Red-Black Trees

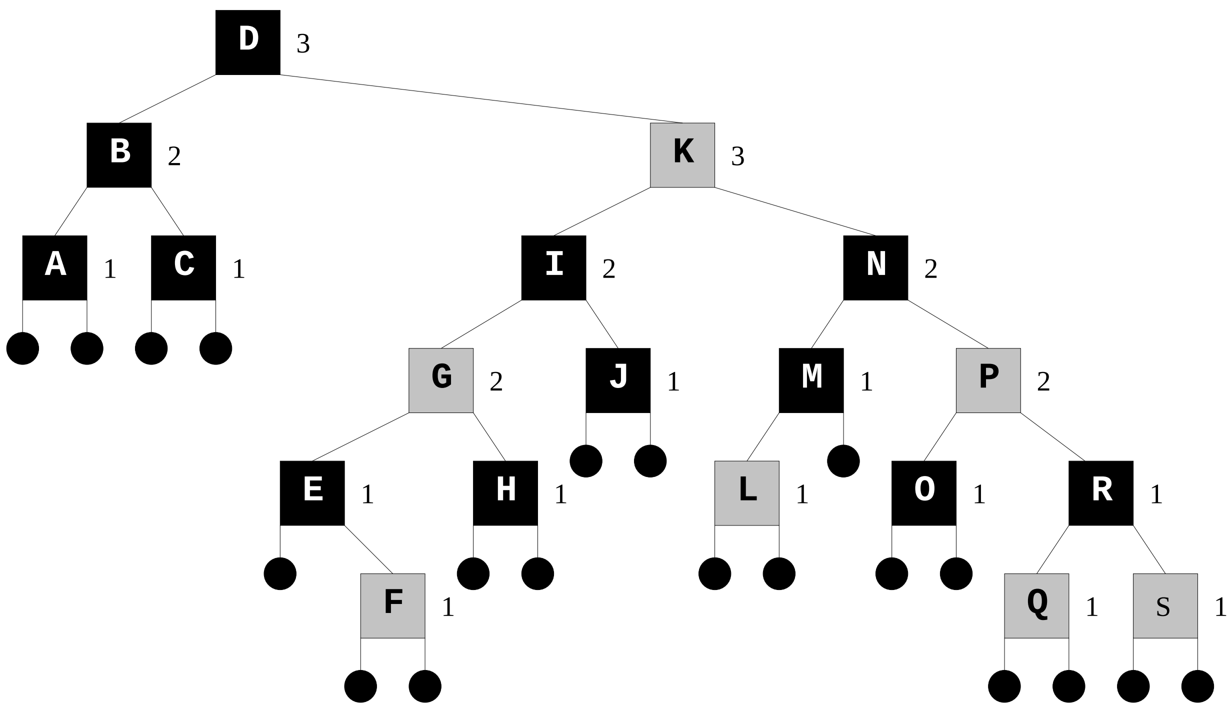
Reading: *CLRS* Chapter 13; 14.1 – 14.2; *CLR* Chapter 14; 15.1 – 15.2.

Definition

A **red-black tree (RBT)** is a binary search tree that satisfies the following **red-black properties**:

1. Every node has a color that is either red or black.
2. Every leaf is black.
3. If a node is red, both children are black.
4. Every path from a given node down to any descendant leaf contains the same number of black nodes. The number of black nodes on such a path (not including the initial node but including leaves) is called the **black-height (bh)** of the node.
5. The root of the tree is black (not a *CLR* property, but should be).

Example



Balance Property of Red-Black Trees

Consider a subtree with n nodes (non-leaves) rooted at any node x within a red-black tree. Then the following relationships hold:

- $\text{height}(x)/2 \leq \text{bh}(x) \leq \text{height}(x)$
- $2^{\text{bh}(x)} - 1 \leq n < 2^{\text{height}(x)}$
- $\lg(n) < \text{height}(x) \leq 2\lg(n+1)$

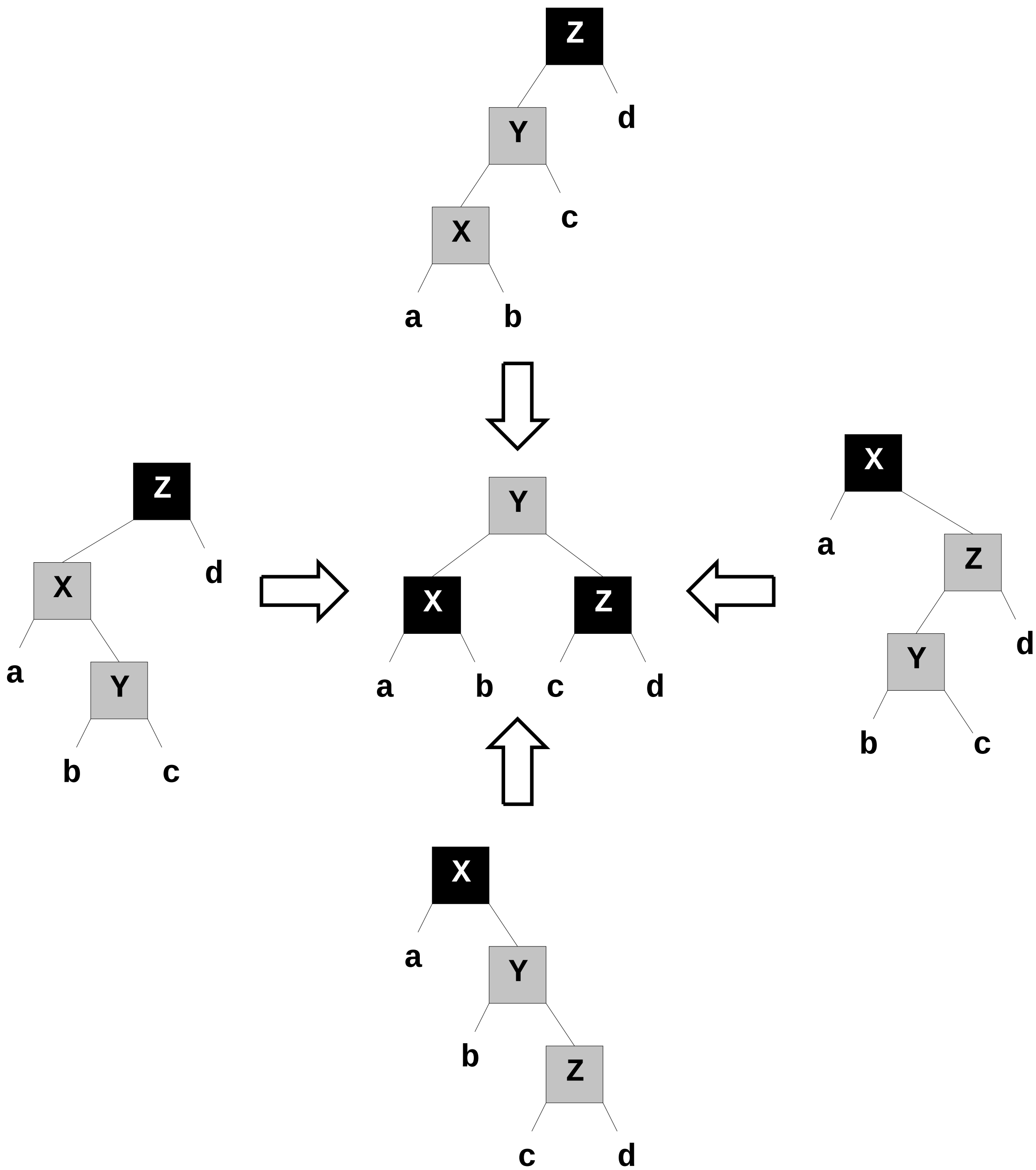
The last relationship is the sense in which a red-black tree is balanced.

Red-Black Tree Insertion

To insert value V into red-black tree T:

Step 1: Use usual BST insertion algorithm, coloring new node red. RBT properties (1), (2), and (4) do not change. RBT property (3) will not hold if there is a **red-red violation**.

Step 2: Remove any red-red violation via the following rotation rules.[1] It may be necessary to apply the rules multiple times. What is the maximal number of times the rules can be applied in the worst case?

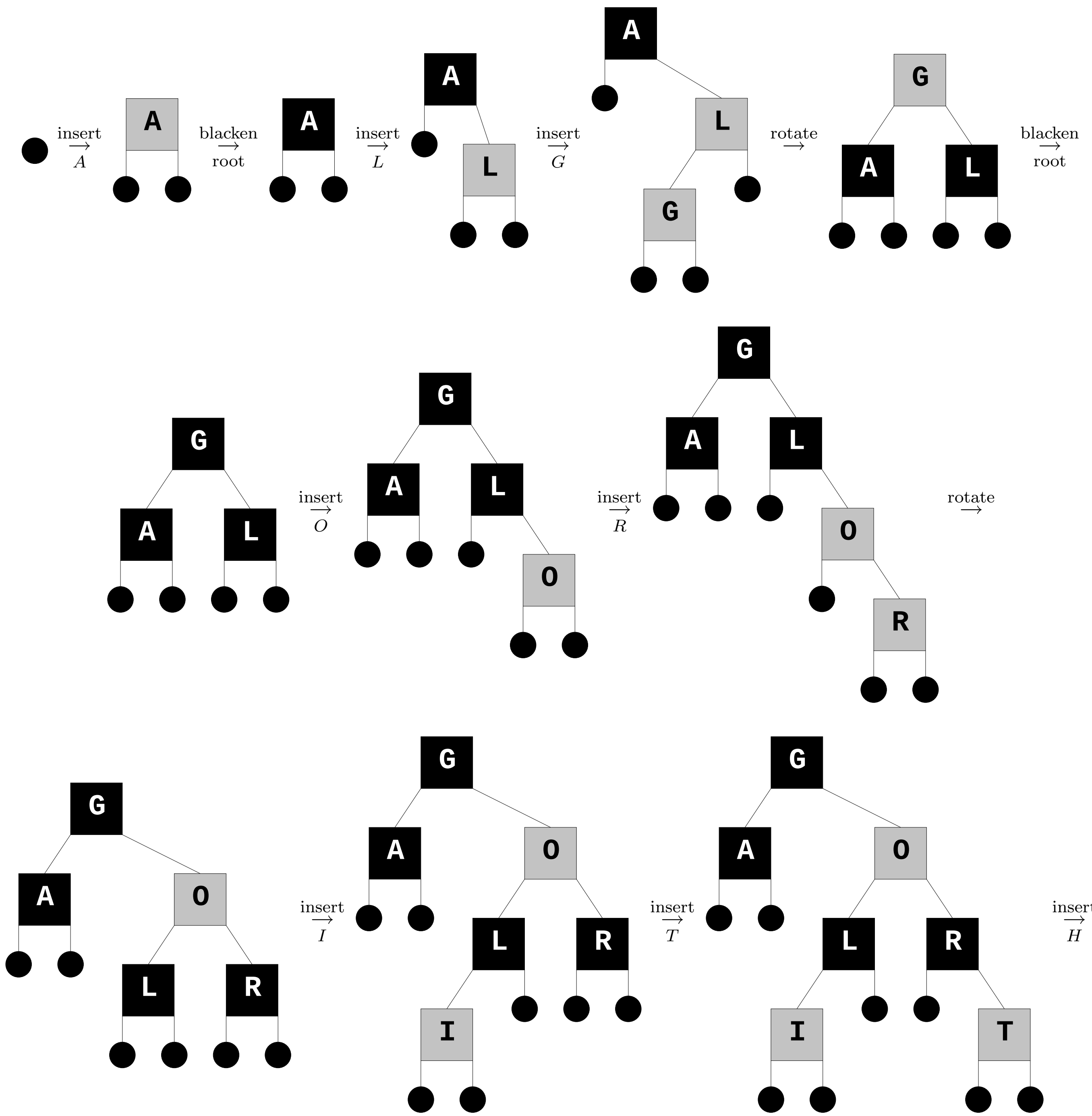


Step 3: If Step 1 or Step 2 leaves the root of the tree red, reassert RBT property (5) by blackening the root. Why is this necessary? (Hint: look at the rules in Step 2.)

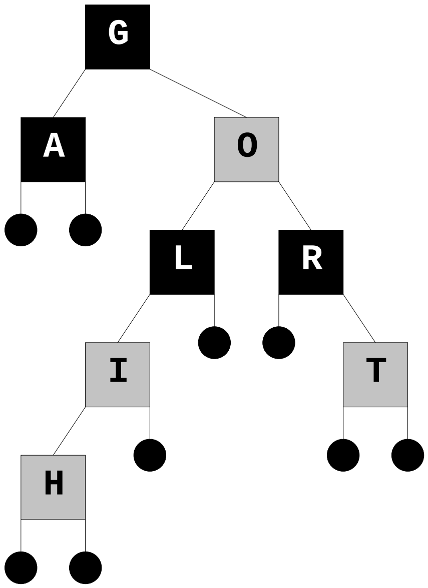
[1]These are simpler, but less efficient than, those in CLR. They are due to Chris Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1998.

Red-Black Tree Insertion Example

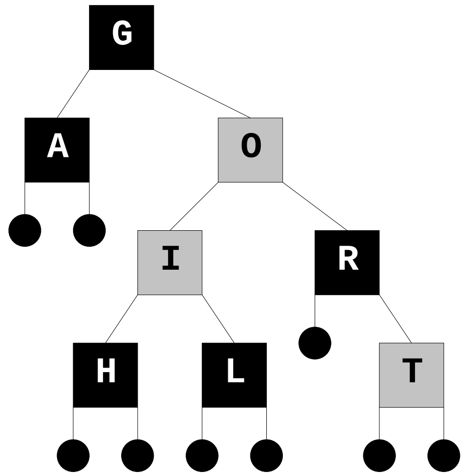
Insert the letters `A L G O R I T H M` in order into a red-black tree.



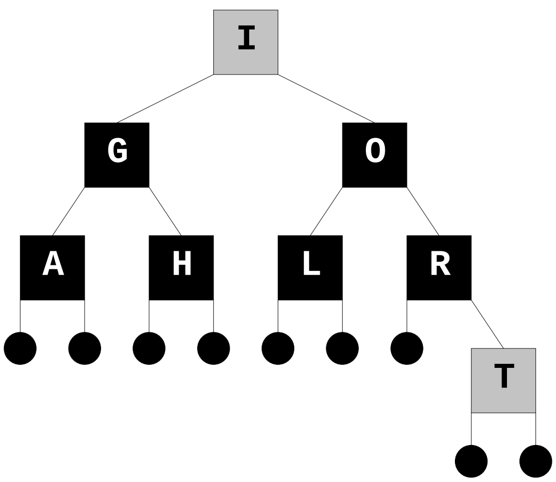
Red-Black Tree Insertion Example (continued)



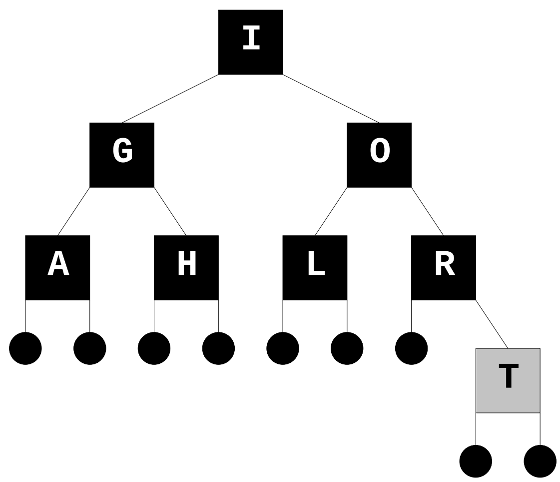
$\xrightarrow{\text{rotate}}$



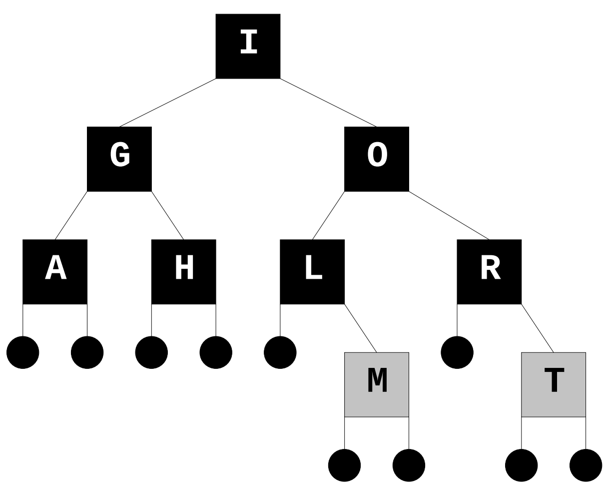
$\xrightarrow{\text{rotate}}$



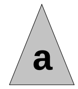
$\underset{\text{root}}{\xrightarrow{\text{blacken}}}$

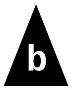


$\underset{M}{\xrightarrow{\text{insert}}}$



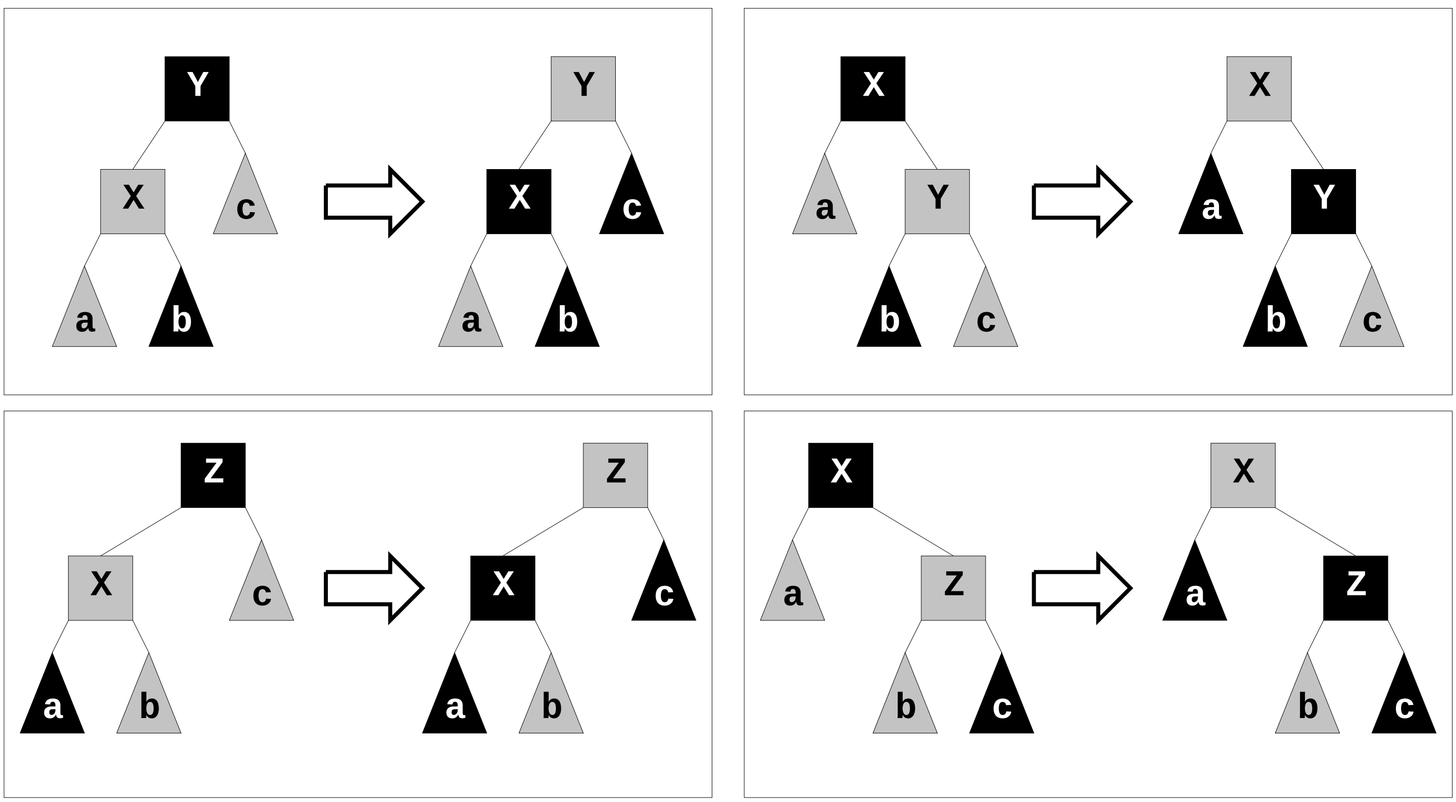
More Efficient Red-Red Violation Elimination, Part 1

The rotation rules presented earlier for eliminating red-red violations are simple but can require a number of rotations proportional to the height of the tree. *CLR* present more complex but efficient rules. Here, we present their rules in a different style, using the notation 

to stand for a red-black tree named `a` rooted at a red node and 

to stand for a red-black tree named `b` rooted at a black node.

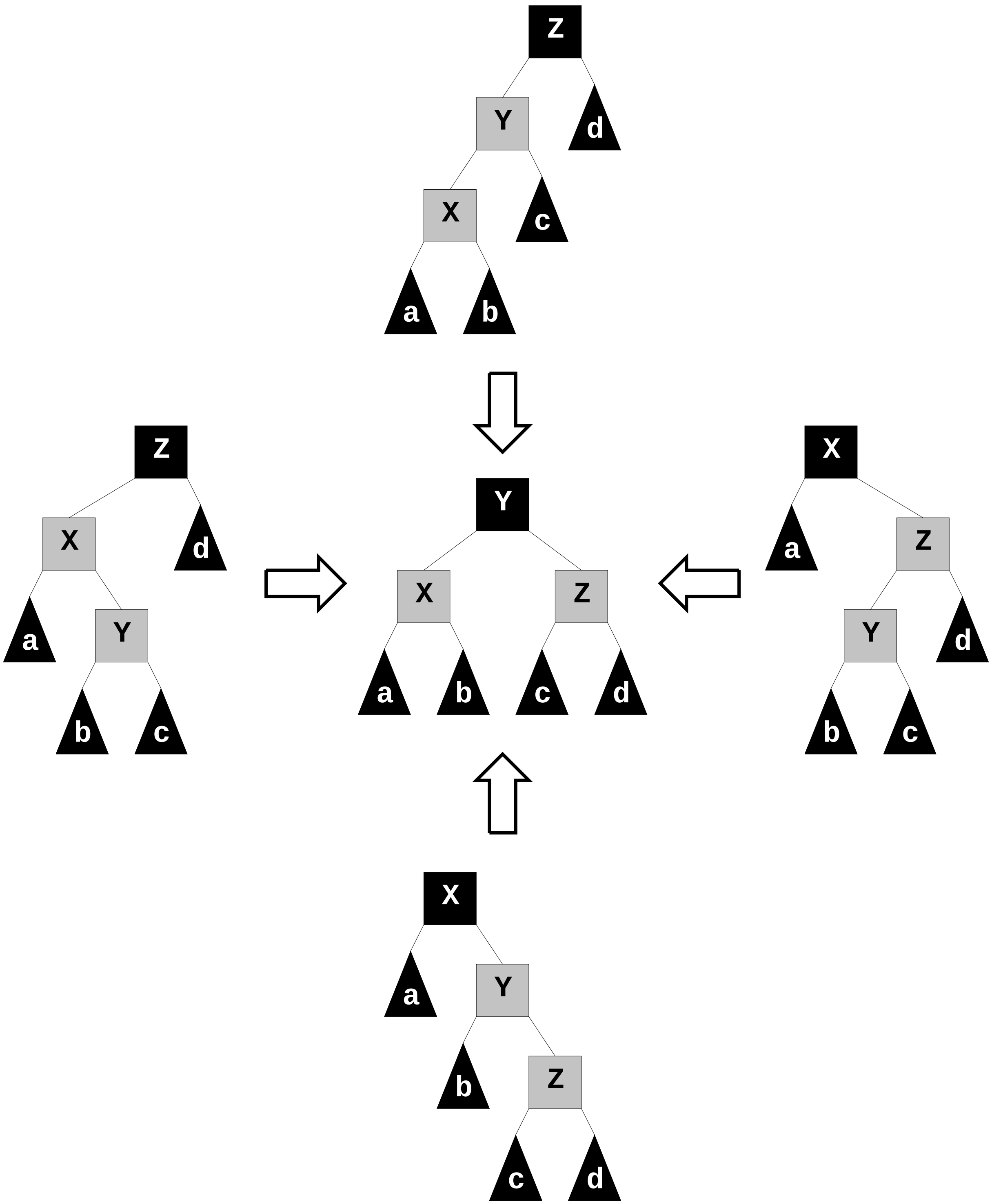
Case 1 of *CLR*'s `RB-Insert` handles the case where the sibling of the top red node N of red-red violation is red. In this case, the blackness of N's parent can be distributed among N and its sibling, as illustrated by the following rules:



Note that no rotations are required, but the red-red violation may move up the tree.

More Efficient Red-Red Violation Elimination, Part 2

Case 2 and Case 3 of *CLR*'s `RB-Insert` handle the situation where the sibling of the top red node N of red-red violation is black. In this case, a single rotation[2] eliminates the red-red violation, as illustrated by the following rules:



This more efficient approach of eliminating red-red violations needs at most $\Theta(\lg(n))$ rule applications and one rotation.

[2] *CLR* presents the rules in such a way that two rotations may be required, but this can be optimized to one.

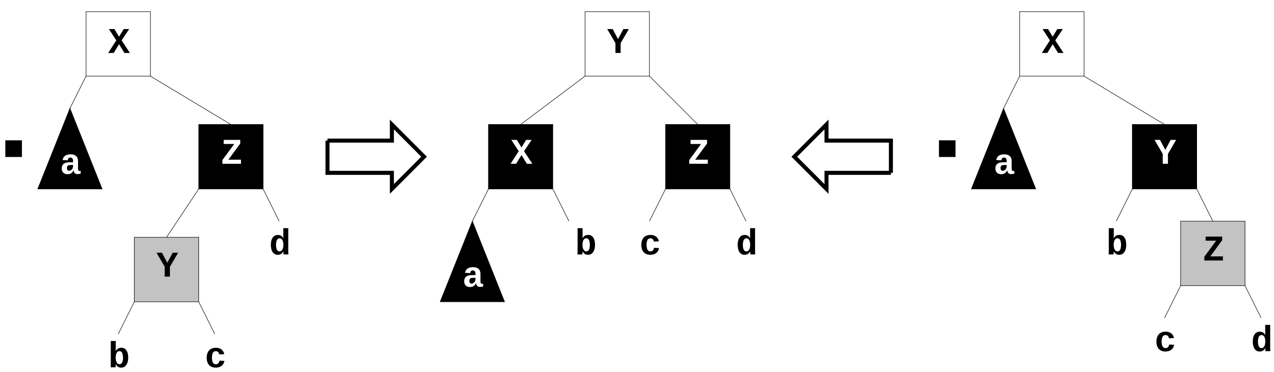
Red-Black Tree Deletion

To delete value V from red-black tree T:

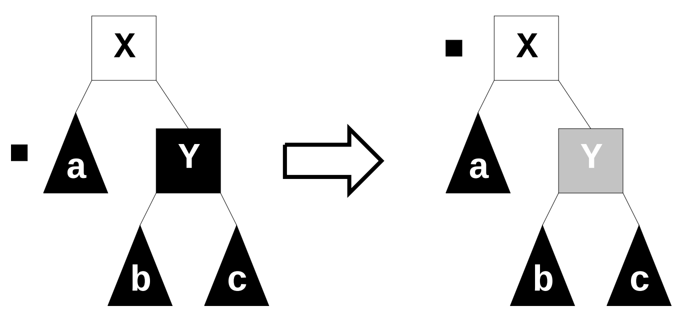
Step 1: Use usual BST deletion algorithm, replacing a deleted node by its predecessor (or successor) in the case where neither child is a leaf. Let N be the node with a child leaf that is deleted. If N is black, property (4) (uniformity of black-height) is violated. Reassert it by making the "other" child of N **doubly-black**

Step 2: Propagate double-blackness up the tree using the following rules.[3] The blackness token ■ turns a black node doubly-black and turns a red node black. Using these rules, the black height invariant can be reasserted with $\Theta(\lg(n))$ rule applications and at most 2 rotations.

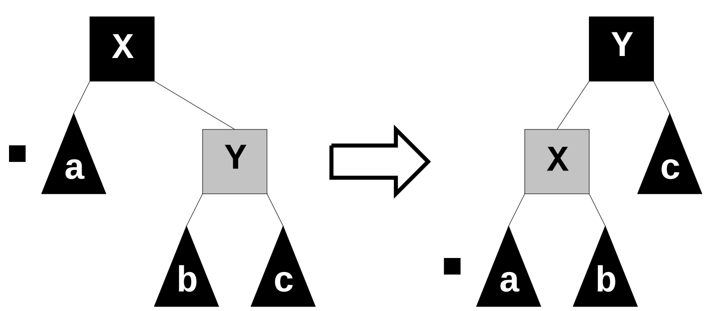
- A. *The sibling of the doubly-black node is black and one nephew is red* (*CLR*'s `RB-Delete` Cases 3 and 4). This rule eliminates the blackness token with one rotation.



- B. *The sibling and both nephews of the doubly-black node are black* (*CLR*'s `RB-Delete` Case 2). This rule propagates the blackness token upward without rotation.



- C. *The sibling of the doubly-black node is red* (*CLR*'s `RB-Delete` Case 1). This enables Case A or B.

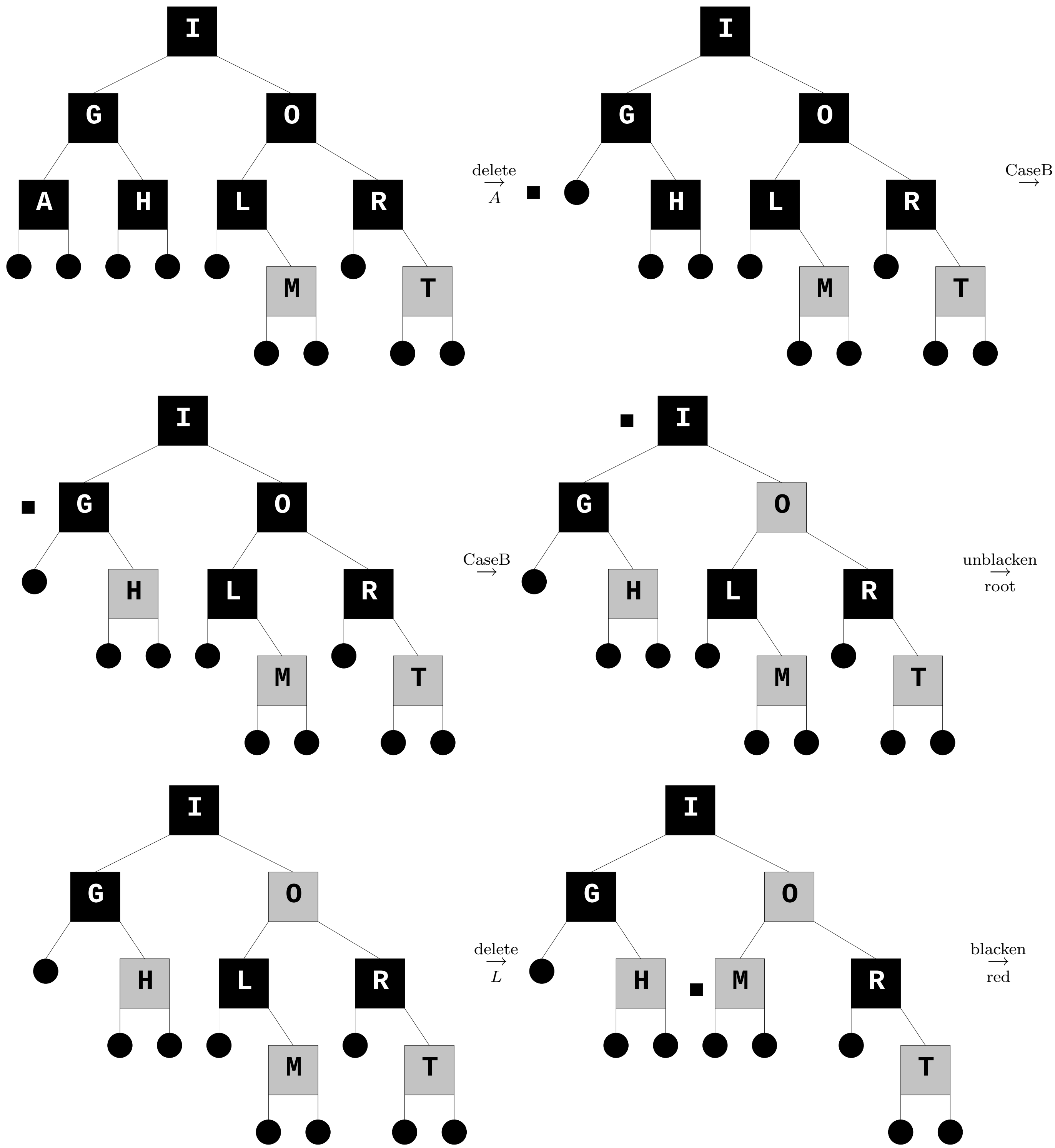


Step 3: If the doubly black token progagates to root, remove it.

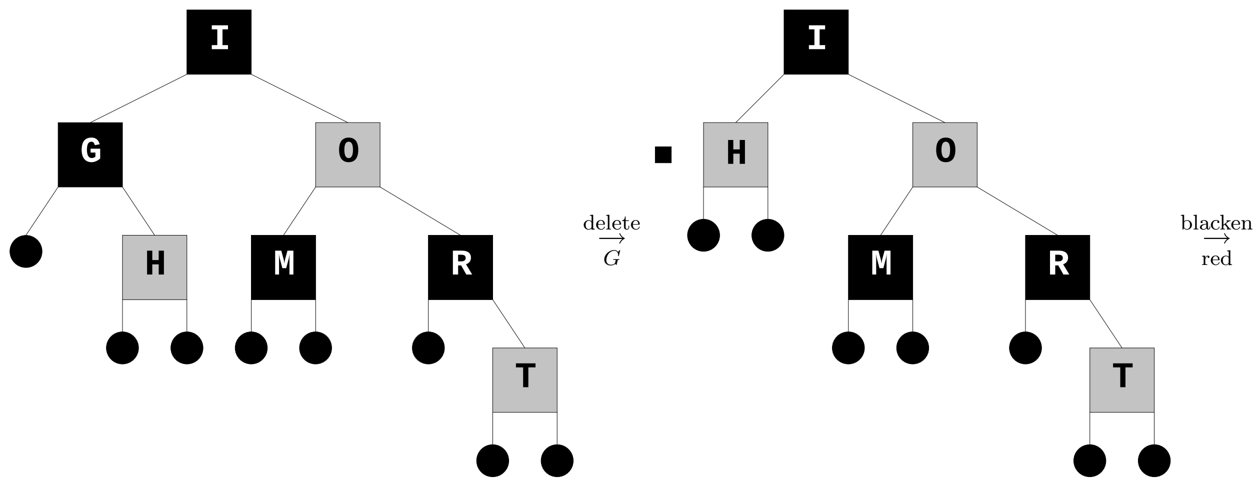
[3] Only the cases where the doubly-black node is a left child are shown; the cases where the doubly-black node is a right child are symmetric.

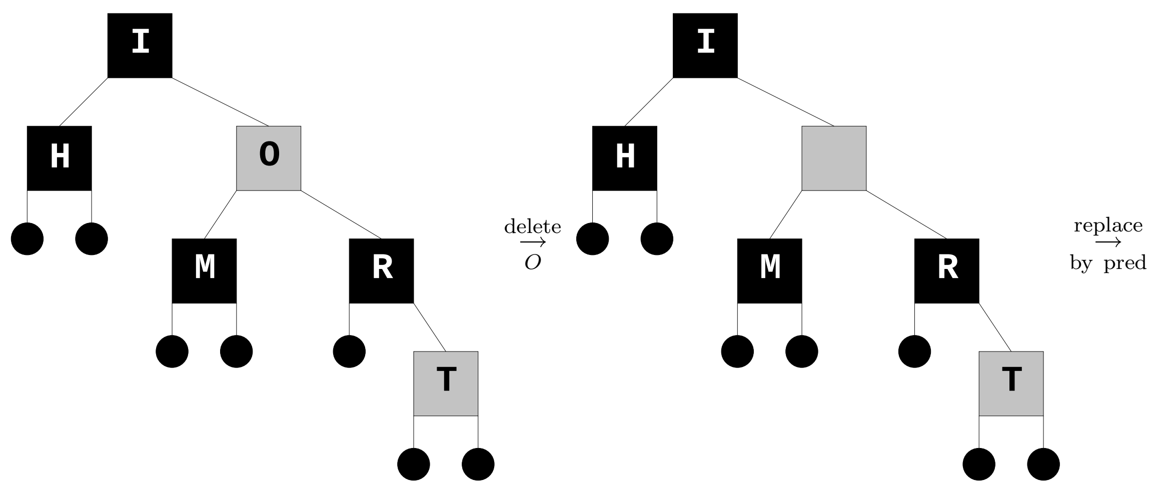
Red-Black Tree Deletion Example

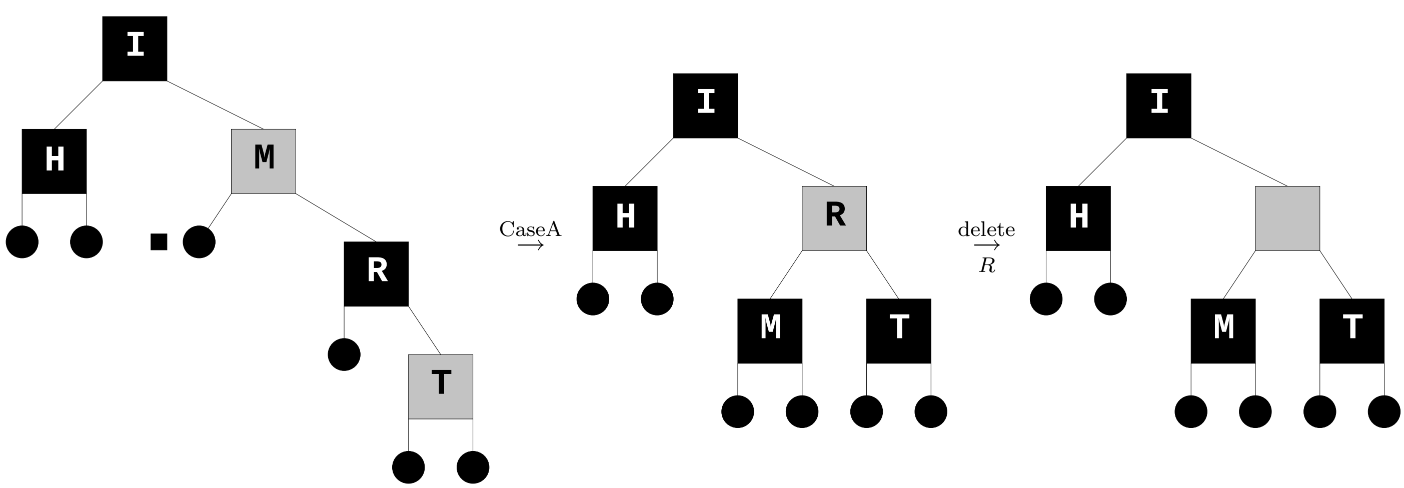
Delete the letters `A L G O R I T H M` in order from the red black tree constructed before.



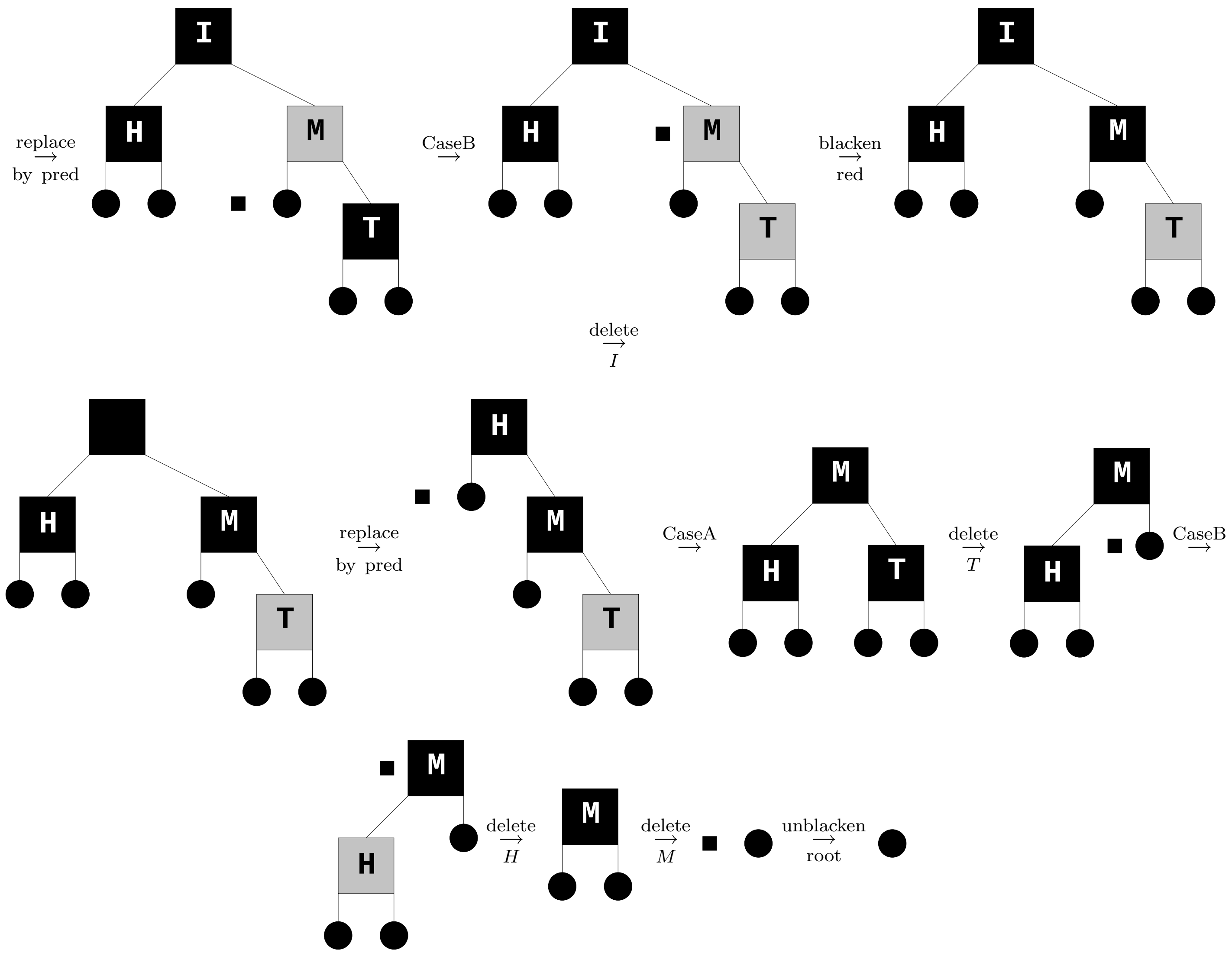
Red-Black Tree Deletion Example (continued)







Red-Black Tree Deletion Example (continued)



Augmenting Red-Black Trees

Can often improve running time of additional operations on a data structure by caching extra information in the header node or data nodes of a data structure. Must insure that this information can be updated efficiently for other operations.
Examples:

1. Store in a header node the length of a linked or doubly-linked list.

2. Store in a header node the maximum value of a linked list representation of a bag. Can also store a pointer to the list node containing the maximum value. Does it matter whether the list is sorted vs. unsorted? Linked vs. doubly-linked?

3. Store the size of every red-black subtree in the root of that subtree.

   ```
   size[leaf] =  0
   size[node] = 1 + size[left[node]] + size[right[node]]
   ```

   Can use size field to:

   - Determine size of tree in $\Theta(1)$ worst-case time.
   - Perform Select(T, k) (i.e. find the kth order statistic) in $\Theta(\lg(n))$ worst-case time.
   - Determine the rank of a given key x in $\Theta(\lg(n))$ worst-case time.

   `Insert` and `Delete` can update the size field efficiently (i.e., without changing the asymptotic running time of `Insert` and `Delete`):

   Insert: In downward phase searching for insertion point, increment sizes by one. In upward "fix-up" phase, update sizes at each rotation.

   Delete: After deleting node y, decrement sizes on path to root[T] by one. In upward "fix-up" phase, update sizes at each rotation.

In general, can efficiently augment every node x of a red-black tree with the a field that stores the result of any function f that depends only on `key[x]`, `f(left[x])`, and `f(right[x])`.

- for `Insert`, field only needs to be updated on path from insertion point to root.
- for `Delete`, only needs to be updated on path from deletion point to root.
- easy to update at every rotation.