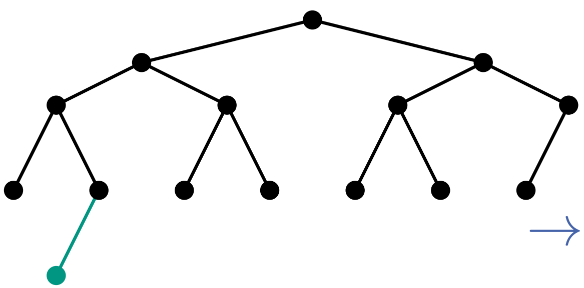
all black
regular leaves



all black
external leaves



internal all red
external leaves



Inserted key

No amount of local rebalancing will help.

$\rightarrow$ Properties make local rebalancing is possible.

Other balanced BSTs maintain balance differently: *AVL trees*

Balance in red-black trees

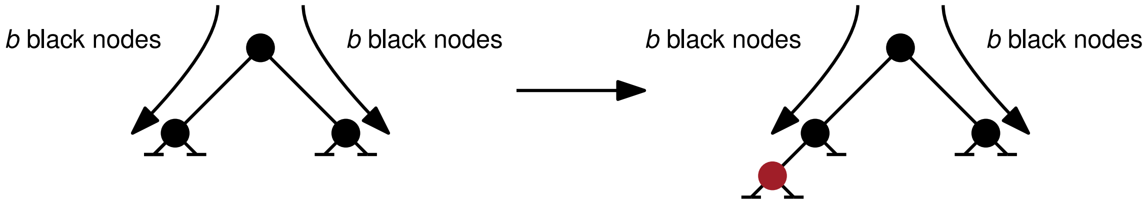
Lemma: A red-black tree on n elements has height at most $2\lg(n+1)$.

Proof.

Red-black trees: insertion

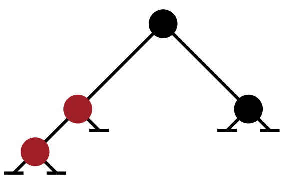
Insert as leaf with color red

$\rightarrow$ does not change number of black nodes on any path.



(make new external node children)

$\rightarrow$ but red node may have red child

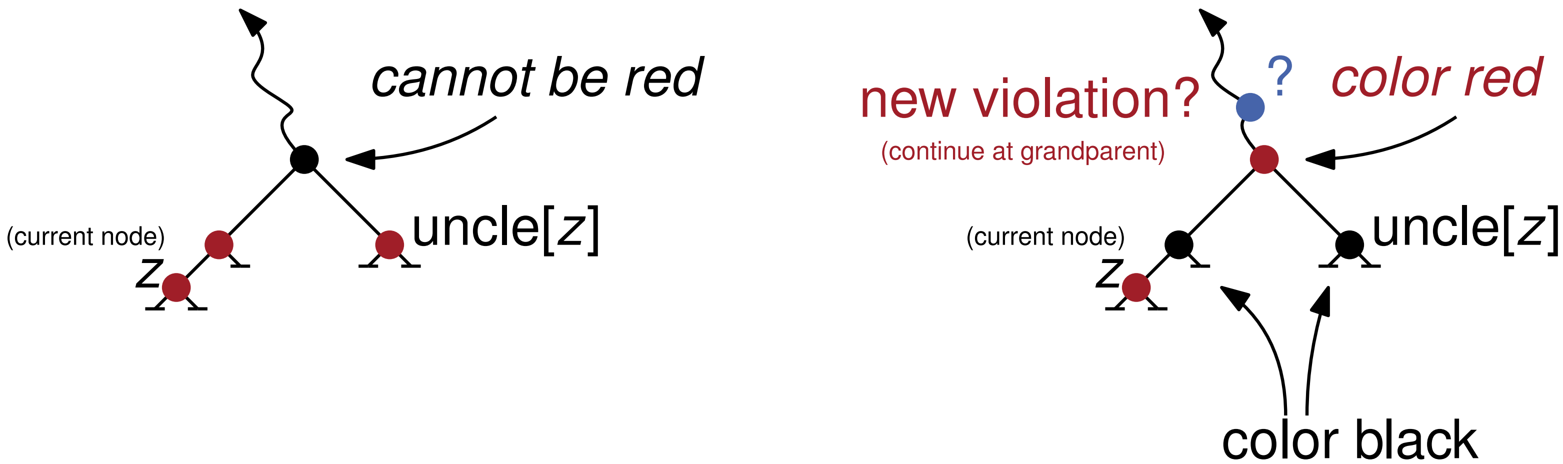


Goal: Fix while maintaining black height.

$\rightarrow$ fix one red node at a time. May introduce another red node.

Red-black trees: insertion, case 1

Case 1: current red node z has a red *uncle*.



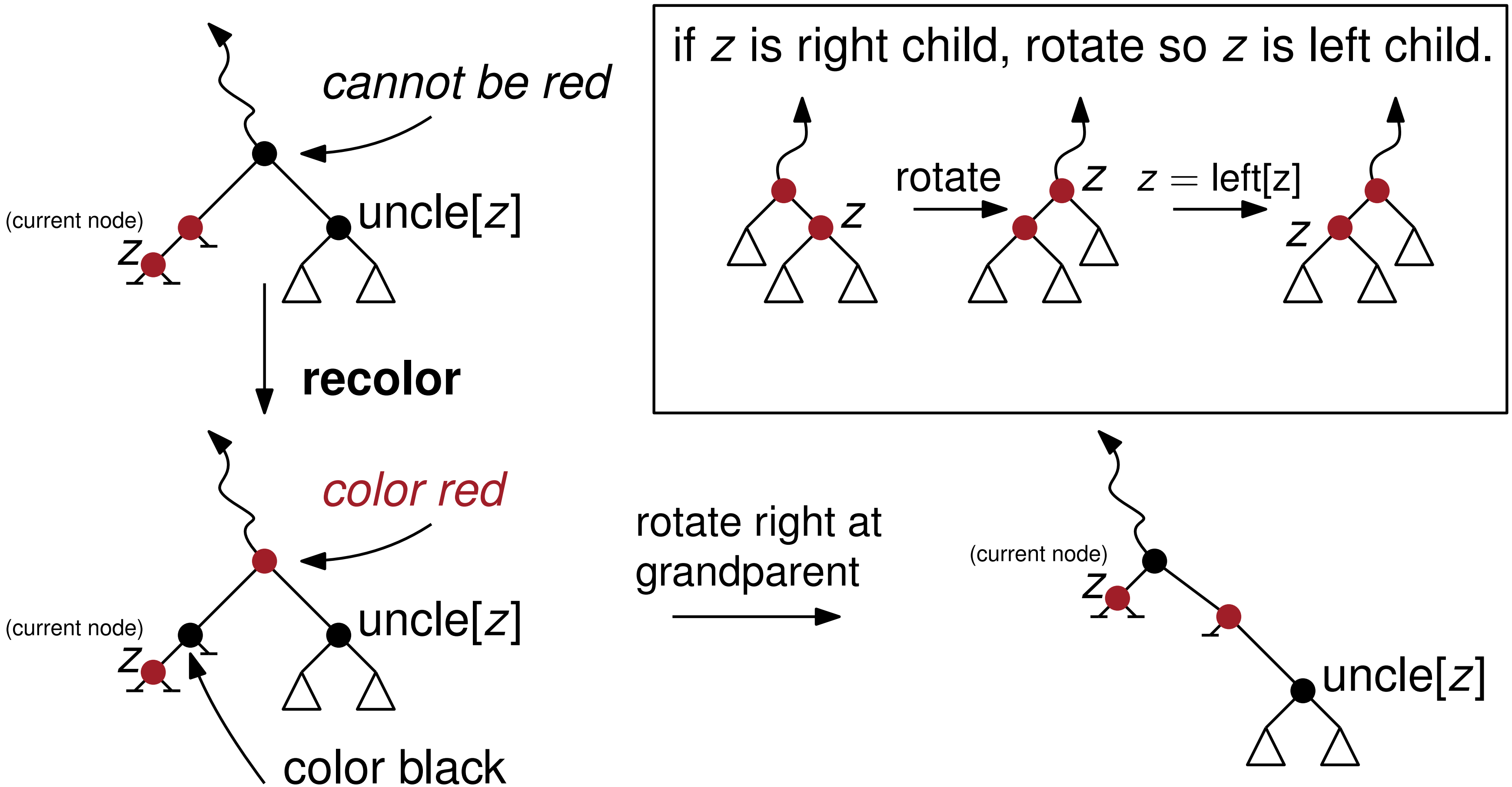
To ensure paths with equal numbers of black nodes:

- Color z's grandparent $p[p[z]]$ red
- Color $p[p[z]]$'s children black.

$\rightarrow$ Set $z = p[p[z]]$ and test again for violations.

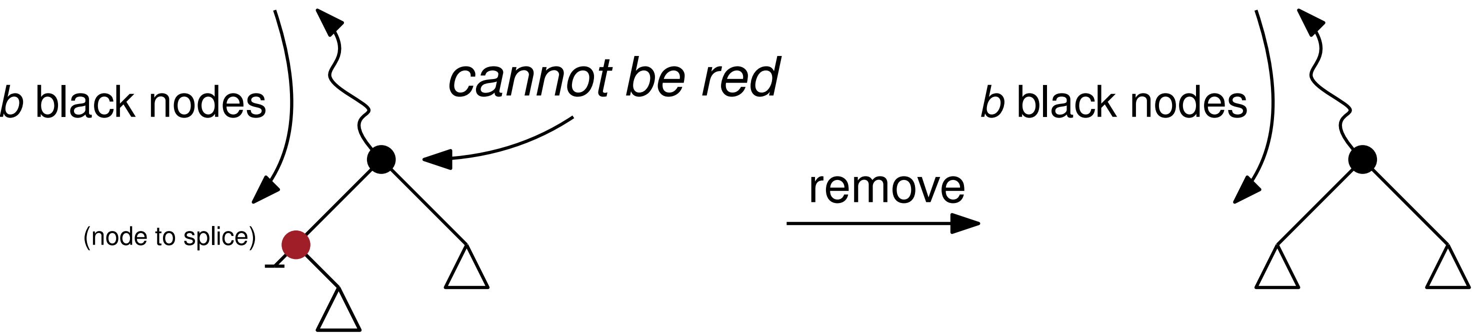
Red-black trees: insertion, case 2

Case 2: current red node *z* has a black uncle.



Red-black trees: deletion, case 0

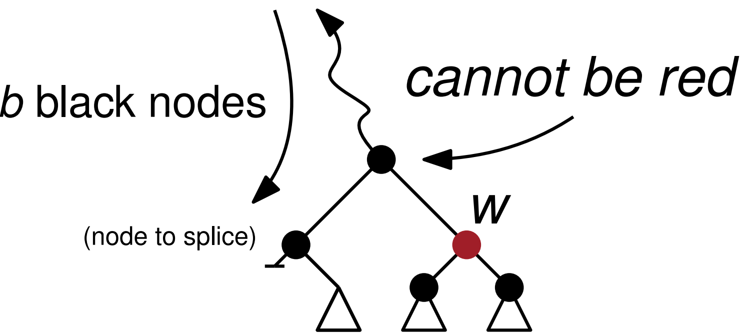
Case 0: spliced node is red.



$\rightarrow$ No rotations or recoloring required.

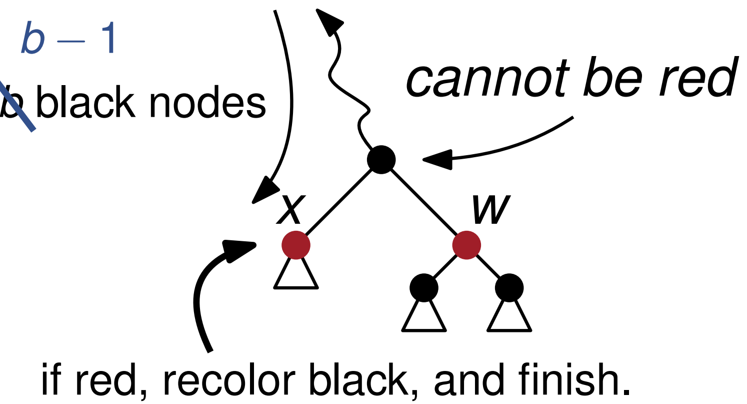
Red-black trees: deletion, cases 1 and 2

Case 1: spliced node is black and sibling is red.



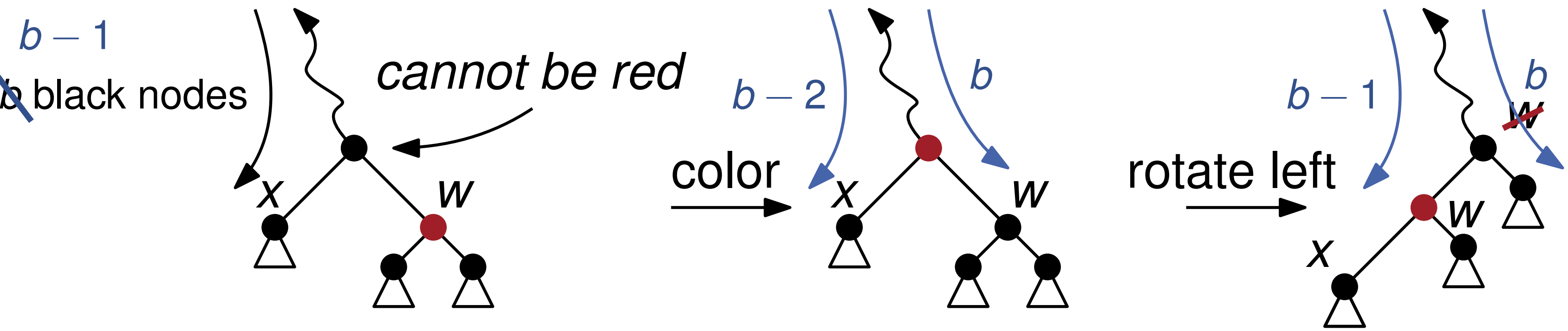
Red-black trees: deletion, cases 1 and 2

Case 1: spliced node is black and sibling is red.



Red-black trees: deletion, cases 1 and 2

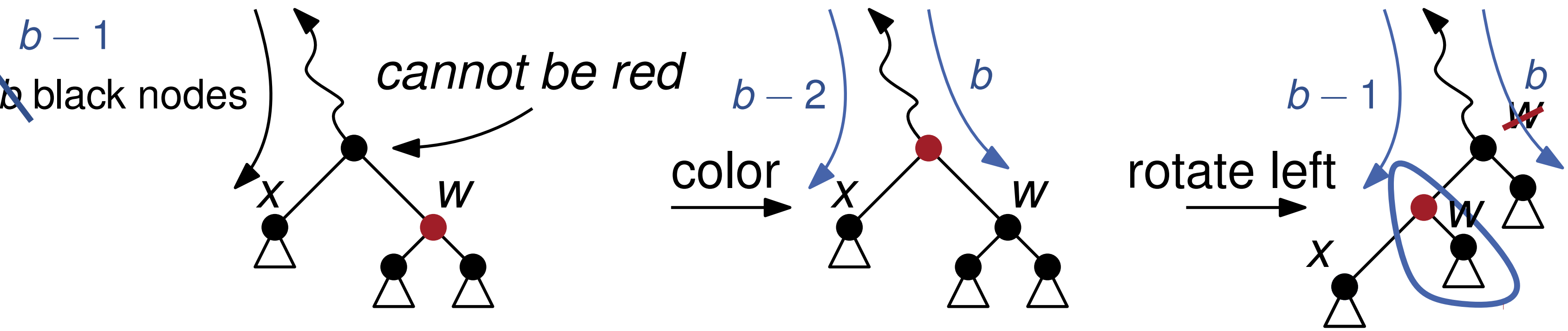
Case 1: spliced node is black and sibling is red.



(black node path lengths change)

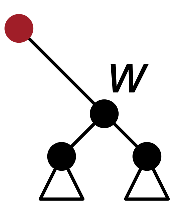
Red-black trees: deletion, cases 1 and 2

Case 1: spliced node is black and sibling is red.



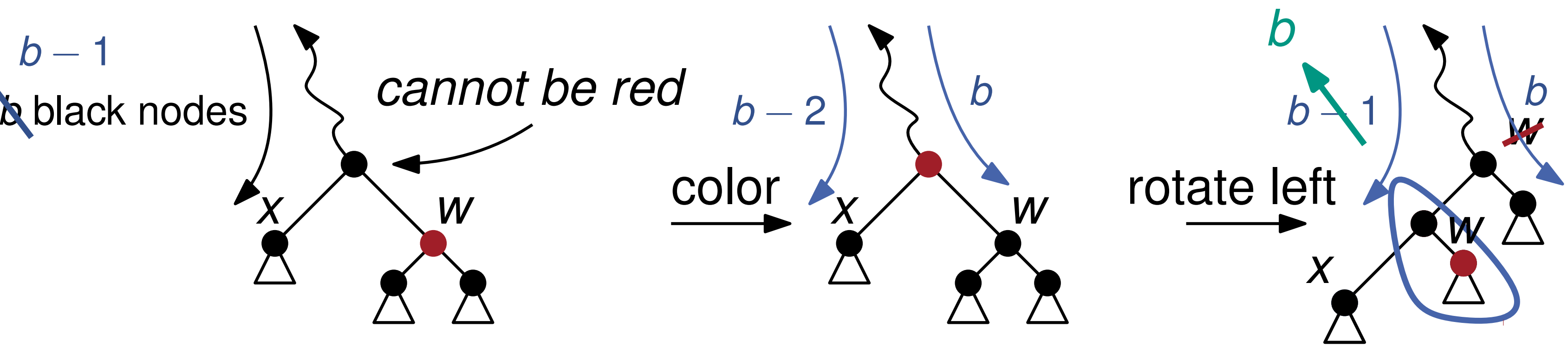
(black node path lengths change)

Case 2: make p[*w*] black, *w* red



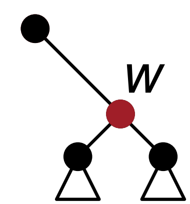
Red-black trees: deletion, cases 1 and 2

Case 1: spliced node is black and sibling is red.



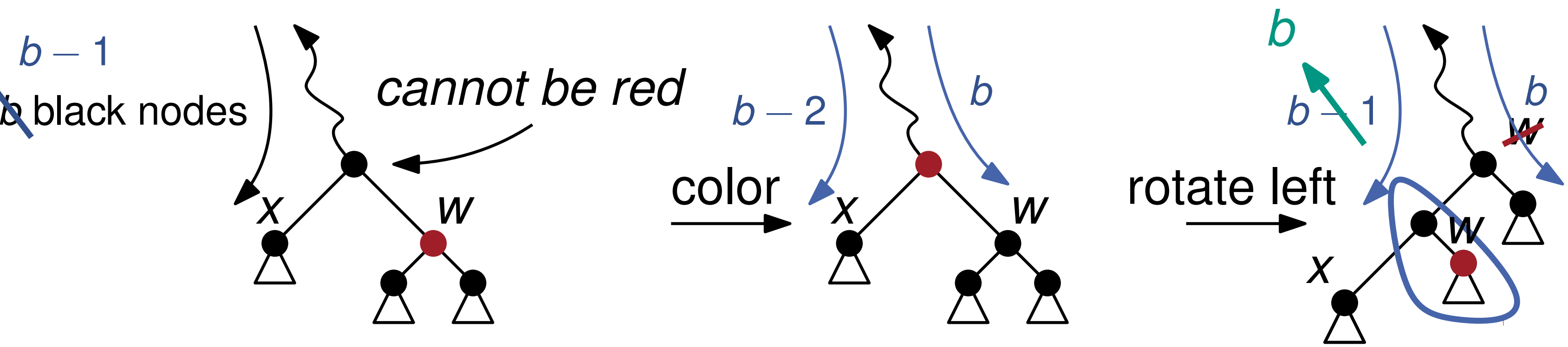
(black node path lengths change)

Case 2: make p[*w*] black, *w* red



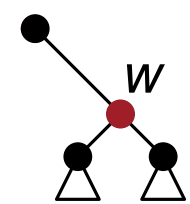
Red-black trees: deletion, cases 1 and 2

Case 1: spliced node is black and sibling is red.



(black node path lengths change)

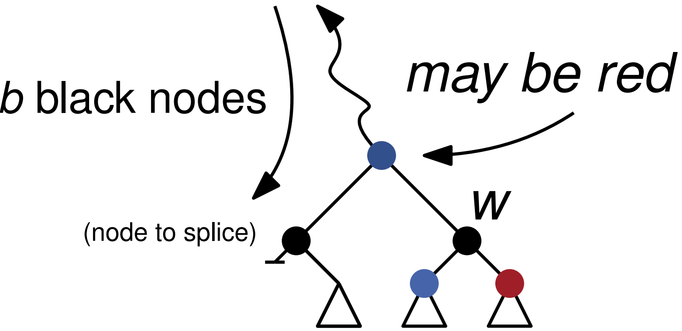
Case 2: make p[*w*] black, *w* red



If case 2 does not apply, then need to add black node with case 3 or 4…

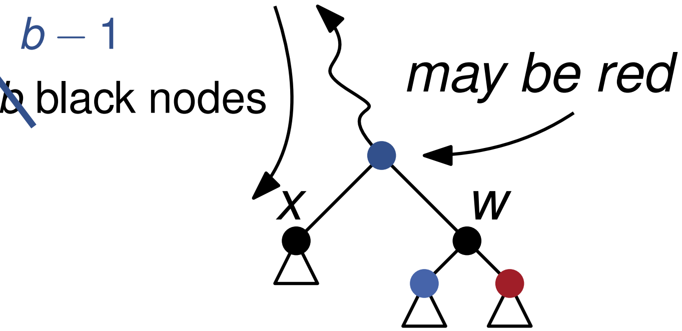
Red-black trees: deletion, cases 3 and 4

Cases 3/4: spliced node is black with a black sibling with a red child



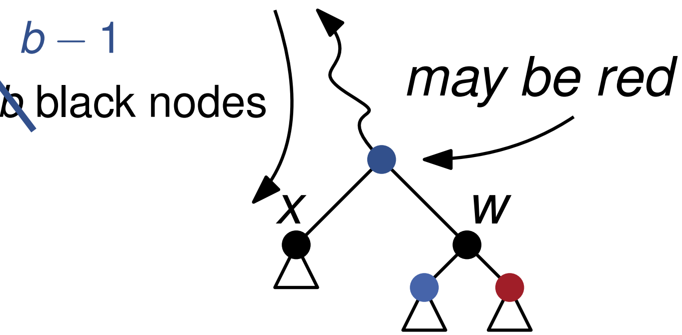
Red-black trees: deletion, cases 3 and 4

Cases 3/4: spliced node is black with a black sibling with a red child



Red-black trees: deletion, cases 3 and 4

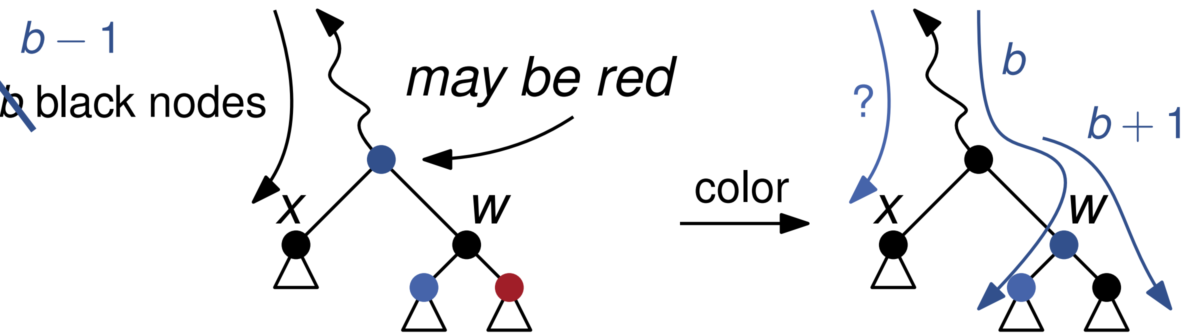
Cases 3/4: spliced node is black with a black sibling with a red right child



Case 4: Right child is red

Red-black trees: deletion, cases 3 and 4

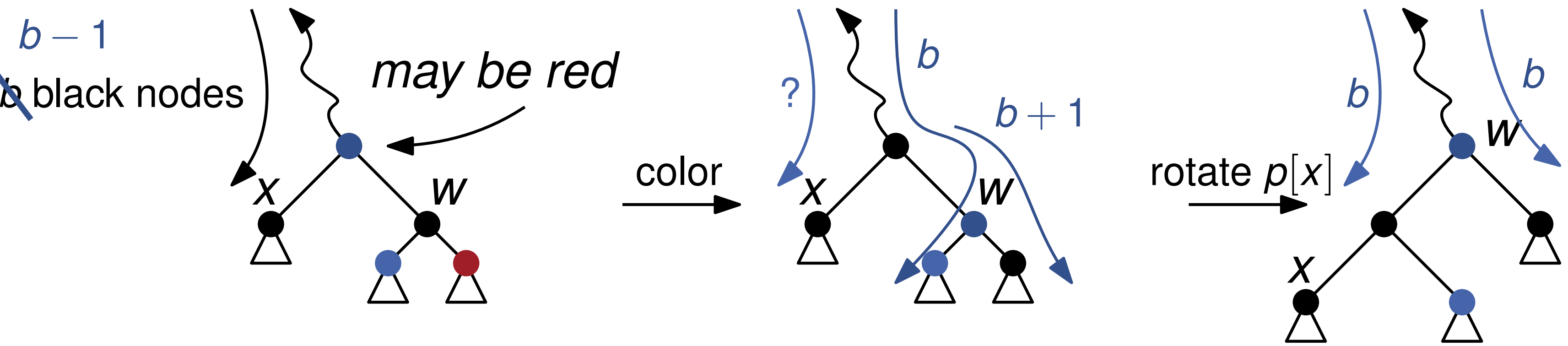
Cases 3/4: spliced node is black with a black sibling with a red right child



Case 4: Right child is red

Red-black trees: deletion, cases 3 and 4

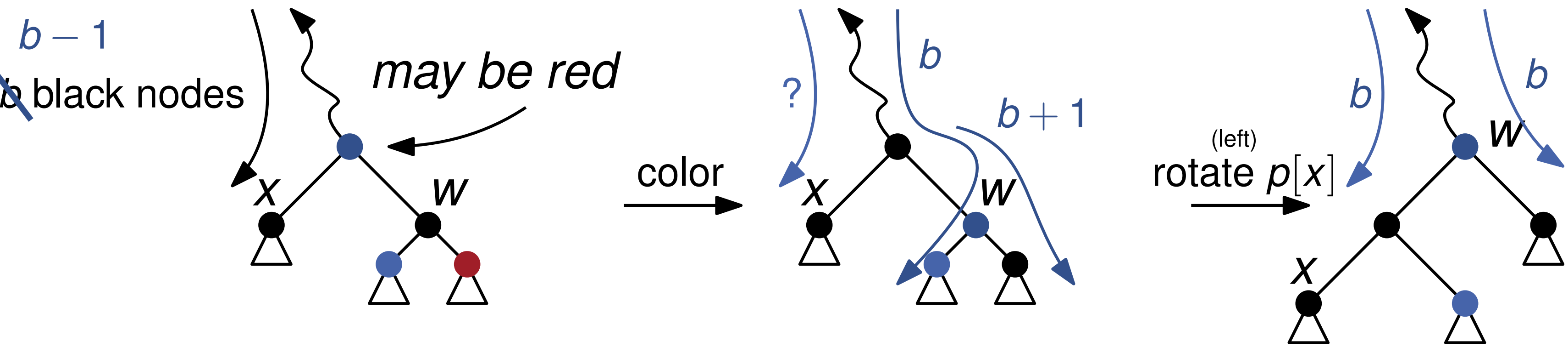
Cases 3/4: spliced node is black with a black sibling with a right red child



Case 4: Right child is red

Red-black trees: deletion, cases 3 and 4

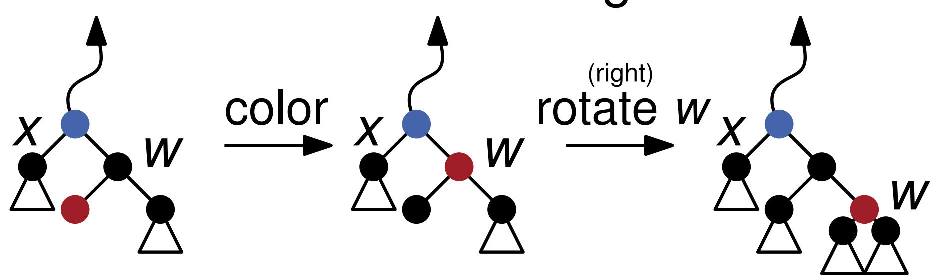
Cases 3/4: spliced node is black with a black sibling with a red child



Case 4: Right child is red

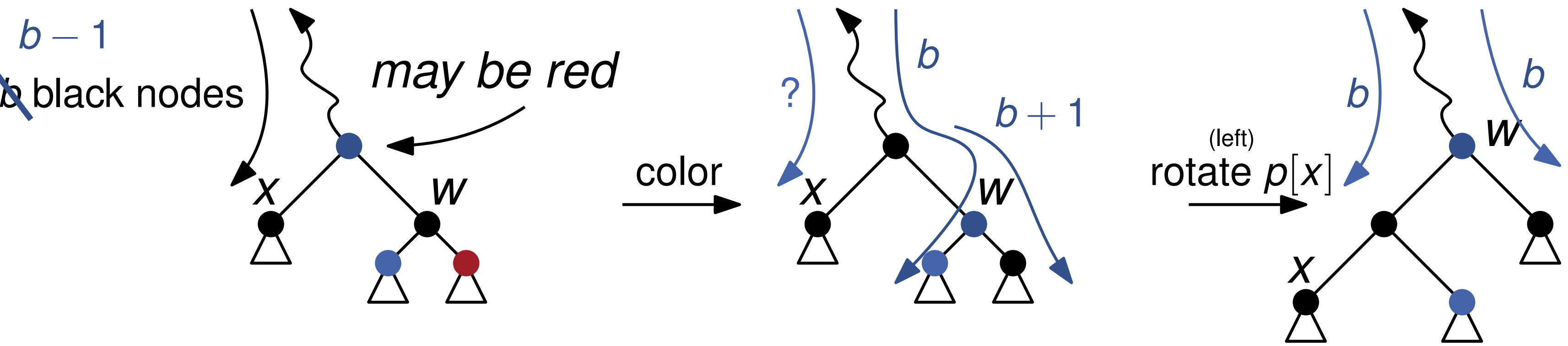
Case 3: Right child is black (and left child is red)

Color and rotate so that right child is red.



Red-black trees: deletion, cases 3 and 4

Cases 3/4: spliced node is black with a black sibling with a red child



Case 4: Right child is red

Case 3: Right child is black (and left child is red)

Color and rotate so that right child is red.