CPTS 223 Advanced Data Structure C/C++

Tree: Red-Black Trees

Overview

- Tree data structure
- Binary search trees
  - Support O(lg(n)) operations
  - Balanced trees
- STL set and map classes
- B-trees for accessing secondary storage
- Applications of Tree

This time: RB trees (practically used self-balanced BSTs)

Red-black trees

- Operations take worst case O(log N) time
- Less rotations or no rotations (reduces stack overhead)
- It has some interesting properties that generate the kinds of behavior we want (not as obvious why at first).
- Unlike Splay Trees, Red-Black trees definitely in use
  - e.g., STL set and map

R-B trees v.s. AVL trees?

- Search: AVL trees provide slightly faster lookups than R-B trees because of their stricter balance
- Insertion/Deletion: R-B trees provide faster insertion and removal since they end up with fewer rotations due to less strict balance
- Storage: AVL height information must be an int while a R-B color can be a bit
- R-B trees are used in STL like maps, multimap, multiset in C++ while AVL trees are used more in databases for faster retrievals

R-B trees v.s. AVL trees?

After inserting same number of items, R-B Tree is slightly higher/deeper than AVL Tree

- Search: AVL trees provide slightly faster lookups than R-B trees because of their stricter balance

Recall AVL condition

- Insertion/Deletion: R-B trees provide faster insertion and removal since they end up with fewer rotations due to less strict balance
- Storage: AVL height information must be an int while a R-B color can be a bit
- R-B trees are used in STL like maps, multimap, multiset in C++ while AVL trees are used more in databases for faster retrievals

R-B trees: definition

- It is a BST
- Every node is colored either red or black
- The root is black
- If a node is red, its children must be black (no adjacent red nodes)
- Every path from a node to a null pointer must contain the same number of black nodes
- Null pointers (NIL nodes) are treated as black nodes

R-B coloring rules

R-B trees: definition

- It is a BST
- Every node is colored either red or black
- The root is black
- If a node is red, its children must be black (no adjacent red nodes)
- Every path from a node to a null pointer must contain the same number of black nodes
- Null pointers (NIL nodes) are treated as black nodes

Node color property

R-B coloring rules

R-B trees: definition

- It is a BST
- Every node is colored either red or black
- The root is black
- If a node is red, its children must be black (no adjacent red nodes)
- Every path from a node to a null pointer must contain the same number of black nodes
- Null pointers (NIL nodes) are treated as black nodes

Root property

R-B coloring rules

R-B trees: definition

- It is a BST
- Every node is colored either red or black
- The root is black
- If a node is red, its children must be black (no adjacent red nodes)
- Every path from a node to a null pointer must contain the same number of black nodes
- Null pointers (NIL nodes) are treated as black nodes

Red property

R-B coloring rules

R-B trees: definition

- It is a BST
- Every node is colored either red or black
- The root is black
- If a node is red, its children must be black (no adjacent red nodes)
- Every path from a node to a null pointer must contain the same number of black nodes
- Null pointers (NIL nodes) are treated as black nodes

R-B coloring rules

Black property

R-B trees: definition

- It is a BST
- Every node is colored either red or black
- The root is black
- If a node is red, its children must be black (no adjacent red nodes)
- Every path from a node to a null pointer must contain the same number of black nodes
- Null pointers (NIL nodes) are treated as black nodes

R-B coloring rules

Black property

Black height: the number of black nodes on the path from root to a leaf node

R-B trees: definition

- It is a BST
- Every node is colored either red or black
- The root is black
- If a node is red, its children must be black (no adjacent red nodes)
- Every path from a node to a null pointer must contain the same number of black nodes
- Null pointers (NIL nodes) are treated as black nodes

R-B coloring rules

Black property

Black height: the number of black nodes on the path from root to a leaf node

Black height >= h/2

R-B trees: definition

- It is a BST
- Every node is colored either red or black
- The root is black
- If a node is red, its children must be black (no adjacent red nodes)
- Every path from a node to a null pointer must contain the same number of black nodes
- Nu[illegible]inters (NIL nodes) are treated as black nodes

R-B coloring rules

Black property

Black height: the number of black nodes on the path from root to a leaf node

Black height >= h/2

h <= 2 log2(n+1) with n nodes

R-B trees: definition

- It is a BST
- Every node is colored either red or black
- The root is black
- If a node is red, its children must be black (no adjacent red nodes)
- Every path from a node to a null pointer must contain the same number of black nodes
- Null pointers (NIL nodes) are treated as black nodes

R-B coloring rules

Leaf property

R-B trees: definition

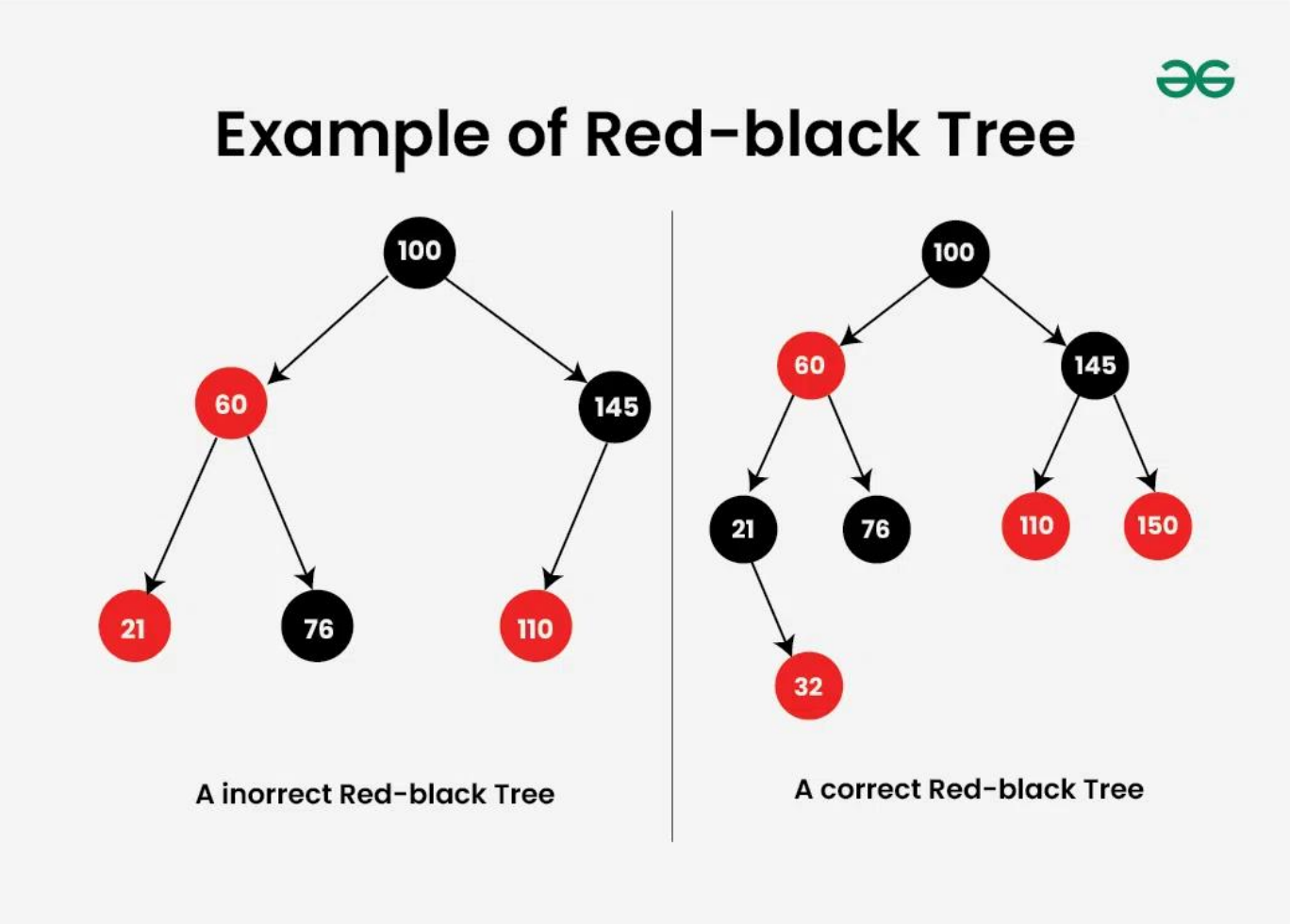
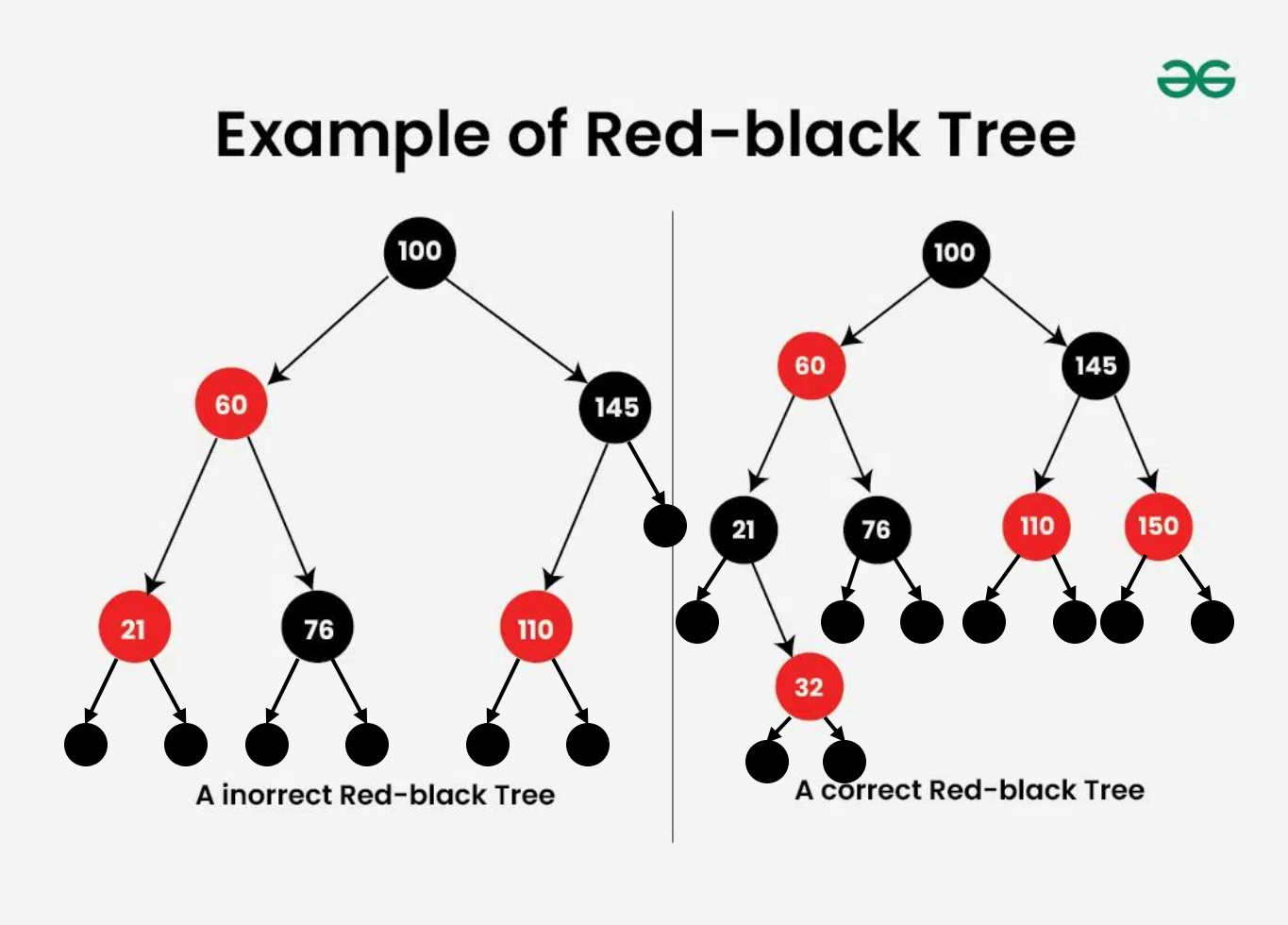


Image credit: https://www.geeksforgeeks.org/introduction-to-red-black-tree/

R-B trees: definition



Leaf property

Null pointer (NIL nodes): treated as black leaf nodes

Image credit: https://www.geeksforgeeks.org/introduction-to-red-black-tree/

R-B trees: definition

Violation of red property:
If a node is red, its children must be black (no adjacent red nodes)

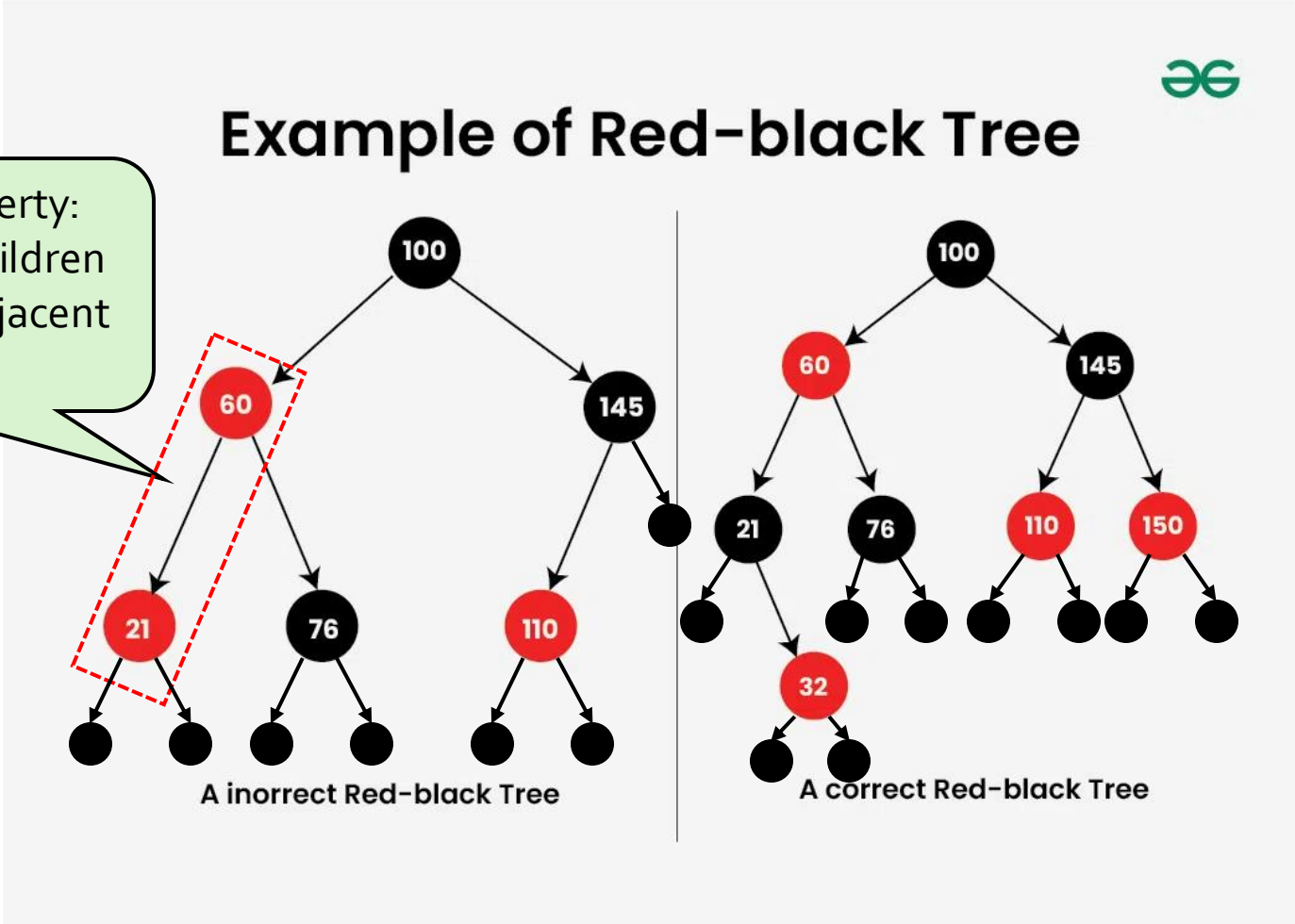
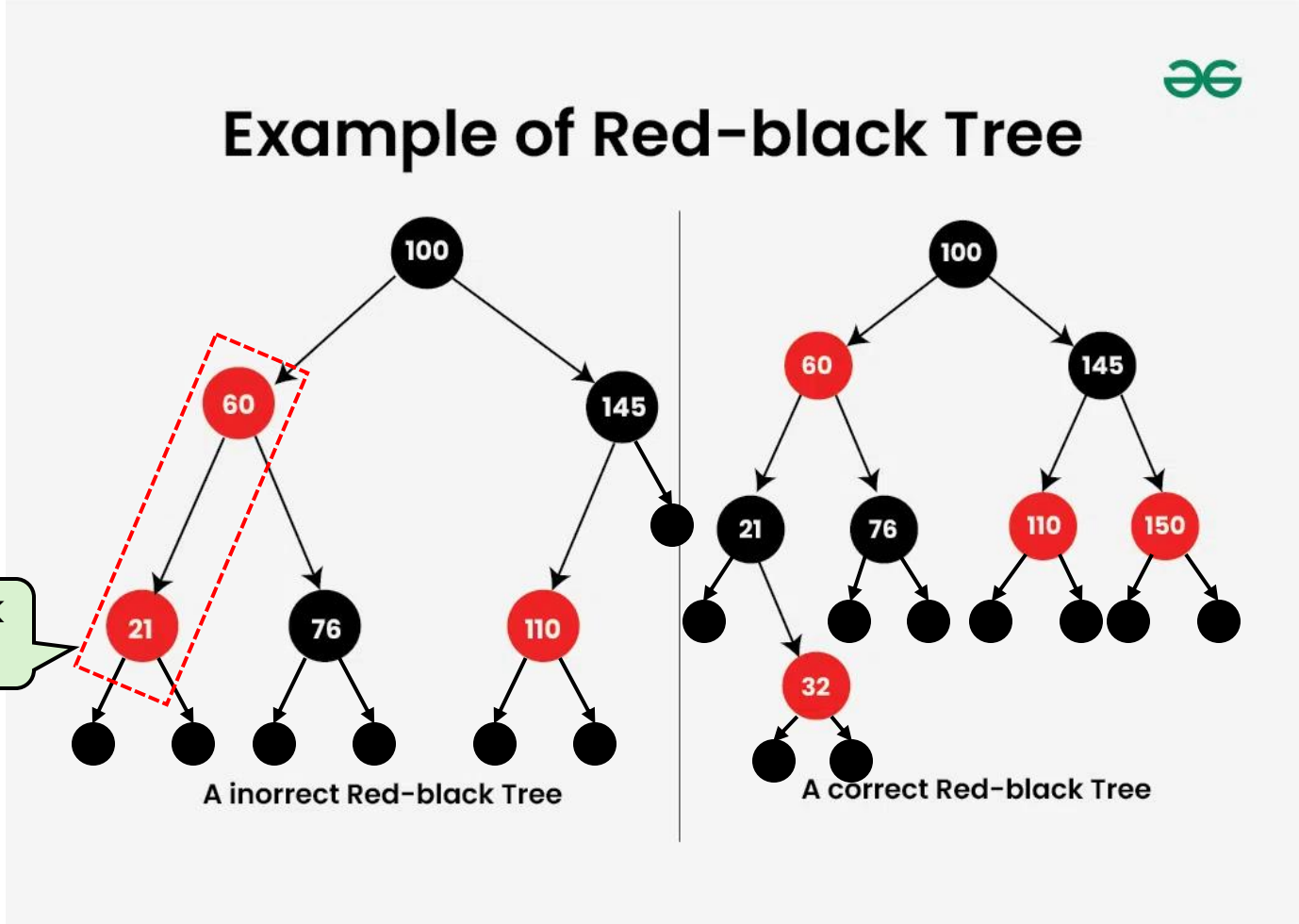


Image credit: https://www.geeksforgeeks.org/introduction-to-red-black-tree/

R-B trees: definition



This path: # of black nodes = 0

Image credit: https://www.geeksforgeeks.org/introduction-to-red-black-tree/

R-B trees: definition

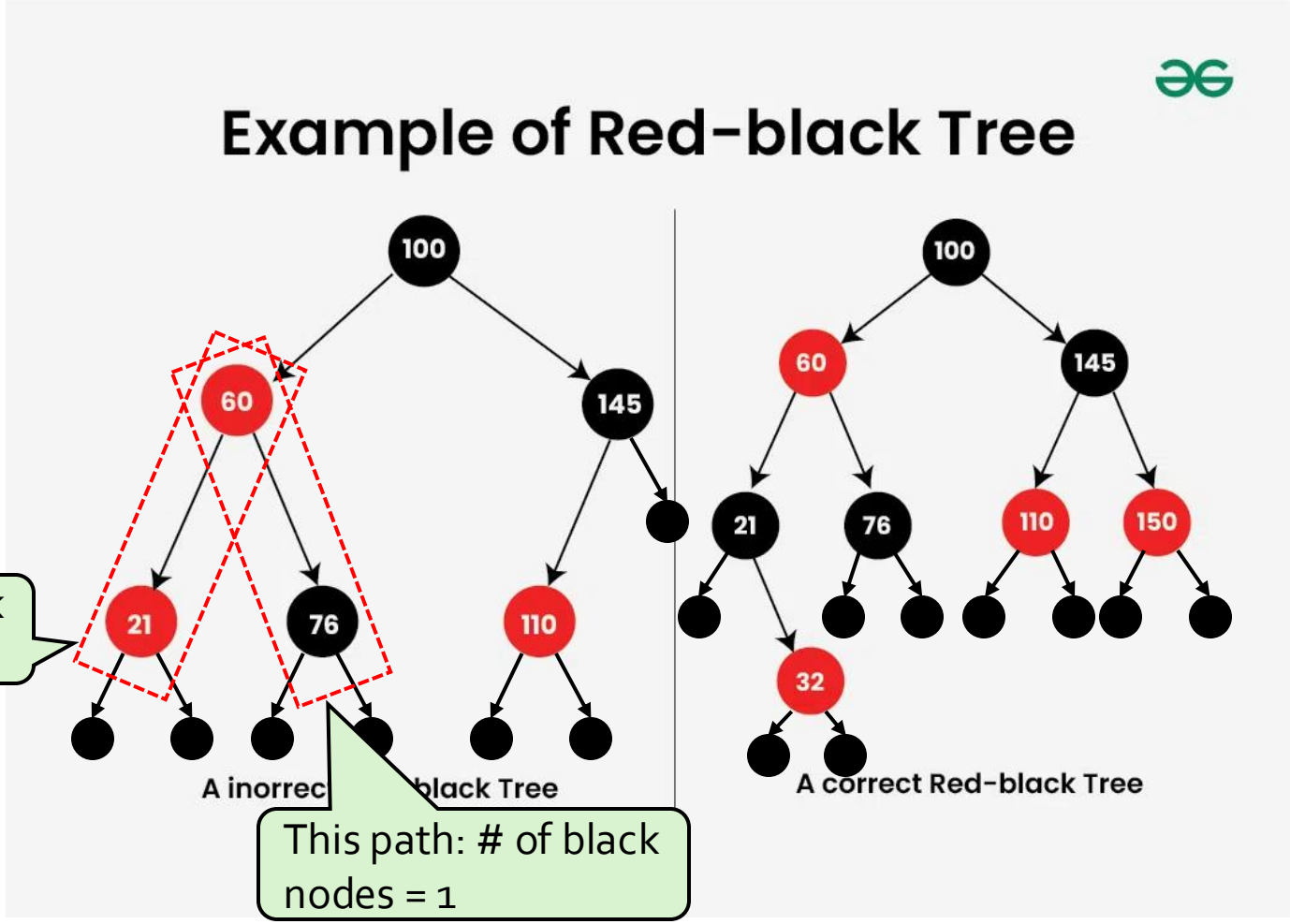
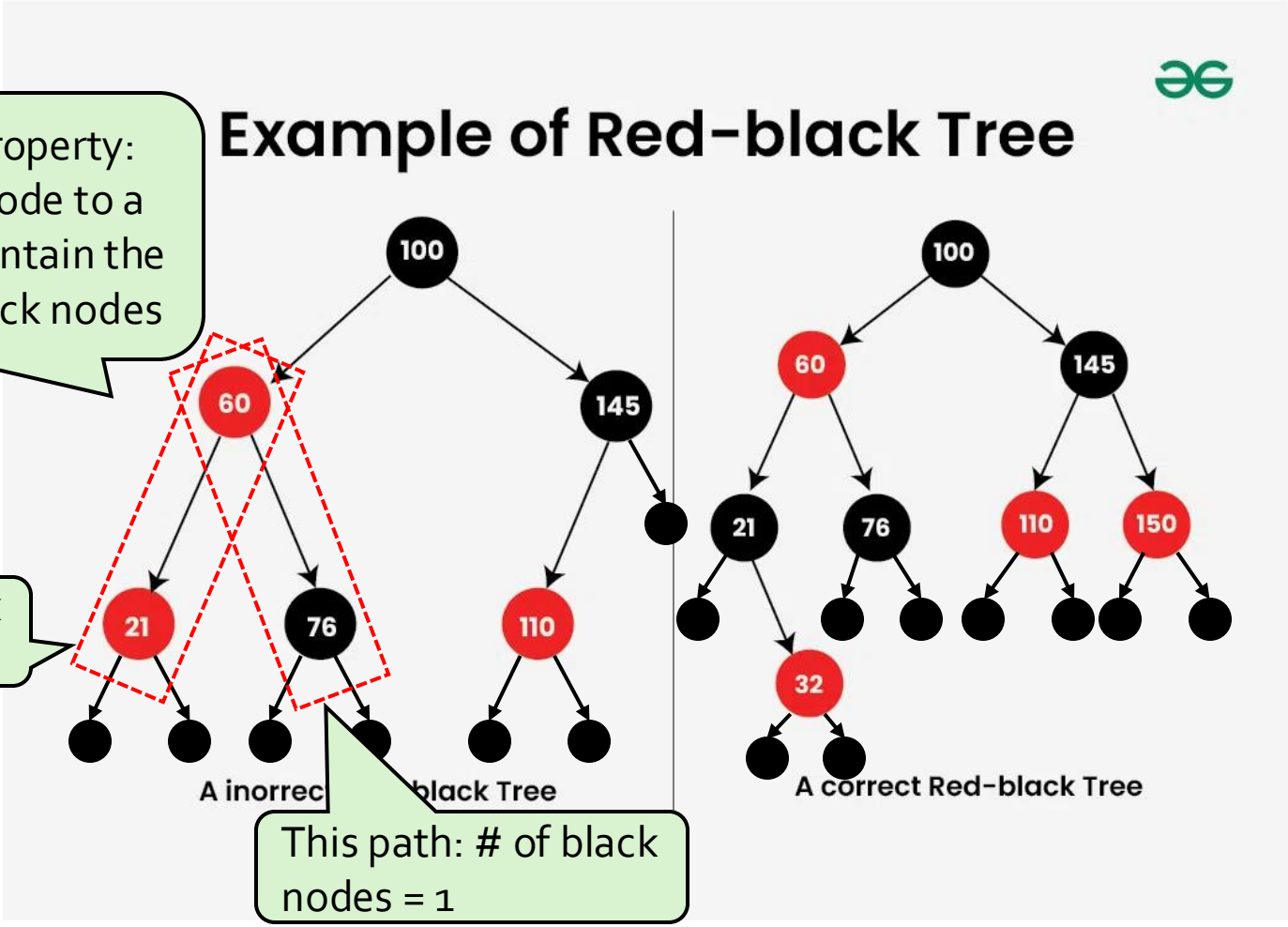


Image credit: https://www.geeksforgeeks.org/introduction-to-red-black-tree/

R-B trees: definition

Violation of black property: Every path from a node to a null pointer must contain the same number of black nodes

This path: # of black nodes = 0



This path: # of black nodes = 1

Image credit: https://www.geeksforgeeks.org/introduction-to-red-black-tree/

R-B tree implementation

R-B tree node

```
25      private:
26        struct RedBlackNode
27        {
28            Comparable    element;
29            RedBlackNode *left;
30            RedBlackNode *right;
31            int           color;
```

Figure 12.14 Class interface and constructor

AVL node

```
1   struct AvlNode
2   {
3       Comparable element;
4       AvlNode   *left;
5       AvlNode   *right;
6       int        height;
```

Figure 4.40 Node declaration for AVL trees

R-B tree implementation

R-B tree node

```
25      private:
26        struct RedBlackNode
27        {
28            Comparable    element;
29            RedBlackNode *left;
30            RedBlackNode *right;
31            int           color;
```

Figure 12.14 Class interface and constructor

- The balance is ensured by the patterns of colors allowed and the rotations/recolorings to fix the tree after inserts and deletes
- This is not a tree that enforces balance like an AVL tree, but it ends up behaving in a way that is just as good (almost as good)

R-B tree: example

In Textbook:
- Double circles: red
- Single circles: black
- Null ptrs omitted

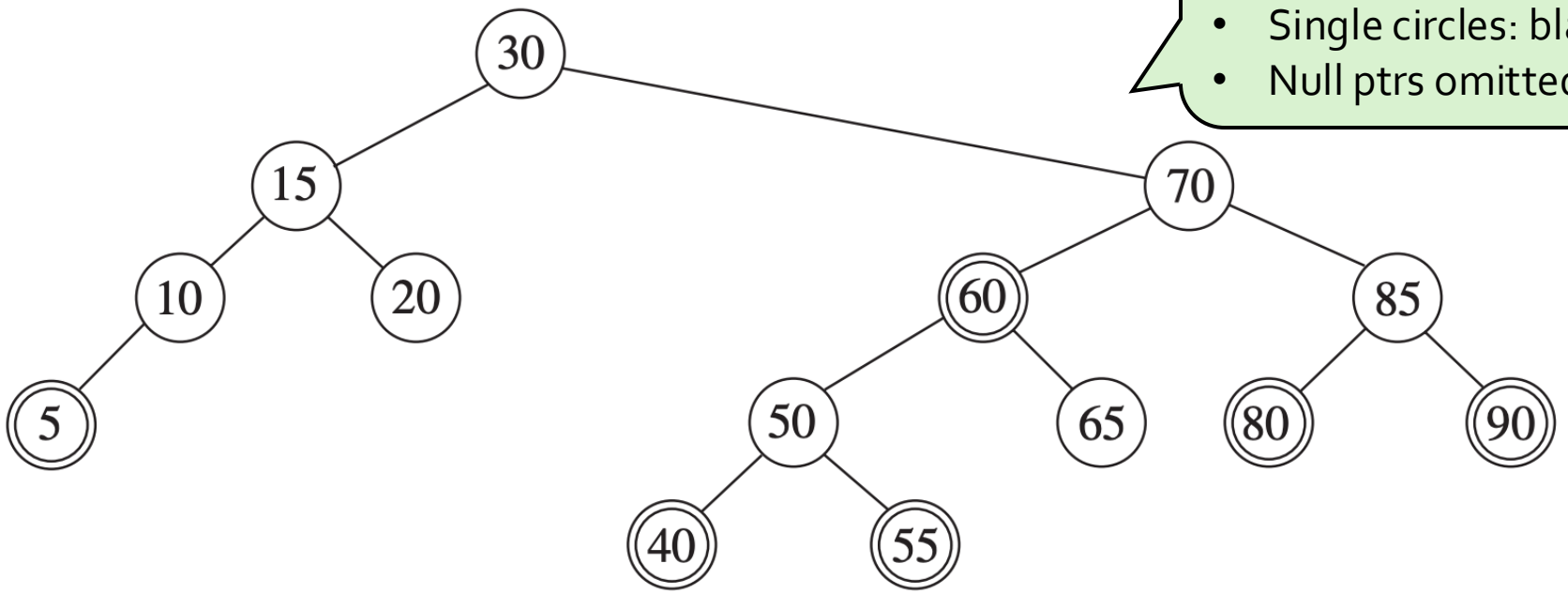


Figure 12.9 Example of a red-black tree (insertion sequence is: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55)

Top-down insert (Chapter 12.2.2)

Try visualization at https://ds2-iiith.vlabs.ac.in/exp/red-black-tree/red-black-tree-oprations/simulation/redblack.html

R-B tree: bottom-up insert

It will change the # of black nodes in path: violates black property

Why not coloring X black?

- Step 1: X is inserted at leaf (null ptr, as BST) and always colored red
- Step 2: adjust the structure and color case by case
  - Case 1: parent of X is black: done
  - Case 2: parent of X is red, then it violates the red property

Red property: no consecutive red nodes

  - Need further adjustment to resolve two consecutive red nodes
    - Case 2.1: aunt (sibling of parent) of X is **black**
      - Rotation
      - Recoloring for case 2.1
    - Case 2.2: aunt (sibling of parent) of X is **red**
      - Rotation
      - Recoloring for case 2.2

The same rotation method

Different recoloring method

R-B tree: bottom-up insert

Notations:
X: the inserted node
P: parent of X
G: grandparent of X
S: sibling of P
A, B, C: subtrees (for general rotations)

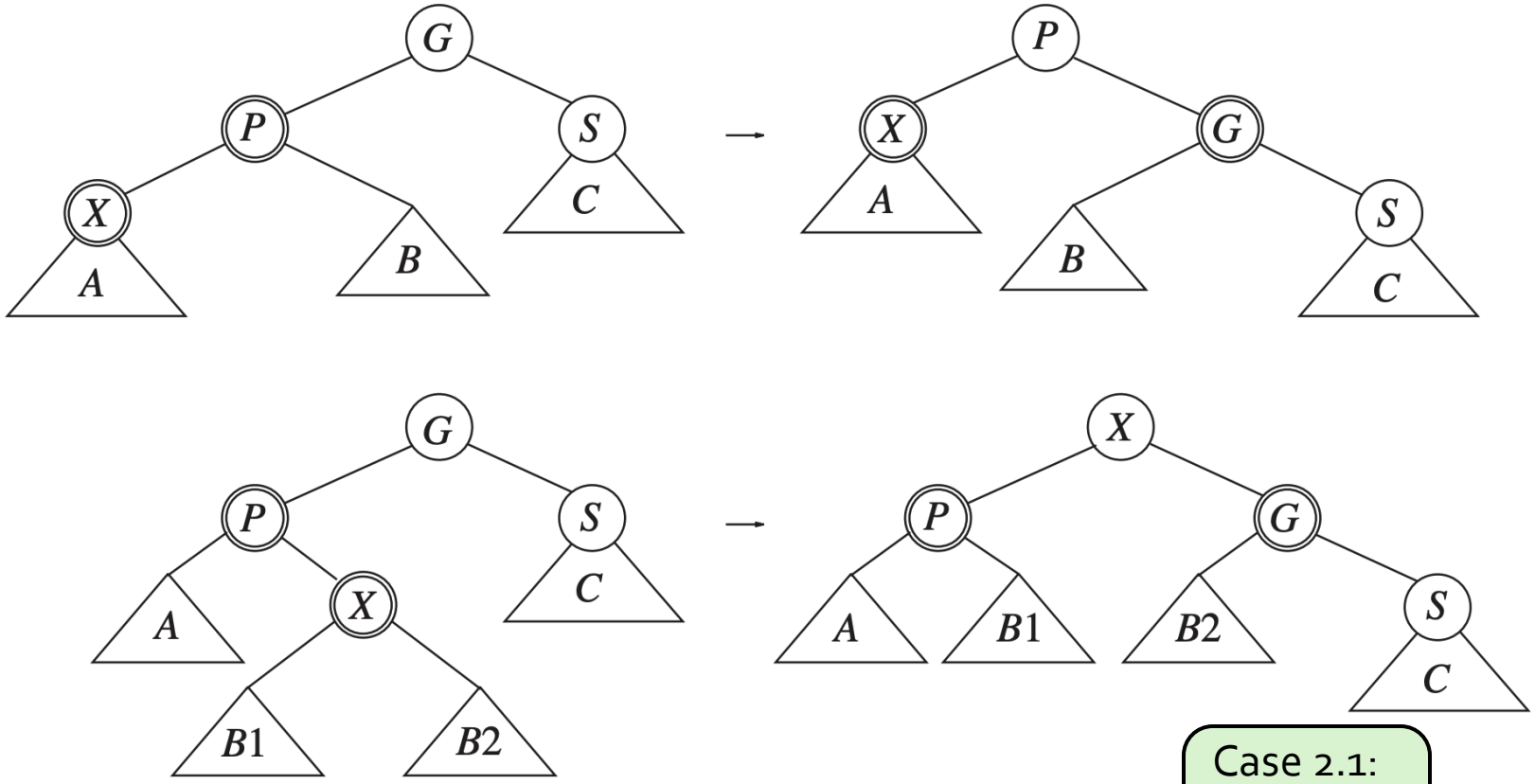


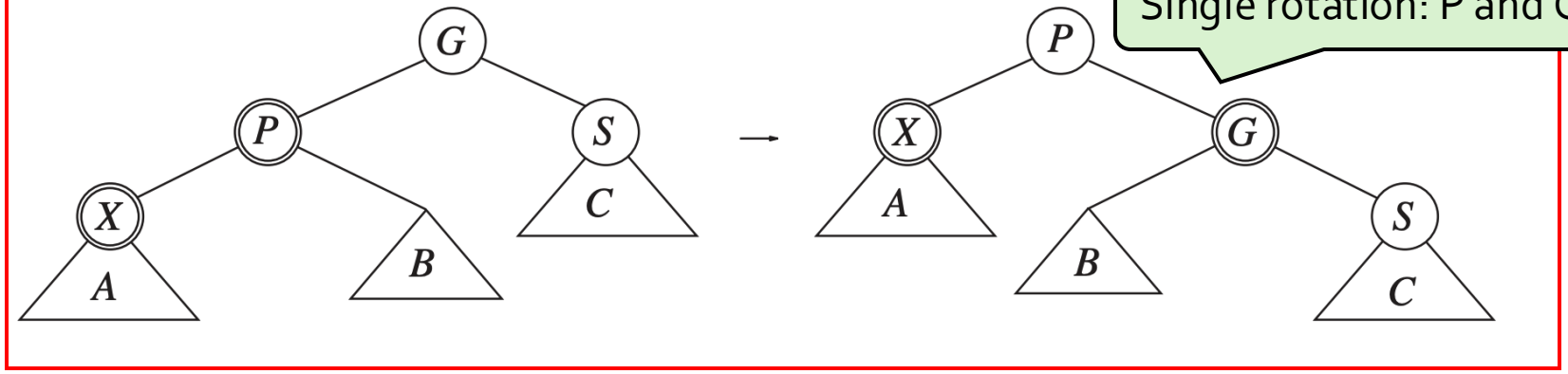
Figure 12.10 *Zig* rotation and *zig-zag* rotation work if *S* is black

Case 2.1:
P is red
S is black

R-B tree: bottom-up insert

Notations:
X: the inserted node
P: parent of X
G: grandparent of X
S: sibling of P
A, B, C: subtrees (for general rotations)

Single rotation: P and G



Double rotation:
First: P and X
Second: X and G

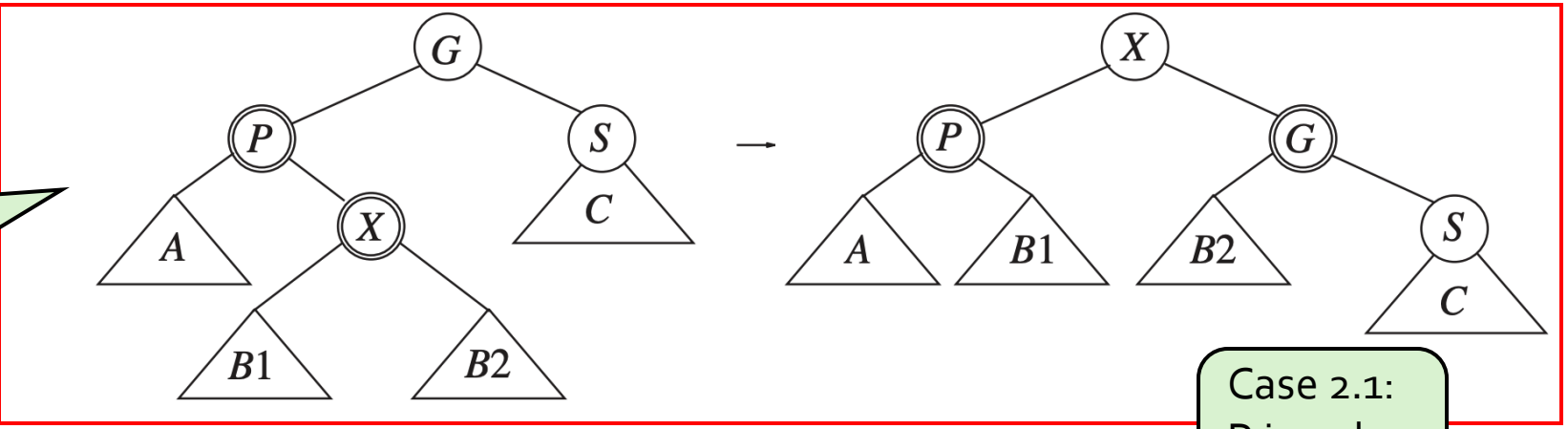


Figure 12.10 *Zig* rotation and *zig-zag* rotation work if *S* is black

Case 2.1:
P is red
S is black

R-B tree: bottom-up insert

Recoloring for case 2.1:
1. New root will be black
2. Nodes at the ends (left and right) need their colors unchanged

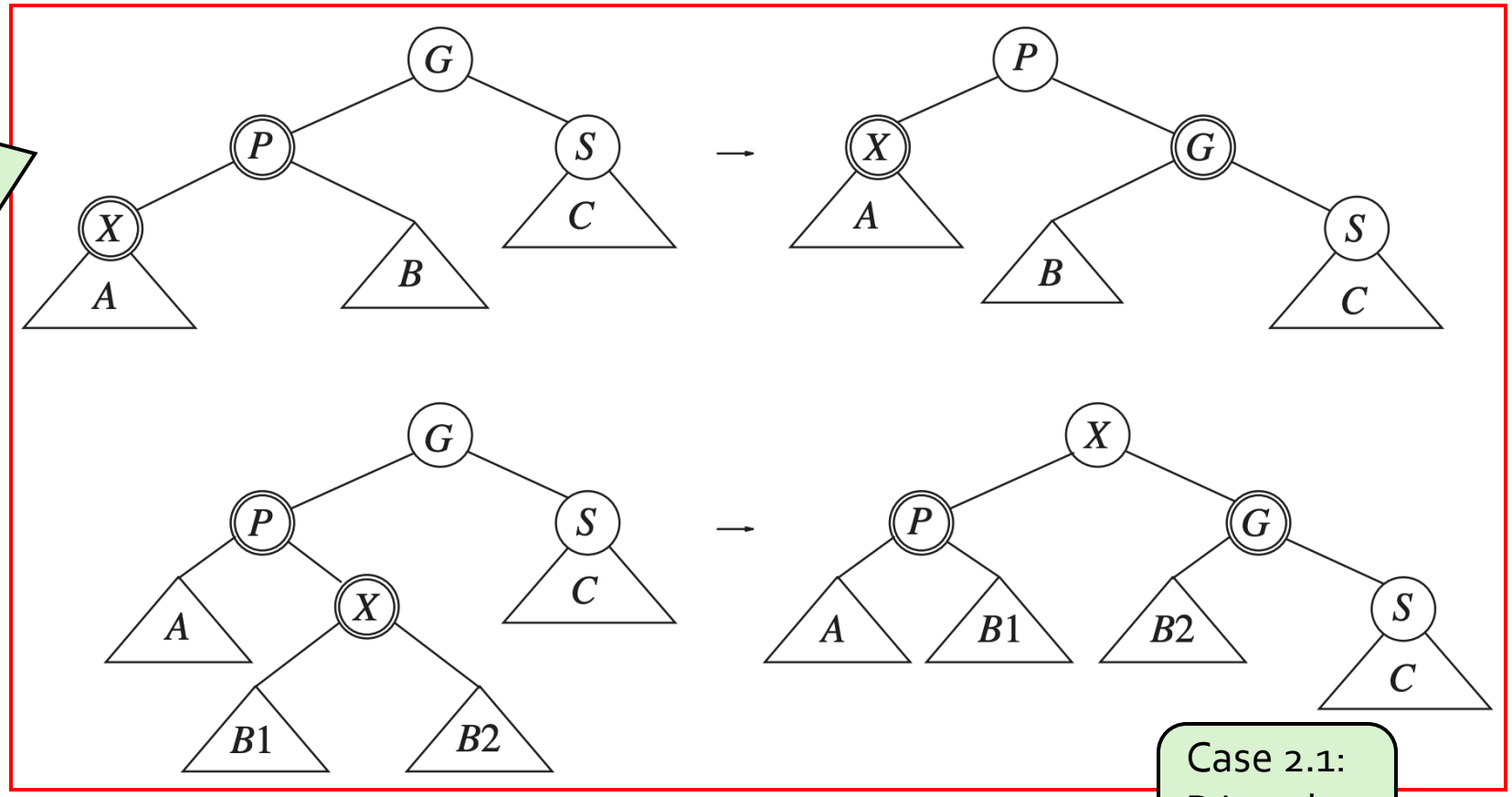
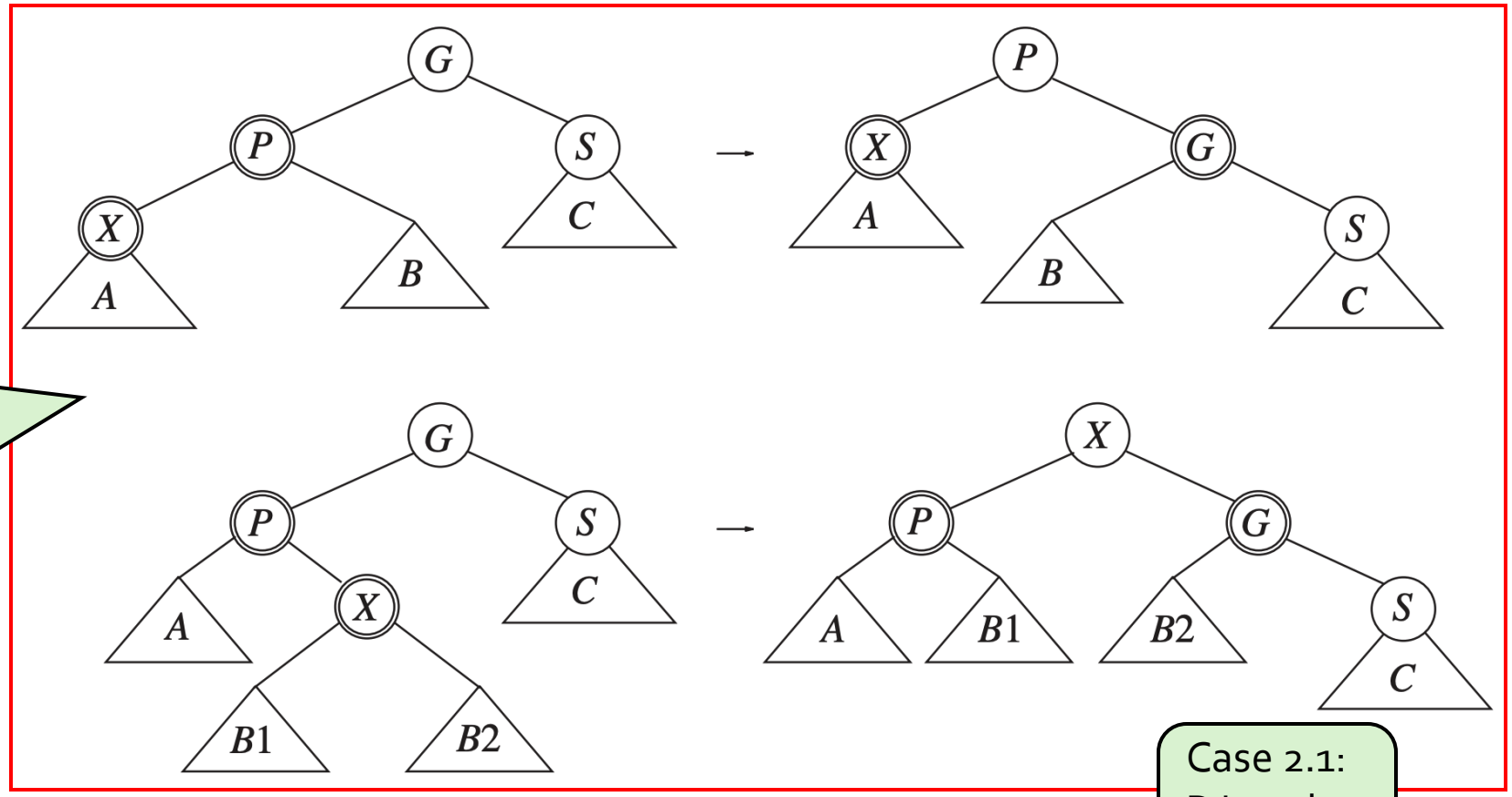


Figure 12.10 *Zig* rotation and *zig-zag* rotation work if *S* is black

Case 2.1:
P is red
S is black

R-B tree: bottom-up insert



Red property ✅
- After adjustment: No consecutive red nodes

Figure 12.10 *Zig* rotation and *zig-zag* rotation work if *S* is black

Case 2.1:
P is red
S is black

R-B tree: bottom-up insert

Black property ✅:
- Black height (BH) of each subtrees:
  BH(A) = BH(B) = BH(C)+1
- # of black nodes from each node to null ptrs remains unchanged

Black property ✅:
- Black height (BH) of each subtrees:
  BH(A) = BH(B1) = BH(B2) = BH(C)+1
- # of black nodes from each node to null ptrs remains unchanged

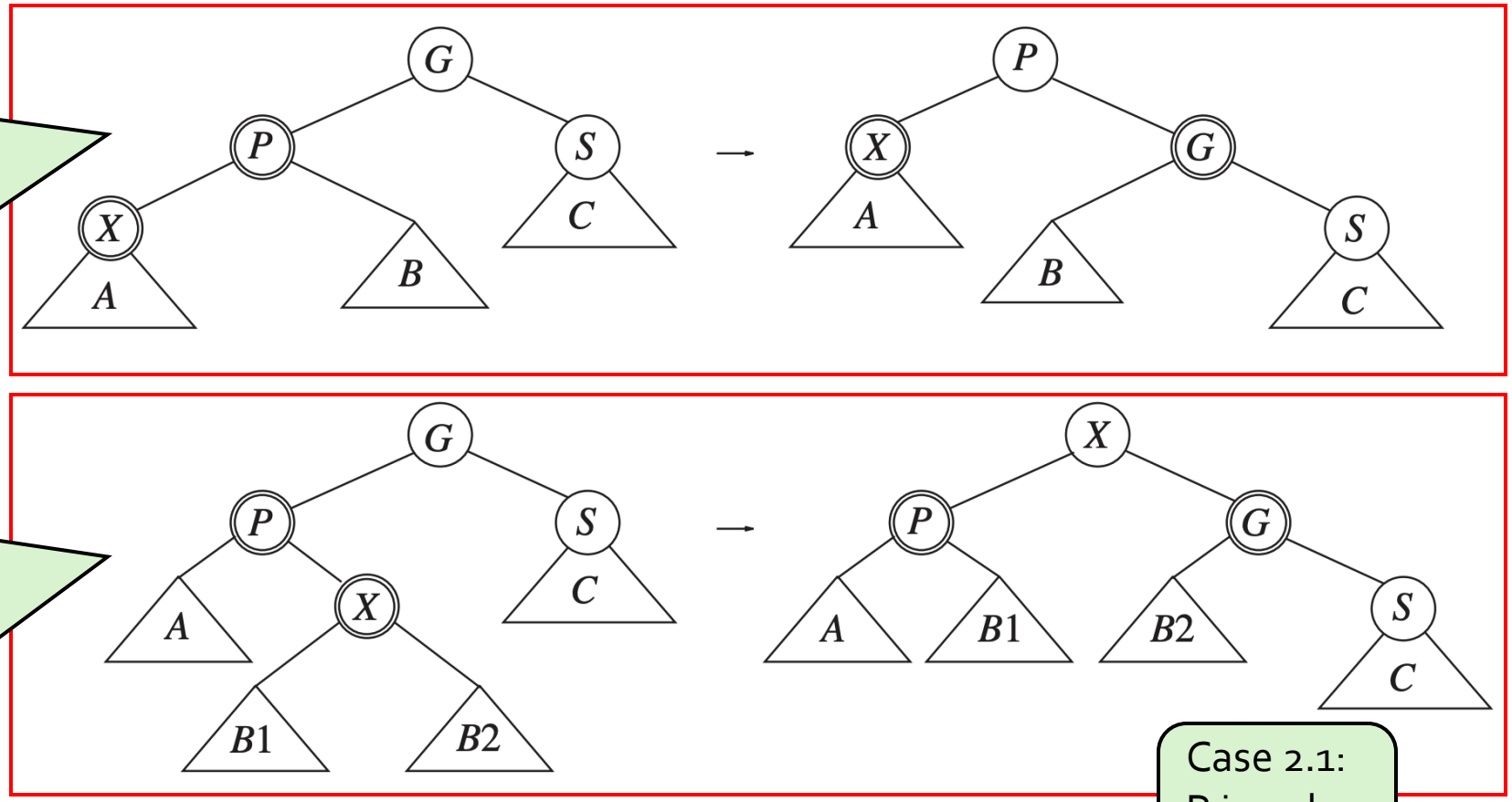
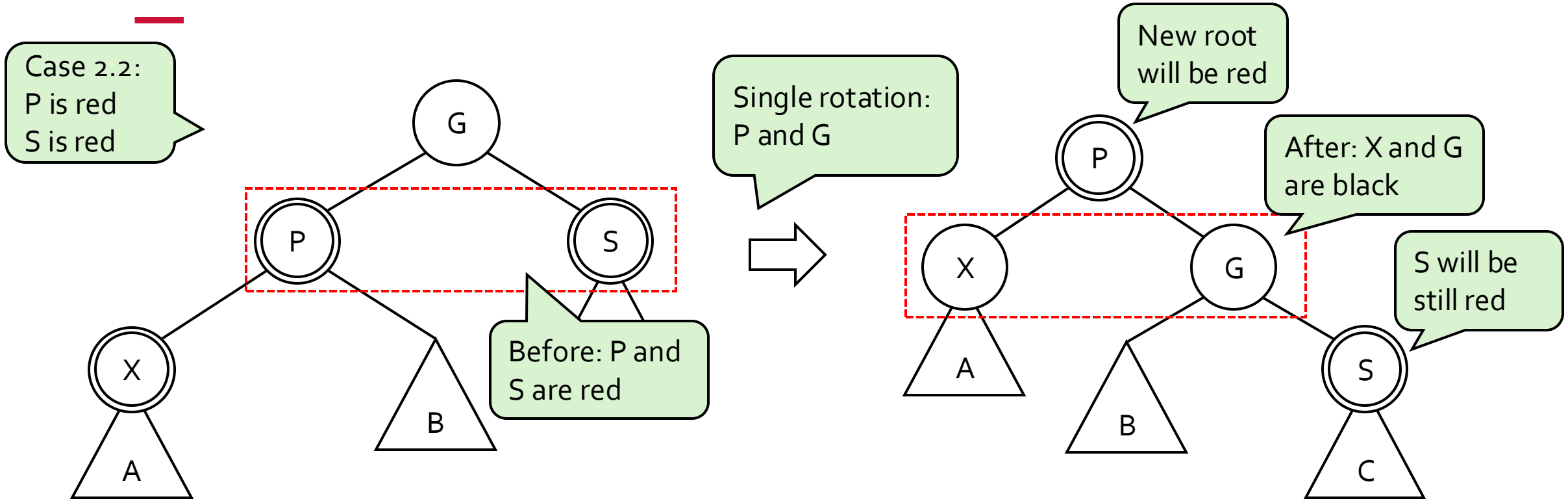


Figure 12.10 *Zig* rotation and *zig-zag* rotation work if *S* is black

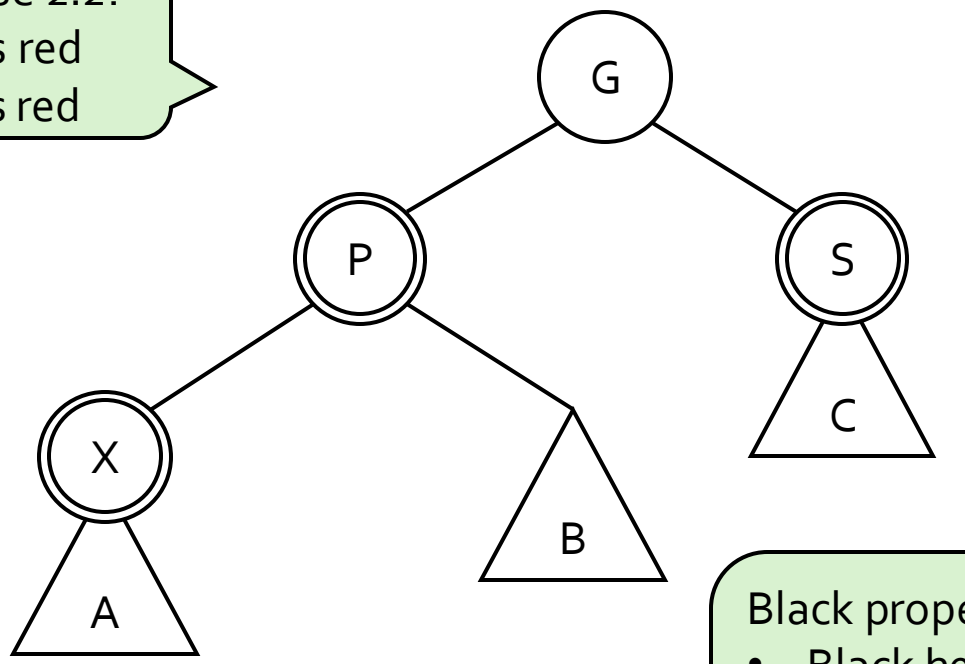
Case 2.1:
P is red
S is black

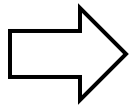
R-B tree: bottom-up insert

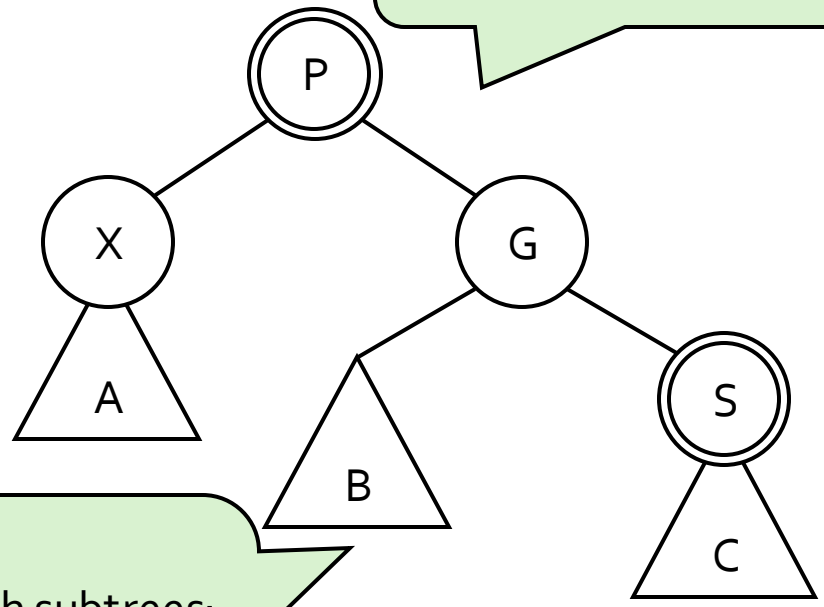


R-B tree: bottom-up insert

Case 2.2:
P is red
S is red



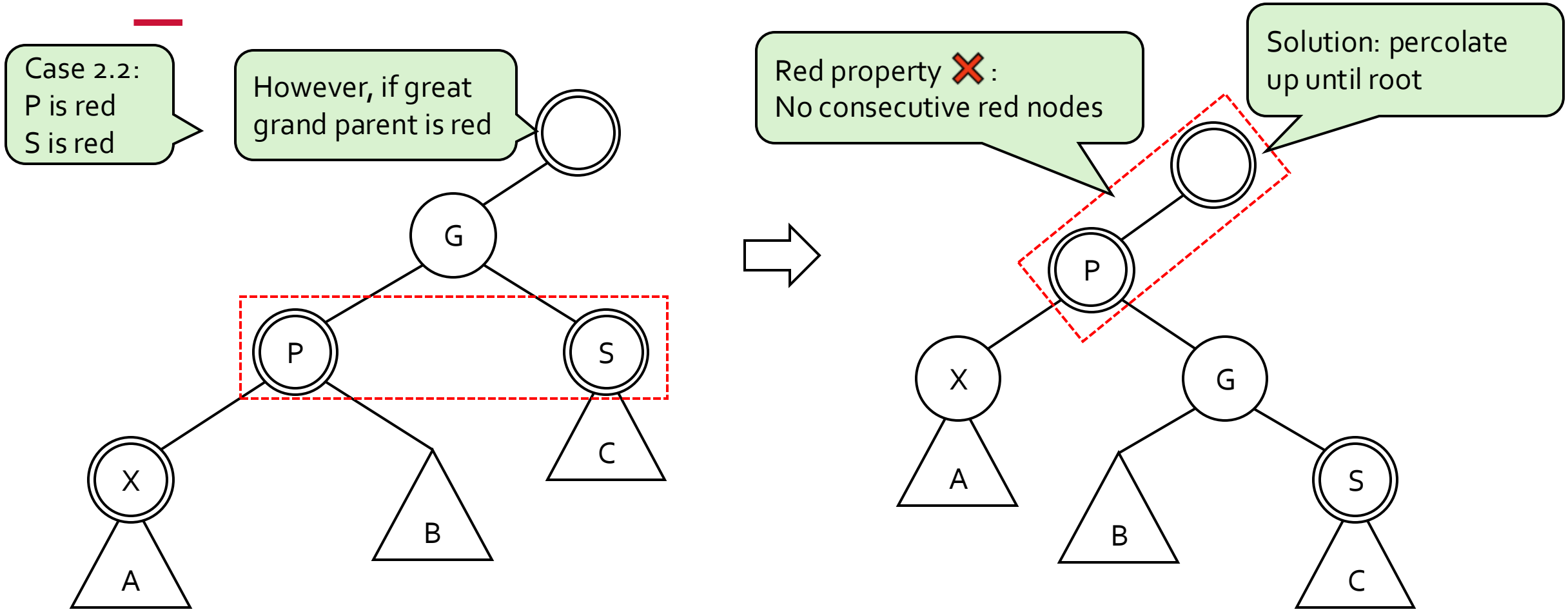




Red property ✅:
No consecutive red nodes

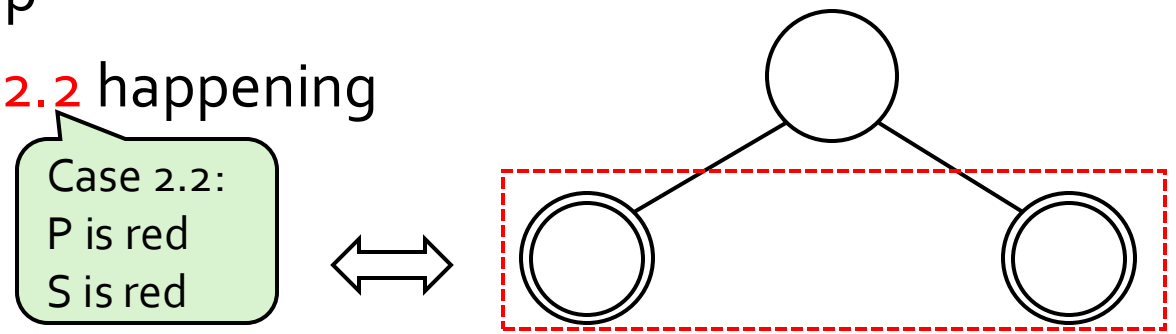
Black property ✅:
- Black height (BH) of each subtrees:
  BH(A) = BH(B) = BH(C)
- # of black nodes from each node to null ptrs remains unchanged

R-B tree: bottom-up insert



R-B tree: top-down procedure

- An alternative to the bottom-up R-B trees
- More practically used
- e.g., Figure 12.9 in Textbook is constructed by a top-down R-B tree (rather than a bottom-up one)
- Technique: color flip
- Target: avoid Case 2.2 happening



Top-down R-B trees: color flip

- On the way down from the root to search null ptr for insertion:
- If we see a node X with two red children, we make X red and the children black
- If X is root, recolor X black
- If X's parent is also red:
  - Fix it with rotation (zig-zag or zig-zig)
- Avoid Case 2.2: no need for percolation up

Color flip is done before finding the position to insert the new node

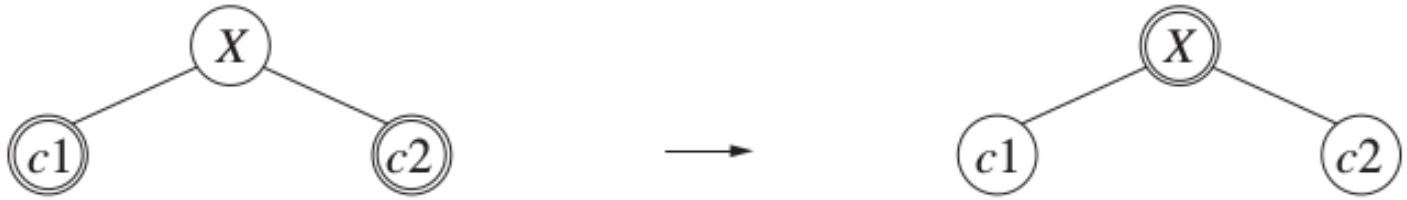
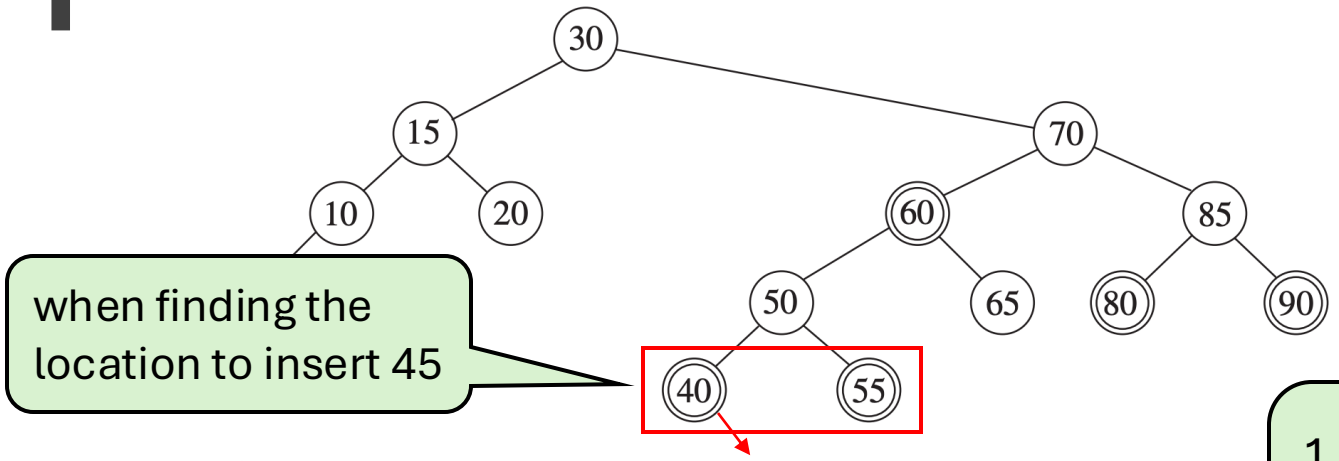


Figure 12.11 Color flip: only if *X*'s parent is red do we continue with a rotation

Top-down R-B trees: example



when finding the location to insert 45

Figure 12.9 Example of a red-black tree (insertion sequence is: 10, 8
30, 50, 65, 80, 90, 40, 5, 55)

1. Color flip at 50 (to red)
2. 50 and 60 are both red
3. Single rotation: 70-60-50

Top-down R-B trees: example

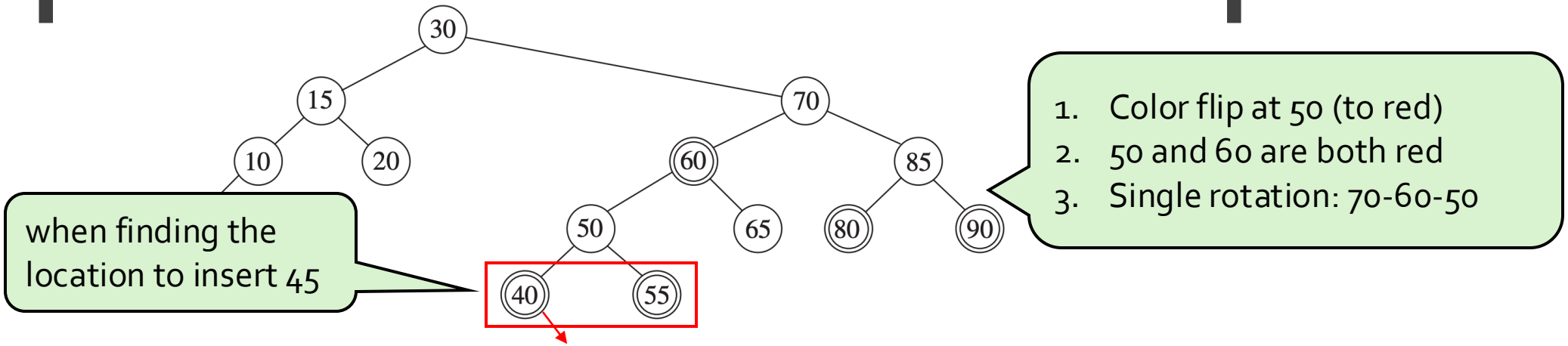


Figure 12.9 Example of a red-black tree (insertion sequence is: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55)

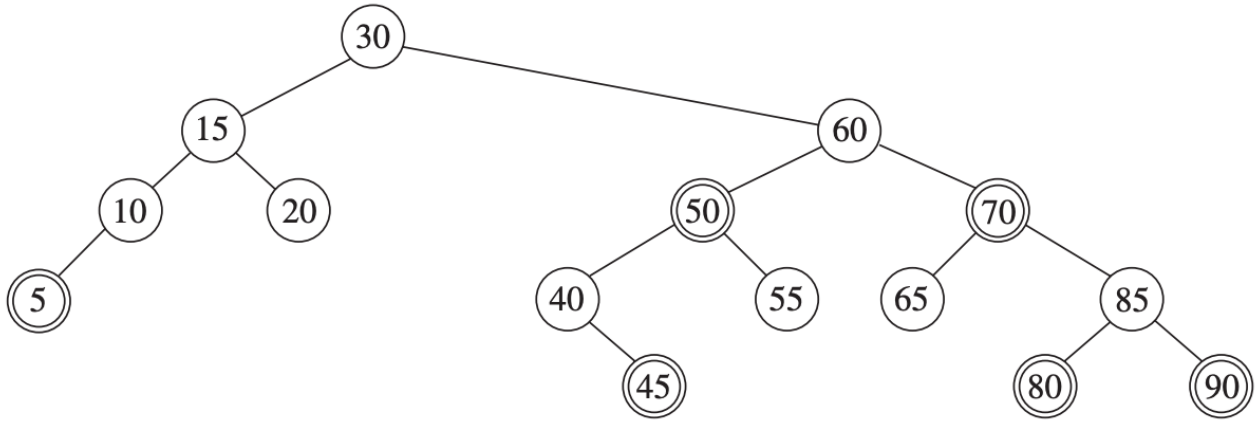


Figure 12.12 Insertion of 45 into Figure 12.9

R-B trees: delete

- Similar to a BST delete, but we need to ensure the R-B properties are maintained if we delete a black node (thereby violating the black property)
- If Node has two children: replace with smallest in right subtree
- If Node only has a right child: ditto (the same)
- If Node with only a left child: replace with largest node in left subtree

BST delete

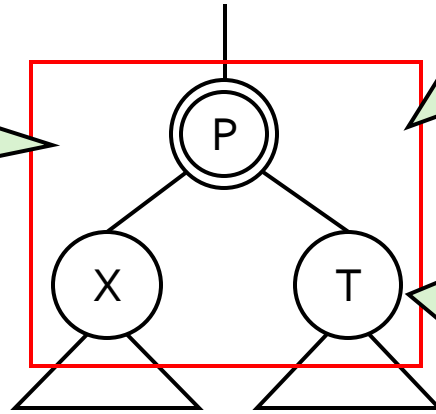
R-B trees: delete

Deleting a red node: lack property ✅

- Case study for the node to be deleted
- If it is red, then make the new node black and quit
- If it is black… that would violate the black property since the tree would lose a black node in the root->nullNode black-height count
- Solution: when **top-down** pass, ensure the leaf node red

Top-down R-B trees: delete

If P is red, then X and T must be **black**



Before top-down pass: X is black

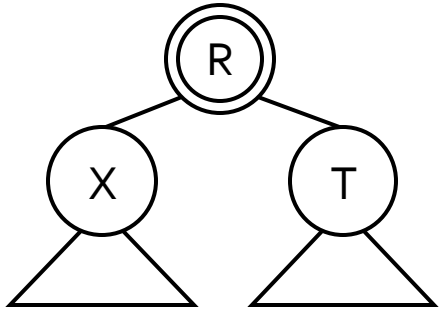
Goal: ensure X is red

- Assume **initial status**:
  - **X** == current node
  - **T** == sibling of X
  - **P** == parent of both

X, T: black

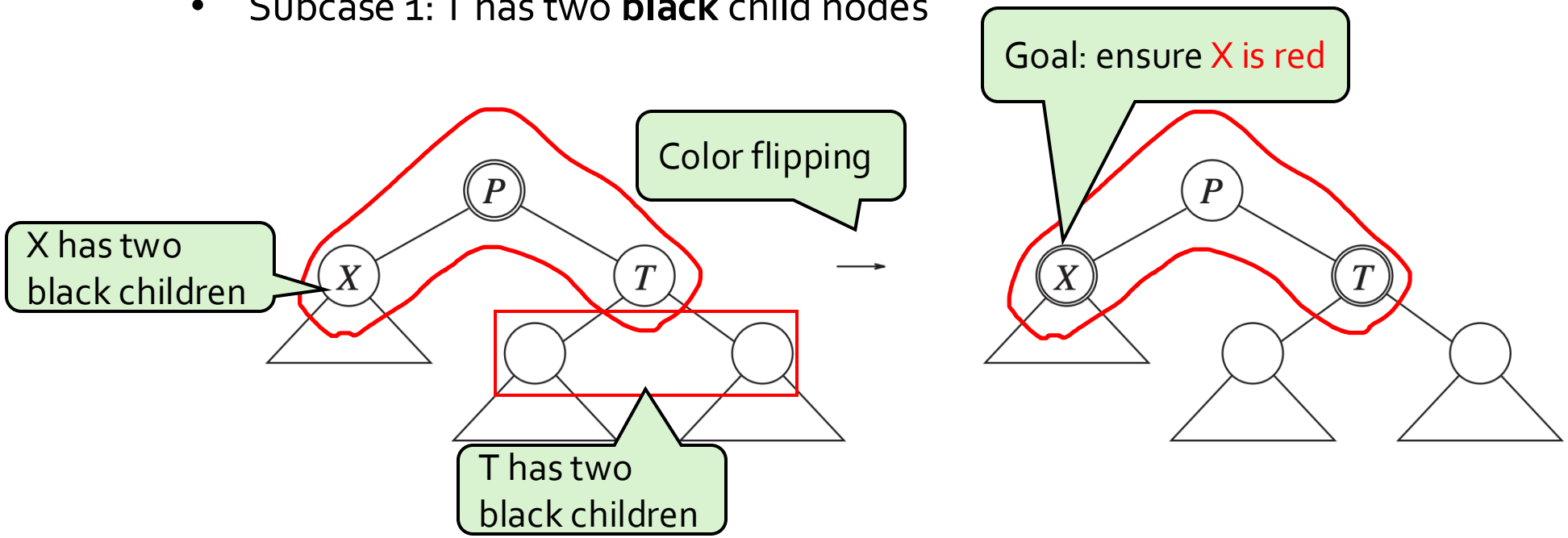
P: red

- As we traverse down the tree, we attempt to ensure X is red
- Root sentinel: temporarily regard root as red



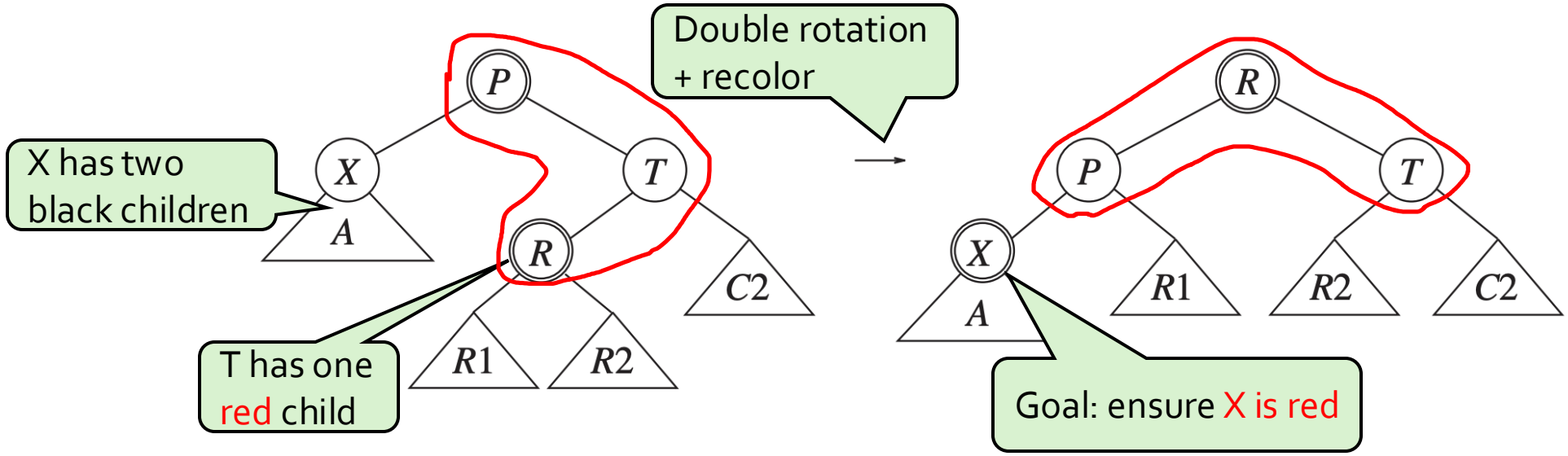
Top-down R-B trees: delete

- Case 1: X has two **black** child nodes
- Three subcases:
  - Subcase 1: T has two **black** child nodes



Top-down R-B trees: delete

- Case 1: X has two **black** child nodes
- Three subcases:
  - Subcase 2: T has one red child node (left)



Top-down R-B trees: delete

- Case 1: X has two **black** child nodes
- Three subcases:
  - Subcase 2: T has one red child node (right)

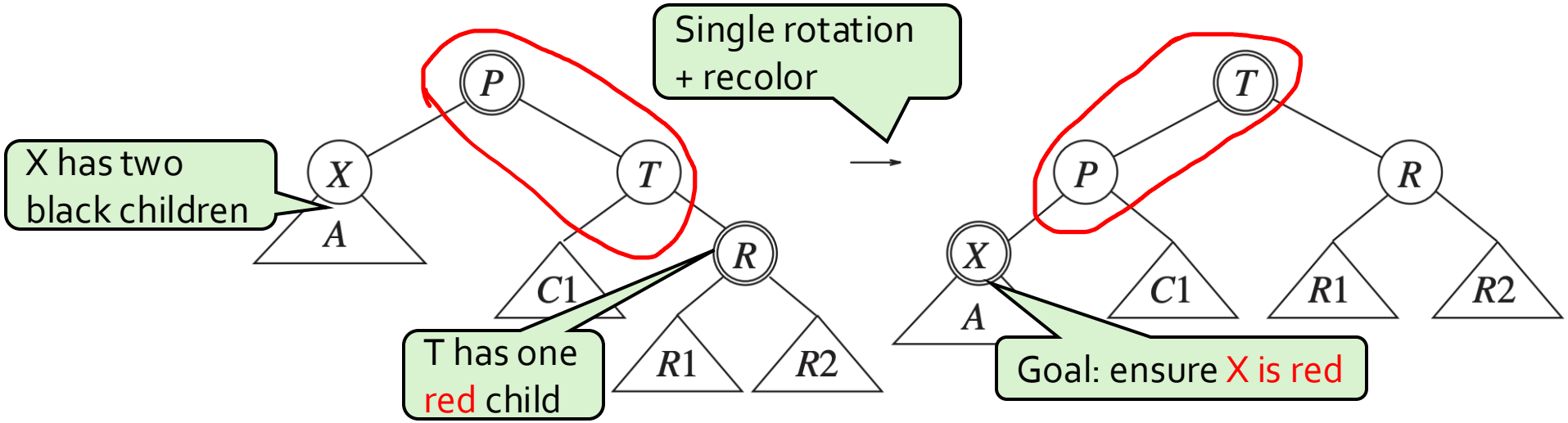
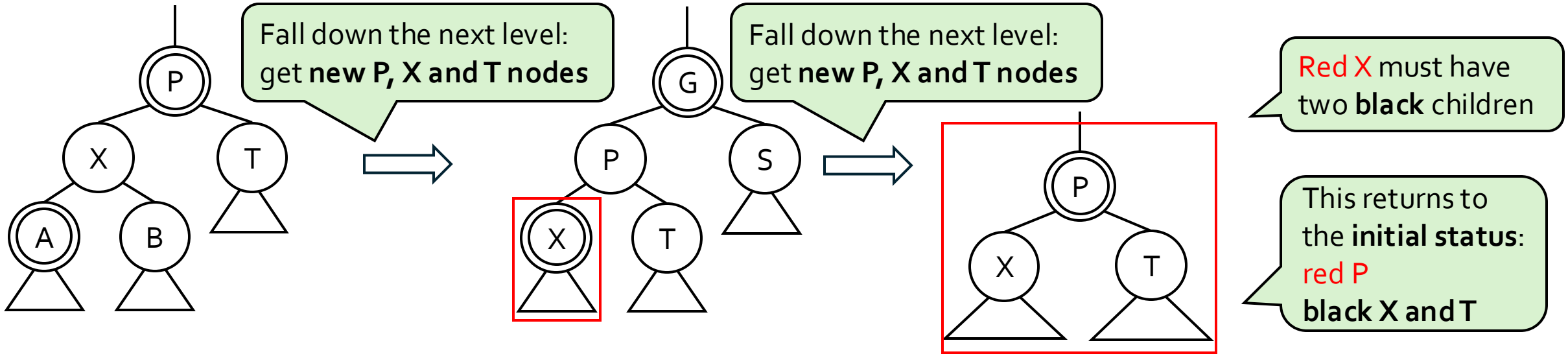


Figure 12.17 Three cases when *X* is a left child and has two black children

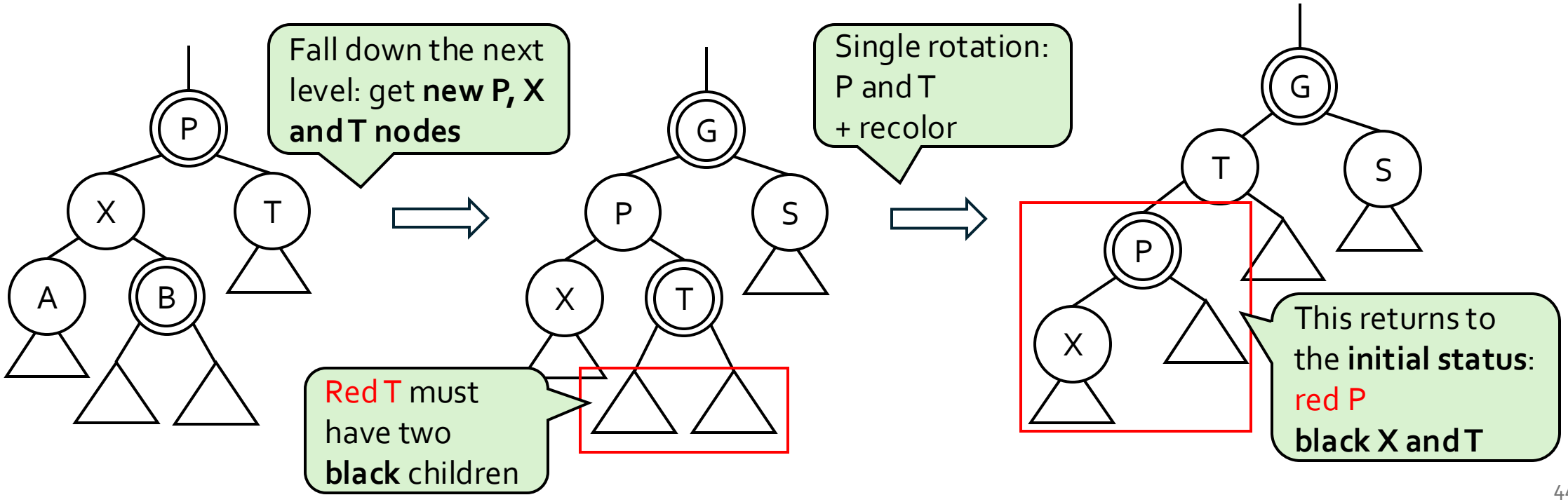
Top-down R-B trees: delete

- Case 2: X has at least a red child node
- Action: fall to the next level and get new P, X, T nodes
  - Subcase 1: top-down pass continues a red node as X



Top-down R-B trees: delete

- Case 2: X has at least a red child node
- Action: fall to the next level and get new P, X, T nodes
  - Subcase 1: top-down pass continues a **black** node as X



R-B tree: summary

- STL set and map classes use balanced trees to support logarithmic insert, delete and search
- Implementation uses top-down red-black trees