Red-black tree

Ch. 10.3

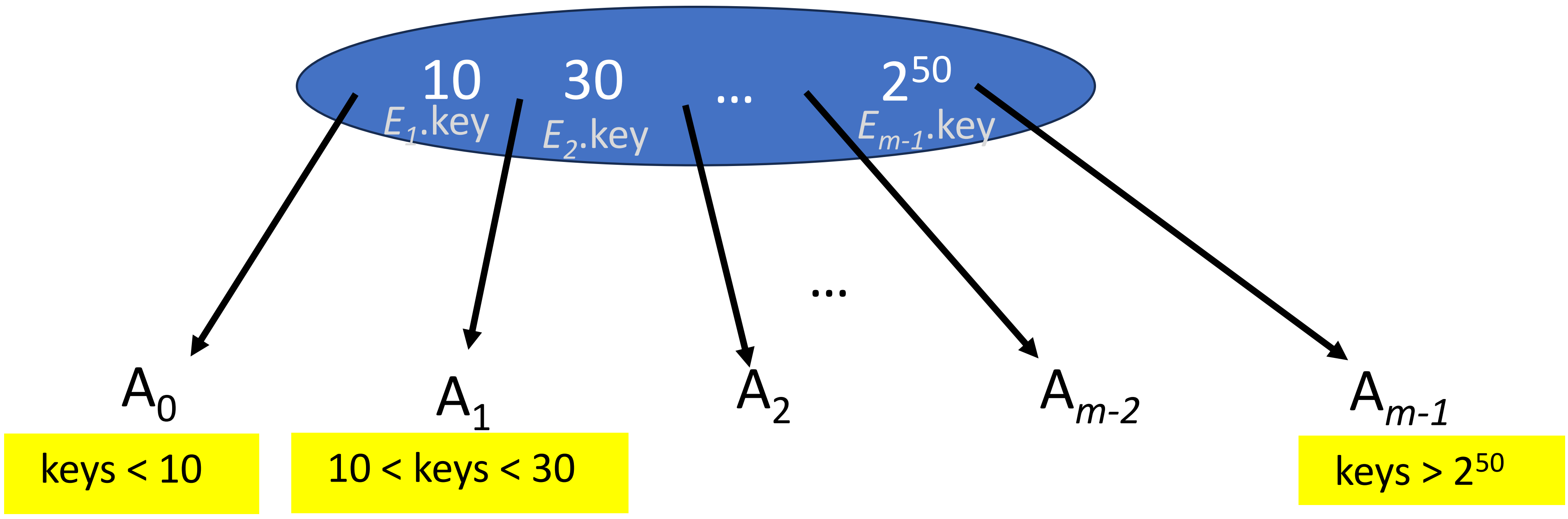
Recall: m-way search tree

- **Ch 10.2 AVL tree**
- **Ch 11.2 B-tree**
  - 2-3 trees (B-tree of order 3)
  - 2-3-4 tree (B-tree of order 4)
- $\boxed{\textbf{Ch 10.3 Red-black tree} \text{ (An extension of 2-3-4 trees)}}$

We are here

- **Ch 11.3 B$^+$-tree**

Recall: m-way search tree

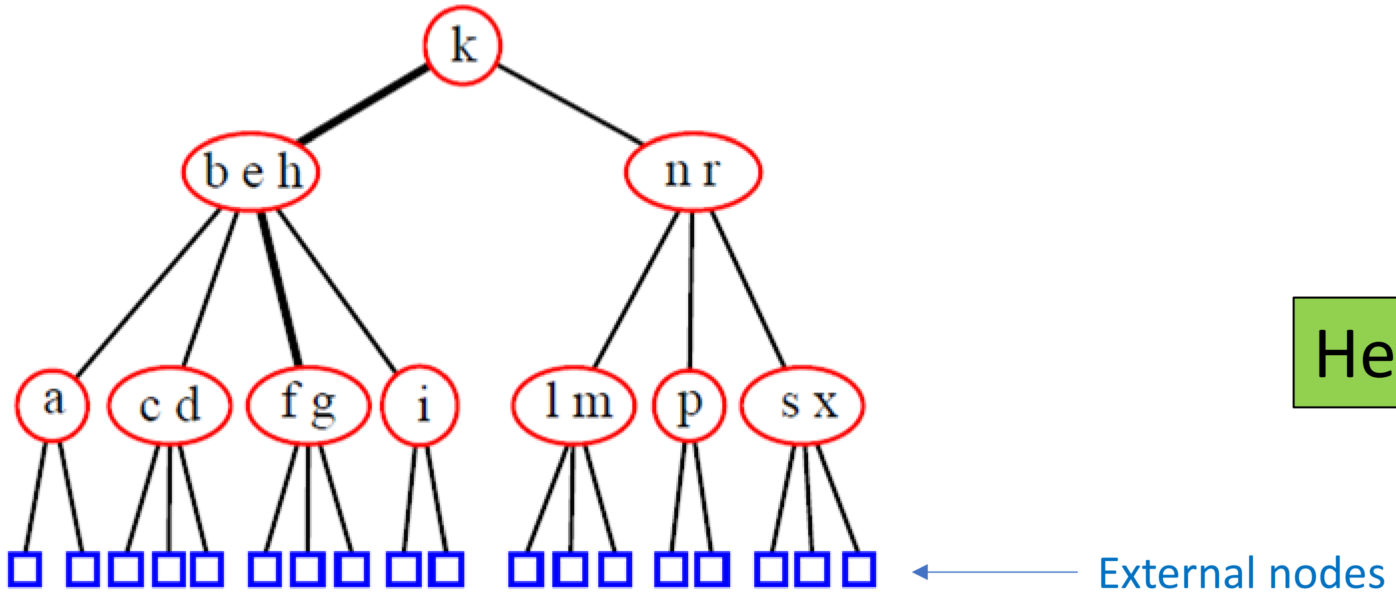


- If *m* is large, the node size becomes large and thus the cost of searching within a node increases.
- Can we just store one data pair in a node? ➔ Binary tree

Red-black tree is the binary tree simulating 4-way search tree (2-3-4 tree).

Recall: 2-3-4 tree

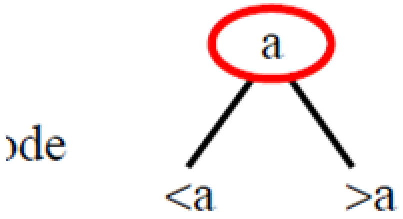
- The nodes store 1, 2 or 3 data pairs and have 2, 3 or 4 children, respectively.
- All external nodes are at the same level.



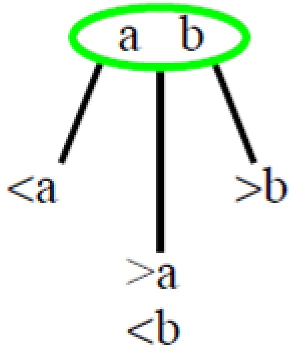
Height = O(log n)

Recall: 2-3-4 tree nodes

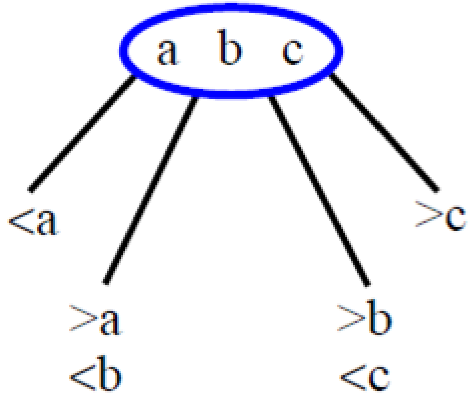
- 2-node
  - Same as a binary node



- 3-node
  - 2 data pairs, 3 children



- 4-node
  - 3 data pairs, 4 children



Recall: Bottom-up insertion

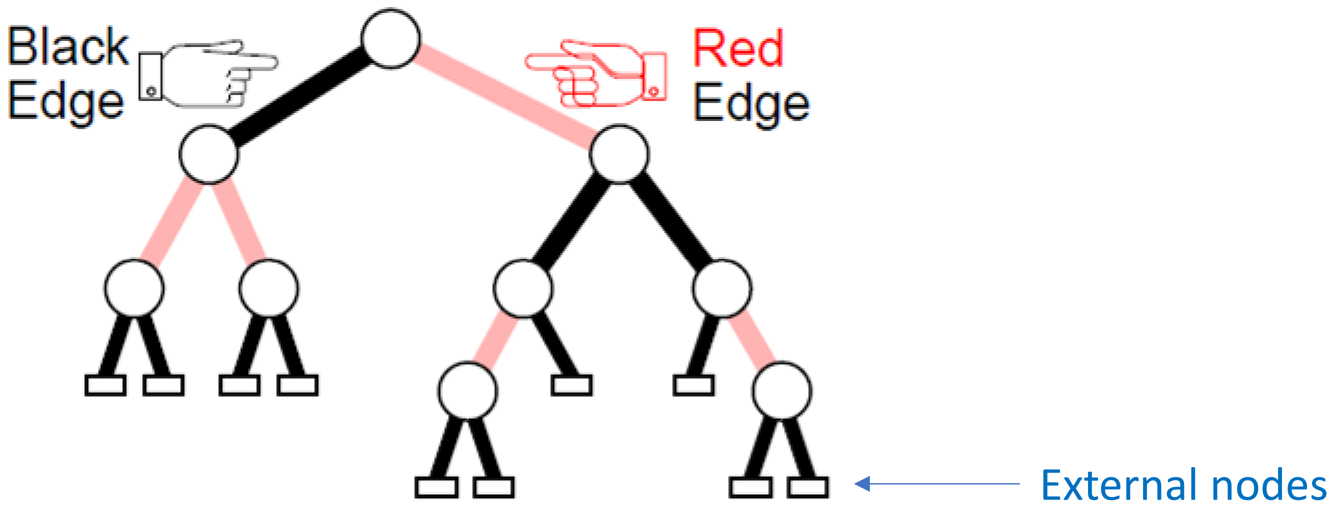
In 2-3-4 tree, if a data pair is inserted into a 4-node:

- The node will be split and the mid data pair will be inserted into to the parent node.
- If the parent is also a 4-node, the splitting process will be repeated. (bottom-up splitting)

Red-Black tree

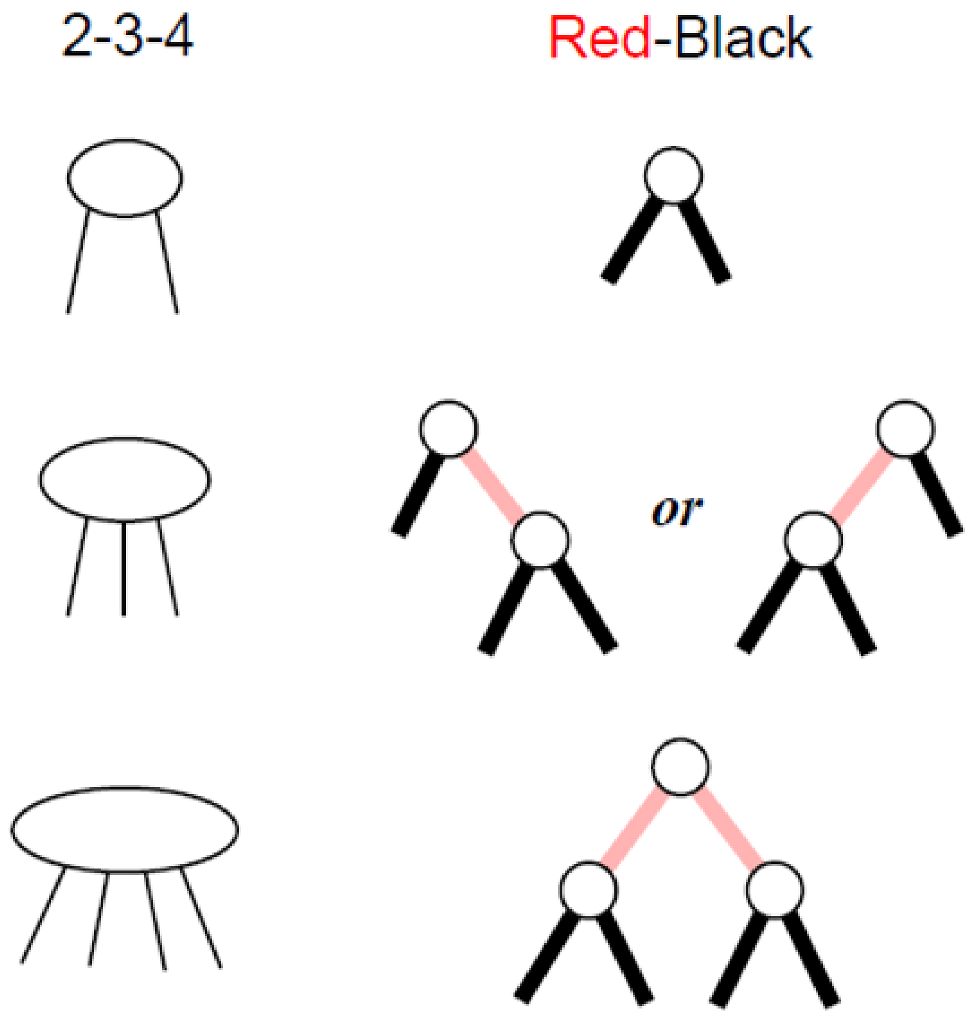
A binary search tree with the following properties:

- Edges are colored **red** or **black**.
- ***No two consecutive red edges*** on any root-external node path
- ***Same number of black edges*** on any root-external node path (=***black height*** of the tree)
- ***Edges connecting to external nodes are black***.



Relate 2-3-4 tree to red-black tree

- 2 node
  - One data pair
- 3-node
  - Two data pairs
- 4-node
  - Three data pairs



Nodes connected by red edges represent the data pairs in a single node of 2-3-4 tree.

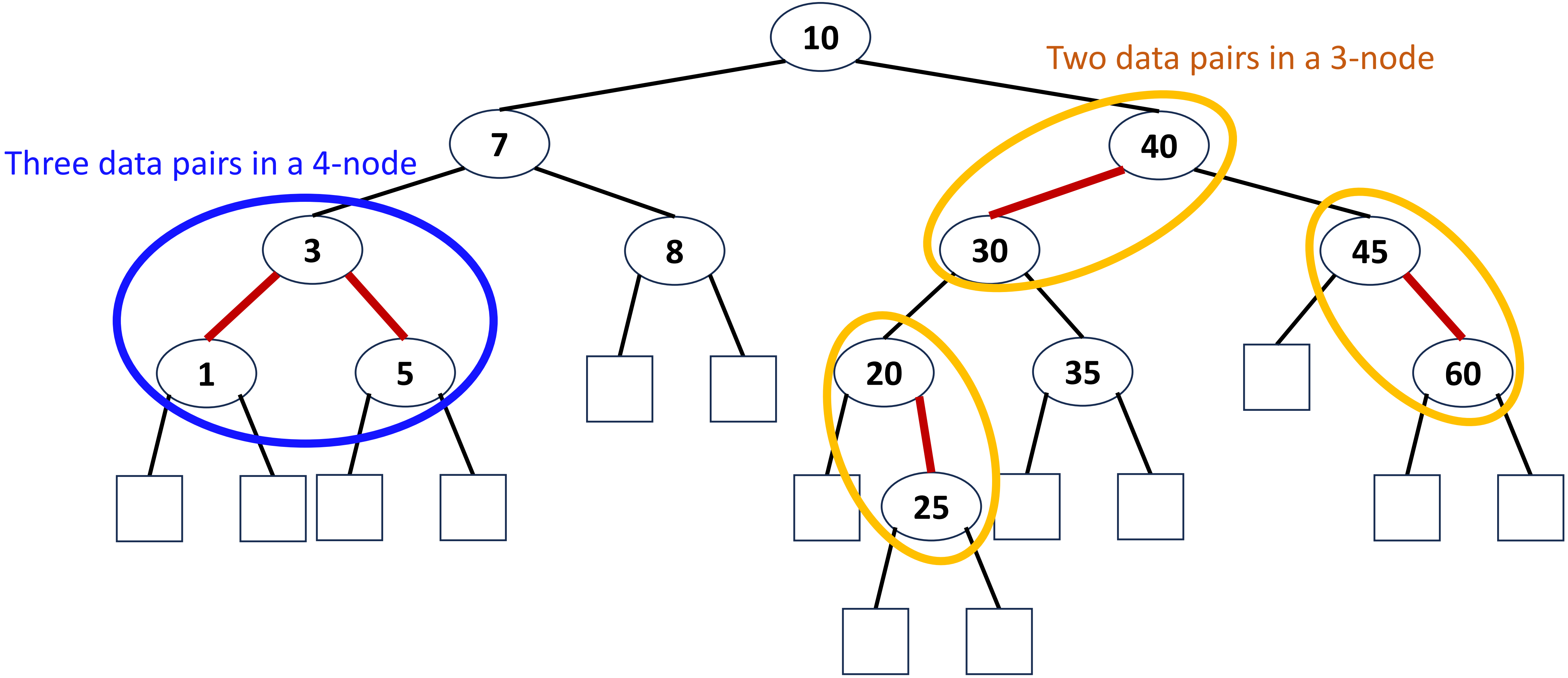
Red-black trees are just a way to represent 2-3-4 trees.

Relate 2-3-4 tree to red-black tree

- Red-black search tree uses binary trees representing m-way search trees.

- Motivation: We need a way to quickly search the data pairs within a node of m-way search trees.

Example of red-black tree



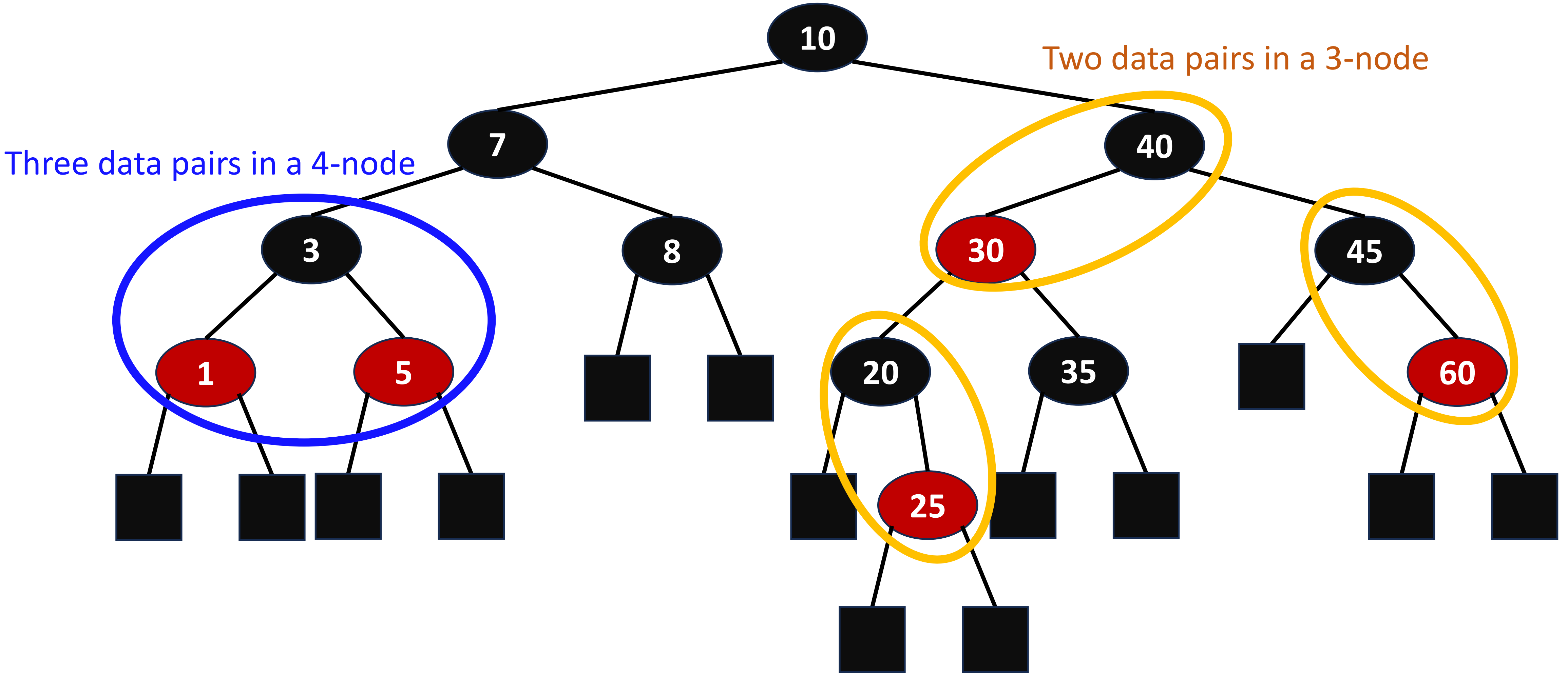
Operations for Red-black trees can be applied to 2-3-4 trees.

Red-Black tree (Equivalent definition)

A binary search tree with the following properties:

- Nodes are colored **red** or **black**.
- ***No two consecutive red nodes*** on any root-external node path
- ***Same number of black nodes*** on any root-external node path
- ***Root and external nodes are black***.

Example of red-black tree (Node)

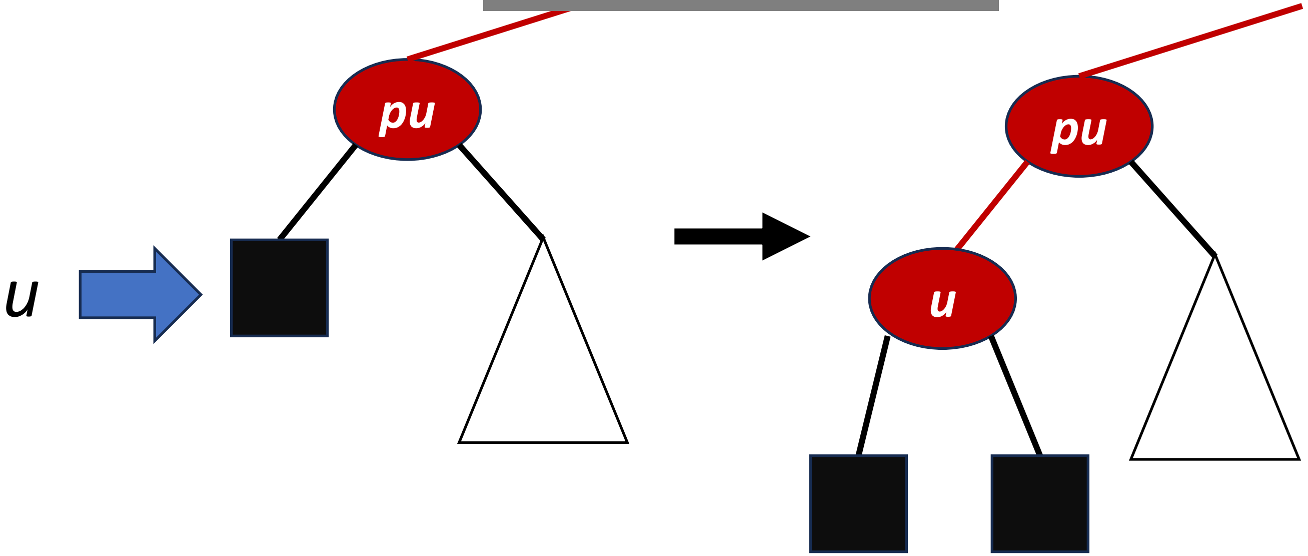


If we know the edge colors, we can deduce the node colors and vice versa.

Operation: Insertion

1. Perform a standard search to find the external node where the key should be added.
2. Replace the external node with an internal node with the new key.
3. Color the new node **red.**
4. Add two new external nodes and color them **black.**
5. If the parent is **red**, we have two consecutive **red** nodes. The tree has to be reorganized.

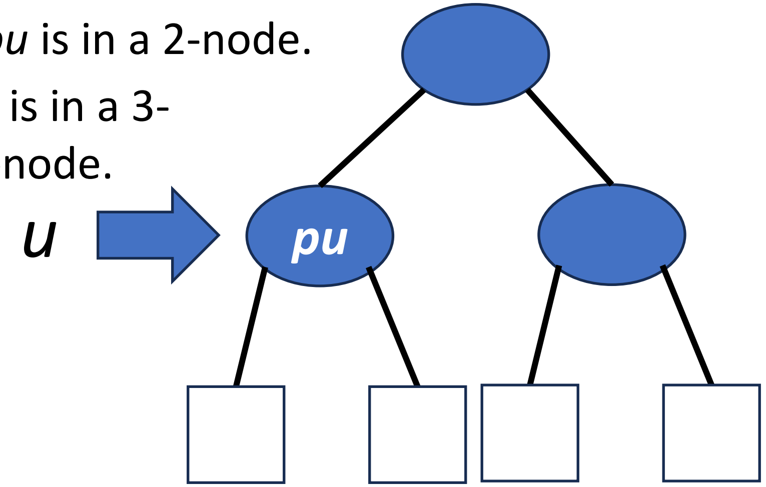
In a red-black tree



In a 2-3-4 tree

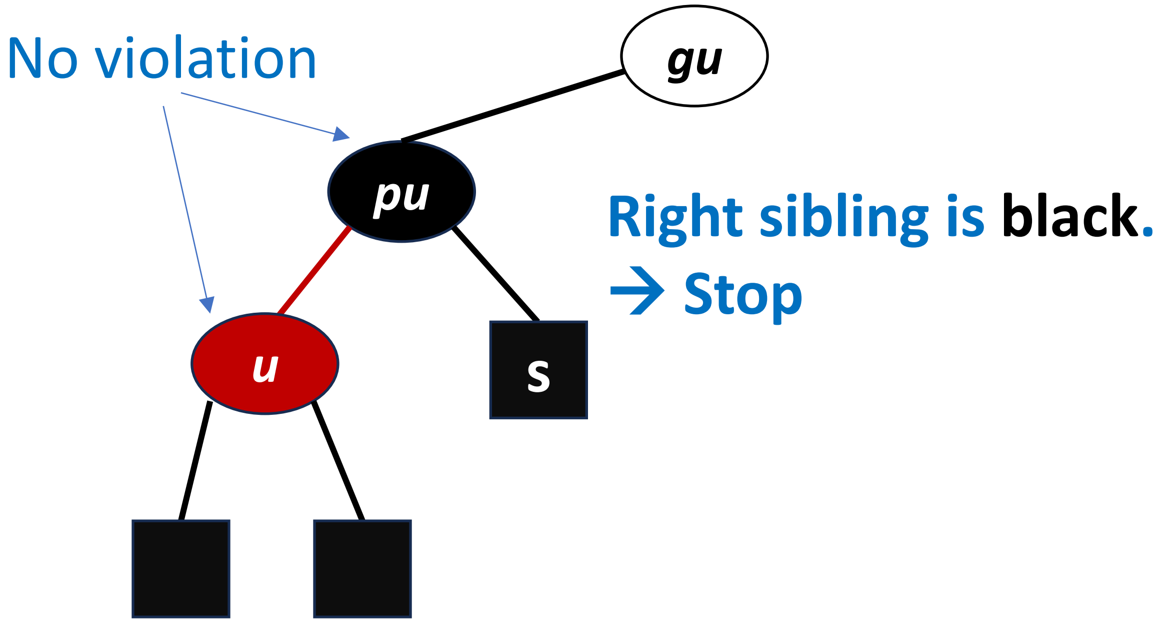
black *pu: pu* is in a 2-node.

red *pu: pu* is in a 3-node or 4-node.

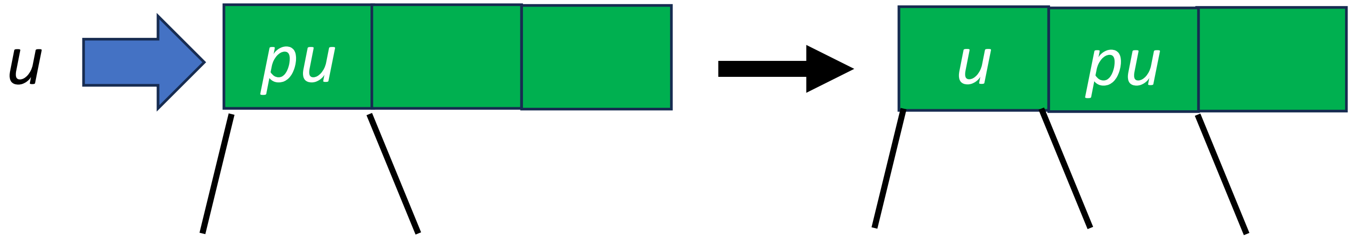


Insertion: parent is black

- *u*: the new node
- *pu*: parent of *u* node
- *gu*: grandparent of *u* node
- *s*: sibling of *u* node
  (may be an external node)

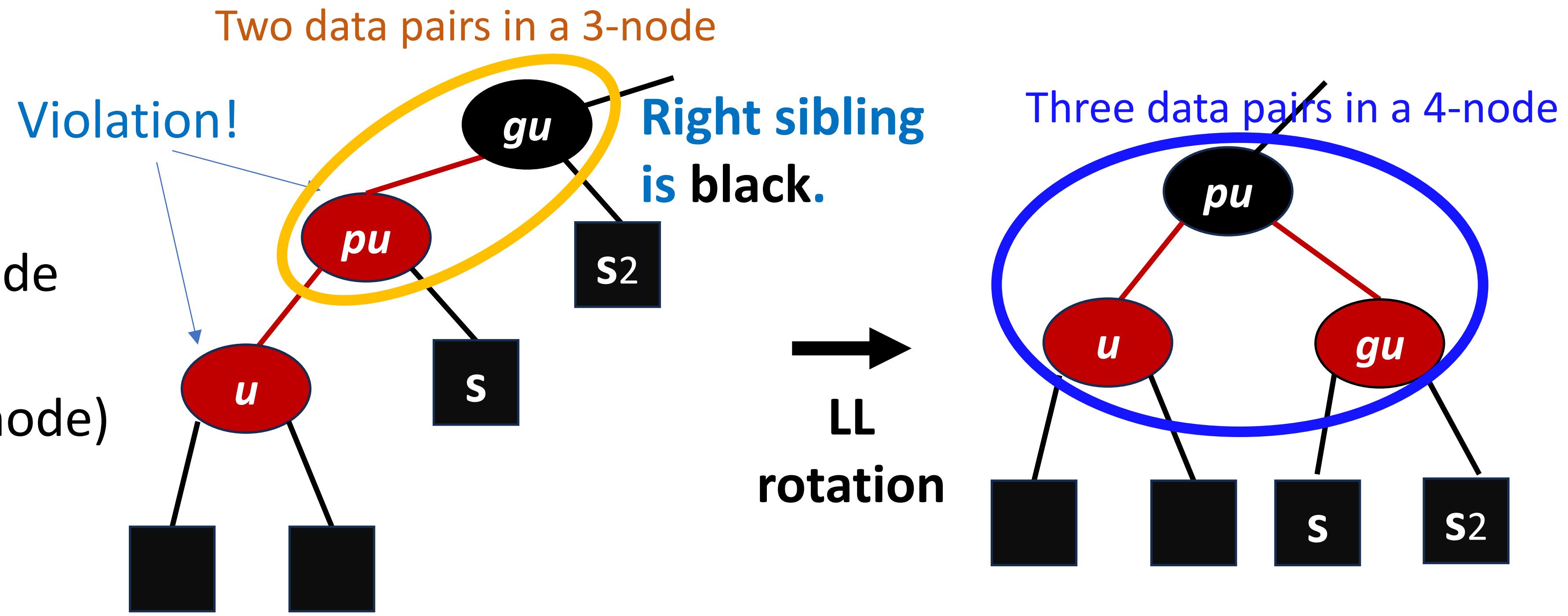


- Equivalent to insert a data pair into a 2-node of 2-3-4 tree

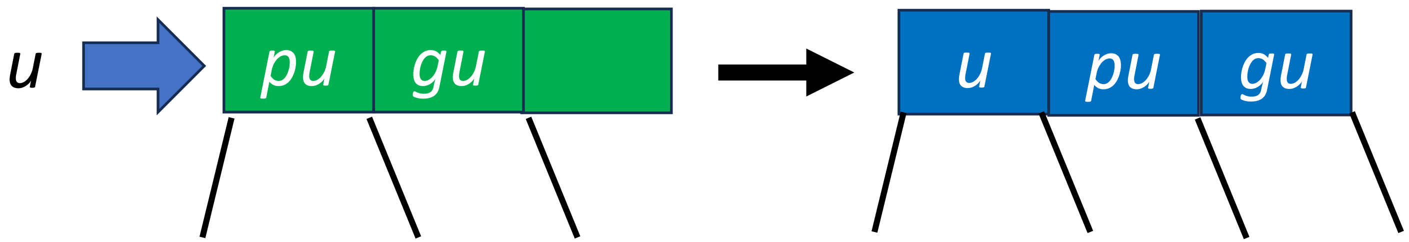


Insertion: parent is red and its sibling is black (1)

- *u*: the new node
- *pu*: parent of *u* node
- *gu*: grandparent of *u* node
- *s*: sibling of *u* node
  (may be an external node)



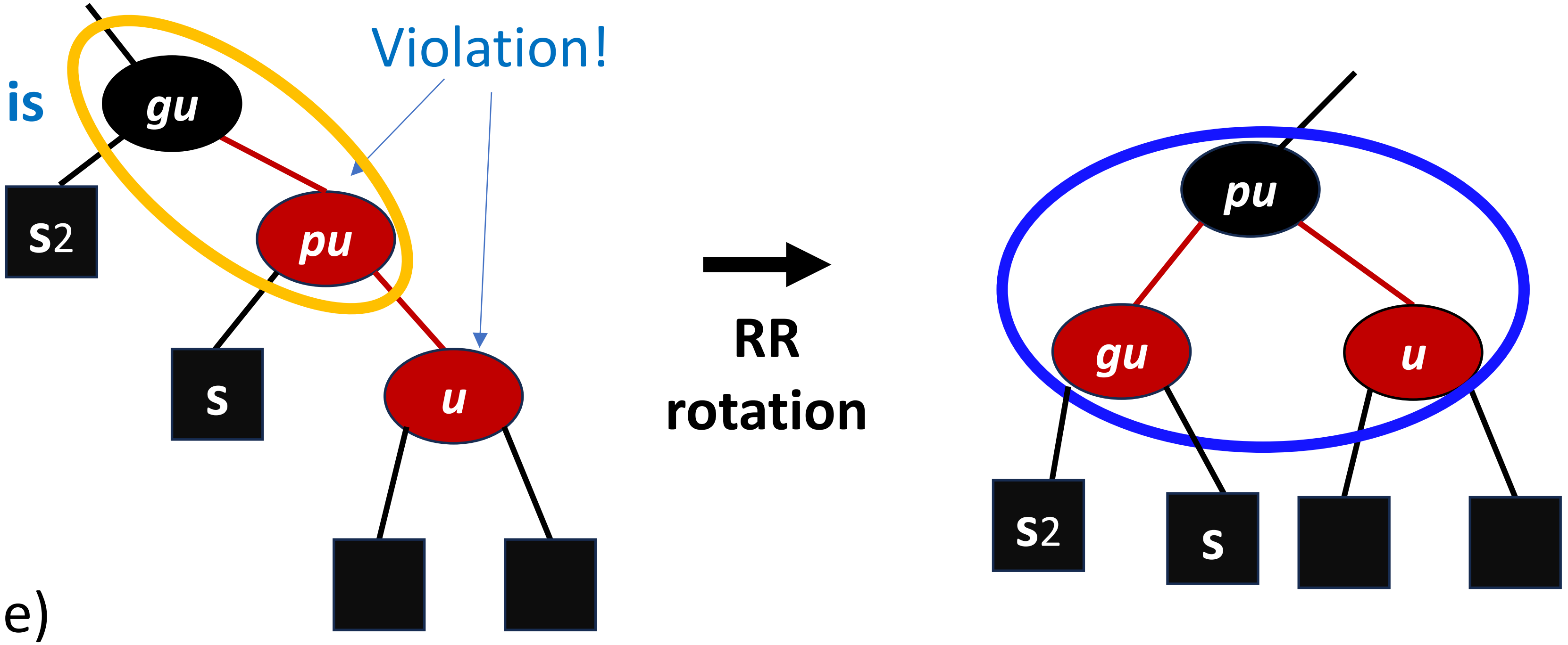
- Equivalent to insert a data pair into a 3-node of 2-3-4 tree and become a 4-node



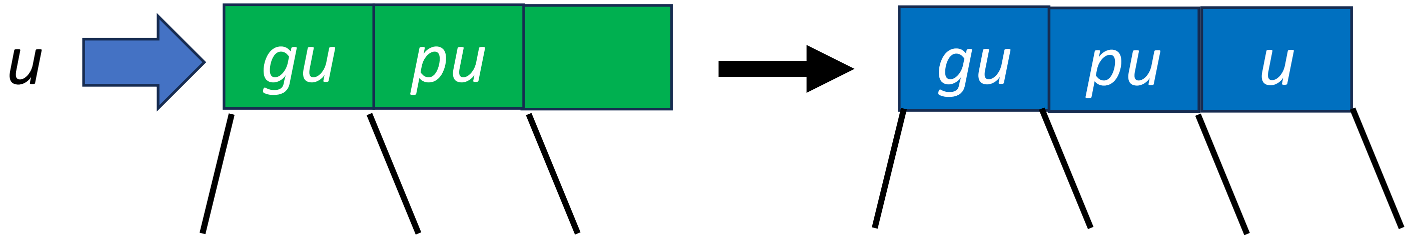
Insertion: parent is red and its sibling is black (2)

Left sibling is black.

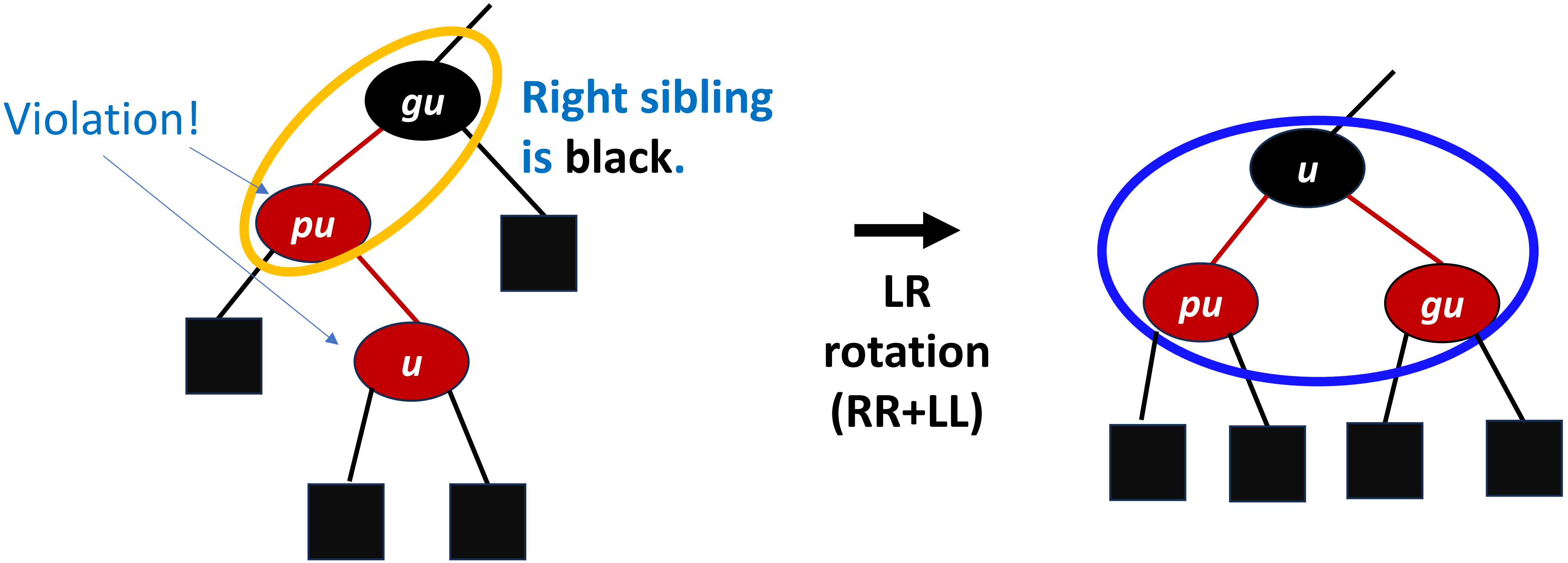
- *u*: the new node
- *pu*: parent of *u* node
- *gu*: grandparent of *u* node
- *s*: sibling of *u* node
  (may be an external node)



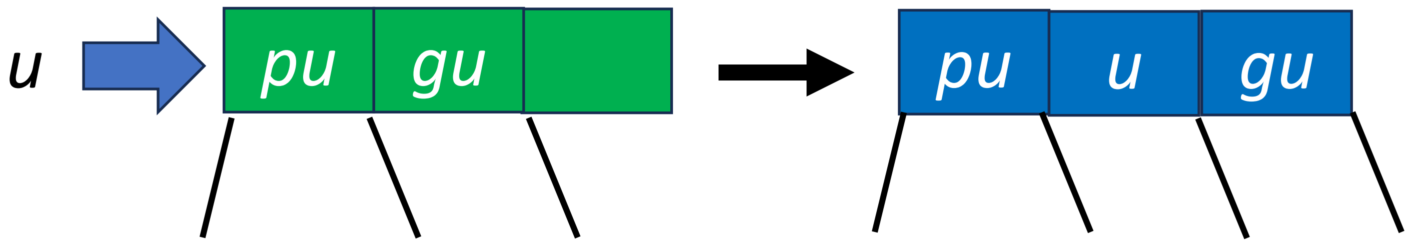
- Equivalent to insert a data pair into a 3-node of 2-3-4 tree and become a 4-node



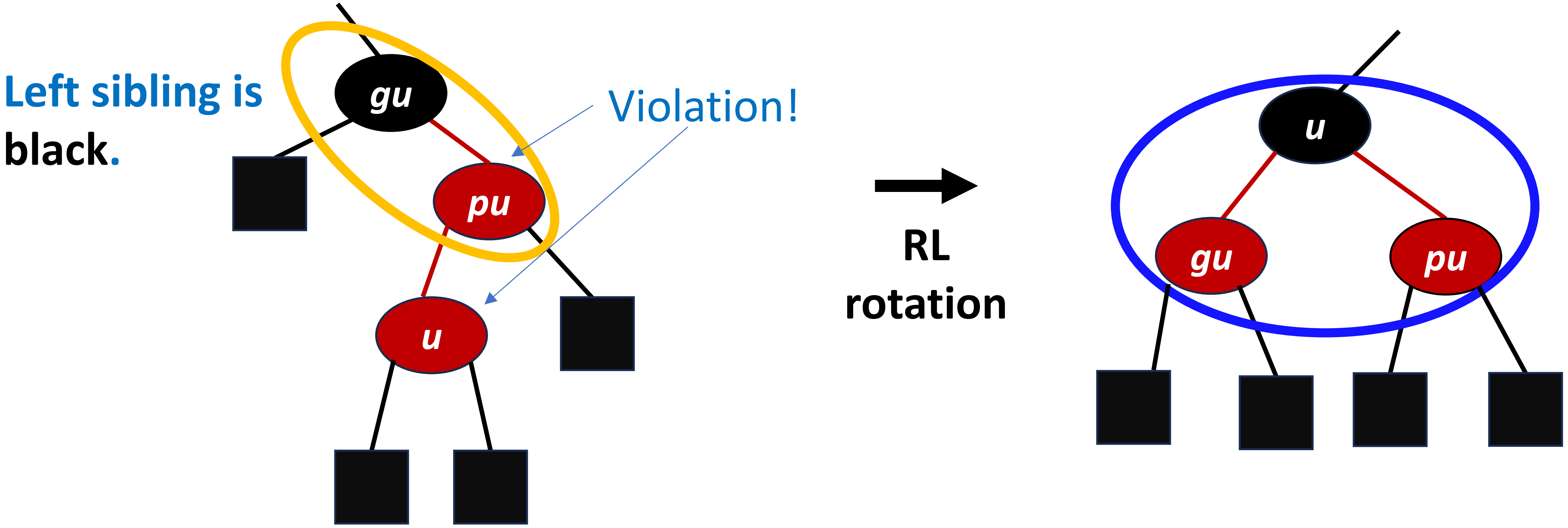
Insertion: parent is red and its sibling is black (3)



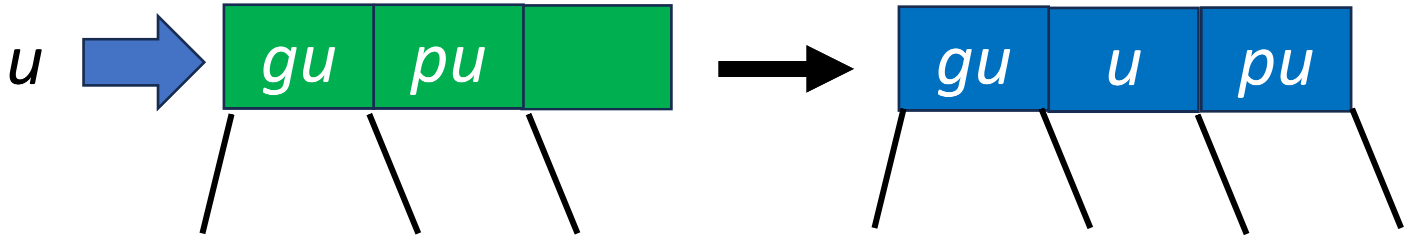
- Equivalent to insert a data pair into a 3-node of 2-3-4 tree and become a 4-node



Insertion: parent is red and its sibling is black (4)



- Equivalent to insert a data pair into a 3-node of 2-3-4 tree and become a 4-node

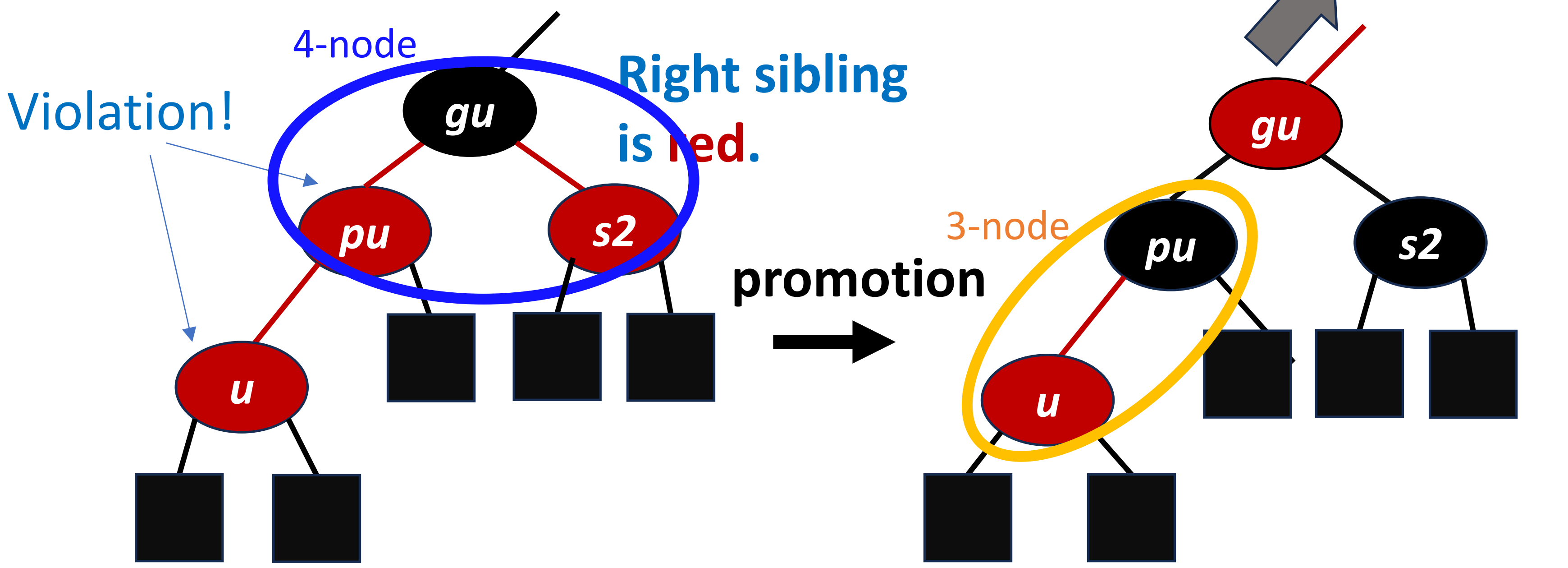


Insertion: parent is red and its sibling is red

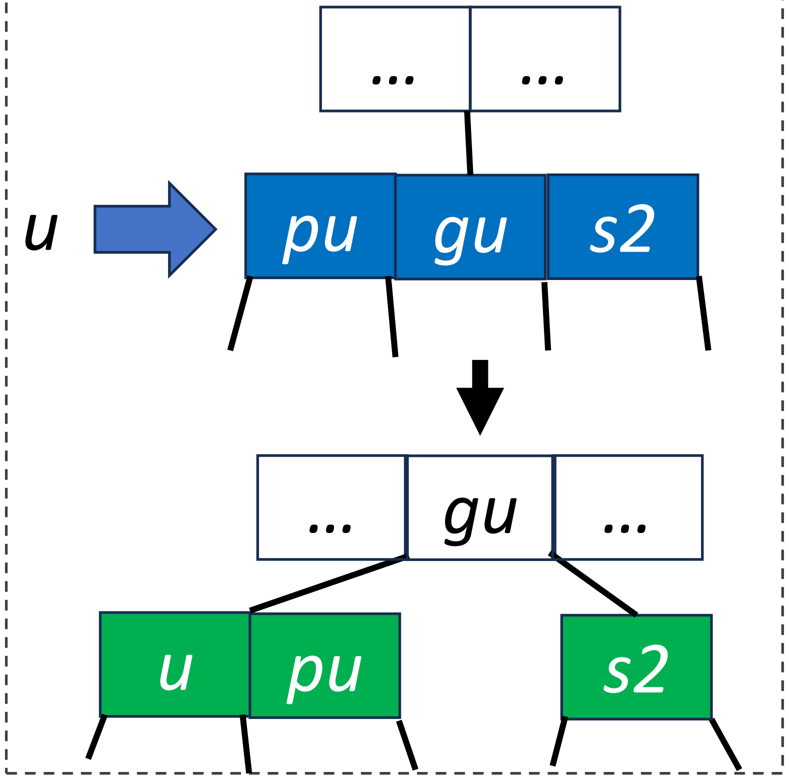
Bottom-up rebalancing

- Change the color of the parent ***pu***, and parent's sibling ***s2*** to **black.**
- Change the color of the grandparent ***gu*** to **red.**
- Continue upward beyond ***gu*** if necessary.
  (rename ***gu*** as ***u*** and repeat)

If gu is root, color it as black.



Equivalent to insert a data pair into a 4-node of 2-3-4 tree. Thus, split is required.



Note that the *black depth* remains unchanged for all the descendants of ***gu***.

Performance

- Lemma 10.1: Let P and Q be the two root-to-external node paths in a red-black tree. Then $length(P) \le 2length(Q)$.

*Length: The number of edges on the path

Red, **black**, **red**, **black**, …, **red**, and **black**

All edges are **black**.

- Lemma 10.2: Let h be the height of a red-black tree, n be the number of internal nodes, and r be the rank of the root.

*Rank: The number of black edges on the path to external nodes.

a) $h \le 2r$

Lemma 10.1

b) $n \ge 2^r-1$

If all internal nodes are black, the number of internal nodes n is 2^r-1.
$r \le \log_2(n+1)$

c) $h \le 2\log_2(n+1)$

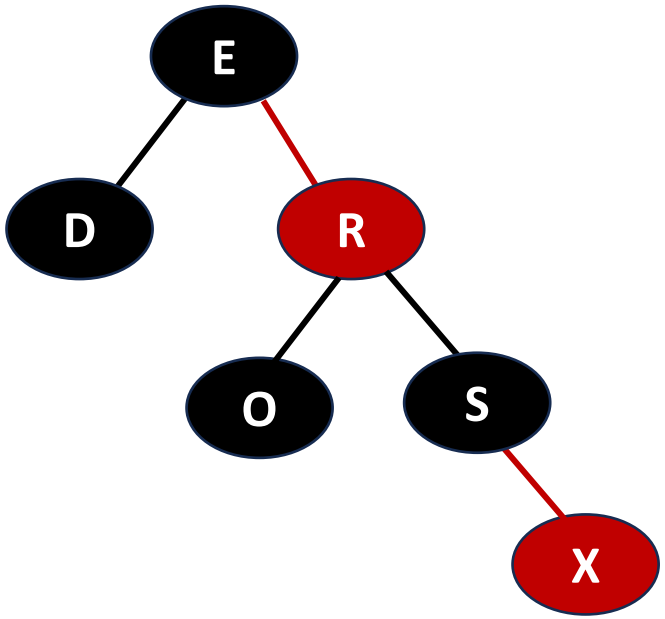
Combine a) and b)

Time complexity of insertion depends on tree height h: O(log n)

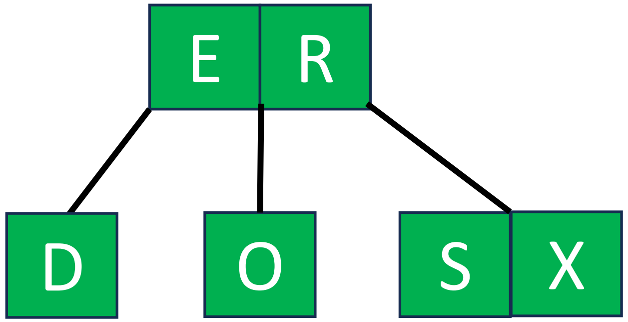
Example of insertion

- Start by inserting “REDSOX” into an empty tree

Red-black tree



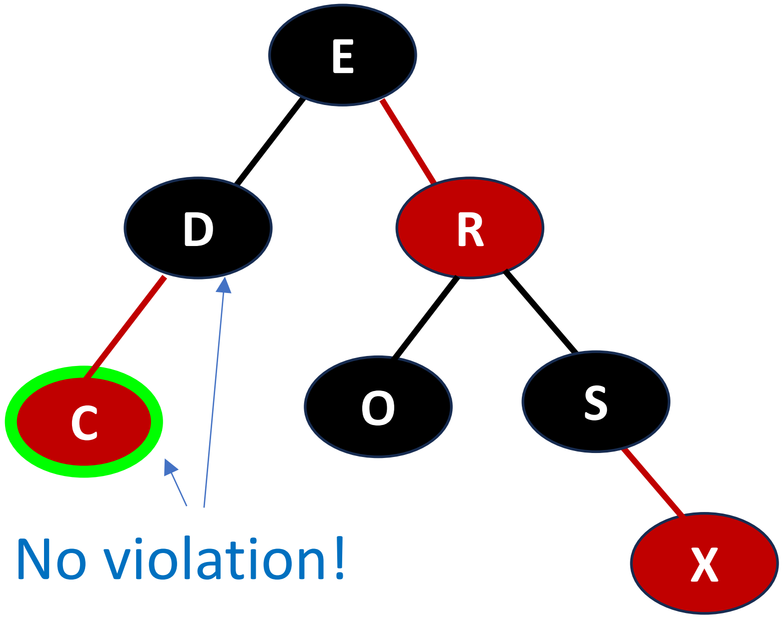
2-3-4 tree

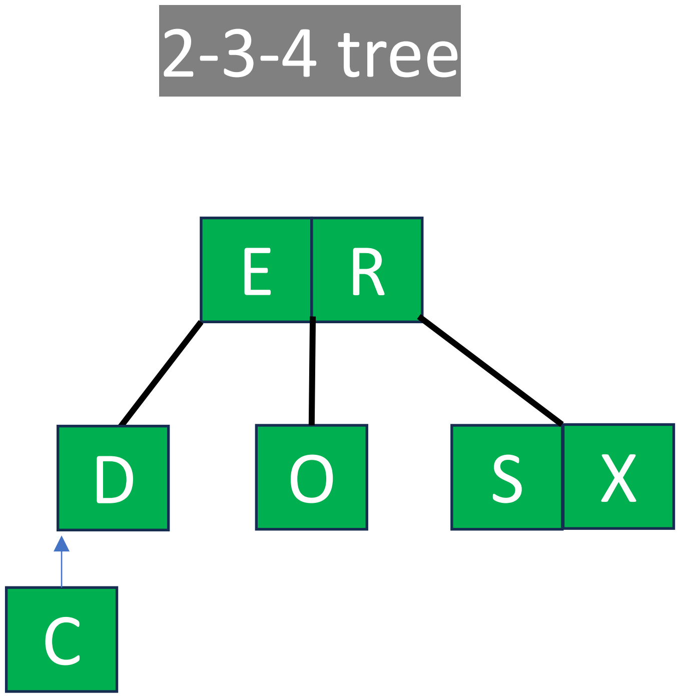


- Let’s insert C, U, B, and S

Example of insertion

- Insert C

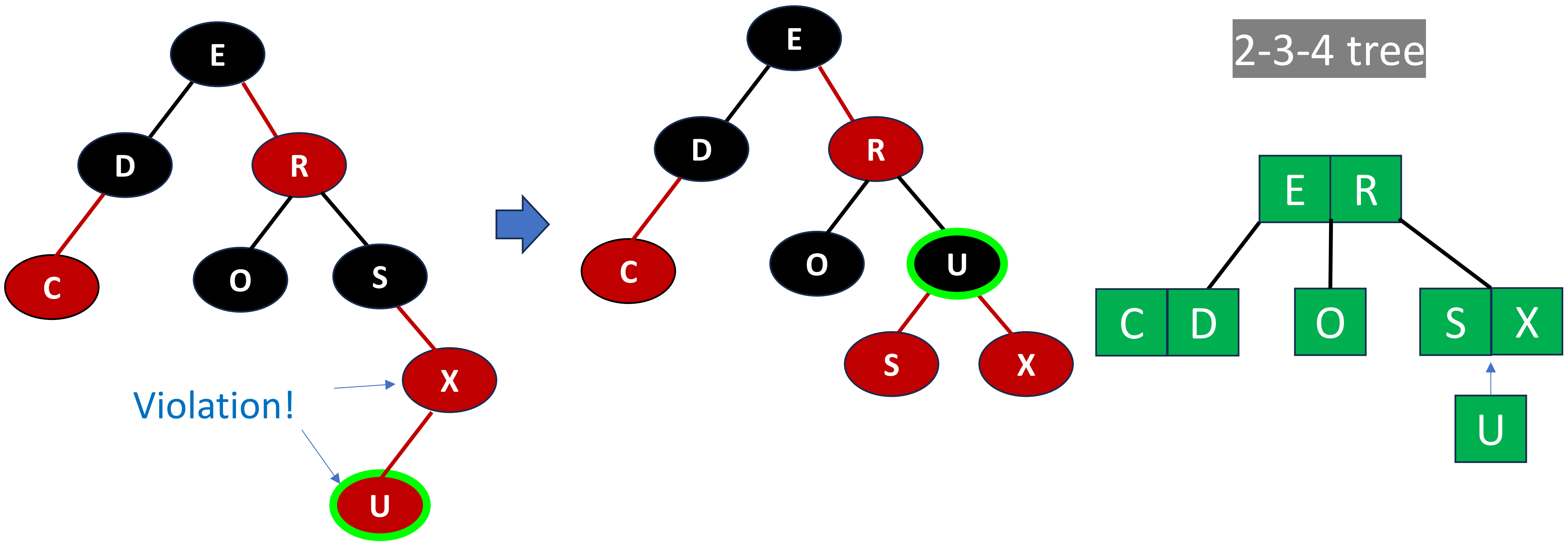




Parent is **black.**

Example of insertion

- Insert U

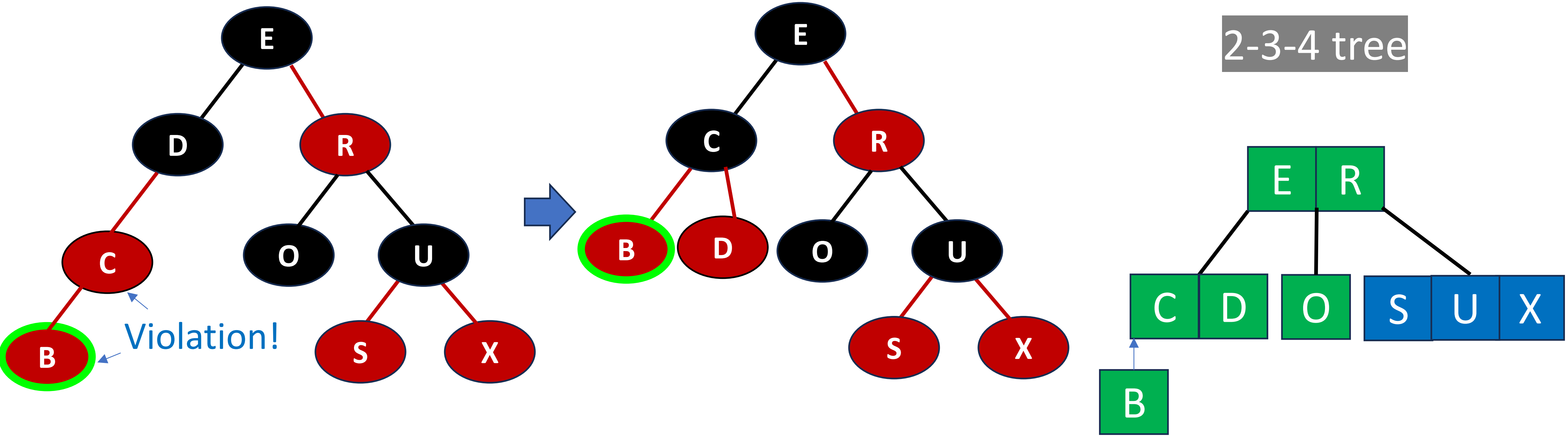


Parent is **red** and its sibling is **black.**

→RL rotation

Example of insertion

- Insert B

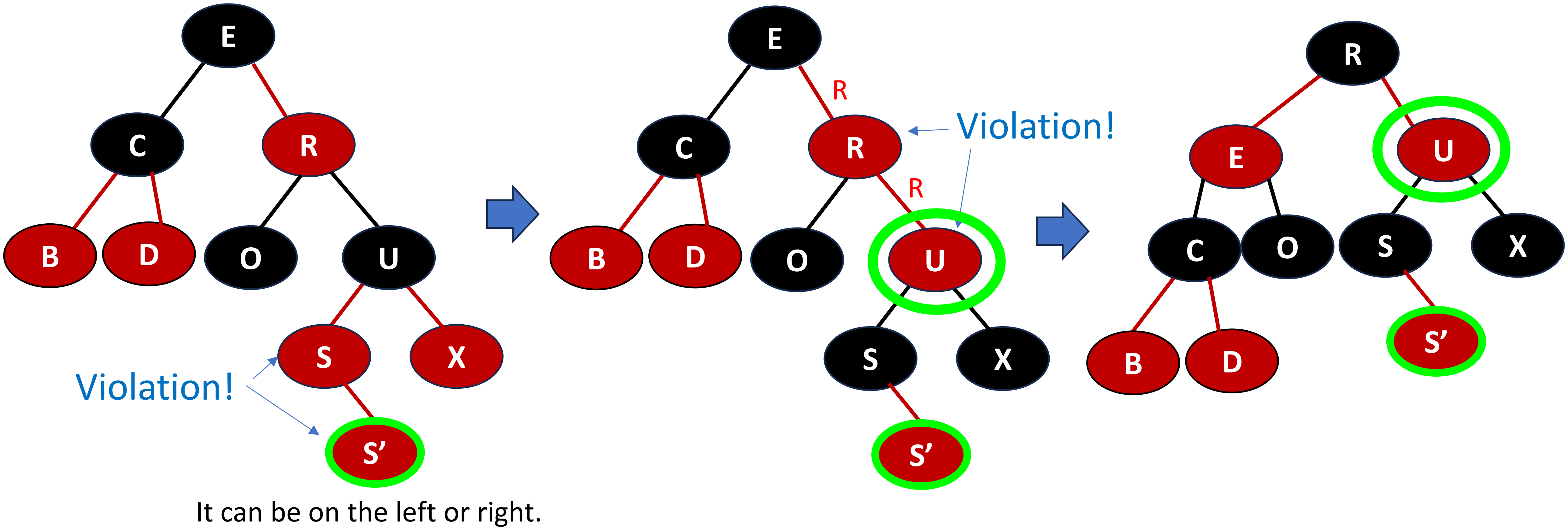


Parent is **red** and its sibling is **black.**

→LL rotation

Example of insertion

- Insert S

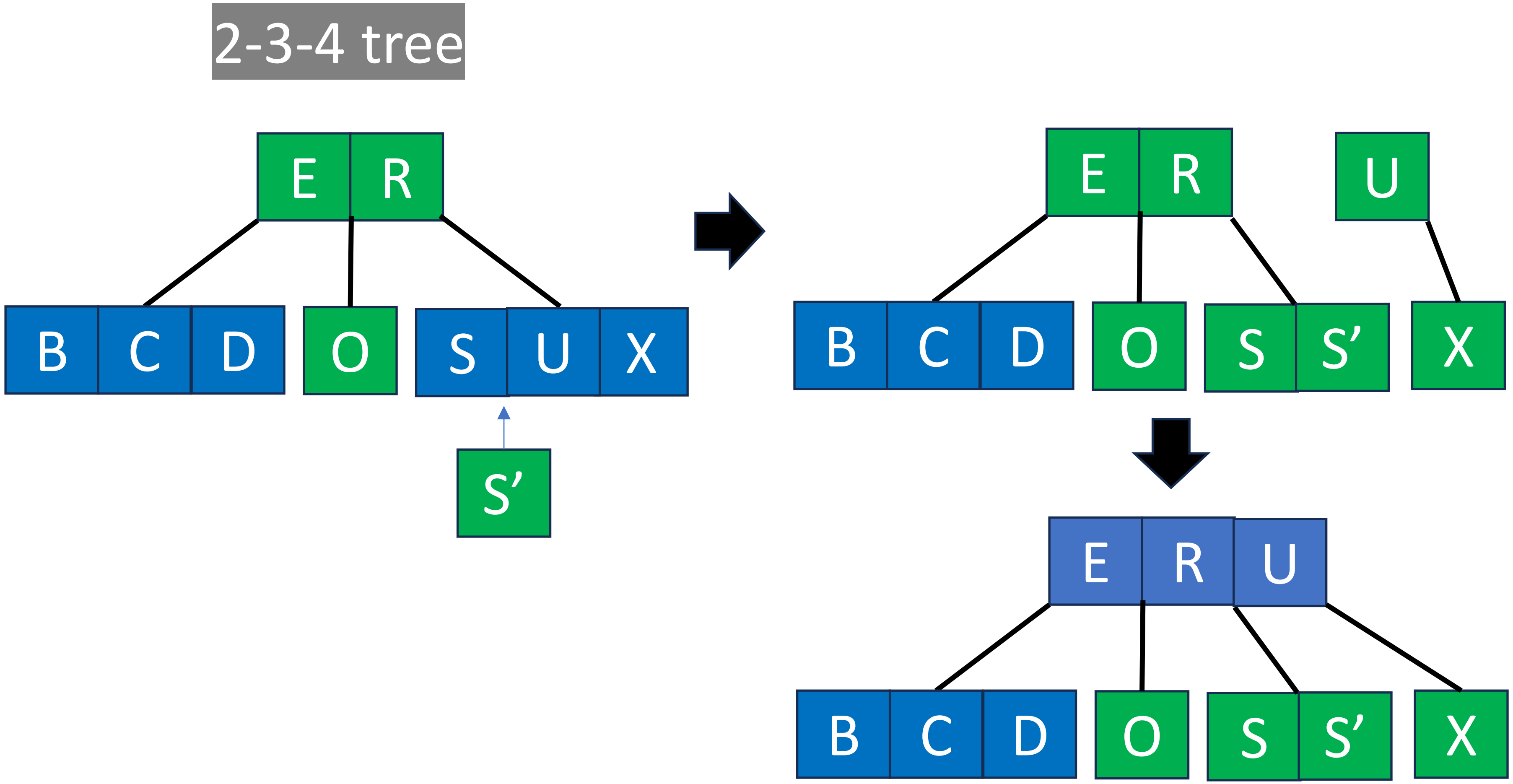


Parent is **red** and its sibling is **red**.
→promotion (color change)

Parent is **red** and its sibling is **black.**
→RR rotation

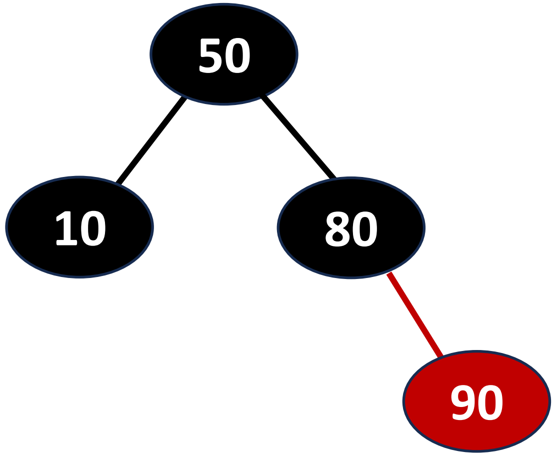
Example of insertion

- Insert S



Exercise

- Given the following red-black tree.
  - Q1: Please insert 70. What is(are) the key(s) in red nodes?
  - Q2: (Continue 1) Please insert 60. What is(are) the key(s) in red nodes?
  - Q3: (Continue 2) Please insert 65. What is(are) the key(s) in red nodes?
  - Q4: (Continue 3) Please insert 62. What is(are) the key(s) in red nodes?

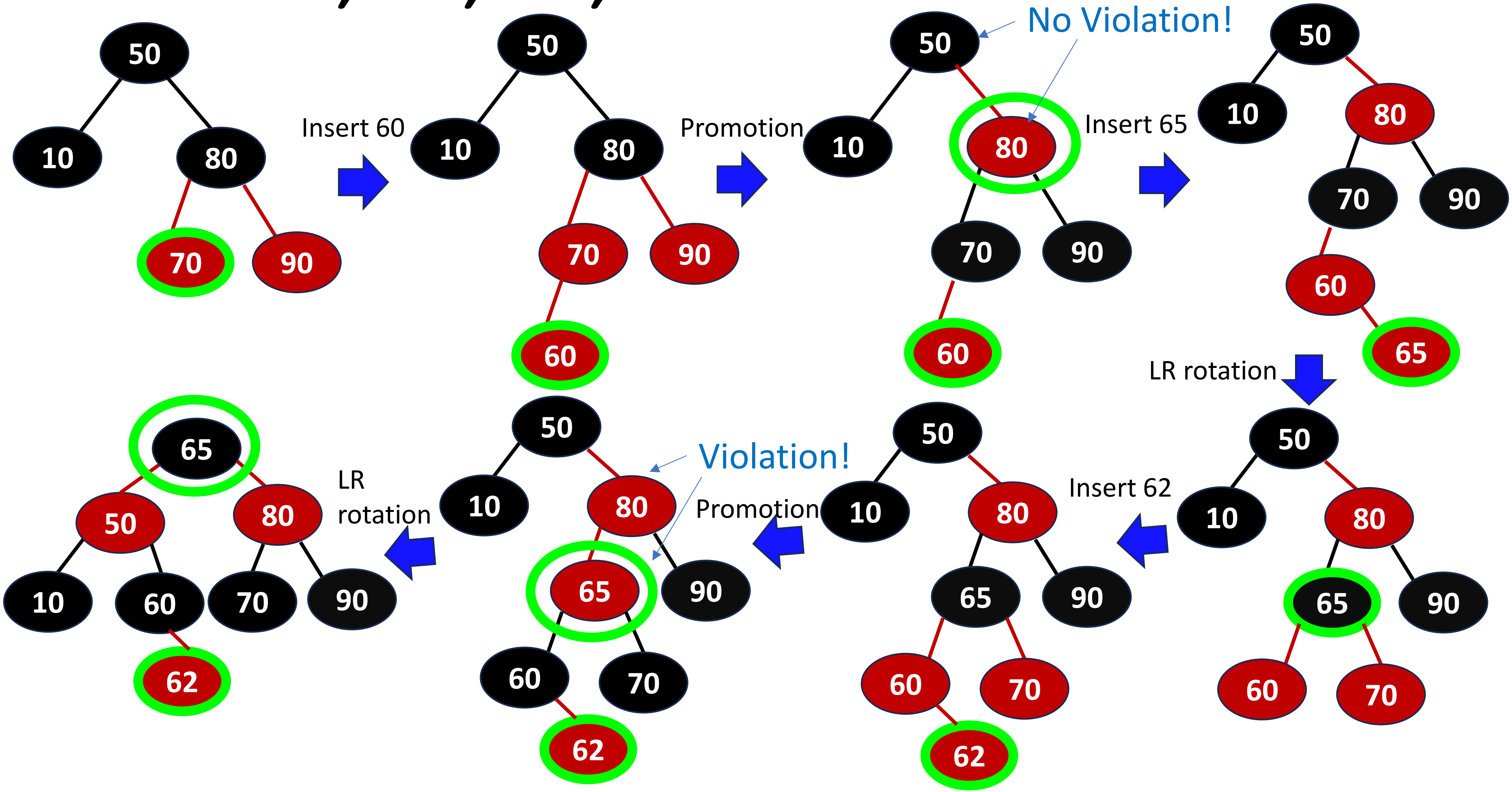


Please reply your answers of Q1-Q4 via the following link:

https://forms.gle/erneNgrctngmjem1A

Group members: 2~4 people

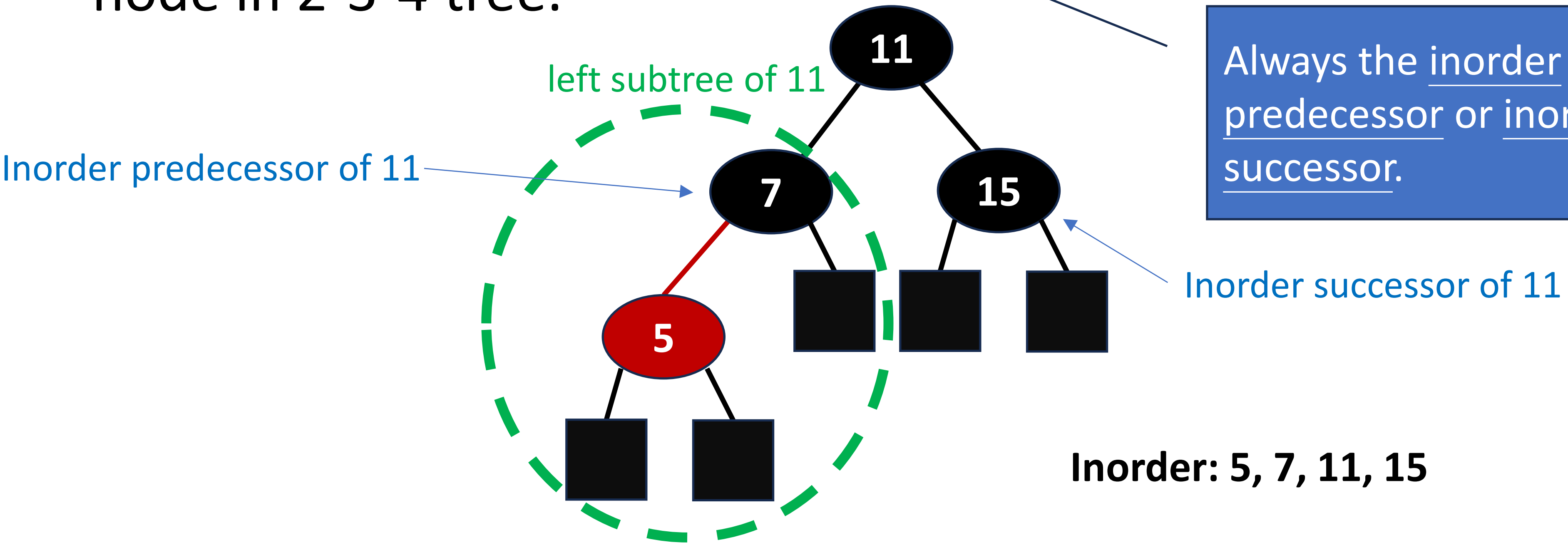
Insert 70, 60, 65, 62



Operation: Deletion

- An exercise in the textbook
- We will only discuss about the deletion of nodes nearby leaf nodes.
  - Reason: The node to replace the deleted node is always from a leaf node in 2-3-4 tree.

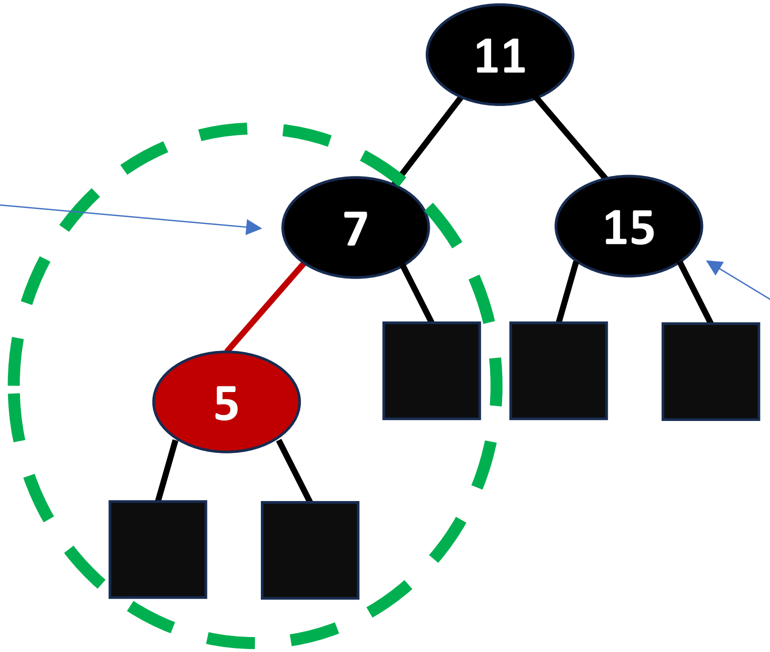
Always the inorder predecessor or inorder successor.



Inorder: 5, 7, 11, 15

The inorder predecessor always has at most one child, since it has the largest key in the left subtree.
The inorder successor always has at most one child, since it has the smallest key in the right subtree.

Inorder predecessor of 11

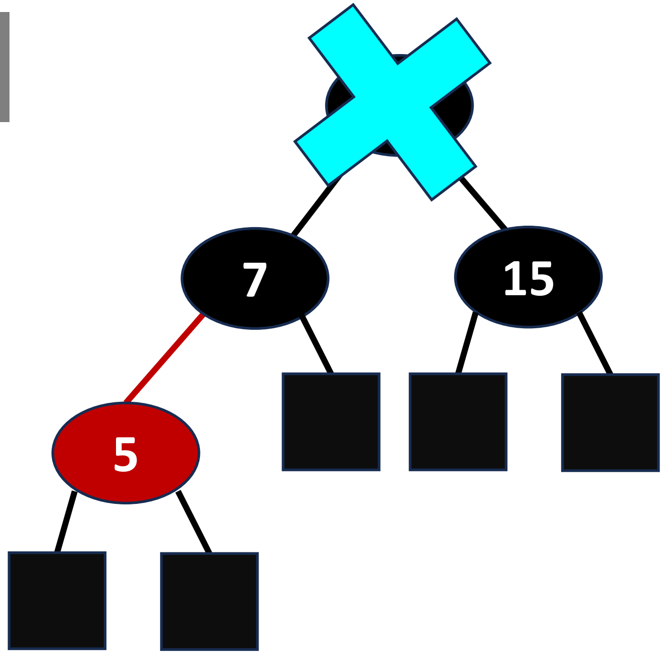


Inorder successor of 11

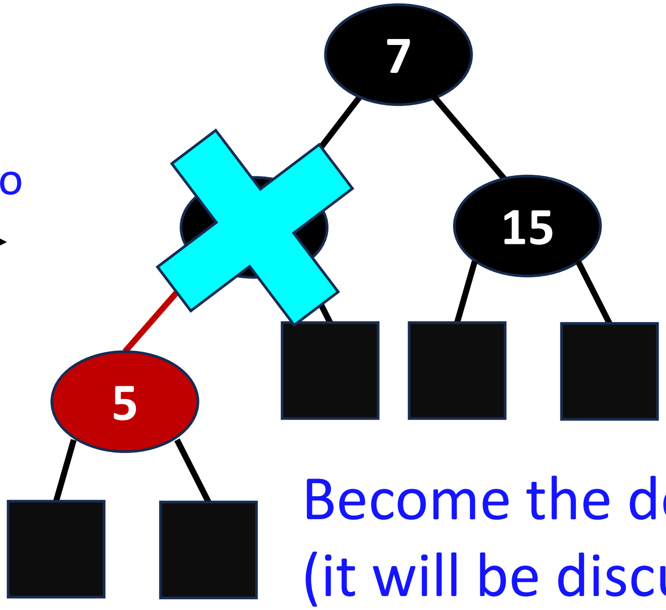
Another case: when 7 is red

After transformation, 7 will become in a black node and we don’t have to change the structure.

Delete 11

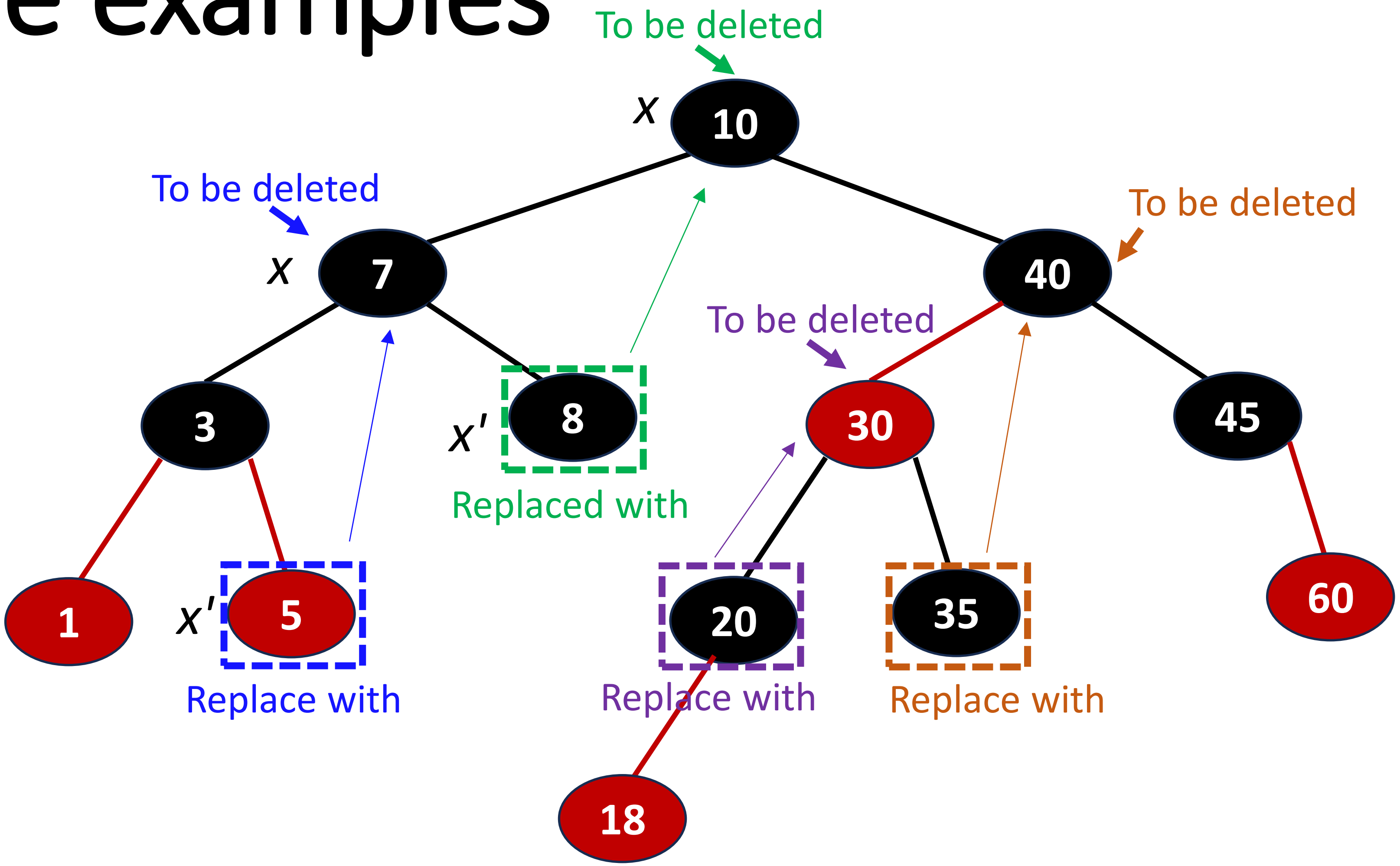


Transform to



Become the deletion in case y1 (it will be discussed later).

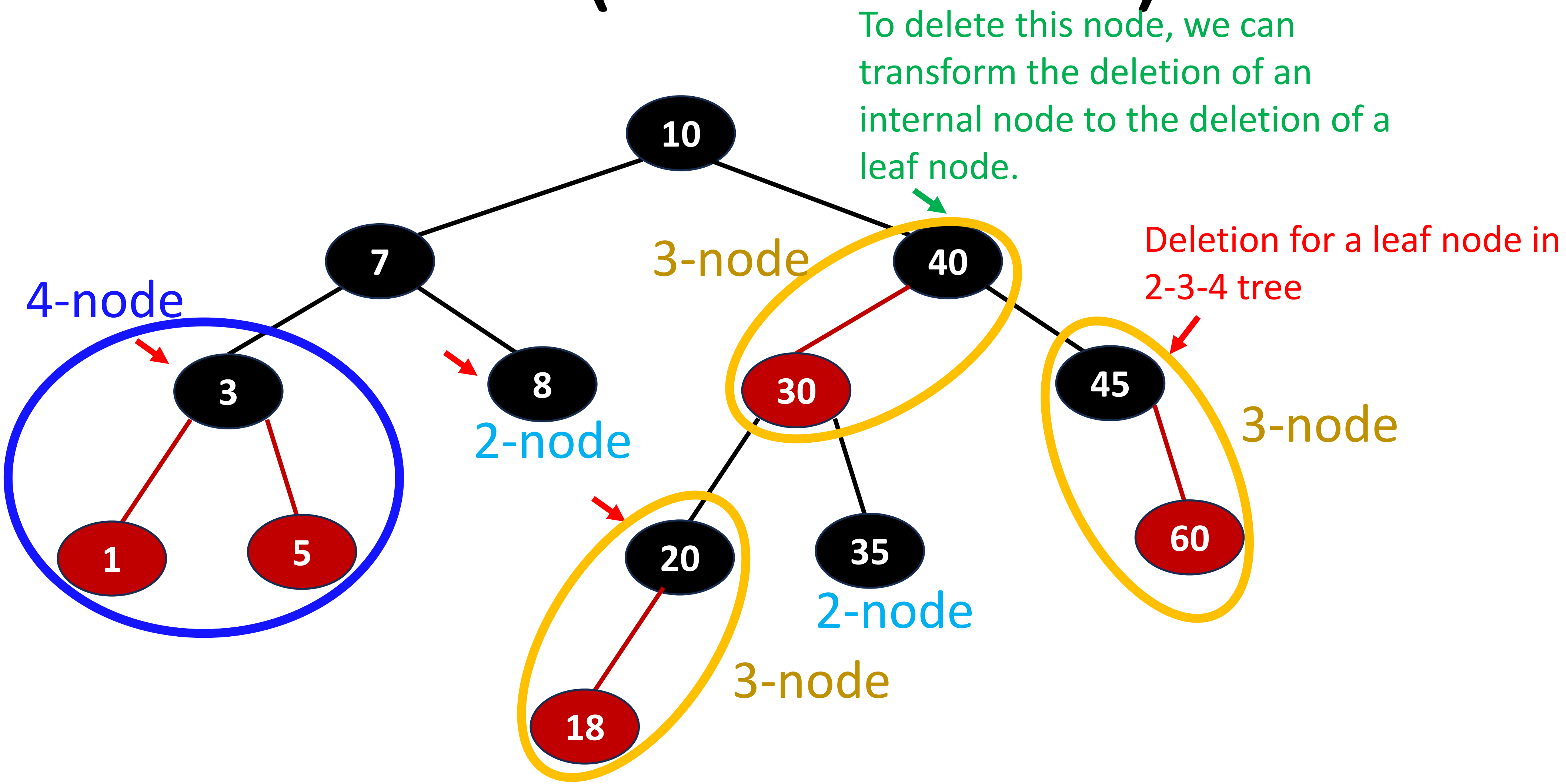
More examples



x'

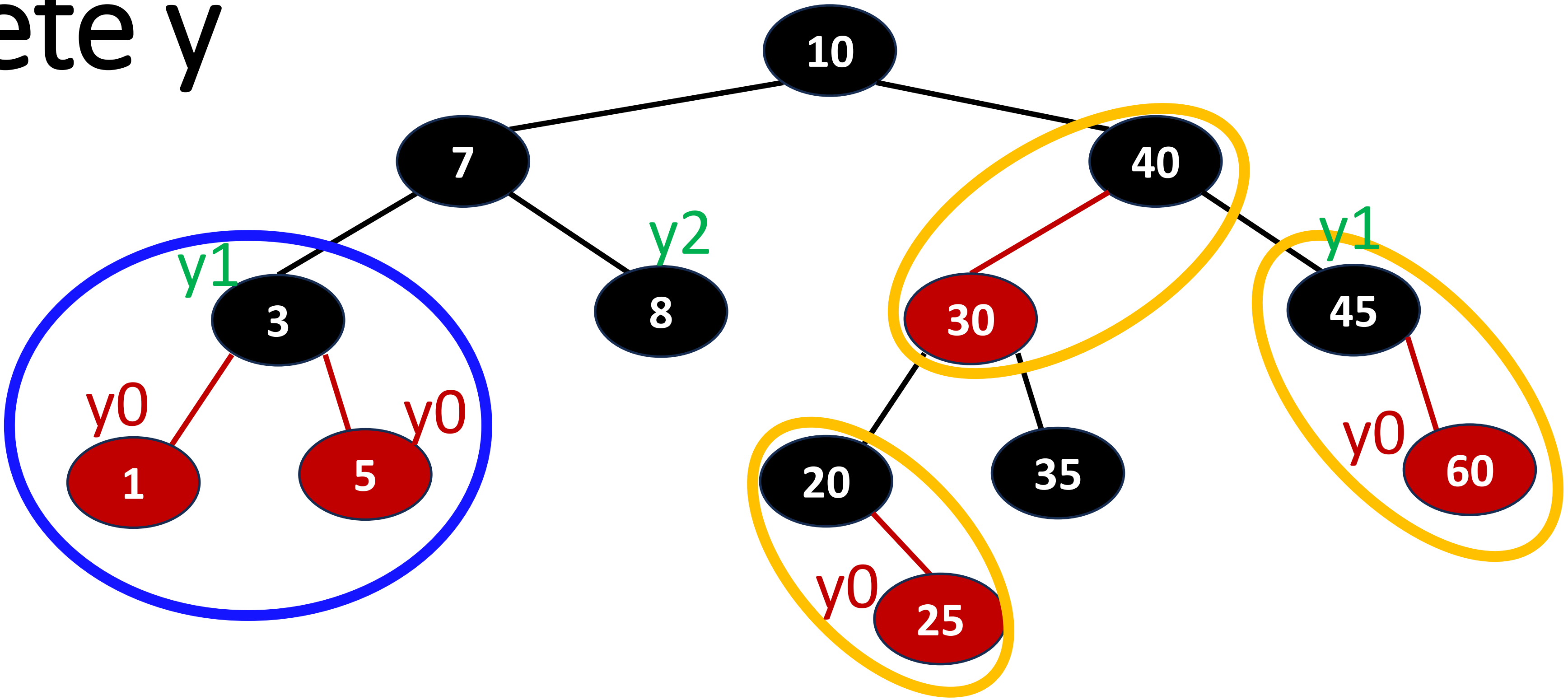
Delete x and replace with x':
After transformation, x' becomes the node to be deleted and it is always in the leaf node of a 2-3-4 tree.

Cases of deletion (in 2-3-4 tree)



Recall: How do we delete the data pair in the 2-node, 3-node, and 4-node in 2-3-4 tree?

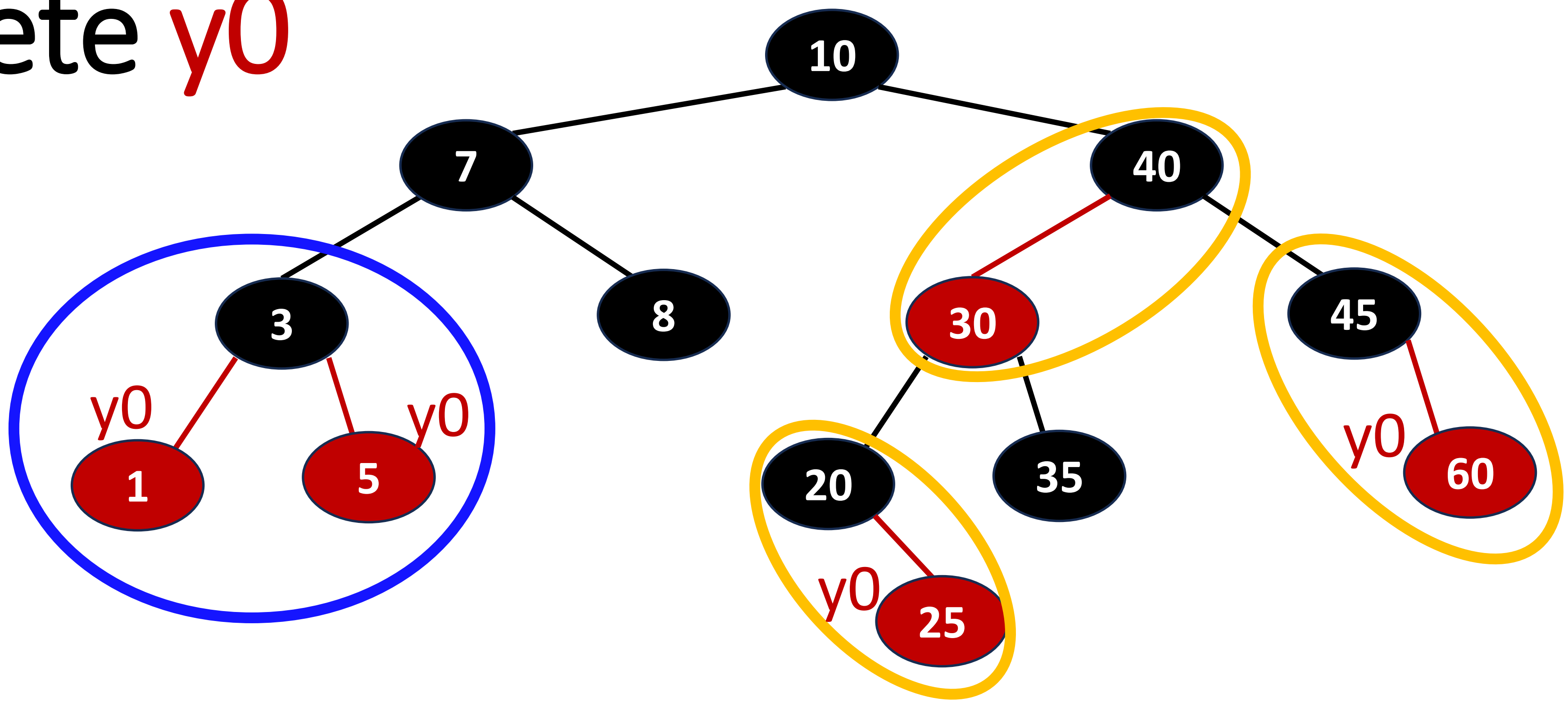
Delete y



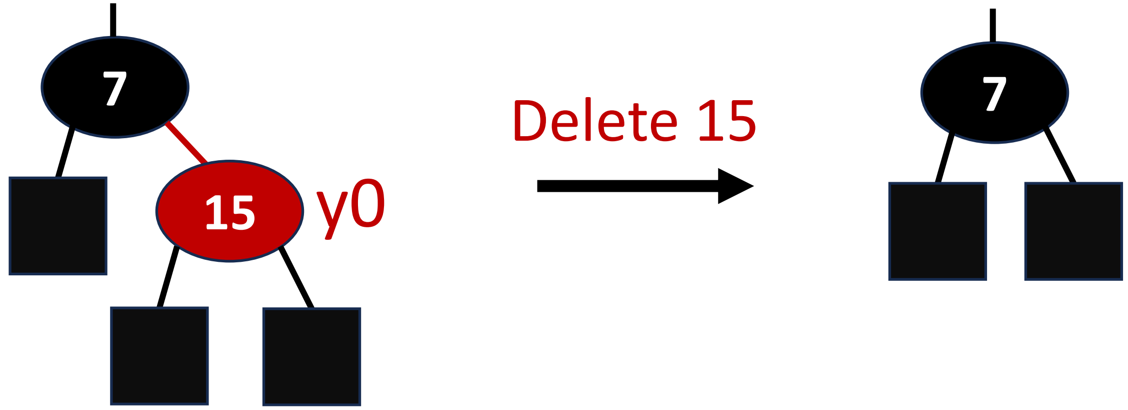
y0, y1, y2 are different types of nodes to be deleted:

- y0: the node connects its parent with a **red link** or it's a red node.
  - It's within a 3- or 4-node in 2-3-4 tree. → Simply delete y.
- y1 and y2: the nodes connect to its parent with a **black link**.
  - y1: with red child node. It's within a 3- or 4-node in 2-3-4 tree.
  - y2: without red child node. It affects balance in 2-3-4 trees and needs reshape.

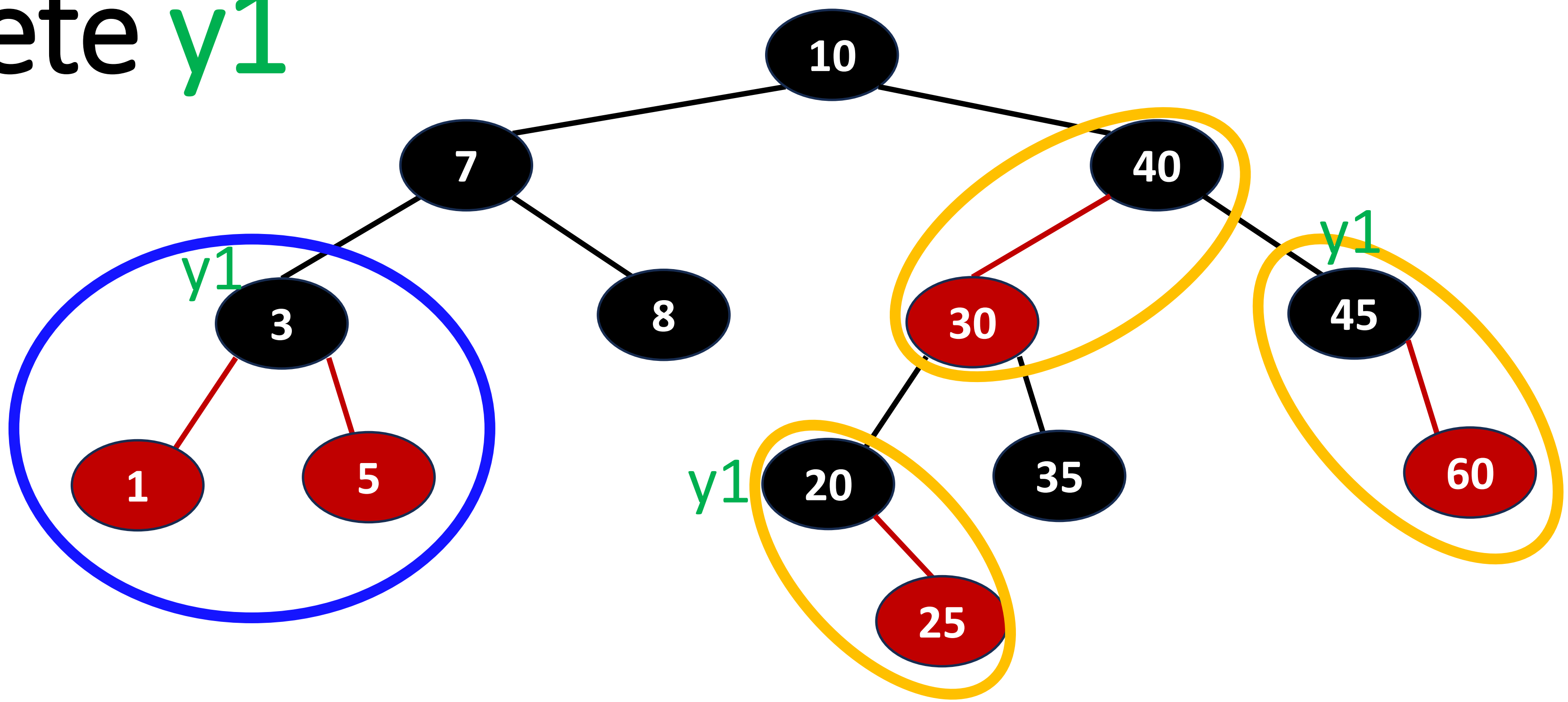
Delete y0



- y0: the node connects its parent with a **red link** or it's a red node.
- Delete y0 and promote one of its external nodes.



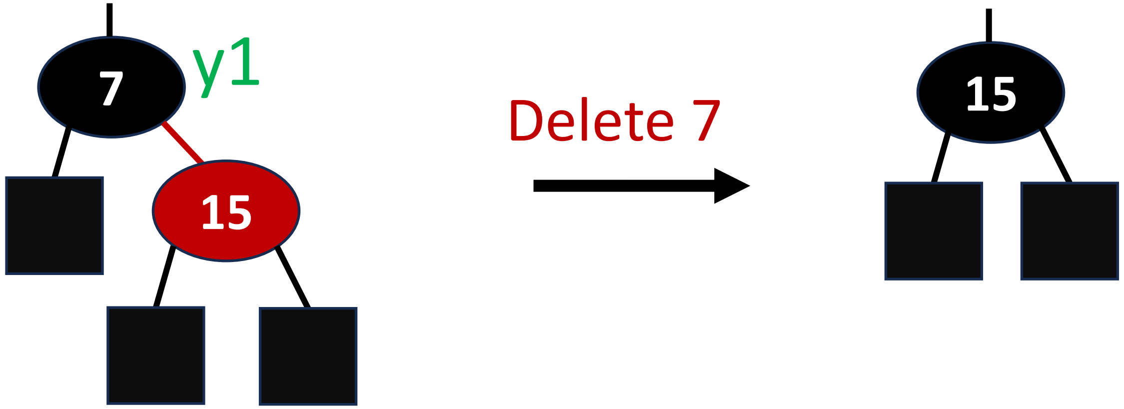
Delete y1



- y1: it's a **black node** with at least one **red child** node.

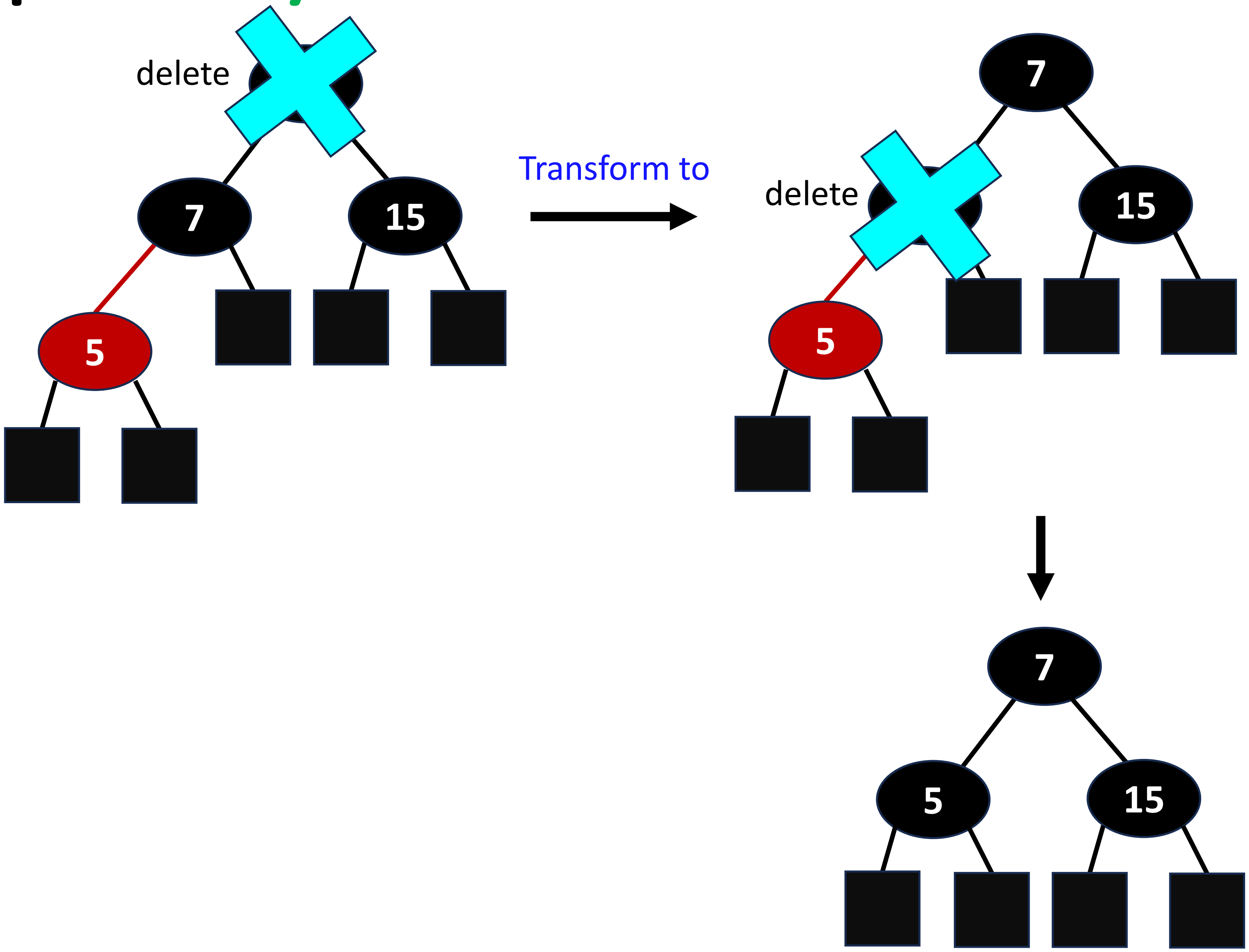
Within a 3- or 4-node in 2-3-4 tree.

- Delete y1, promote one of its child nodes, which is then colored **black**.



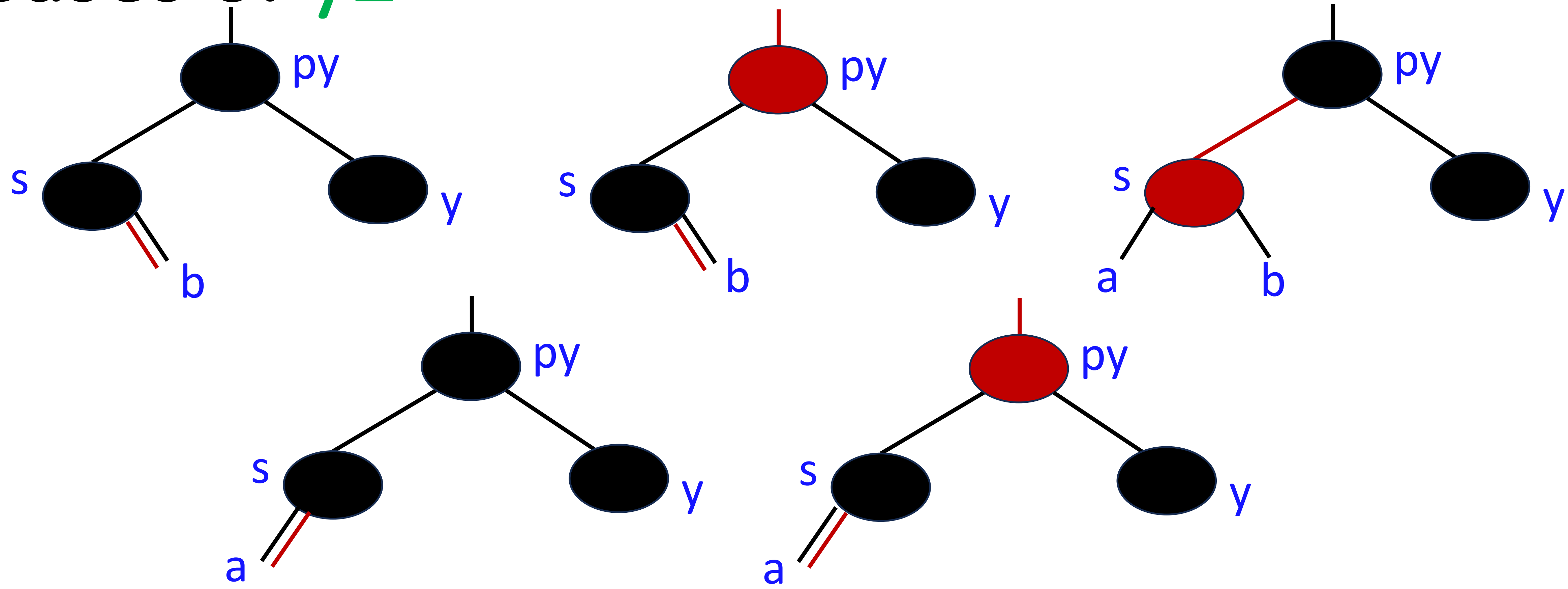
Example of y1 deletion

Delete 11



Cases of y2

y2: it's a **black node** without **red child** node.

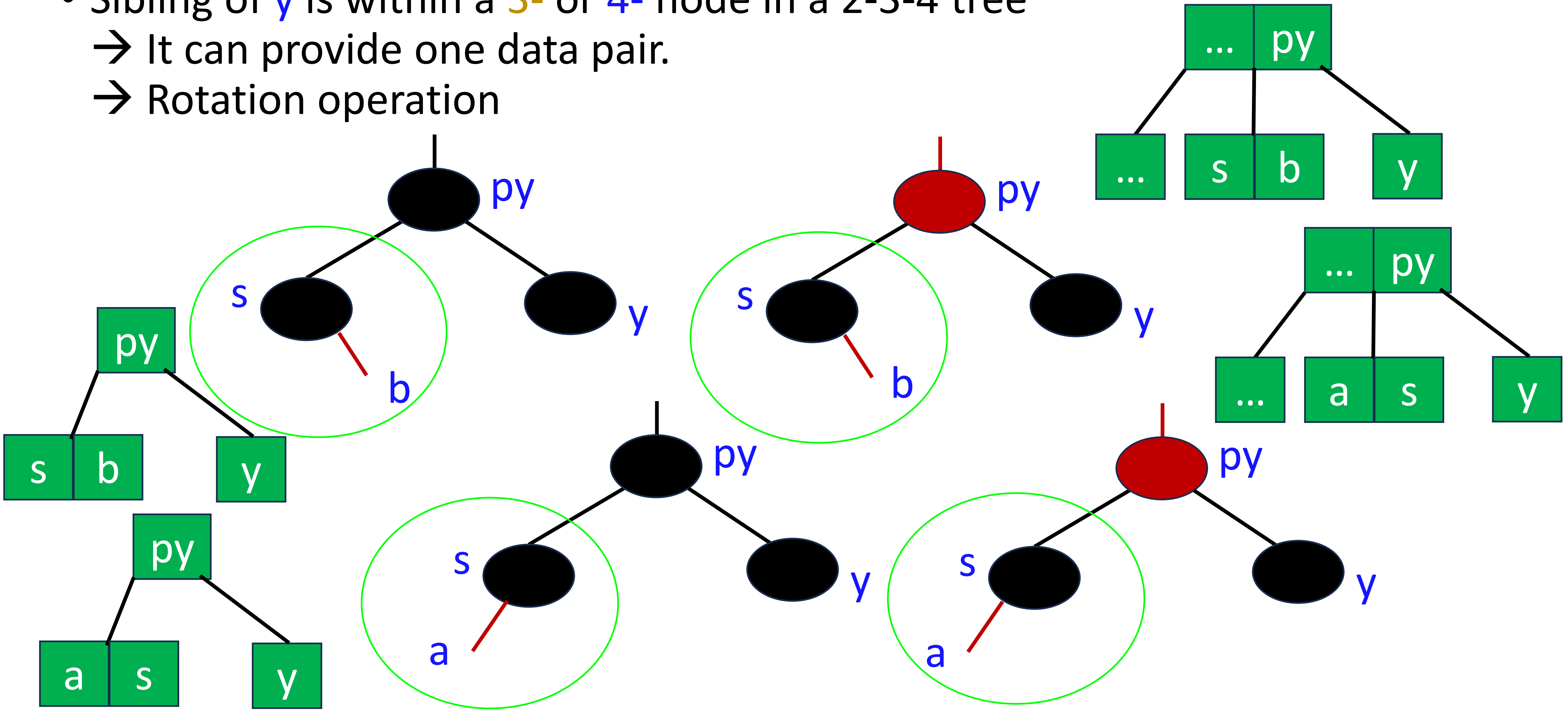


- Delete y:
  - It's better that the tree height remains the same.
  - If the reshaping propagates towards root, tree height may be reduced by one.
- Before reshaping, it's a b-tree (2-3-4 tree). After reshaping, it's still a b-tree.

Type I of y2

Recall: In m-way search tree, **rotation** has high priority. Otherwise, **combine** may reduce the height by 1.

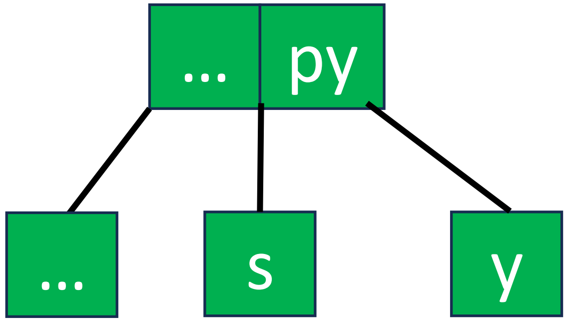
- Sibling of y is within a 3- or 4- node in a 2-3-4 tree
  → It can provide one data pair.
  → Rotation operation

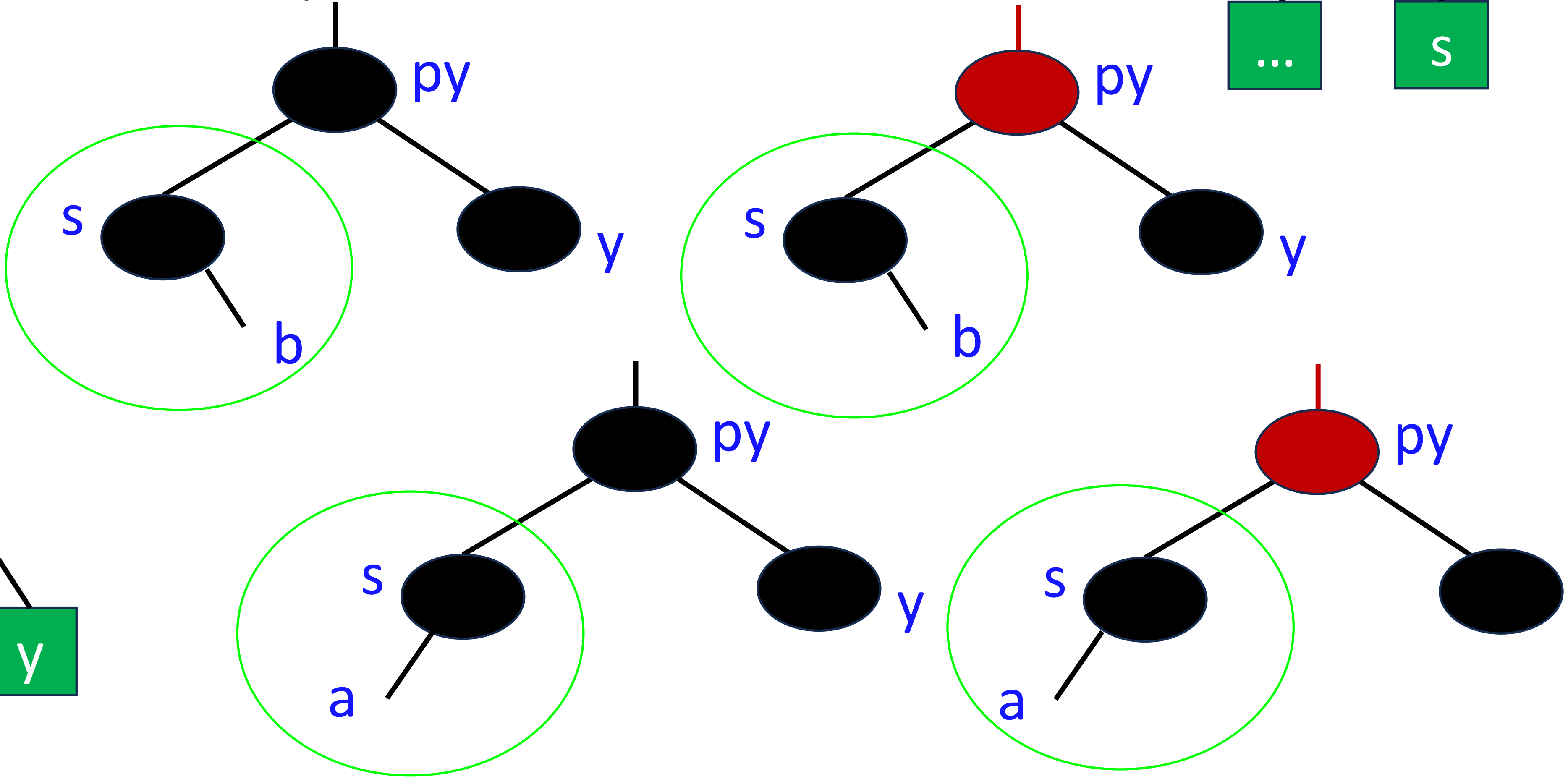


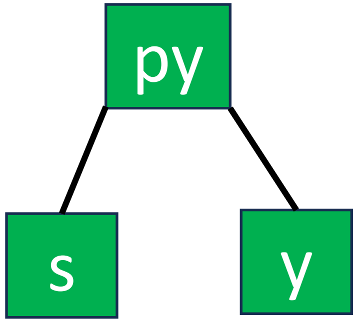
Type II of y2

Recall: In m-way search tree, **rotation** has high priority.
Otherwise, **combine** may reduce the height by 1.

- Sibling of y is within a 2-node in 2-3-4 tree
  → It cannot provide one data pair.
  → We cannot do rotation.
  → Combine operation



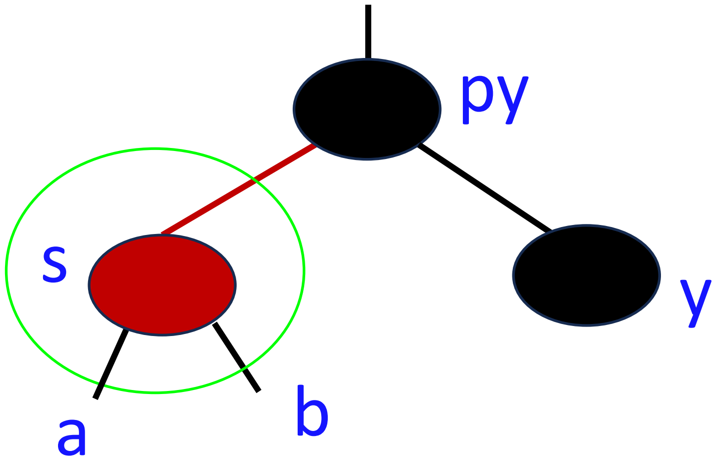


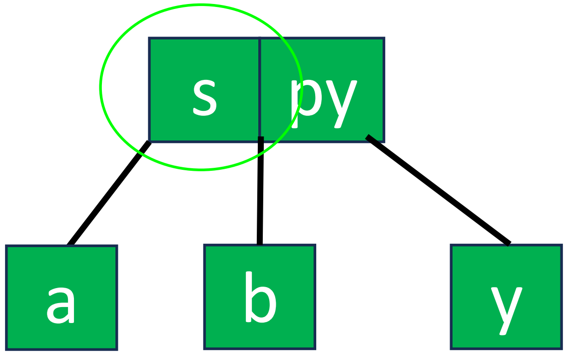


Type III of y2

Recall: In m-way search tree, **rotation** has high priority.
Otherwise, **combine** may reduce the height by 1.

- In red-black tree, y has a red sibling s.
  In 2-3-4 tree, s is not y's sibling. Actually, y's siblings are a and b.
  → Reshape the red-back tree, so that y's sibling is a black node.





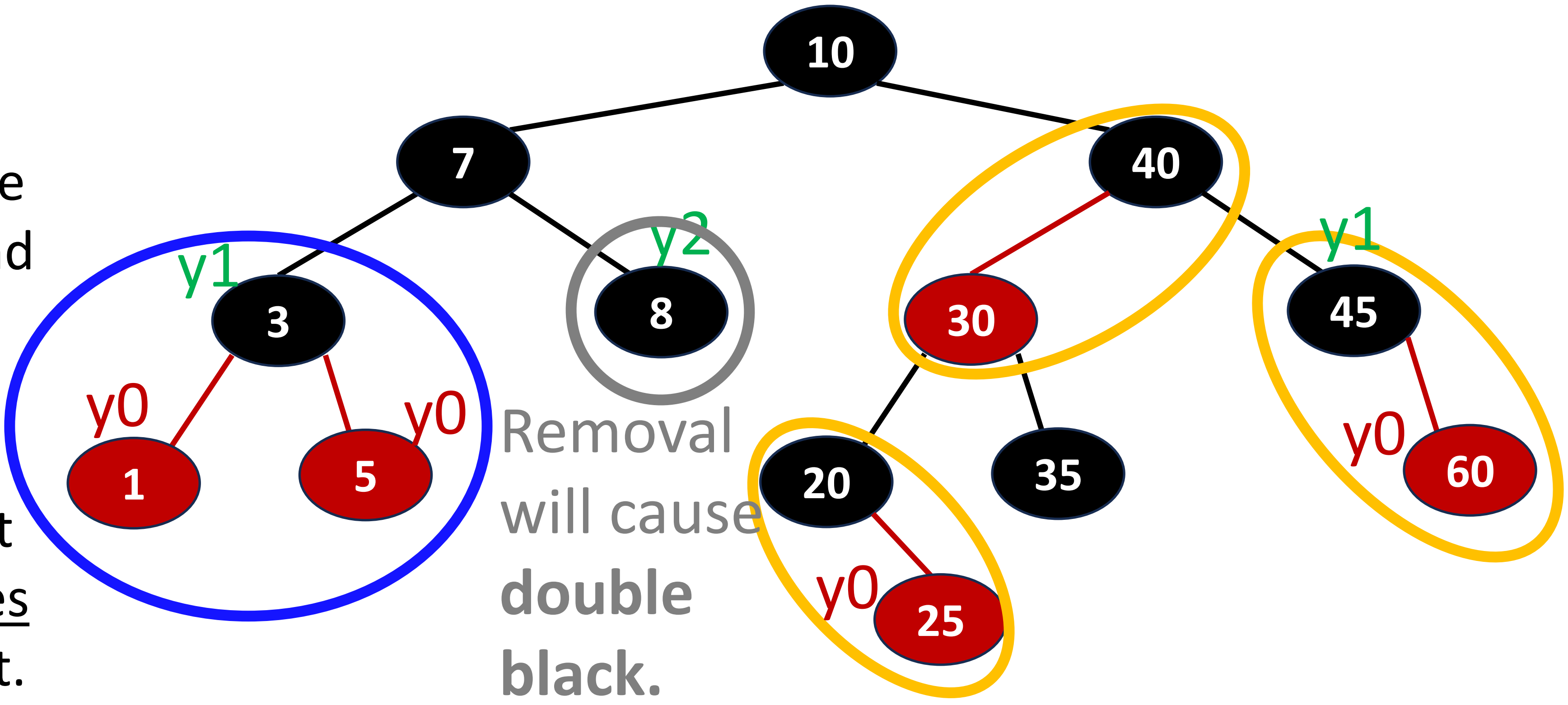
- The deletion may cause chain reaction.

Note of deletion

- We only adjust the nodes which are located at nearest two levels in the corresponding 2-3-4 tree.
- The adjustment may cause chain reaction (bottom-up towards root).

→Resulting from "**double black** in y".

- Originally, a path has two consecutive black nodes.
- Because of the deletion, one black node *y* is removed, and the other node is moved to the location of *y*.
- The node *y* is marked as **double black** to show that it should have <u>two black nodes</u> to maintain the black height.

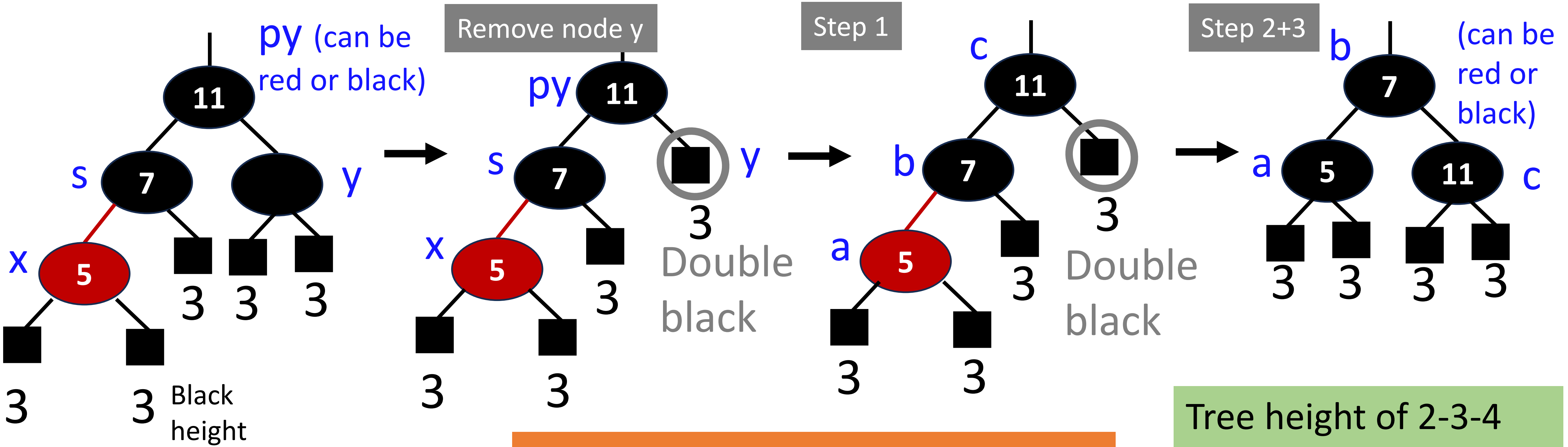


Type I of y2

Type I: Sibling s is **black** and has a **red child** x.

Generally, relabel to let key a < key b < key c

1. Relabel nodes: x → a, s → b, and py → c (For LL case in this example)
2. Replace the original py with b. Make a and c its children. Keep inorder relationships unchanged.
3. Color a and c **black**, and color b the former color of py.



y's sibling can provide a data pair.

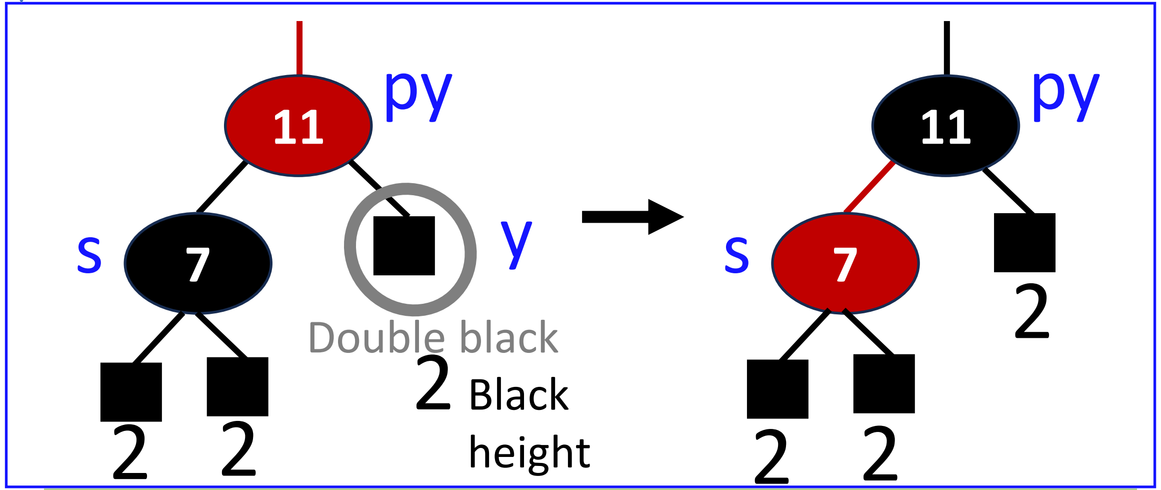
Tree height of 2-3-4 tree is not changed.

Type II of y2

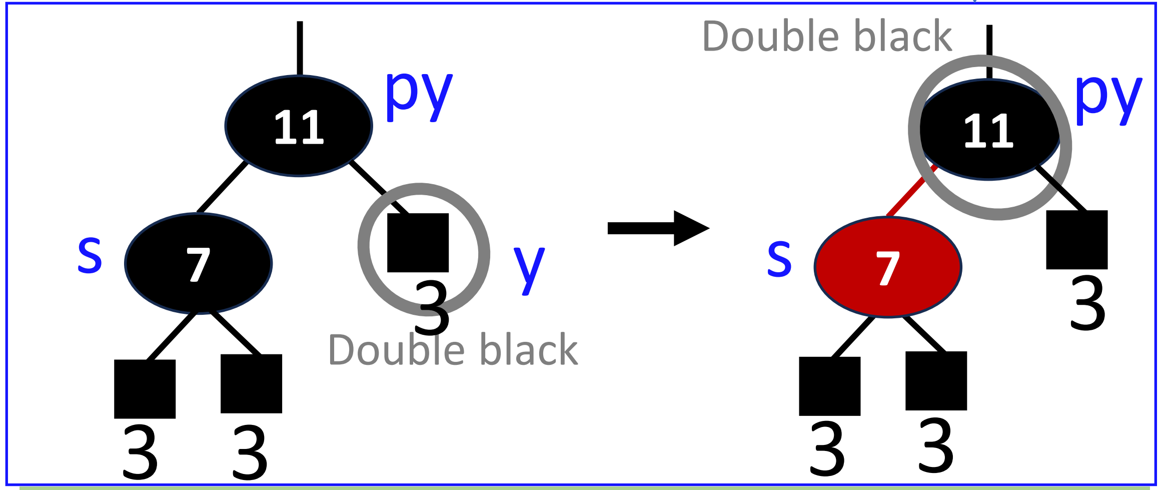
<u>Type II: Sibling s is **black** and has **black children**.</u>

1. Color s **red**.
2. If py is **red**, we color py **black**.
3. Else if py is not the root, we color py **double black** and repeat resolving for py.

May repeat Type I, Type II, Type III of y2



Tree height of 2-3-4 tree is not changed.



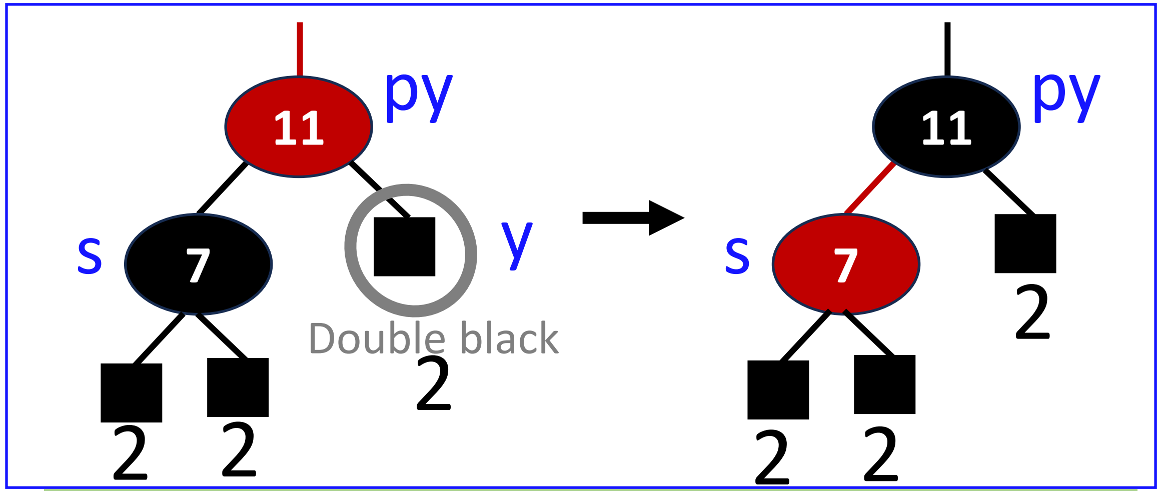
Tree height of 2-3-4 tree reduces by one if py is root.

May result in chain reaction.

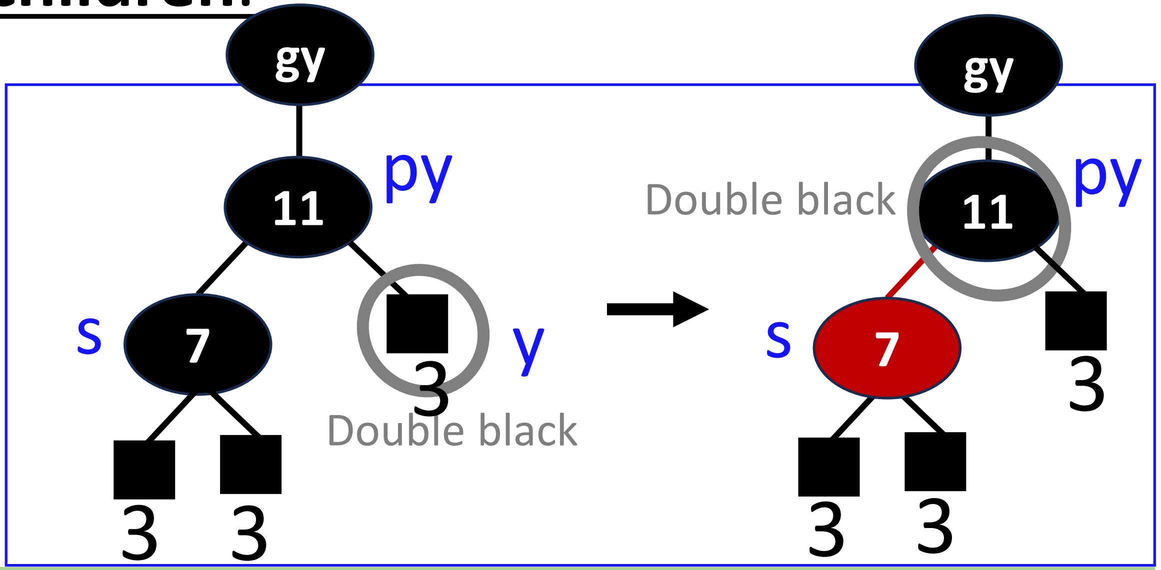
y's sibling cannot provide a data pair. Need the data pair from y's ancestor.

Type II of y2

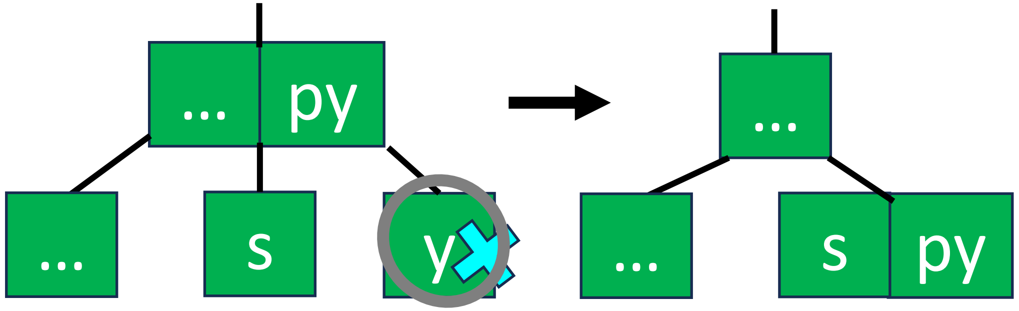
Type II: Sibling s is **black** and has **black children**.

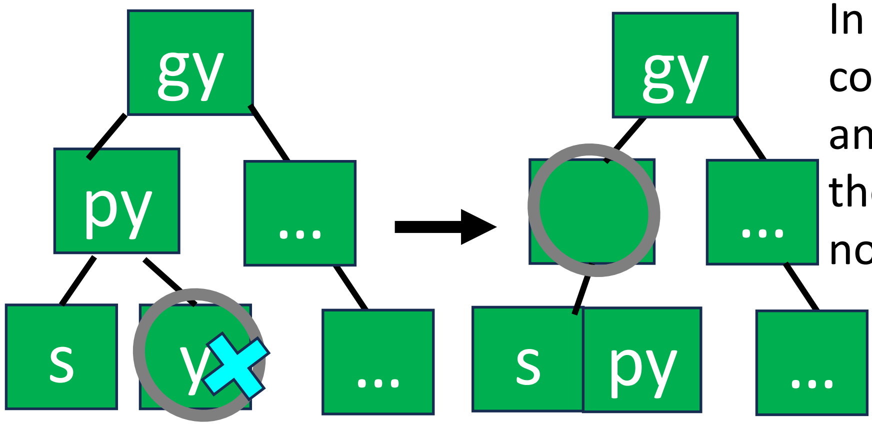


Tree height of 2-3-4 tree is not changed.



If a root becomes double black, the tree height of 2-3-4 tree reduces by one.





In 2-3-4 tree, after combining the sibling and the parent node, the original parent node becomes empty.

In 2-3-4 tree, removing y represents removing a child from y's parent. We borrow a data pair from the parent node and combine with the sibling.

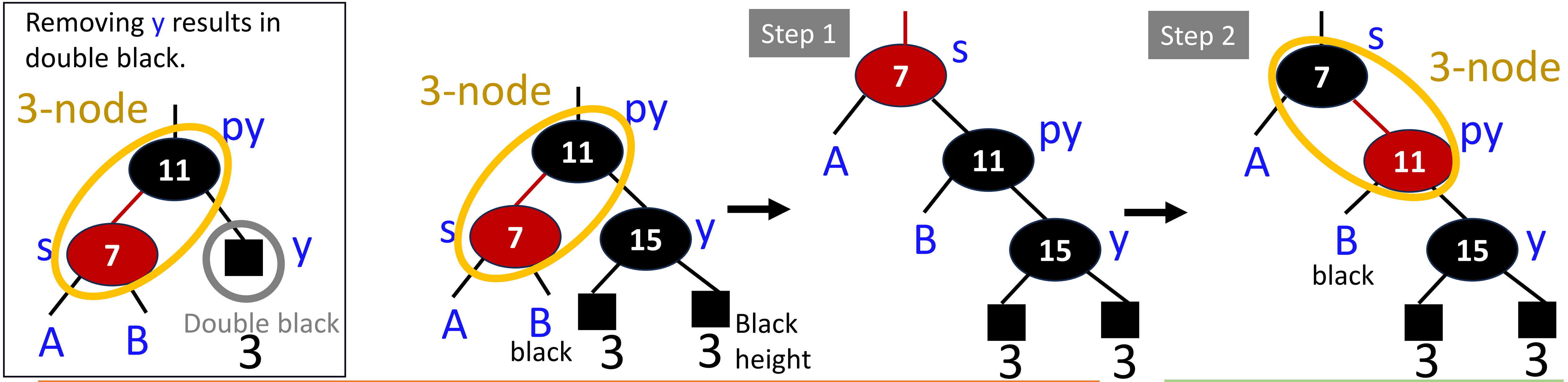
Type III of y2

Type III: Sibling s is red.

As s is **red**, py must be **black**.

1. Rotate around s and py. That is, s becomes the parent of py.
2. Color s **black** and py **red**, and repeat resolving for y.

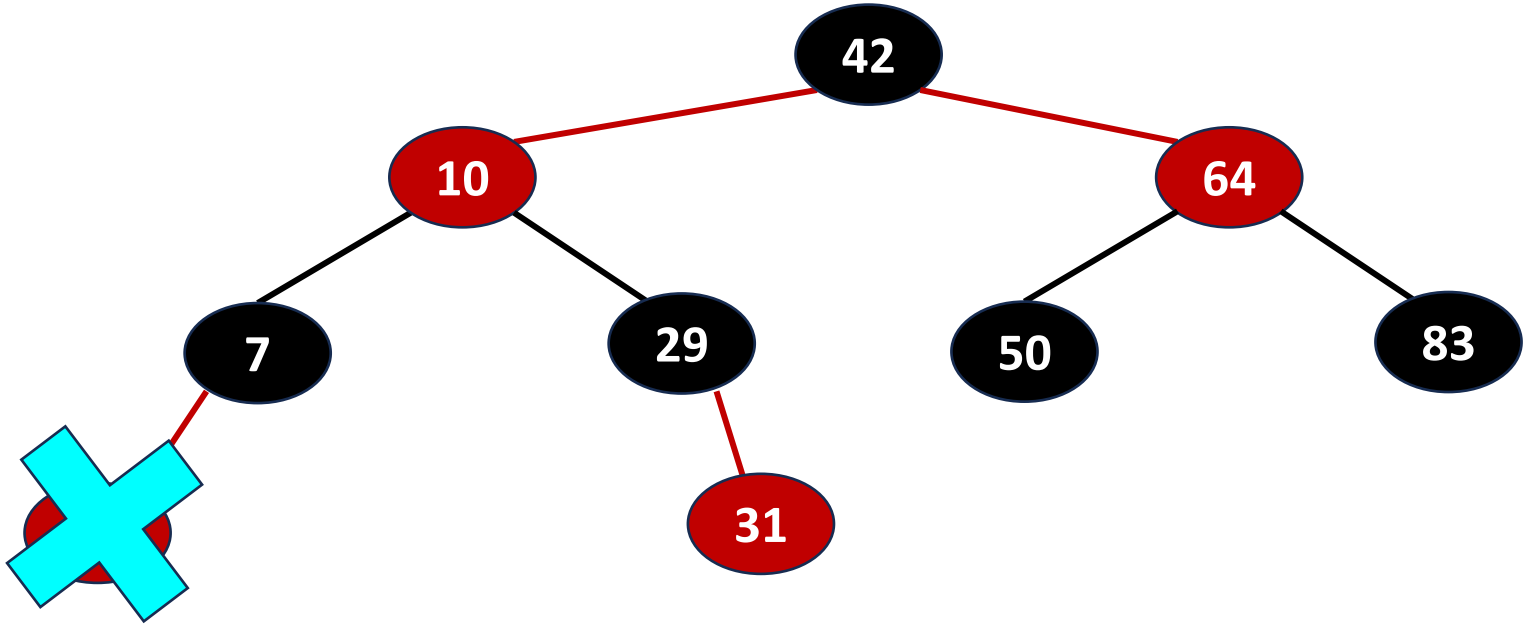
May repeat Type I and Type II of y2 (not Type III)



In red-black tree, y has a **red** sibling. Thus, we create a **black** sibling for y based on 2-3-4 tree. Then, type III of y2 is transformed to type I or II of y2.

Tree height of 2-3-4 tree may finally reduce one.

Example of deletion



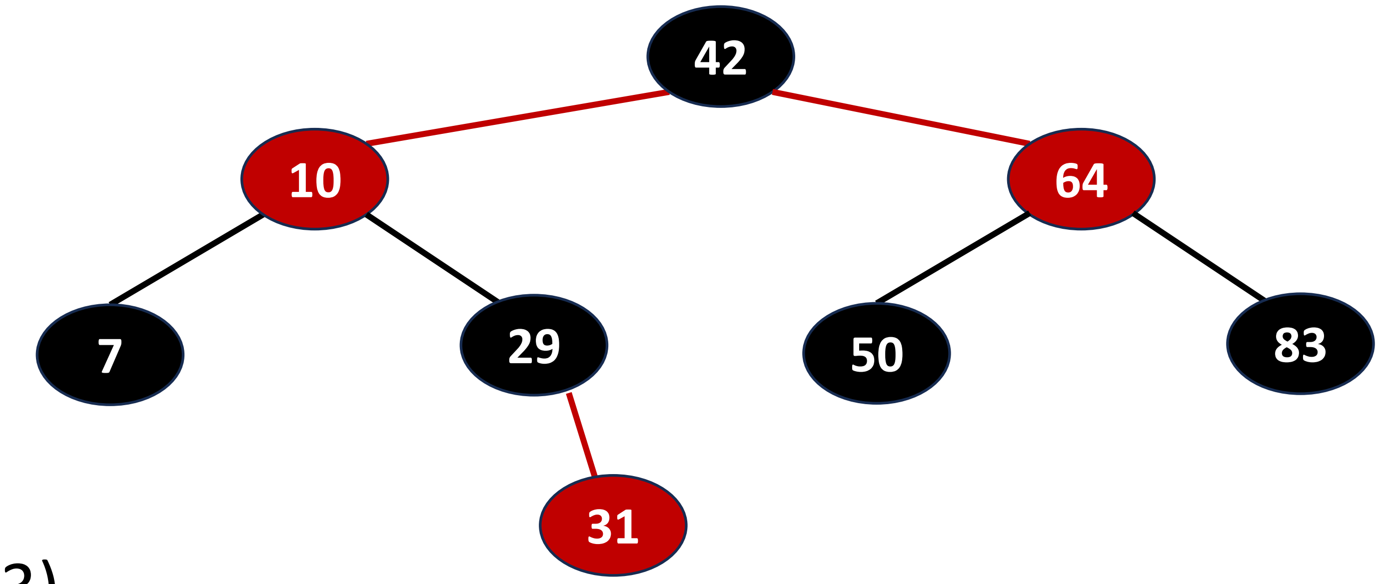
Delete 5

- A **red** leaf node

→ Case y0

→ Simply delete 5

Example of deletion

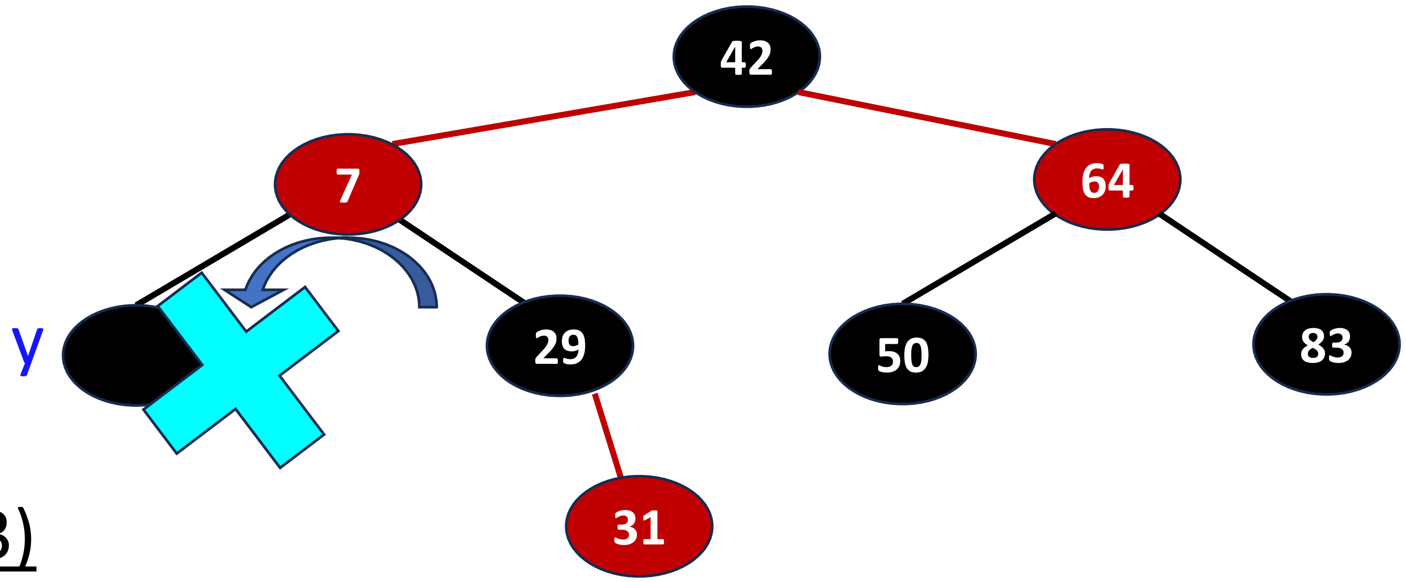


Delete 10 (1/3)

- An internal node

→ Replace with the key of inorder predecessor, that is, 7.

Example of deletion



<u>Delete 10 (2/3)</u>

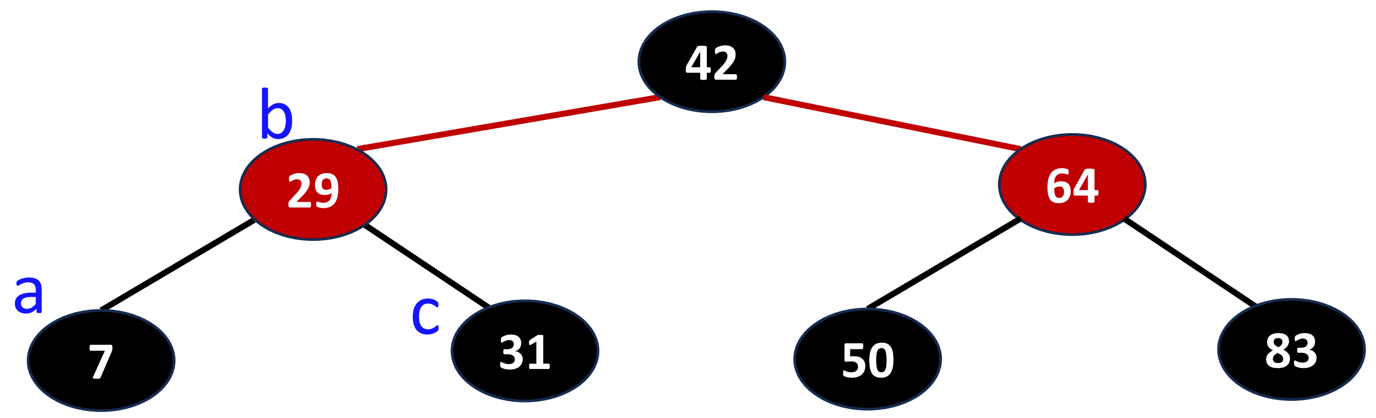
- An internal node

→ Replace with the key of inorder predecessor, that is, 7.

→ Delete the inorder predecessor y.

→ Case y2 (black node without a red child)
Type I (Sibling is black and has a red child)

Example of deletion



Delete 10 (3/3)

- An internal node

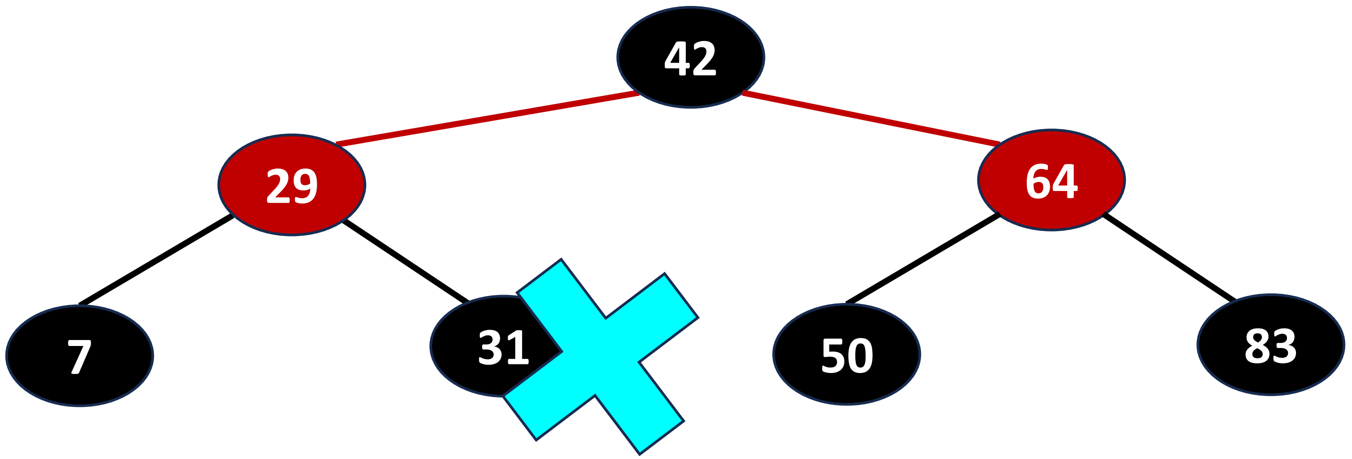
→ Replace with the key of inorder predecessor, that is, 7.

→ Delete the inorder predecessor y.

→ Case y2 (black node without a red child)
Type I (Sibling is black and has a red child)

→Rotation

Example of deletion

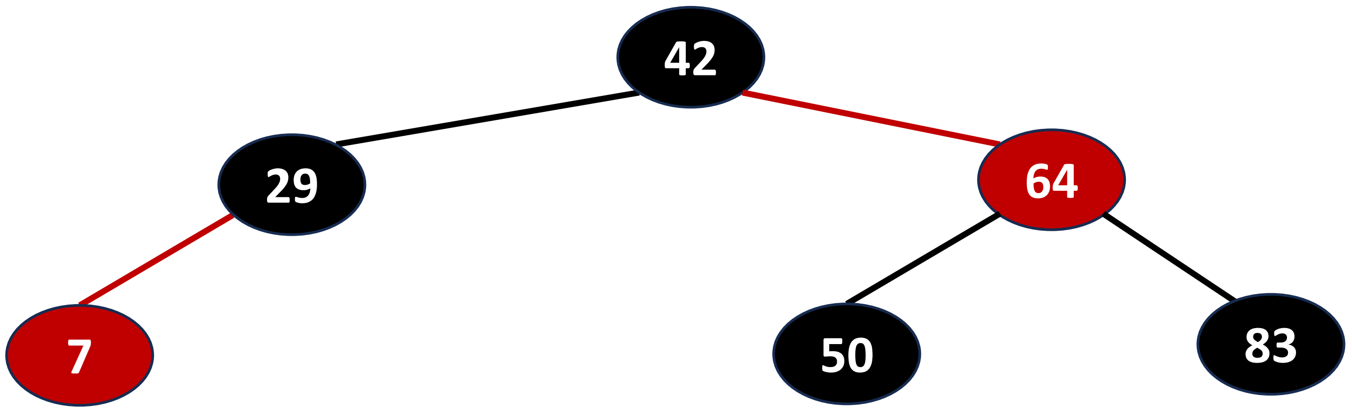


Delete 31 (1/2)

Case y2 (black node without a red child)
Type II (Sibling is **black** and has **black** children)

→ Color its sibling red and its parent black.

Example of deletion

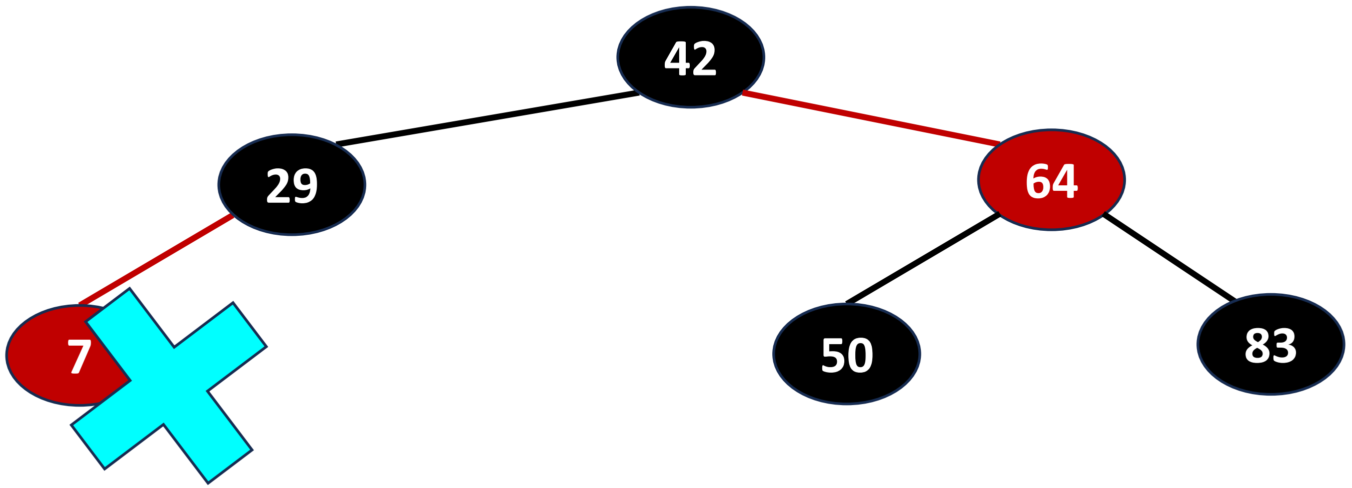


Delete 31 (2/2)

Case y2 (black node without a red child)
Type II (Sibling is **black** and has **black** children)

→ Color its sibling **red** and its parent **black**.

Example of deletion



<u>Delete 7</u>

- A **red** leaf node

→ Case y0

→ Simply delete 7

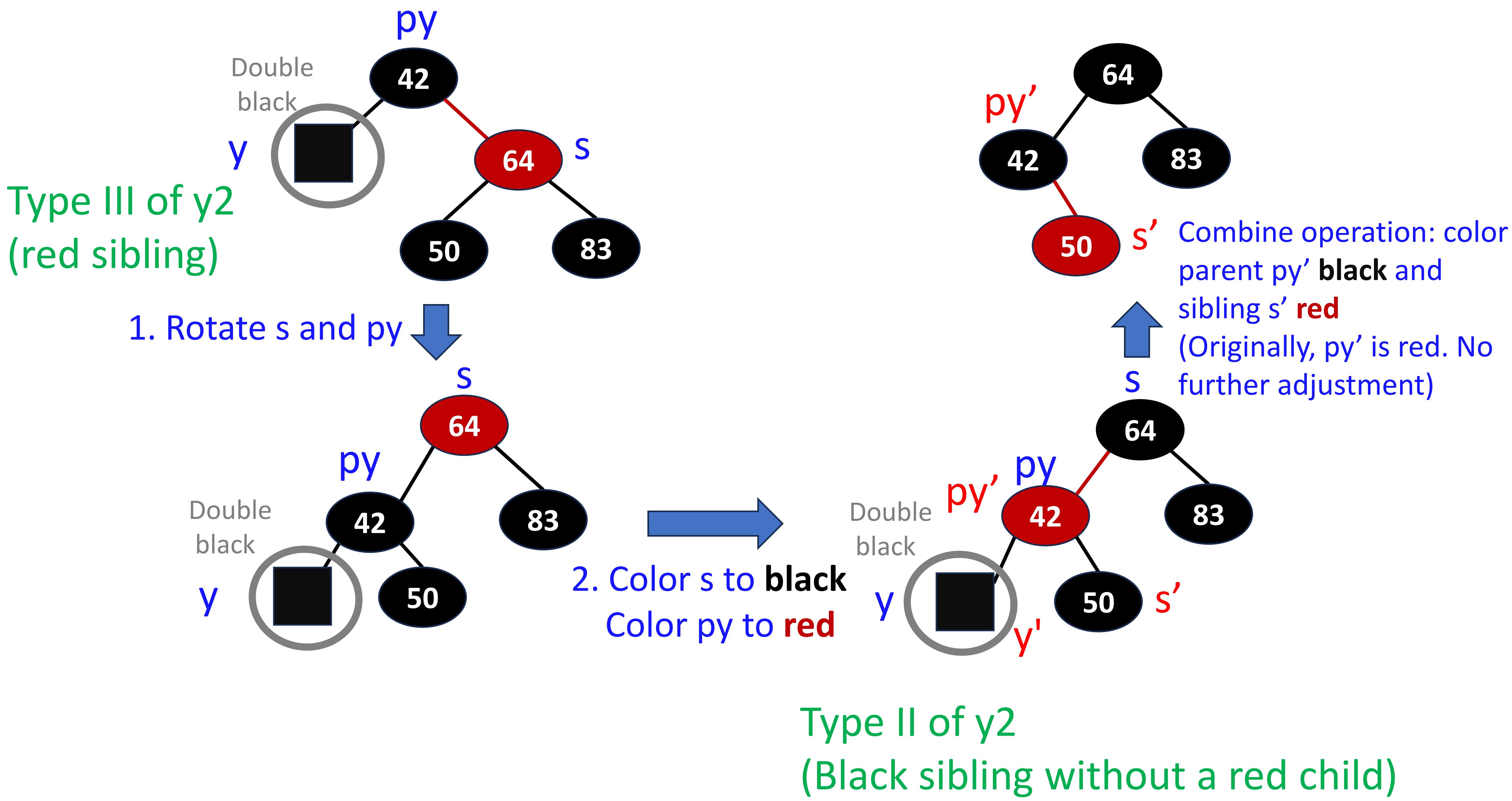
<u>Delete 29</u>

- A **black** leaf node without a **red** child.

→ Case y2.

→ Sibling is **red**.

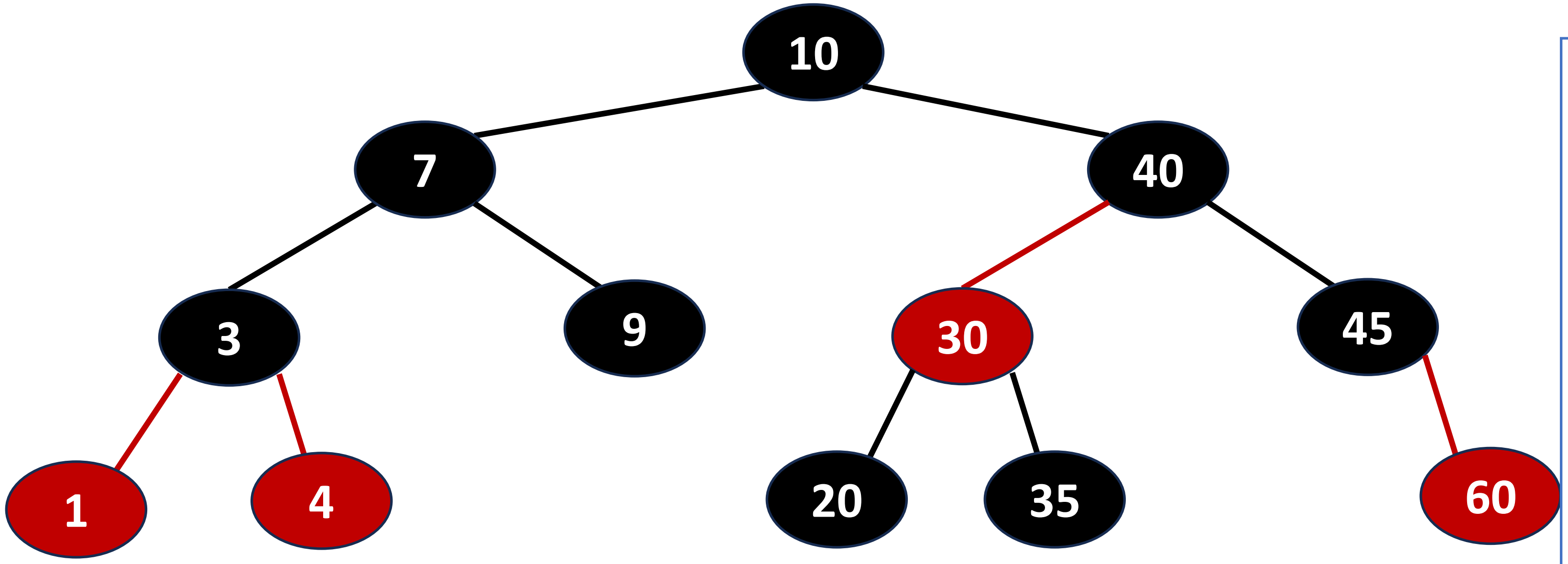
→ Type III of y2



py
Double black
42
y
64
s
50
83
Type III of y2
(red sibling)
1. Rotate s and py
s
64
py
42
83
Double black
y
50
2. Color s to black
Color py to red
64
py'
42
83
50
s'
Combine operation: color parent py' black and sibling s' red
(Originally, py' is red. No further adjustment)
s
64
py' py
42
83
Double black
y
y'
50
s'
Type II of y2
(Black sibling without a red child)

Exercise

- Given the following red-black tree.
  - Q5: Please delete 4. What is(are) the key(s) in red nodes?
  - Q6: (Continue Q5) Please delete 9. What is(are) the key(s) in red nodes?
  - Q7: (Continue Q6) Please delete 40. What is(are) the key(s) in red nodes?
  - Q8: (Continue Q7) Please delete 7. What is(are) the key(s) in red nodes?



Please reply your answers of Q5-Q8 via the following link:

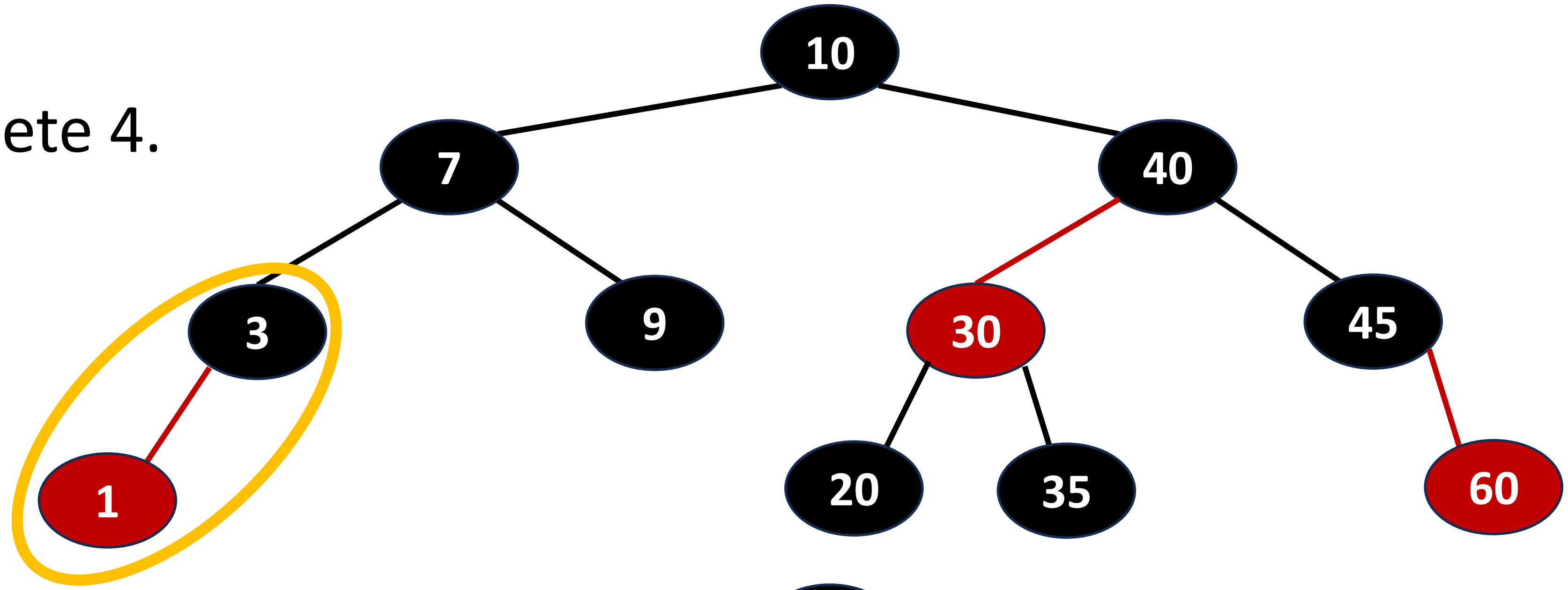
https://forms.gle/erneNgrctngmjem1A

Group members: 2~4 people

Exercise

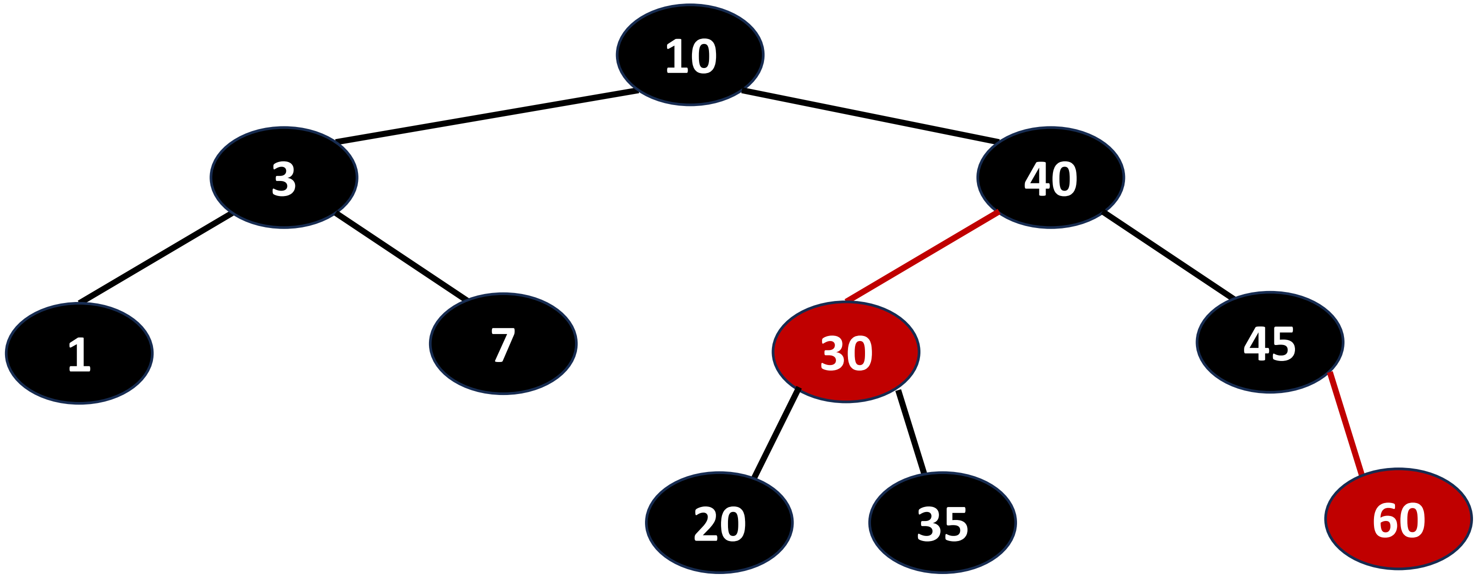
- Q5: Please delete 4.

Case y0



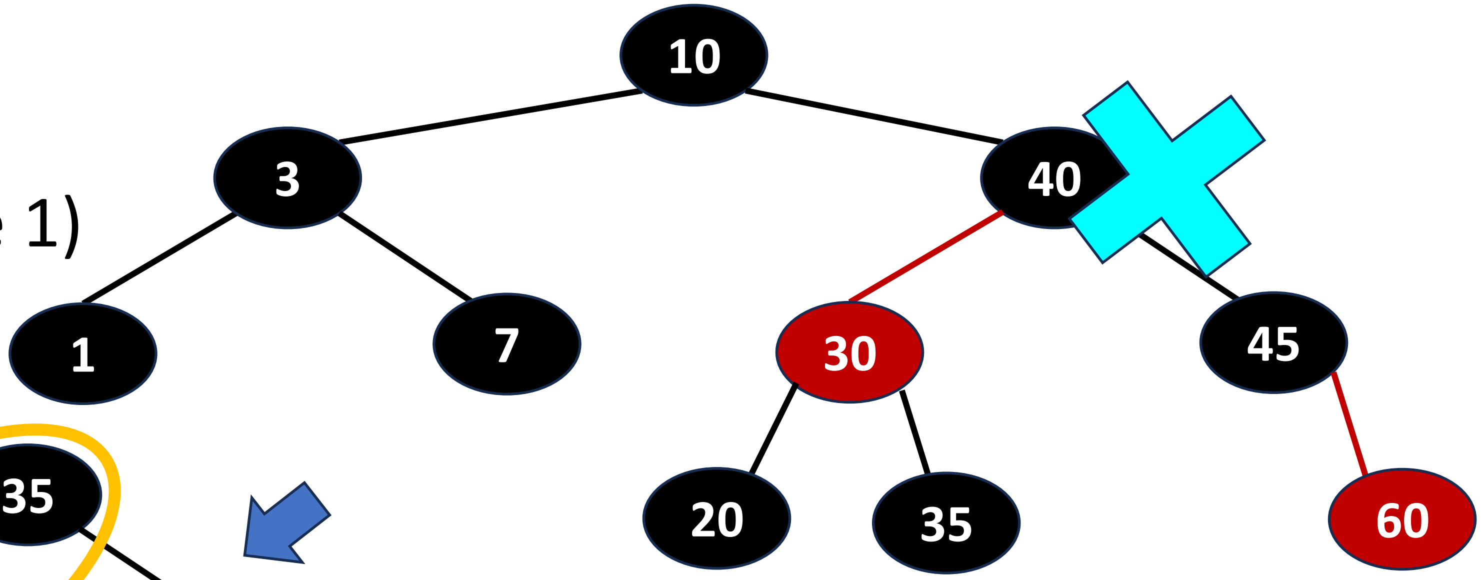
- Q6: Please delete 9.

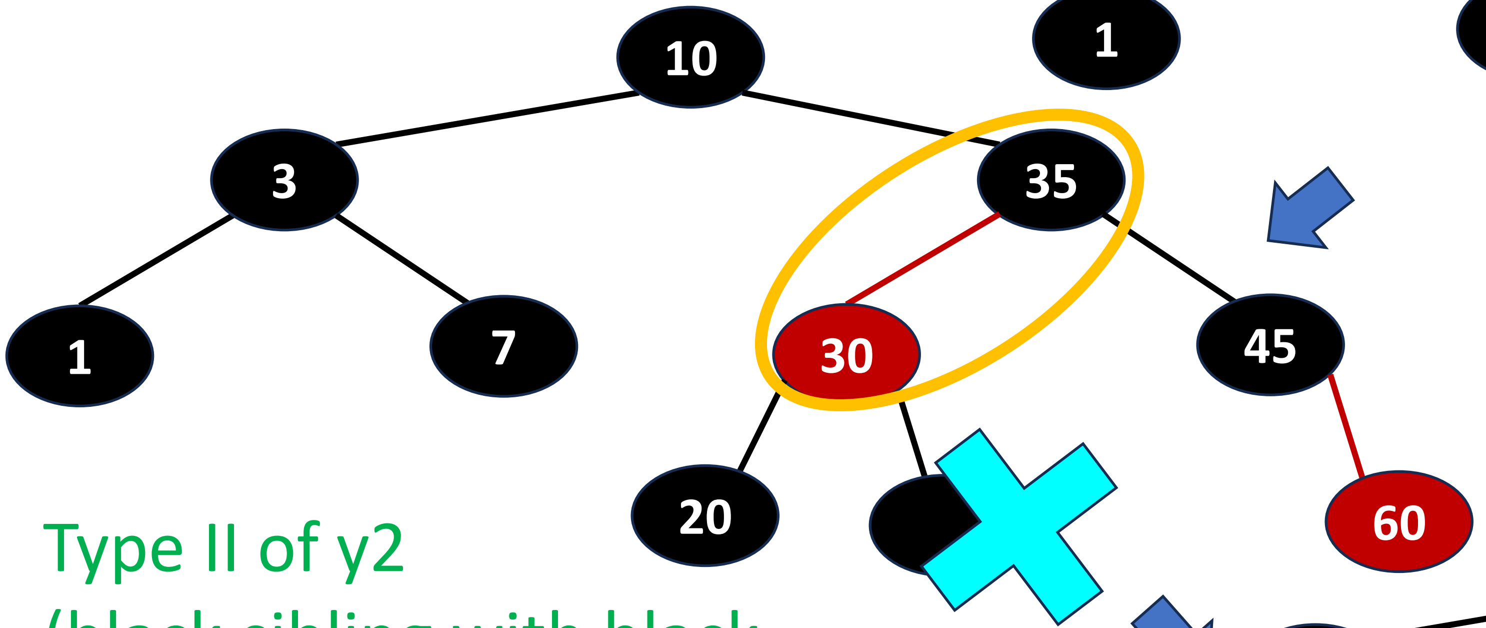
Case y2
Type I



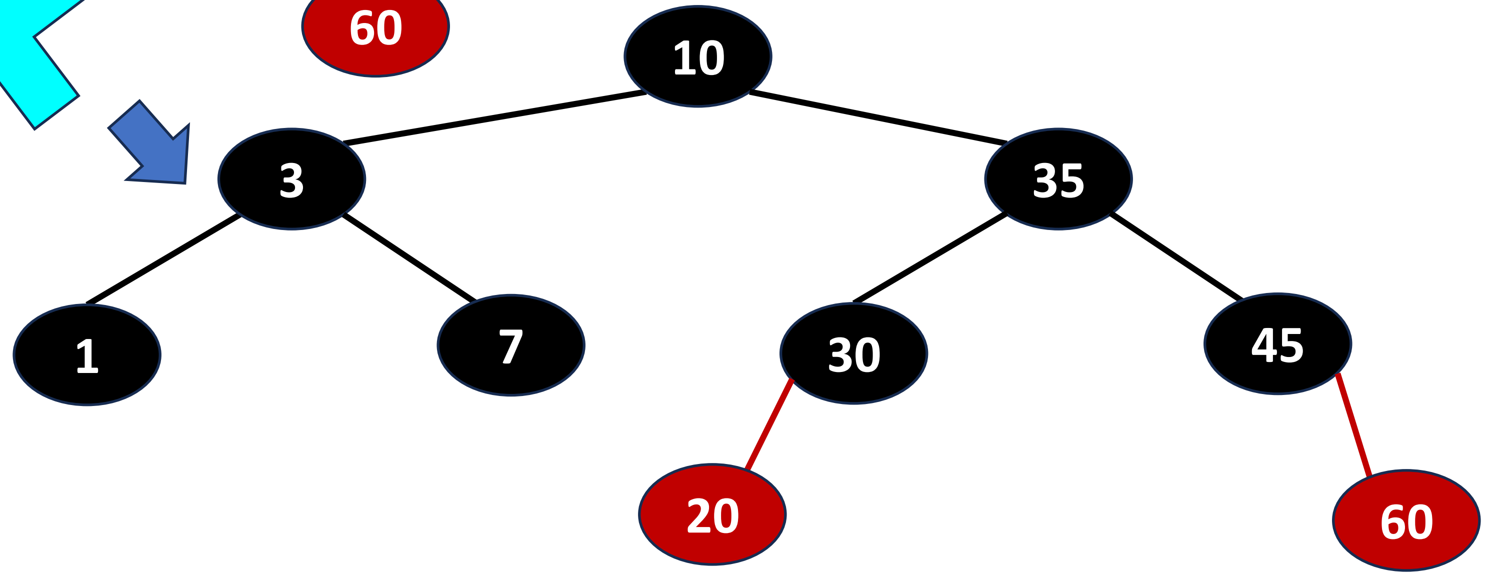
Exercise

- Q7: Please delete 40. (Case 1)



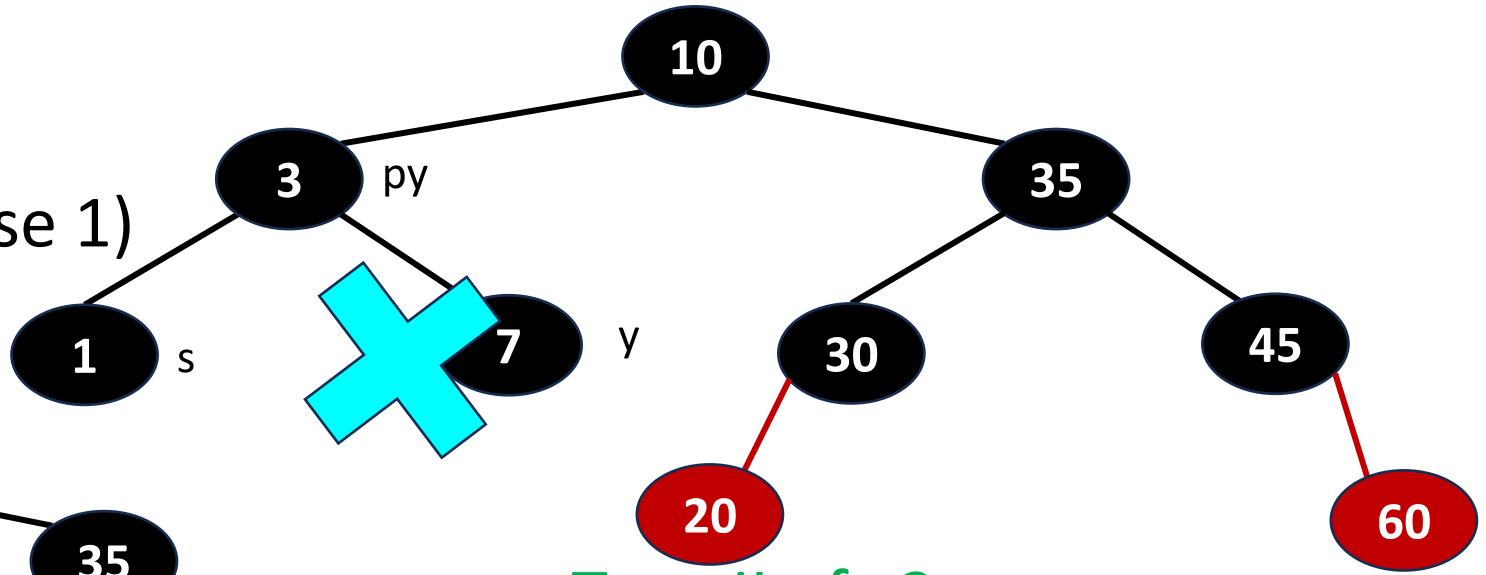


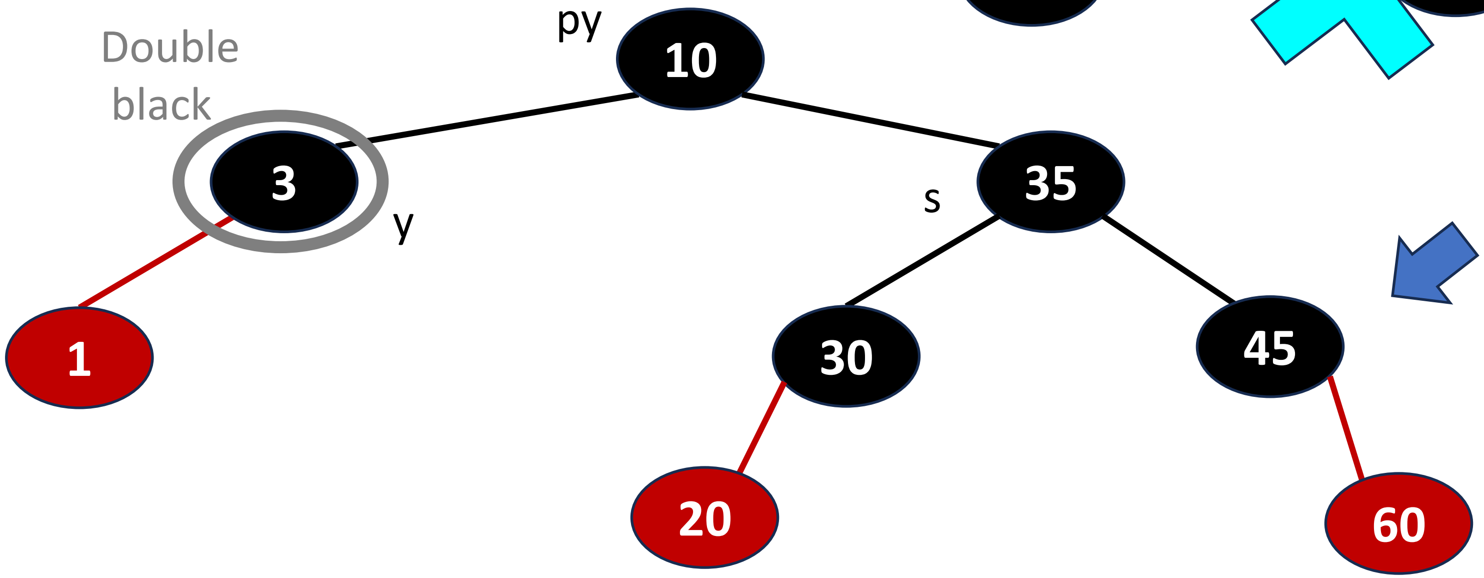
Type II of y2
(black sibling with black
children)
Parent py is **red**.→ stop



Exercise

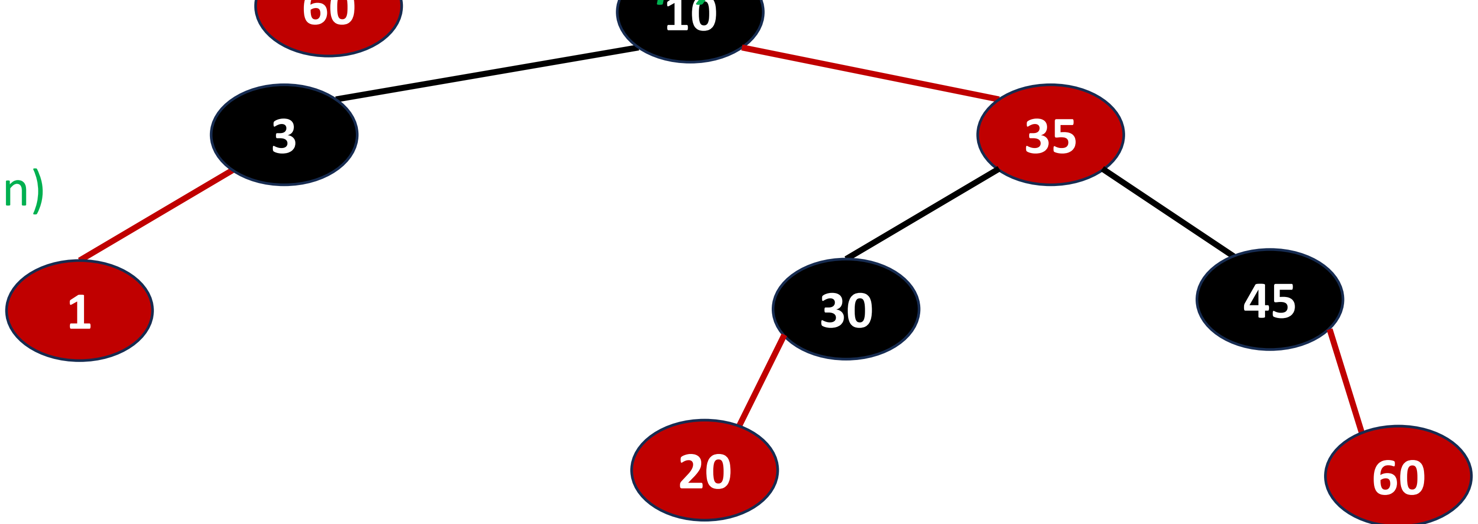
- Q8: Please delete 7. (Case 1)





Type II of y2 (black sibling with black children)

- Set *s* to be red.
- Parent *py* is <u>not</u> **red**. → up
- Set *py* to be double black

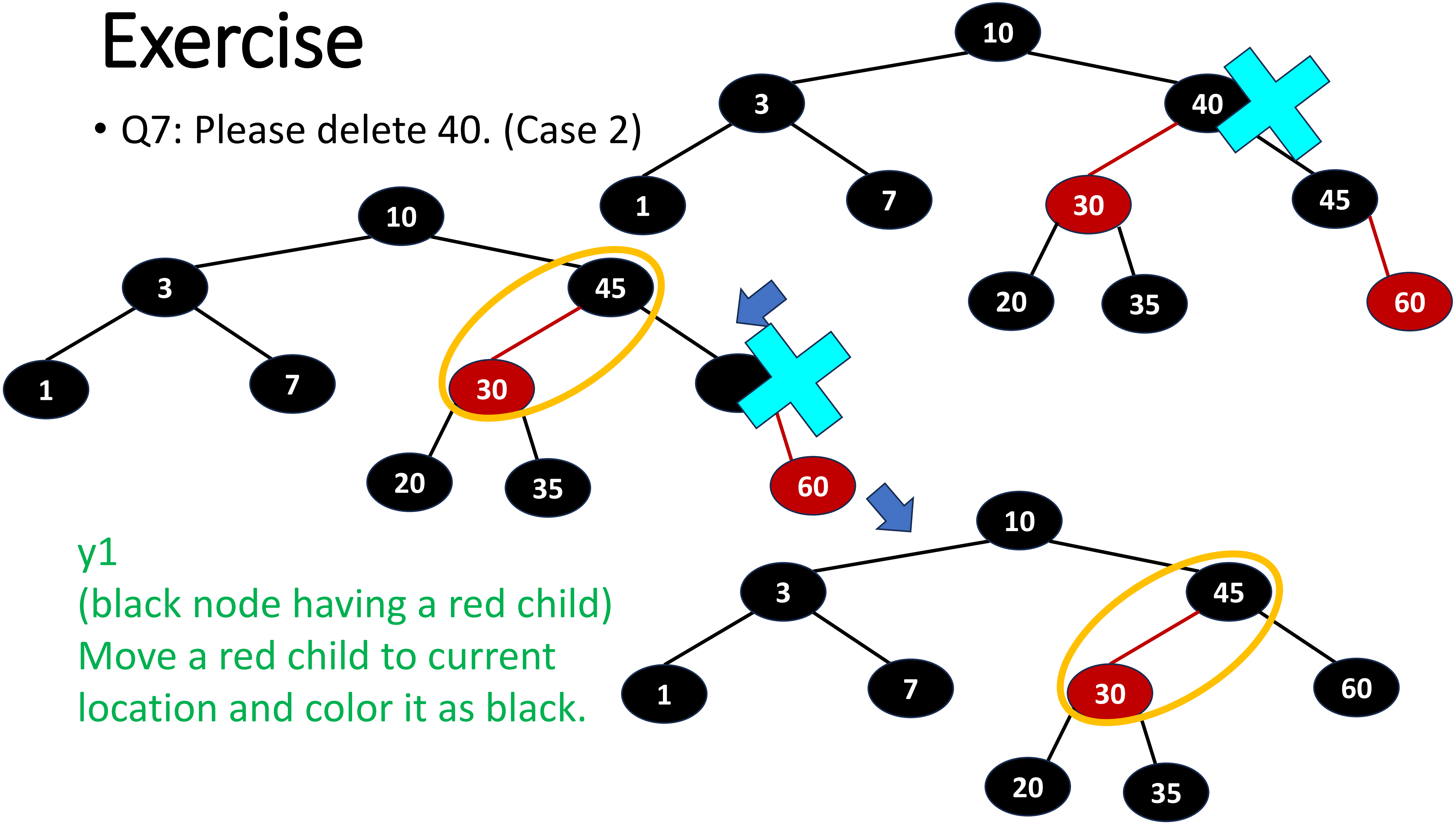


Type II of y2 (black sibling with black children)

- Set *s* to be red.
- Parent *py* is not red but it's the root.→ stop

Exercise

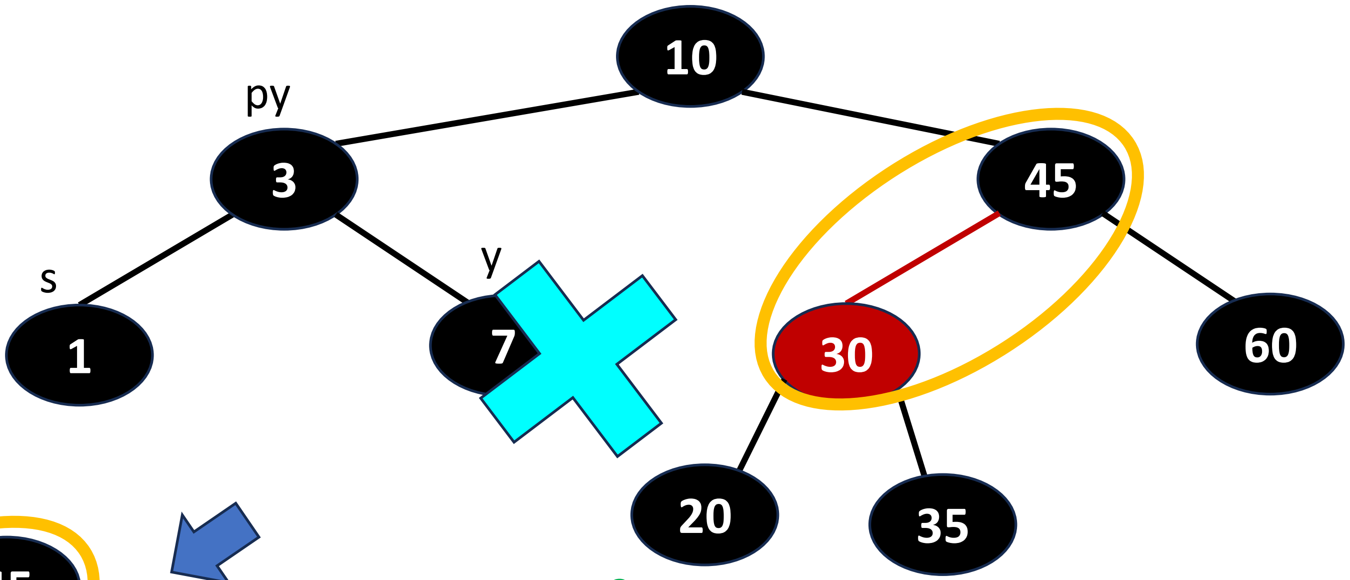
- Q7: Please delete 40. (Case 2)

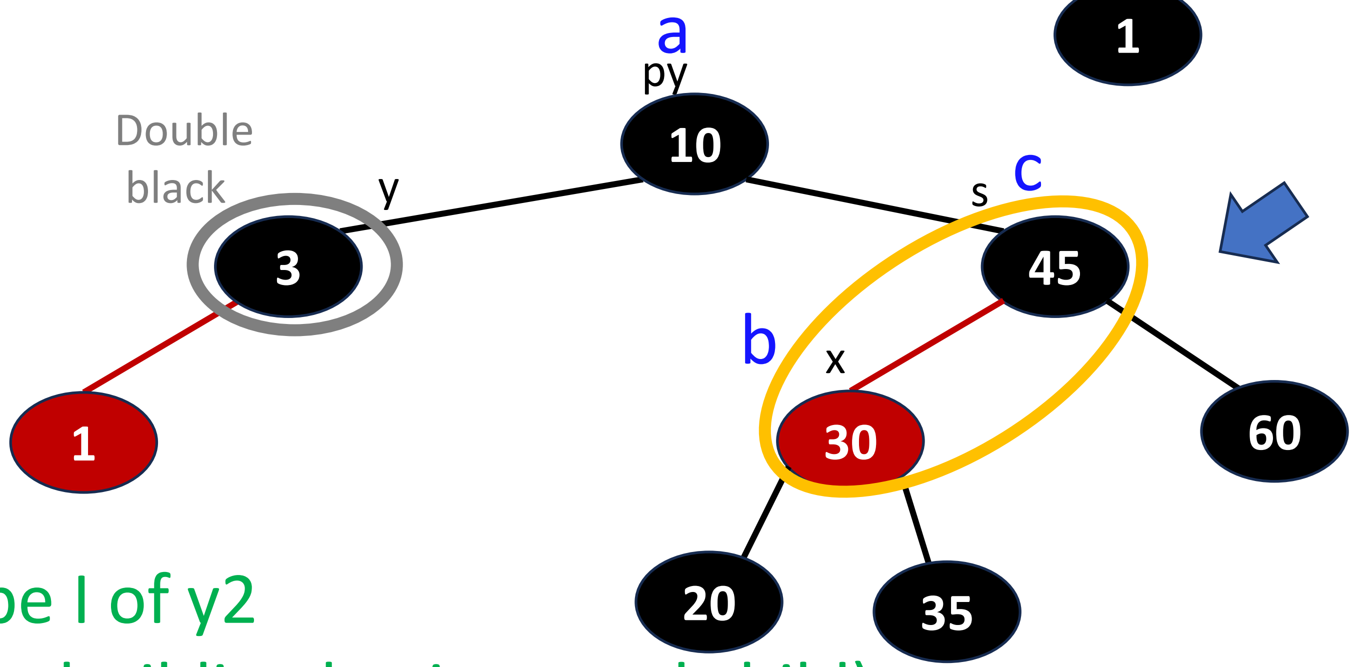


y1
(black node having a red child)
Move a red child to current location and color it as black.

Exercise

- Q7: Please delete 7. (Case 2)

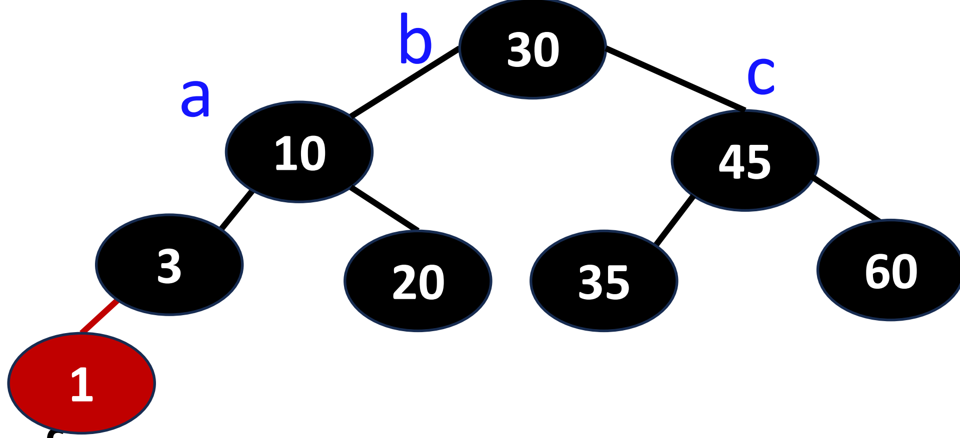




Type II of y2 (black sibling with black children)

- Set *s* to be red.
- Parent *py* is <u>not</u> **red**. → up
- Set *py* to be double black

Type I of y2
(black sibling having a red child)
Rotation of an internal node in 2-3-4 tree



1. py → a, x → b, and s → c. (RL case)
2. Replace the py with b. Make a and c its children.
3. Color a and c **black**, and color b the former color of py.

Insertion and deletion algorithms for Red-Black Trees. Are they unique?

- No!!!
- Given an 2-3-4 tree, several variant red-black trees exist

k-way search tree (*k*>4) → red-black tree

Main Concepts:

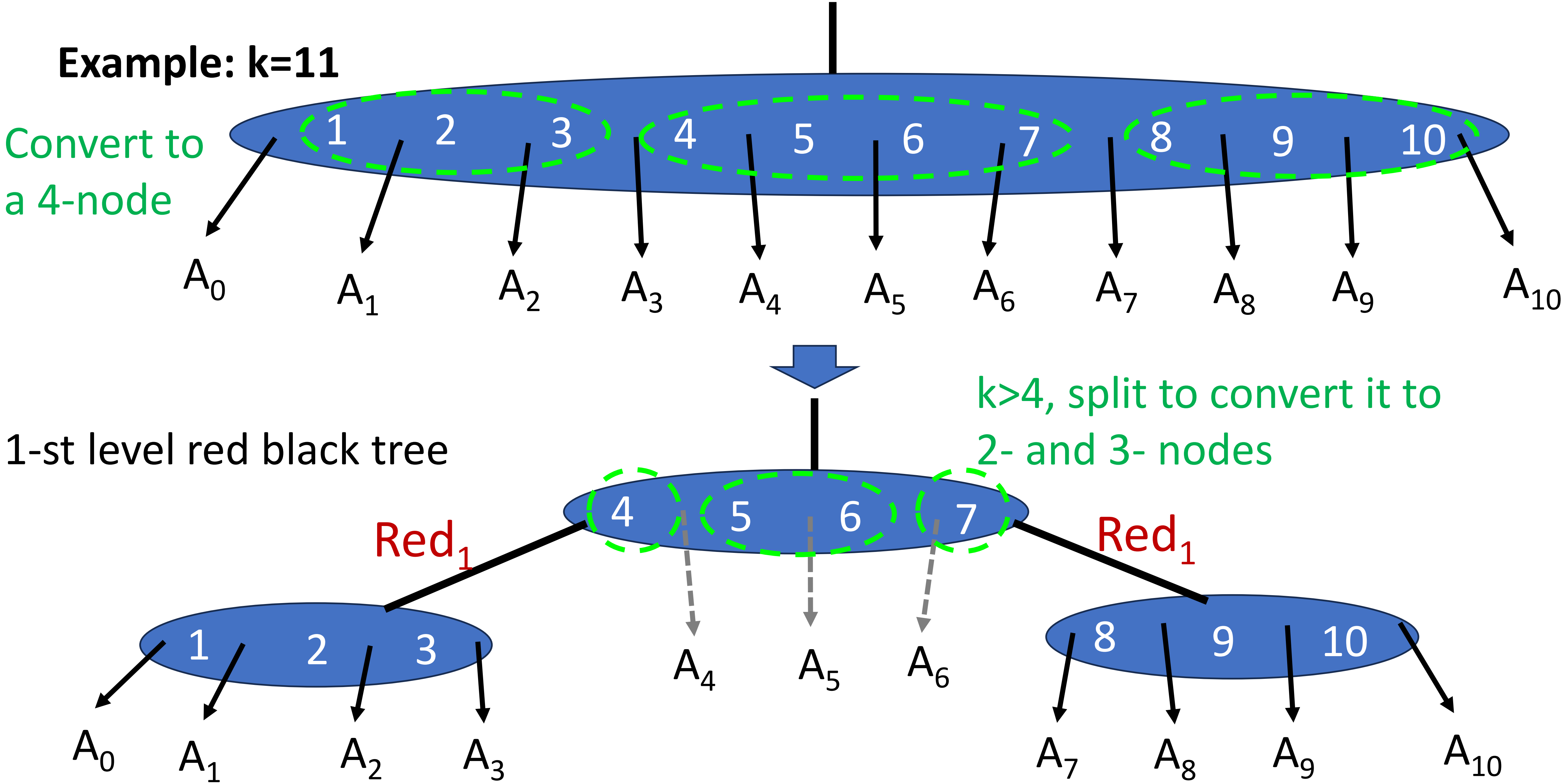
1. Transform a *k*-way search tree to 2-3-4 tree
2. Transform the 2-3-4 tree to red-black tree

Details:

1. Given a node *p* containing # of data pairs > 4
2. Split the keys in *p* into 3 groups, denoted by X, Y, and Z. Note that the # of data pairs in X, Y, or Z may be still larger than 4.
3. Consider the node *p* as a 4-node in 2-3-4 tree and convert X, Y, Z to three nodes in red-black tree.
4. Repeat Step 1 to 3 to build a level-*i* red-black tree (initially, i=1)
5. i++
6. Process the nodes containing # of data pairs larger than 4 using steps 1 to 4, and generate level-*i* red-black tree. (i=2)
7. Repeat steps 1 to 6 until the tree has only 2-, 3-, and 4-nodes. Then we convert the 2-3-4 tree to binary tree.

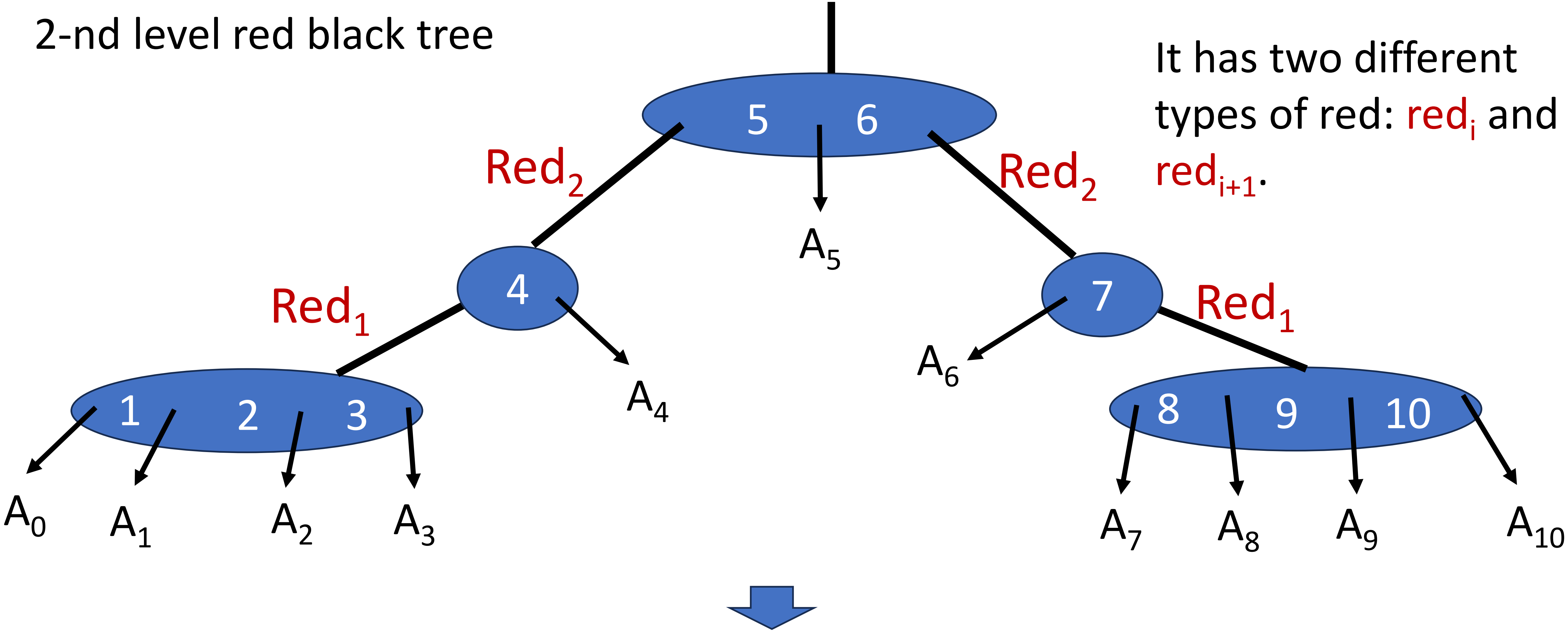
Note: The red-black tree will have more than two types of colors.

k-way search tree (*k*>4) → red-black tree



k-way search tree (k>4) → red-black tree

2-nd level red black tree



It has two different types of red: red$_i$ and red$_{i+1}$.

Convert to red-black tree using the algorithm in textbook (2-3-4 tree to red-black tree)

Summary

- Red-black tree
- The relationship to 2-3-4 tree
- Operations:
  - Insertion
  - Deletion