The rebalancing process of red-black trees

Yuanming Gao
yuanming.gao@gmail.com

Agenda

- Definition
- Some conventions
- Insert a new node and then rebalance the tree
- Remove a node and then rebalance the tree

Note

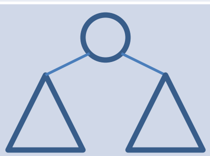
- Make sure that you have read the chapter 12 of the book 《Introduction to Algorithm》(third edition) before you continue reading this article;
- I do not talk about how to insert/remove a node into/from a binary tree, you may find details in the foregoing chapter（12.3 Insertion and deletion）;
- The purpose of the article is to give more logic to the rebalancing process to make it more comprehensible.

Definition

- A Red-black tree is a binary search tree with the following traits:
  1. Each node is either red or black;
  2. If a node is red, then both its children are black;
  3. Every path from a given node to any of its descendant NIL nodes goes through the same number of black nodes;
  4. The root is black;
  5. All leaves (NIL) are black.

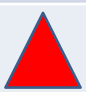
Some conventions

- A binary tree is a recursive data structure, we use a triangle to represent it and a circle to represent a node:

Figure	Meaning
	A node in a binary tree.
	A binary tree, sub-tree, or empty tree. Note: it can be used to represent a tree or sub-tree in which there is only one node.
	A binary tree or sub-tree which has a root node and two sub-trees (it is a non-empty tree).

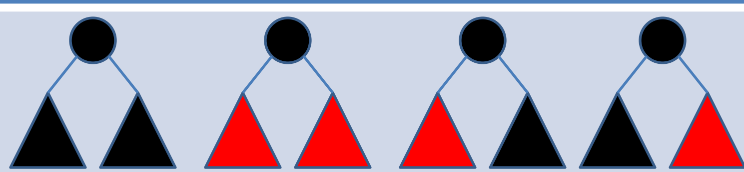
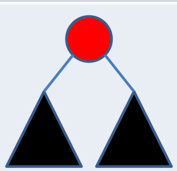
Some conventions (continue…)

- Each node in a Red-Black tree is either red or black, and if a node is red, then both its children are black, so:

Figure	Meaning
	A black node in a red-black tree.
	A red node in a red-black tree.
	A red-black tree or sub-tree whose root node is black, or an empty red-black tree.
	A red-black sub-tree whose root node is red.

Some conventions (continue…)

- At the previous abstraction level, we say there are only two different red-black sub-trees;
- But if we unfold them a bit, we can get five different red-black sub-trees (empty red-black trees or sub-trees are excluded because if we can unfold them, they must not be empty):

Figure	Meaning
	A red-black tree or sub-tree whose root node is black.
	A red-black sub-tree whose root node is red.

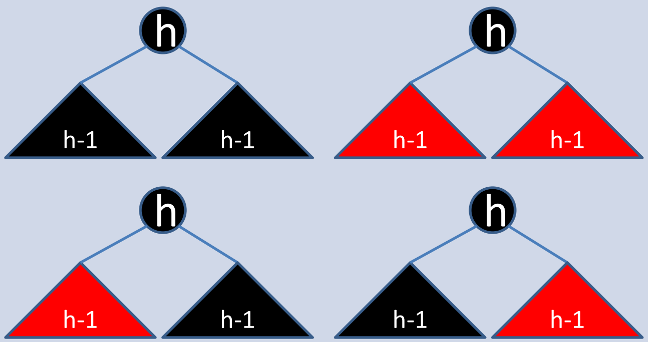
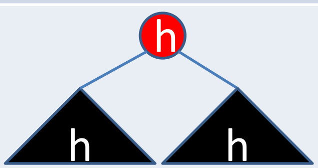
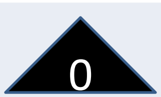
Some conventions (continue…)

- Every path from a given node to any of its descendant NIL nodes goes through the same number of black nodes. That is the **black depth** of a node, so:

Figure	Meaning
	A red-black tree or sub-tree whose root node is black and the black depth of the root node is h (h >= 0). When h > 0, the root node is a normal node. When h == 0, it is an empty red-black tree or sub-tree. We say there is only a NIL node in it.
	A red-black sub-tree whose root node is red and the black depth of the root node is h (h >= 0, if h == 0, there is only a red node in it).

Some conventions (continue…)

- The following six figures can represent all different red-black trees, sub-trees or empty trees:

Figure	Meaning
	A red-black tree or sub-tree whose root node is black, the black depth of the root node is h, and the black depth of its two child nodes is h-1 (h-1 >= 0).
	A red-black sub-tree whose root node is red, the black depth of the root node is h (h >= 0), and the black depth of its two child nodes is h too.
	An empty red-black tree or sub-tree.

Some conventions (continue…)

- If the black depth of a node X is h, we say the black depth of the corresponding sub-tree (tree or empty tree) whose root node is the node X is h too;
- The root of a red-black is black, so:

Figure	Meaning
	The figure can be used to represent a red-black tree, sub-tree or empty tree.
	The figure can be used to represent a red-black sub-tree (or in some intermediate states, the original root node of a red-black tree is replaced with a red node).

Some conventions (continue…)

- How to represent a NIL node:

Figure	Meaning
	A NIL node, whose black depth is 0. Usually they are omitted in our pictures.
	An empty red-black tree, we can say there is only a NIL node in it. Usually they are omitted in our pictures.

- A NIL node is actually a null pointer in our program.

Some conventions (continue…)

- Dyeing a normal black node red will decrease its black depth from h to h-1;
- Dyeing a normal black node red is a dangerous operation, because it may break the rule 2;
- We must make sure that both of the children of the node are black and then we can dye it red. The new tree (or sub-tree) where the node is the root node is regular, but we must continue to check whether its parent (if it exists) is red;

Some conventions (continue…)

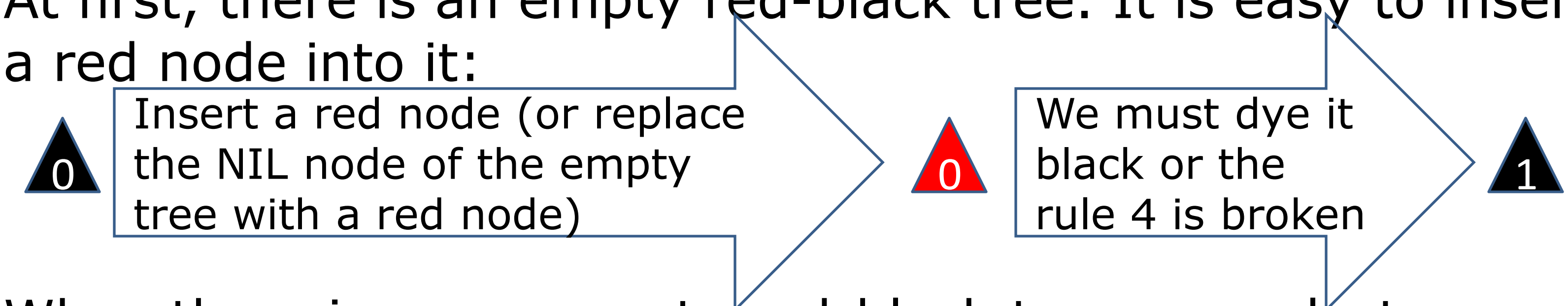
- Dyeing a red node black will increase its black depth from h to h+1;
- Dyeing a red node black is also a dangerous operation because the rule 3 may be broken;
- The new tree (or sub-tree) where the node is the root node is regular, but if the node has parent, it is obvious that the rule 3 is broken for those ancestor nodes;

Some conventions (continue…)

- If we select a NIL node and replace it with a new red node (its black depth is 0), we can find at the very position we replace the root node 0 of the empty sub-tree with a red node 0;
- Naming convention: if we name a tree (sub-tree or empty tree) **X** (or XL, XR, …, etc), usually we name its root node **X** too. The reverse is also the same.

Insert and then rebalance

- We always insert a red node into a red-black tree (or **we replace the NIL node with a red node in an empty sub-tree**).
- At first, there is an empty red-black tree. It is easy to insert a red node into it:



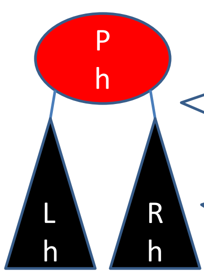
- When there is a non-empty red-black tree, we select an empty sub-tree and then replace the NIL node of the sub-tree with a red node. We call the root node of the very sub-tree **the node N** (now it is red but before that it was black). if N's parent is black, the process is finished; if N's parent is red, the rule 2 is broken (note: the rule 2 is only broken in the upper level sub-tree where N's parent is the root node and N is a child sub-tree. No other rules are broken).

Insert and then rebalance (continue…)

- We say the foregoing operation is the base case of a recursive process (will give more details later);
- Generally speaking, given a sub-tree whose root node's color has been changed from black to red (**by dyeing it red or replacing it with a red node**), we call the root node of the very sub-tree the node **N**. If N's parent is black, the inserting and then rebalancing process is finished; if N's parent is red, the rule 2 is broken (note: the rule 2 is only broken in the upper level sub-tree where N's parent is the root node and N is a child sub-tree. No other rules are broken);
- The foregoing operation may generate irregular sub-trees, we need to rebalance them.

Insert and then rebalance (continue…)

- First we need to know how many such irregular sub-trees could be generated from the color change;
- Because of the precondition: the color of the root node of a sub-tree is changed from black to red and its parent is red, we only need to consider this sub-tree:

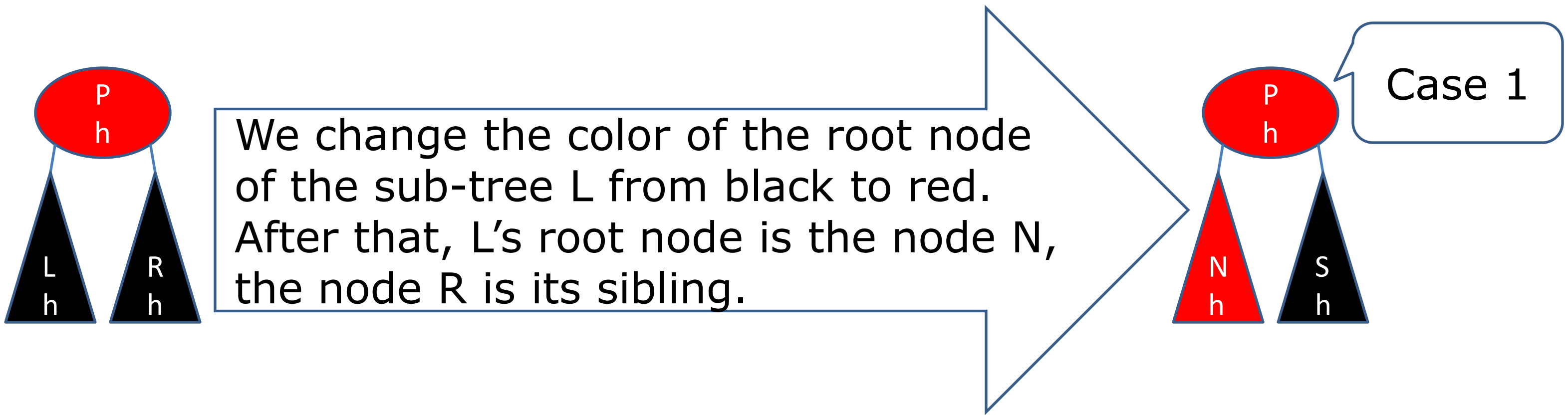


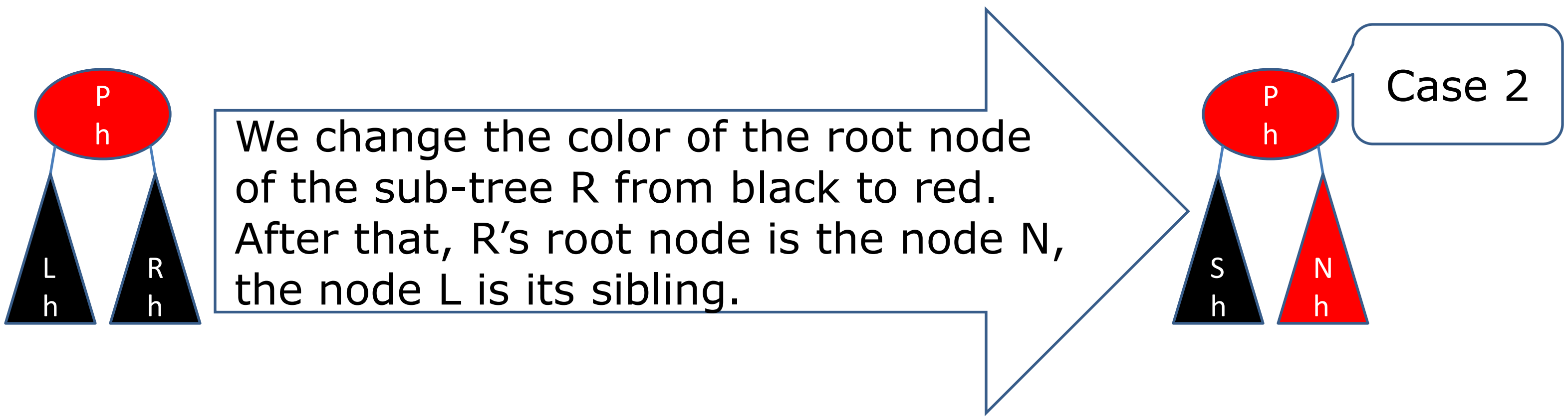
The letters P, L, R are the name of the nodes (L and R are the name of the root node of the two child sub-trees respectively). **h** is their black depth. (Note: h >= 0)

The letter L and R are the name of sub-trees too. When we say the sub-tree P, it includes the node P and the two child sub-trees L and R.

- After the color change is finished, we get two irregular sub-trees to rebalance:

Insert and then rebalance
(continue…)





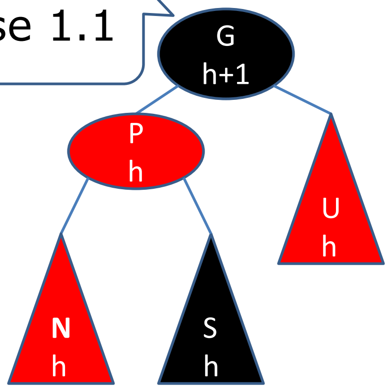
Insert and then rebalance (continue…)

- We do not know how to rebalance the following irregular sub-trees:

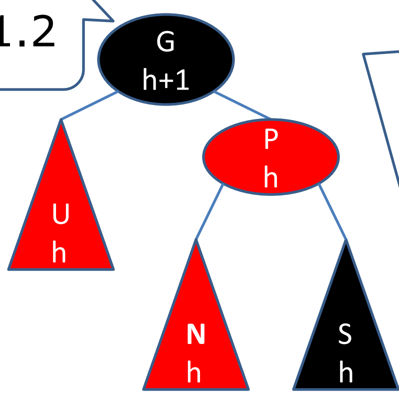
- If we add the grandparent and the uncle of the node N into the pictures, we get eight irregular sub-trees.

Insert and then rebalance (continue…)

Case 1.1



Case 1.2

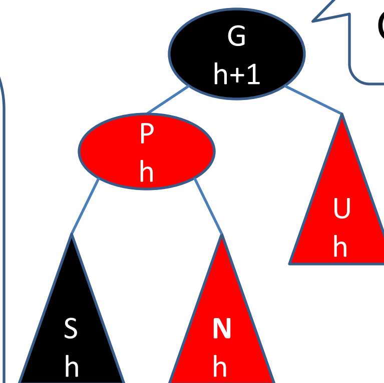


The letters G, P, U, N, S are the name of the nodes or sub-trees, the expression (h[+|-]*number*) below is their black depth.

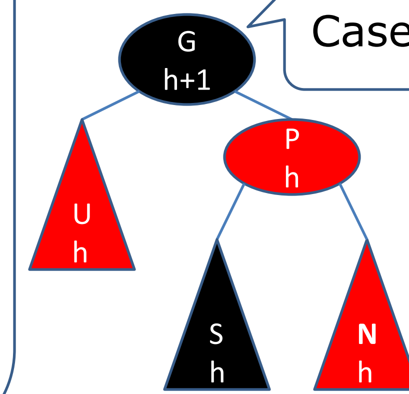
1. **G**: grandparent
2. **P**: parent
3. **U**: uncle
4. **N**: new node (whose color has been changed from black to red)
5. **S**: sibling

(Note: h >= 0)

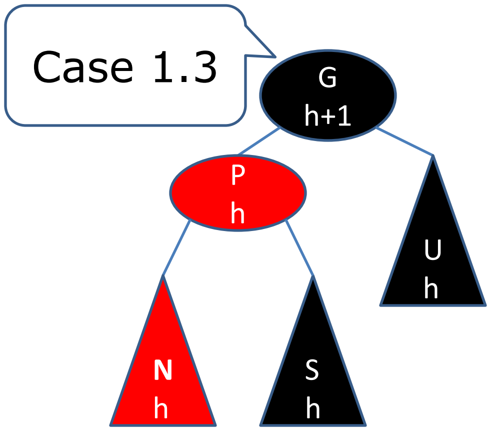
Case 2.1

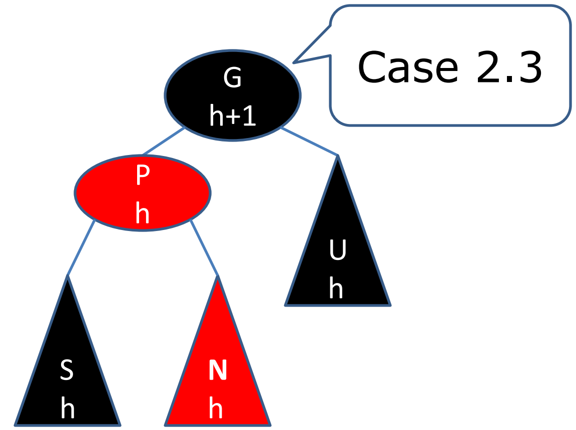


Case 2.2

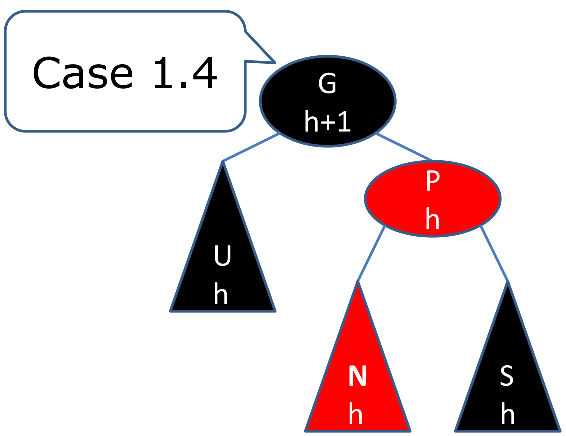


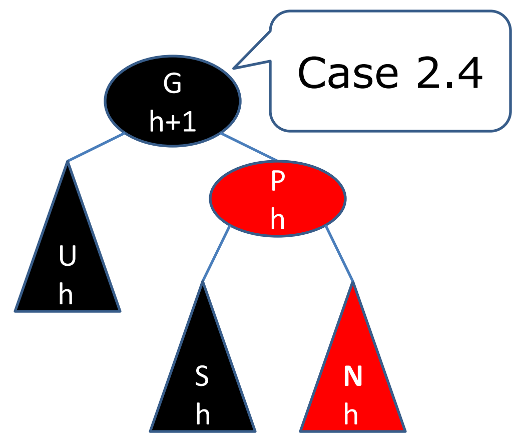
Insert and then rebalance
(continue…)





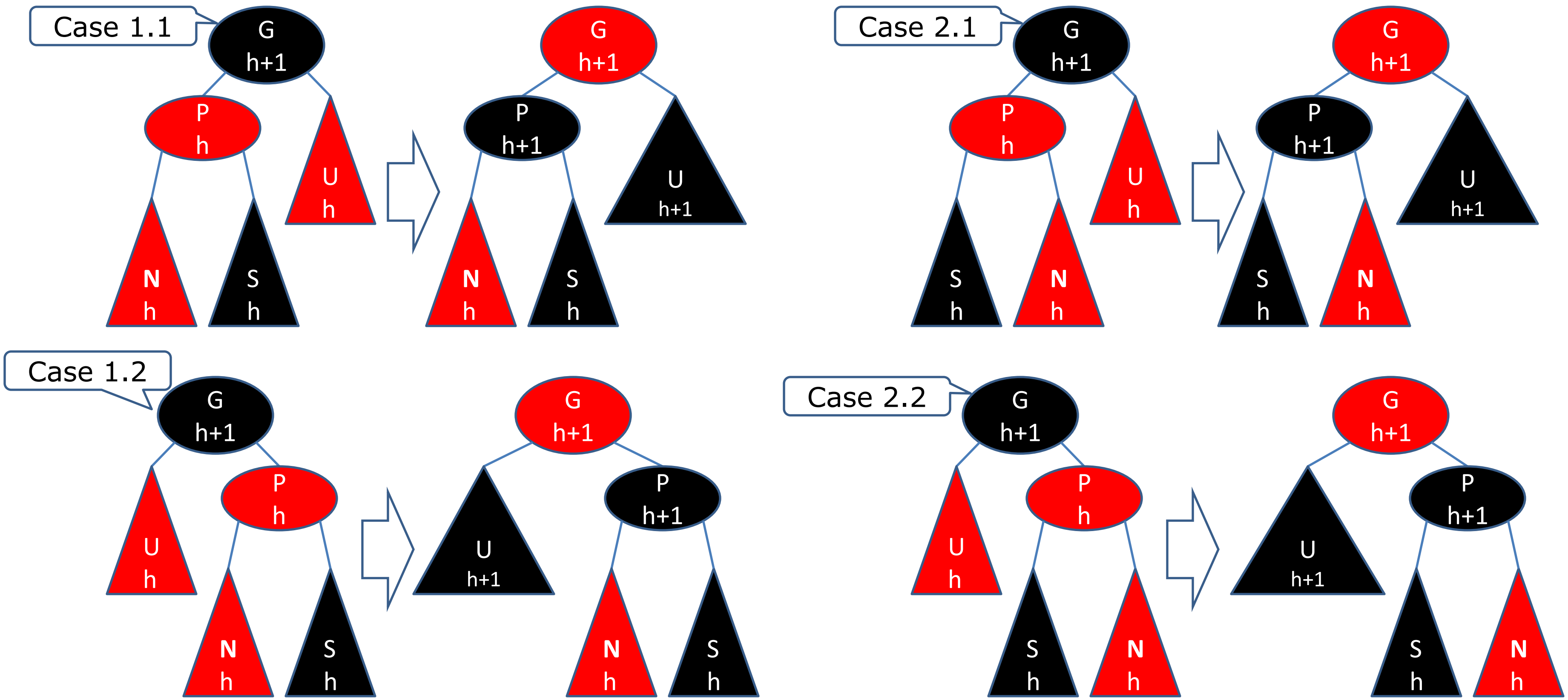
Note: h >= 0





Insert and then rebalance (continue…)

- The method to rebalance the sub-trees 1.1, 1.2, 2.1, 2.2 is as below:

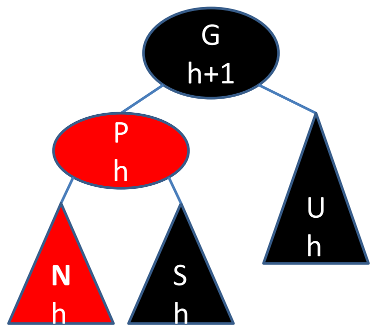


Insert and then rebalance (continue…)

- We dye the nodes P, U black and dye the node G red, all five rules are kept in all the sub-trees;
- But the root node G of the sub-trees is changed from black to red, we need to check:
  - Whether the node is the root node of the whole tree, if it is, the rule 4 is broken;
  - Whether the node G’s parent is red, if it is, the rule 2 is broken in the upper lever sub-tree.
- Apparently, the process is recursive.

Insert and then rebalance (continue…)

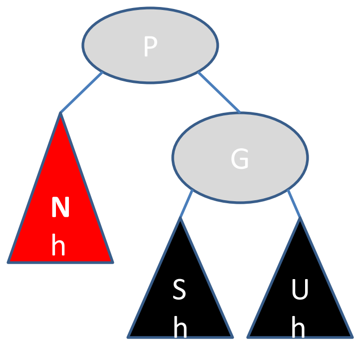
- The method to rebalance the sub-tree 1.3 is as below (note: h >= 0):



- The term "rotation" is used to describe the method to rebalance such a sub-tree, but I think we should treat it as a jigsaw puzzle: we have five pieces, how do we use them to rebuild a regular red-black sub-tree?

Insert and then rebalance
(continue…)

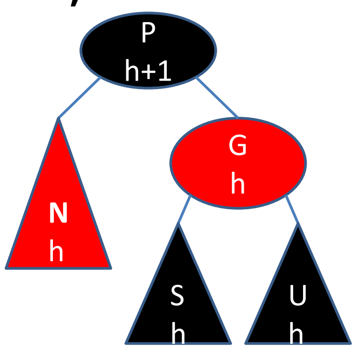
- First we can create a regular binary search tree like this:



- The binary search tree has the following traits:
  - all the sub-trees N, S and U are regular red-black sub-trees (no rule is broken in them);
  - the black depth of all the sub-trees N, S and U is h;
  - until now the color of the nodes P and G is not determined.

Insert and then rebalance (continue…)

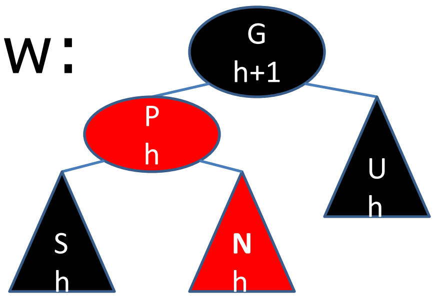
- Before we dyed the node N red, the black depth of the root node G of the sub-tree was h+1, that means that we should let the P's black depth be h+1, so we can color the nodes P and G in this way:



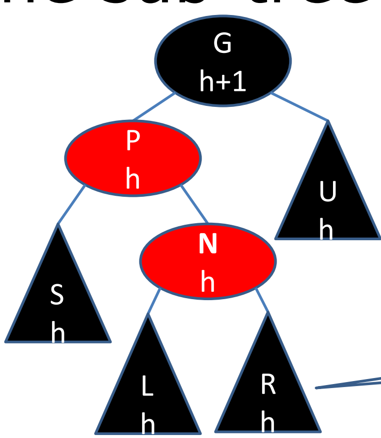
- Then we use the five pieces to rebuild a new red-black sub-tree, the black depth of the root node of the new sub-tree is still h+1, the root node is black, no rule is broken. So the rebalancing process is finished for the case.

Insert and then rebalance (continue…)

- The method to rebalance the sub-tree 2.3 is as below:



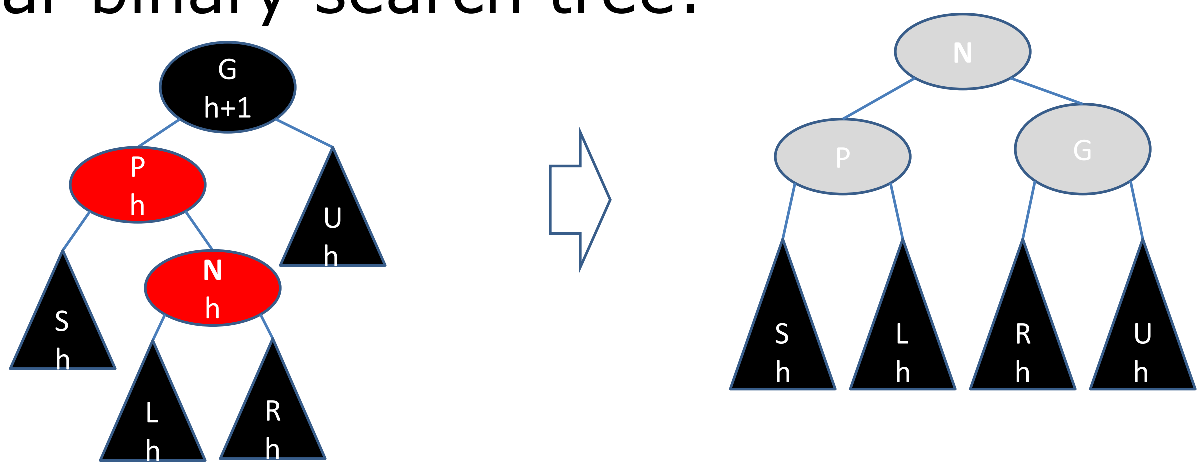
- First we unfold the node N (or the sub-tree N) a bit (we can do this even there is only a red node in the sub-tree N):



The nodes L and R are the children of the node N, and their color must be black, their black depth must be h because the node N is the root node of a regular sub-tree.   (Note: h >= 0)

Insert and then rebalance (continue…)

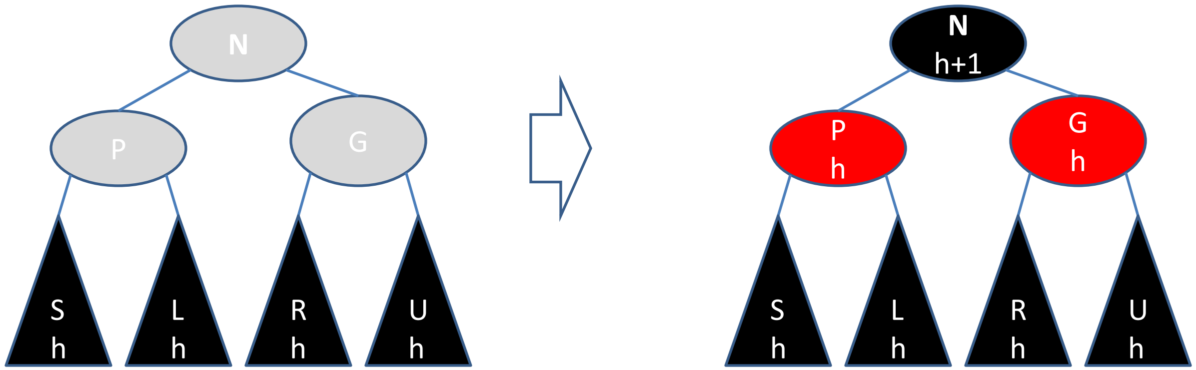
- The term "double rotation" is used to describe the method to rebalance such a sub-tree, but I think we should treat it as a jigsaw puzzle: we have seven pieces, how do we use them to rebuild a regular red-black sub-tree?
- We can reorganize the seven pieces to get such a regular binary search tree:



Insert and then rebalance (continue…)

- The binary search tree has the following traits:
  - all the sub-trees S, L, R and U are regular red-black sub-trees (no rule is broken in them);
  - the black depth of all the sub-trees S, L, R and U is h;
  - until now the color of the nodes N, P and G is not determined.
- Before we dyed the node N red, the black depth of the root node G of the sub-tree was h+1, that means that we should let the N's black depth be h+1, so we can color the nodes N, P and G in this way:

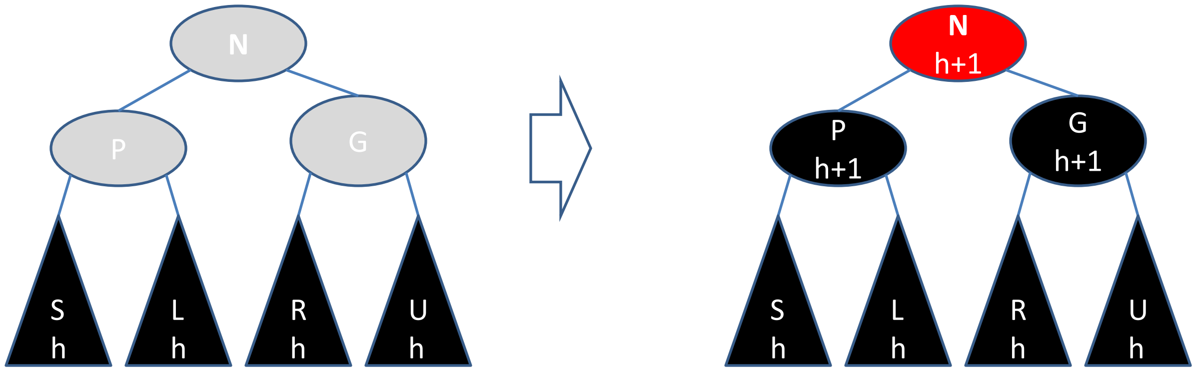
Insert and then rebalance (continue…)



- Now we use the seven pieces to rebuild a new red-black sub-tree, its root node is N with the black depth h+1;
- No rule is broken in the new red-black sub-tree, and certainly no rule is broken in any other sub-tree, it means that the rebalancing process is finished for the case.

Insert and then rebalance (continue…)

- BTW, we can color the nodes P, G and N in other way:

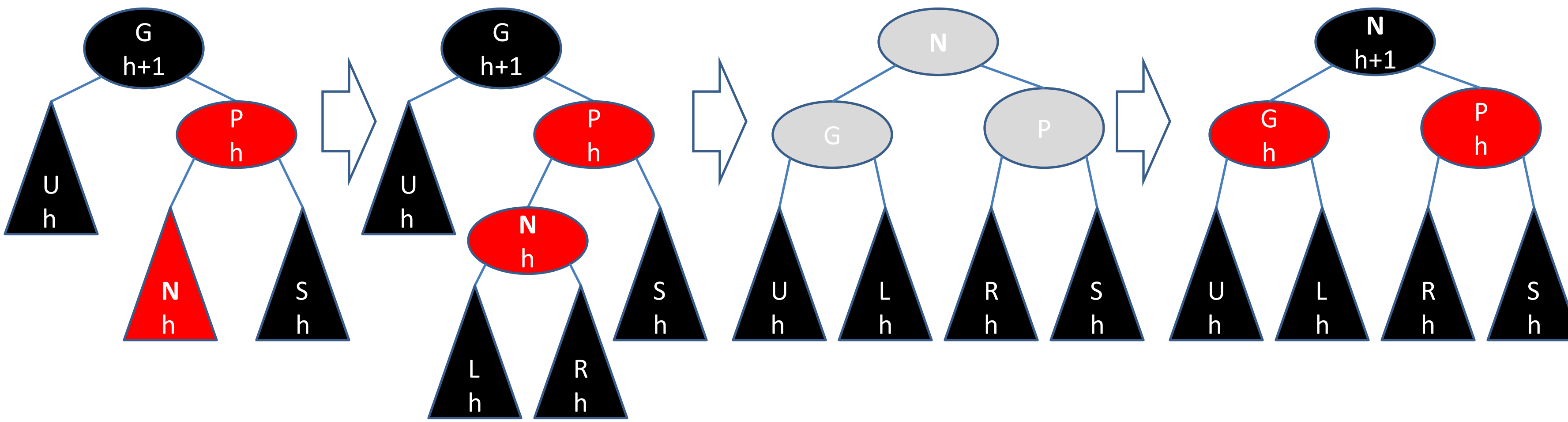


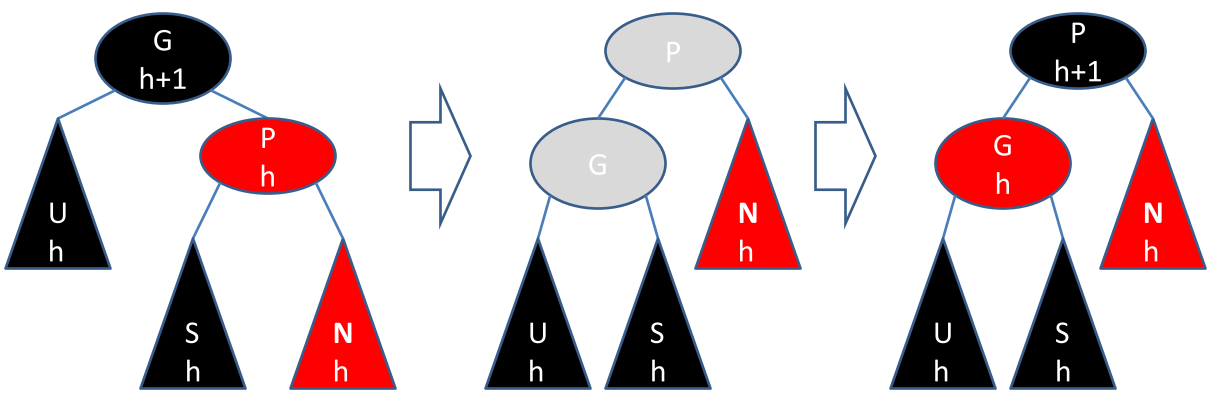
- The resulted sub-tree does not break any rule, its black depth is h+1, but its new root node N is red, the previous root node G is black. So if we select to color the three nodes in this way, the rebalancing process continues;
- For performance we do not do that.

Insert and then rebalance

(continue…)

- We can use the similar method to rebalance the irregular sub-trees 1.4 and 2.4:





Insert and then rebalance (continue…)

- In summary, the inserting and then rebalancing process is recursive:
  - the step 1: given a sub-tree whose root node's color has been changed from black to red, we call its root node the node **N**. Note: this sub-tree is always a regular red-black sub-tree before and after the change;
  - the step 2: if the node **N** is the root node of the whole tree, we dye it black, and then the process is finished;
  - the step 3: if its parent is black, the process is finished;
  - the step 4: if its parent is red too, we get eight different irregular sub-trees to rebalance:
    - For the sub-trees in the cases 1.1, 1.2, 2.1, 2.2, we can dye three nodes in an opposite color in order to get regular sub-trees. But the color of the root node of the very sub-trees is changed from black to red, the root node becomes the node **N**, we return to the step 1;
    - For the sub-trees in the cases 1.3, 1.4, 2.3, 2.4, we use the foregoing method to rebalance them to get regular sub-trees, and then the process is finished.
  - the base case is: we select an empty tree or sub-tree and replace its black root (NIL) node with a new red node.

Remove and then rebalance

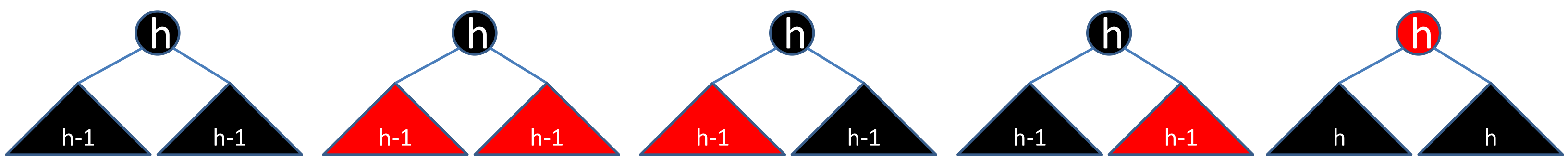
- We always remove such a node from a non-empty tree: its children are two NIL nodes;
- If the node is red, it is finished because no rule is broken;
- If the node is black and it is not the last node, the rule 3 is broken;
- If the node is black and it is the last node, we will get an empty red-black tree;
- Removing such a **black** node is like decreasing the black depth of the corresponding sub-tree from h (1) to h-1 (0).

Remove and then rebalance (continue…)

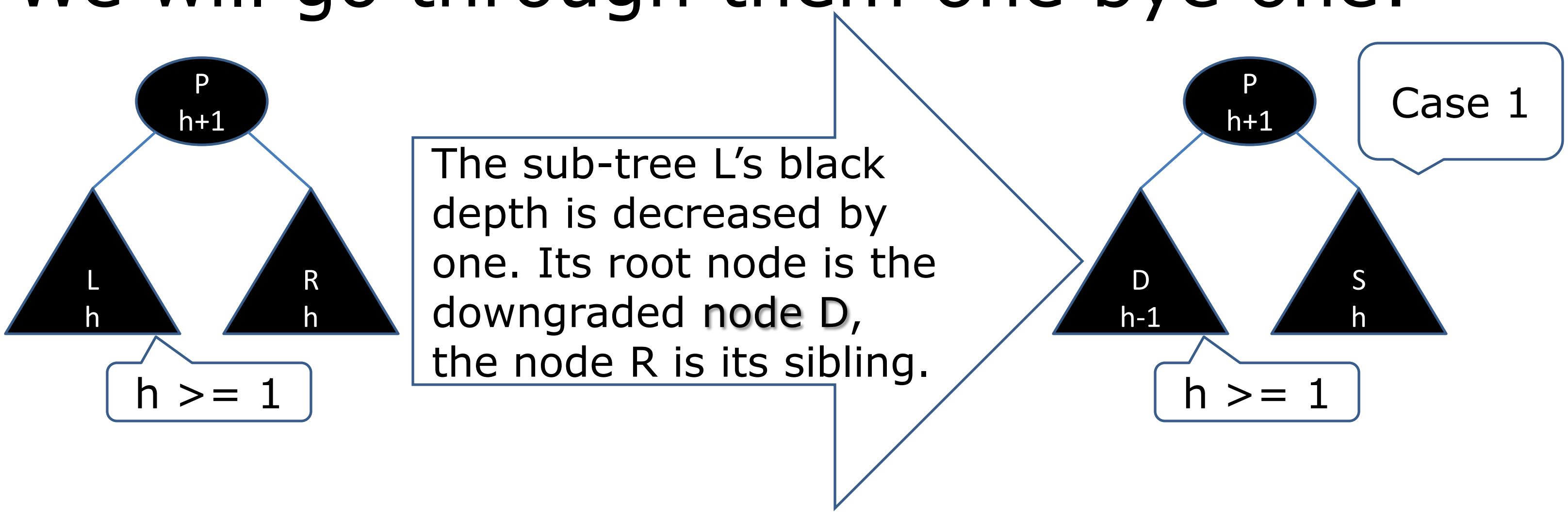
- The removing and then rebalancing process is recursive too;
- The base case is: a **black** node with two NIL child nodes is removed. After that at the very place there is only a NIL node (**still black**);
- The black depth the corresponding sub-tree is decreased from h (1) to h-1 (0) (we call its root node the node **D**), then the rule 3 is broken if it is not the last node, it causes that we need to rebalance the irregular sub-trees;
- How many the irregular sub-trees are there?

Remove and then rebalance
(continue…)

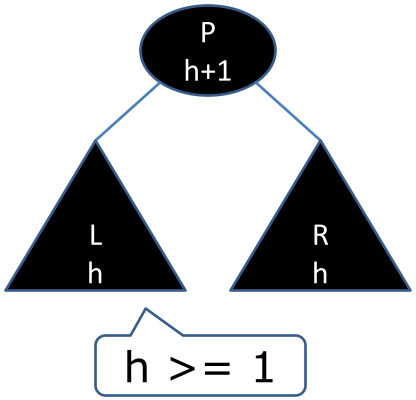
- There are only five different non-empty sub-trees:



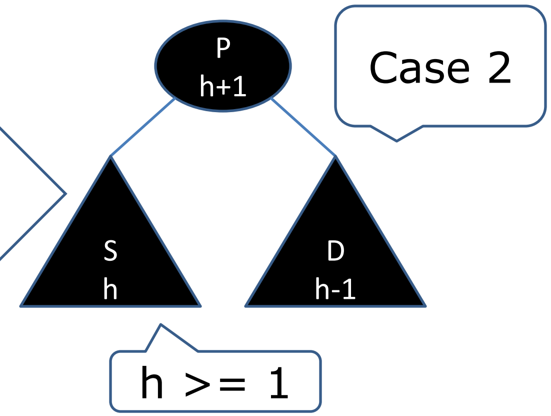
- We will go through them one bye one:

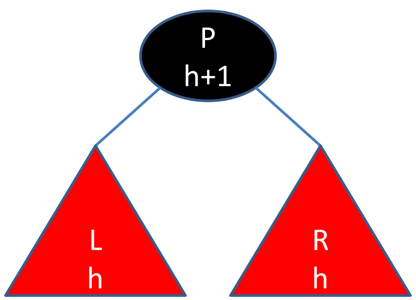


Remove and then rebalance
(continue…)



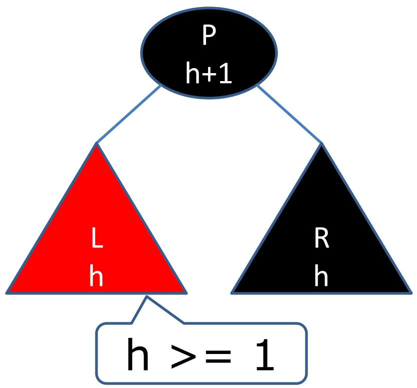
The sub-tree R's black depth is decreased by one. Its root node is the downgraded node D, the node L is its sibling.



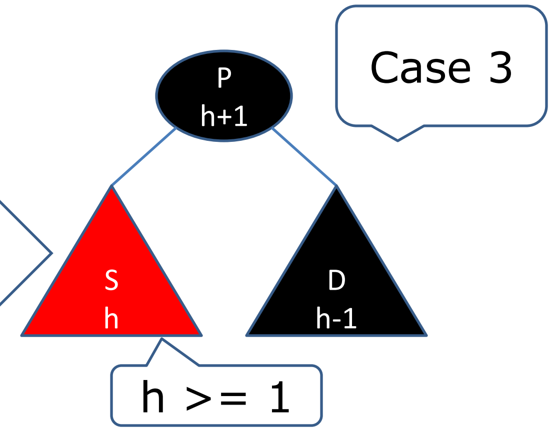


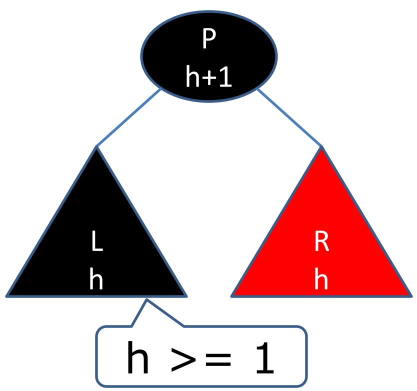
The nodes L and R are red, they cannot be the candidate of the downgraded node.

Remove and then rebalance (continue…)

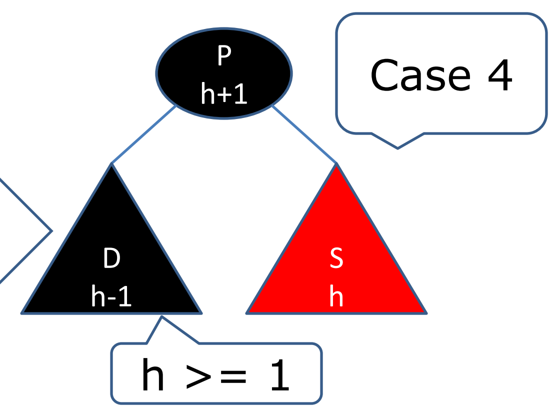


The sub-tree R's black depth is decreased by one. Its root node is the downgraded node D, the node L is its sibling.



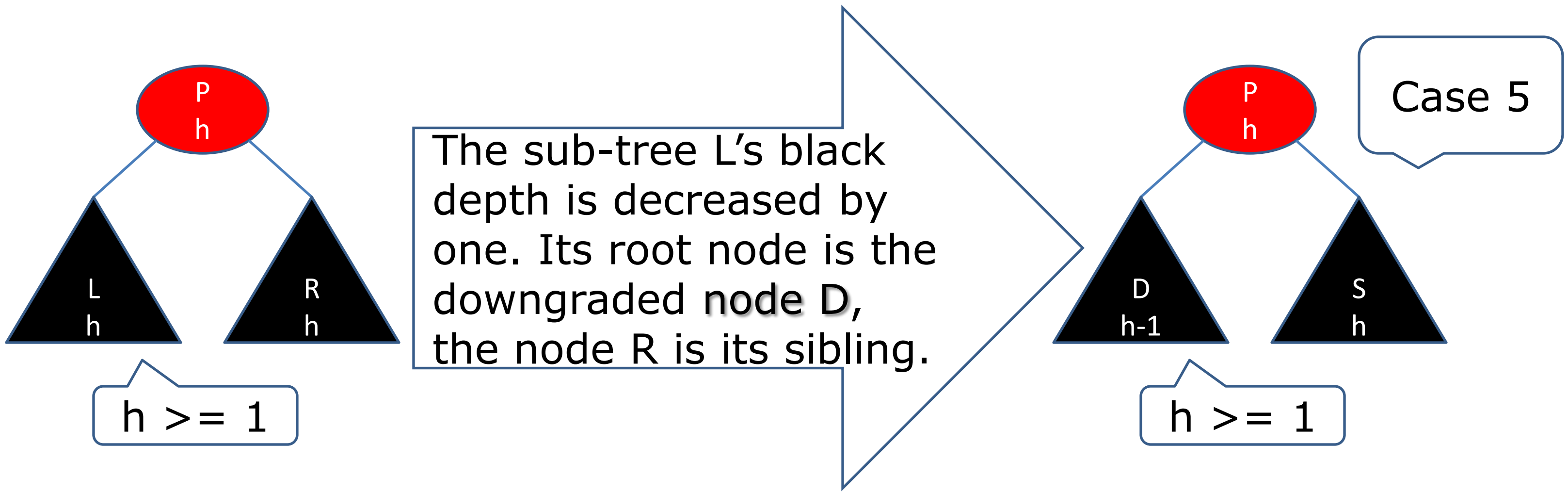


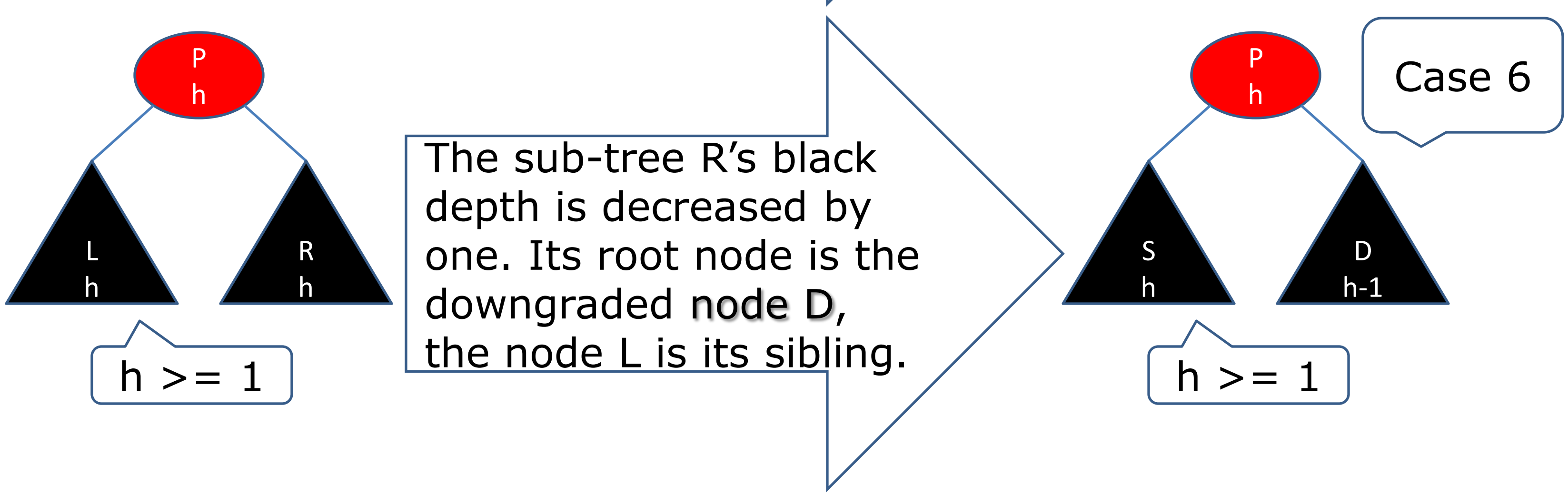
The sub-tree L's black depth is decreased by one. Its root node is the downgraded node D, the node R is its sibling.



Remove and then rebalance

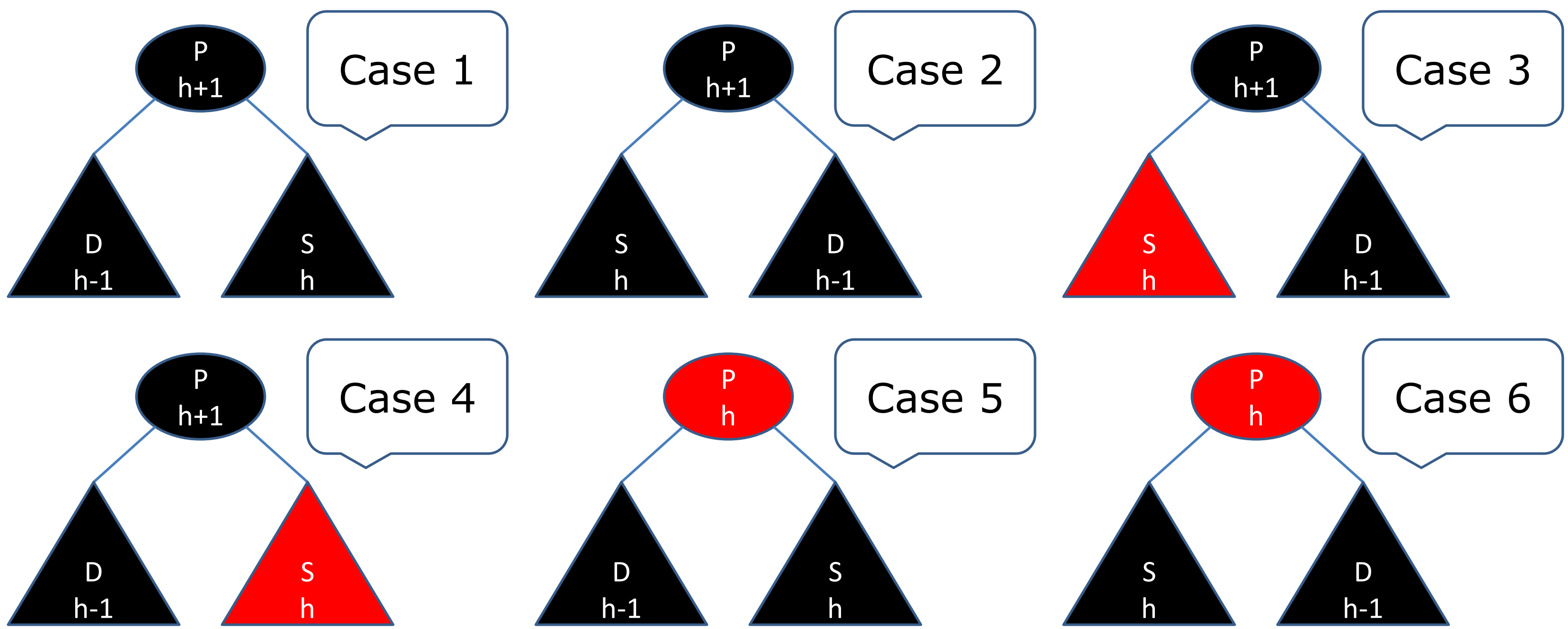
(continue…)





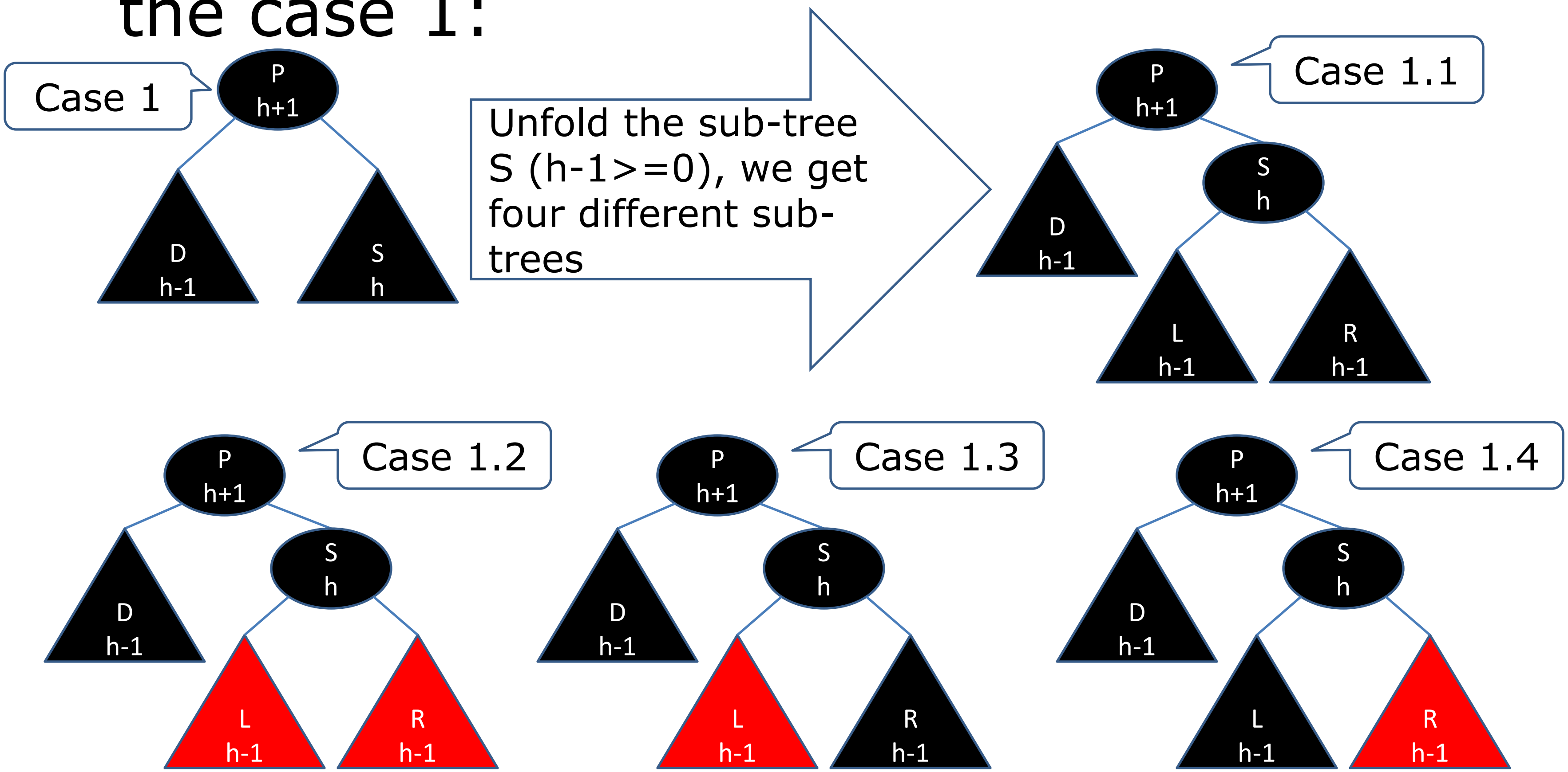
Remove and then rebalance
(continue…)

- Now we get six different irregular red-black sub-trees to rebalance. We do not know how to do it, we need to bring more nodes into consideration.



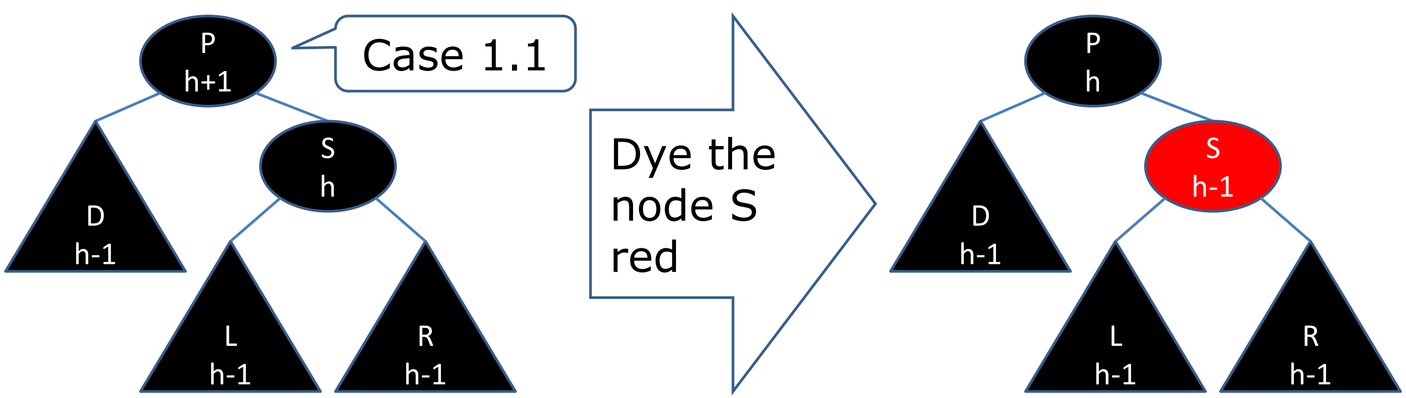
Remove and then rebalance
(continue…)

- The method to rebalance the sub-tree in the case 1:



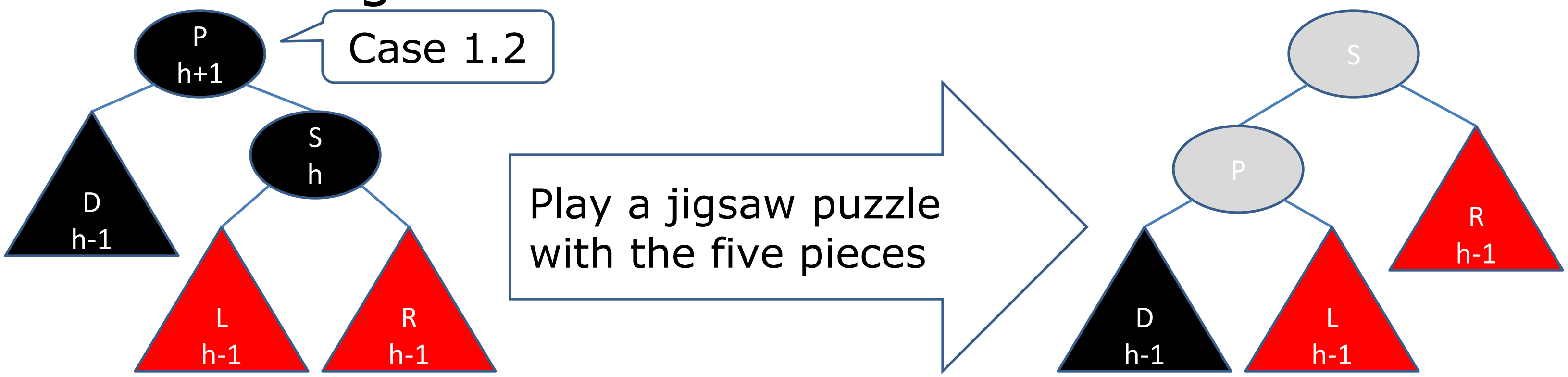
Remove and then rebalance (continue…)

- For the irregular red-black tree 1.1, we can dye the node S red, and then we get a regular red-black tree with the black depth h: the black depth of the root node P is decreased by one. If P is not the root node of the whole tree, it becomes **the node D** and then the recursive process continues.

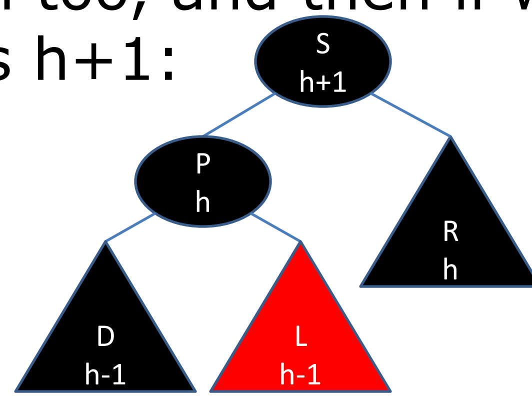


Remove and then rebalance (continue…)

- For the irregular red-black tree 1.2:

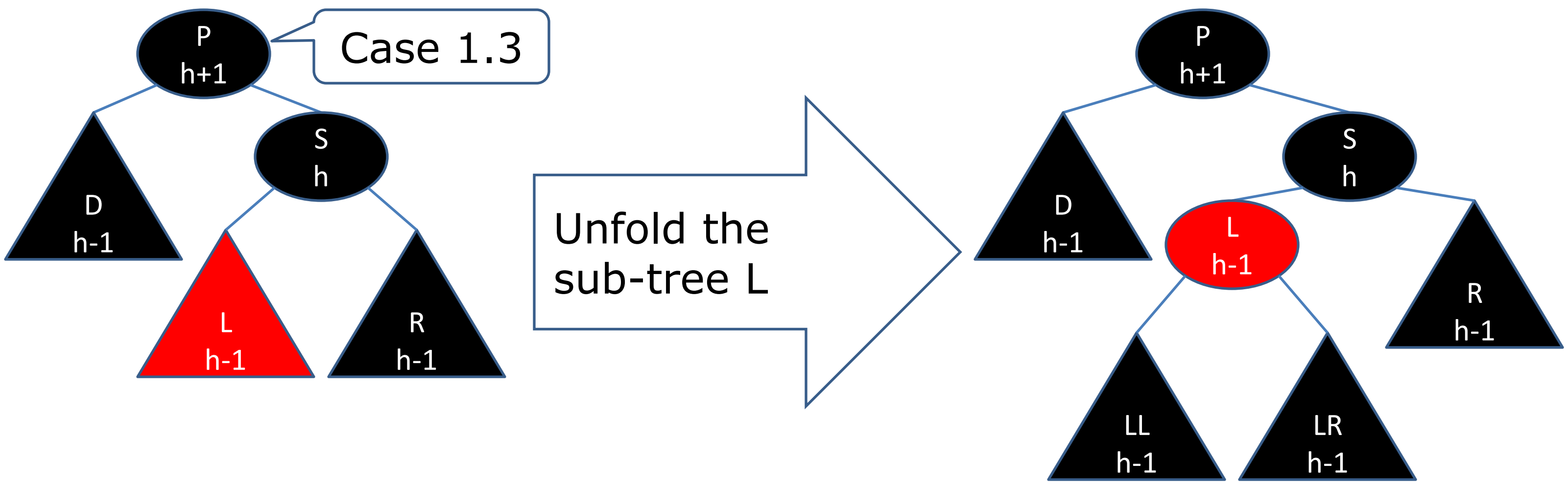


- Because L is red, so P must be black, then P’s black depth will be h, it causes that we must dye R black, so R’s black depth will be h too, and then if we dye S black, S’s black depth is h+1:

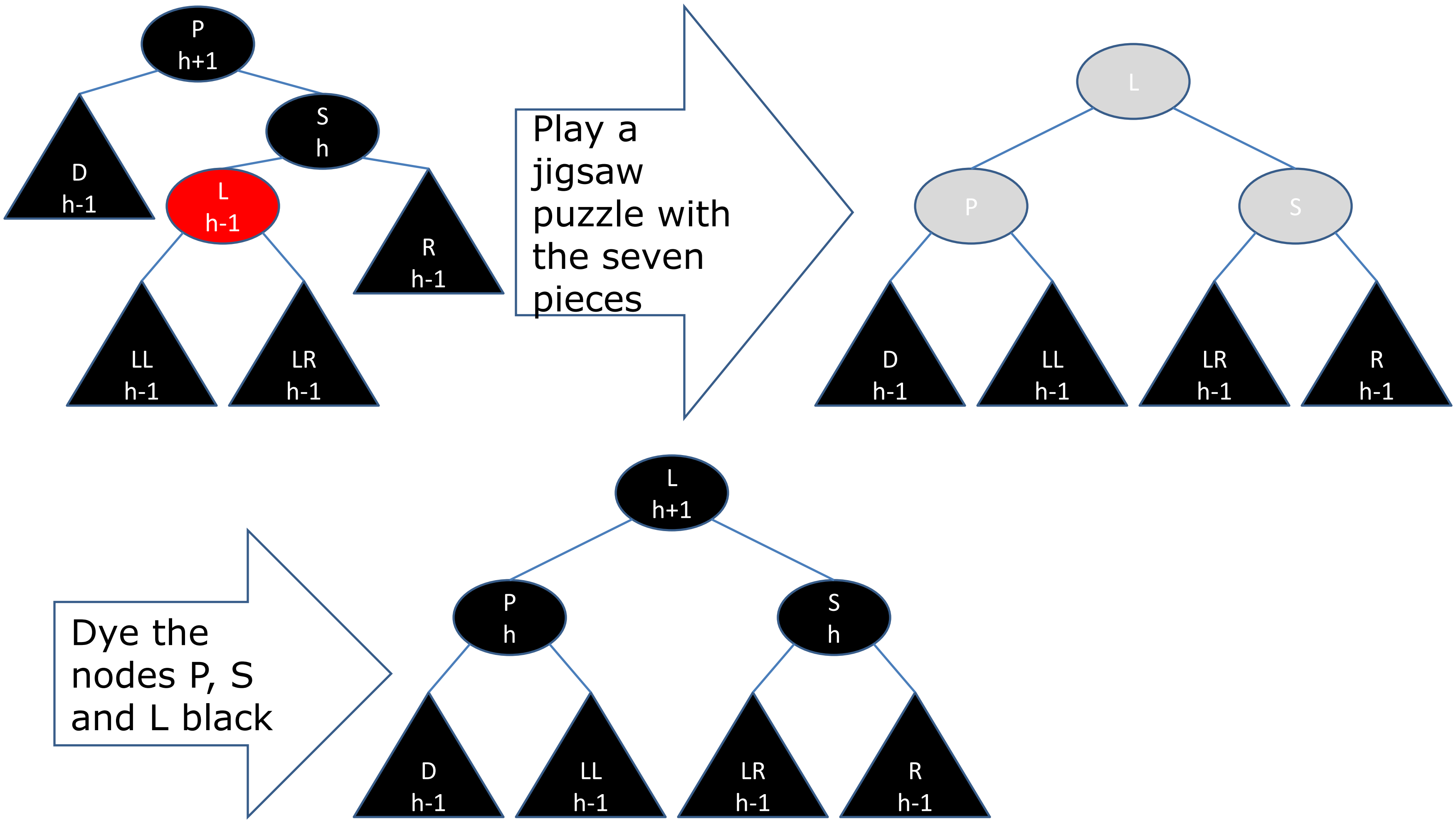


Remove and then rebalance (continue…)

- Our method resolves the case 1.2, no more action is required;
- For the irregular red-black tree 1.3:

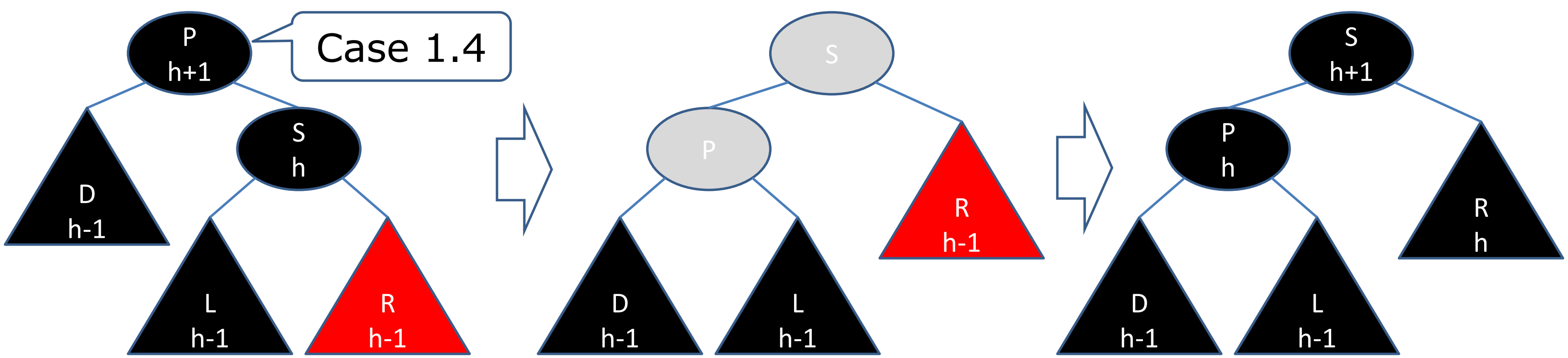


Remove and then rebalance
(continue…)



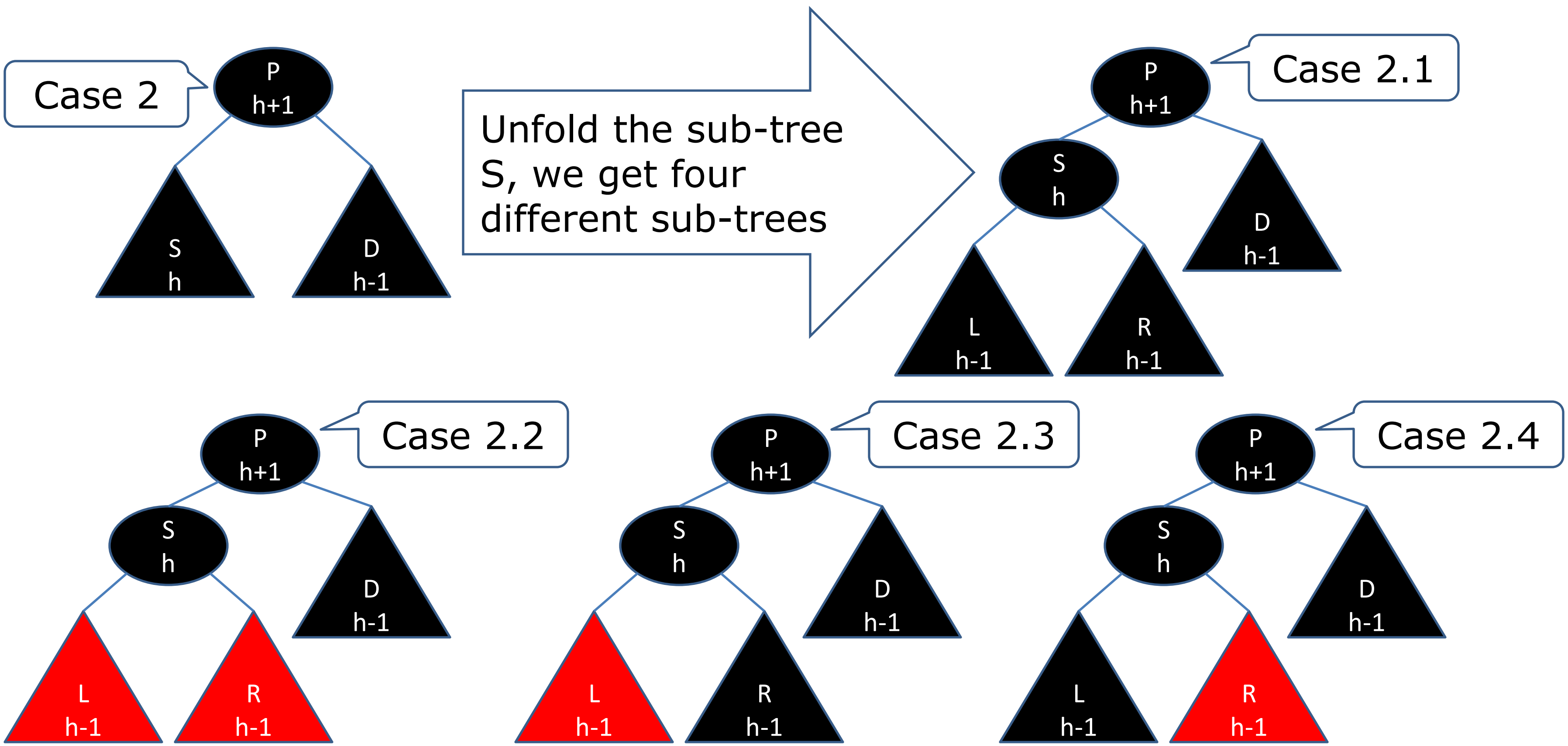
Remove and then rebalance
(continue...)

- Our method resolves the case 1.3, no more action is required;
- For the irregular red-black tree 1.4, we can use the method which is similar to the method 1.2 to resolve it:

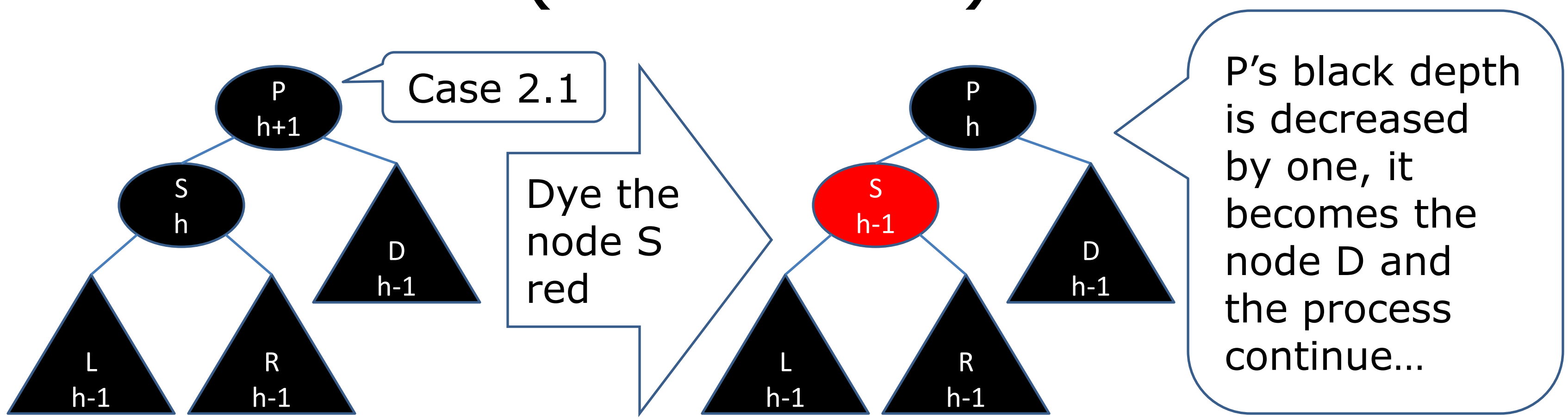


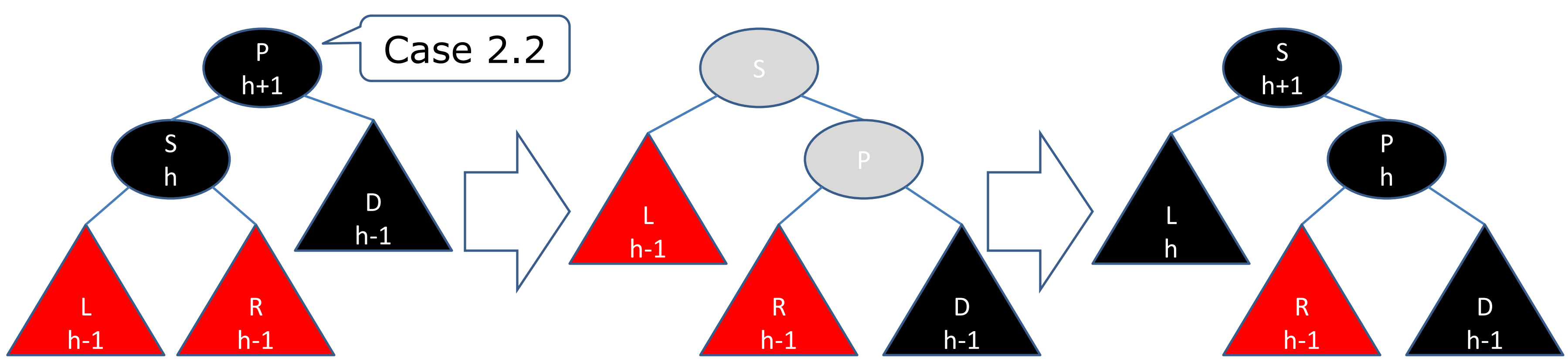
Remove and then rebalance (continue…)

- We can use the similar method to rebalance the sub-tree in the case 2:

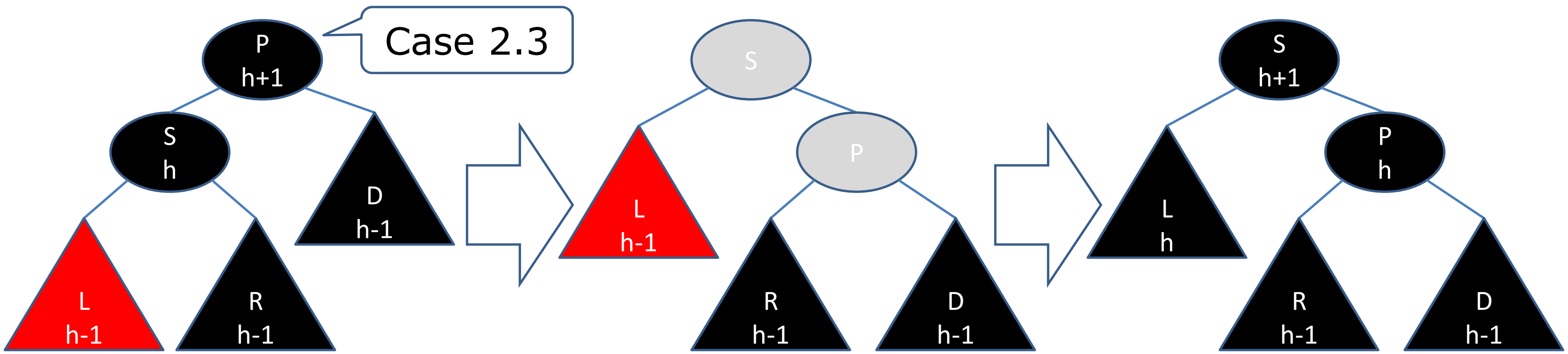


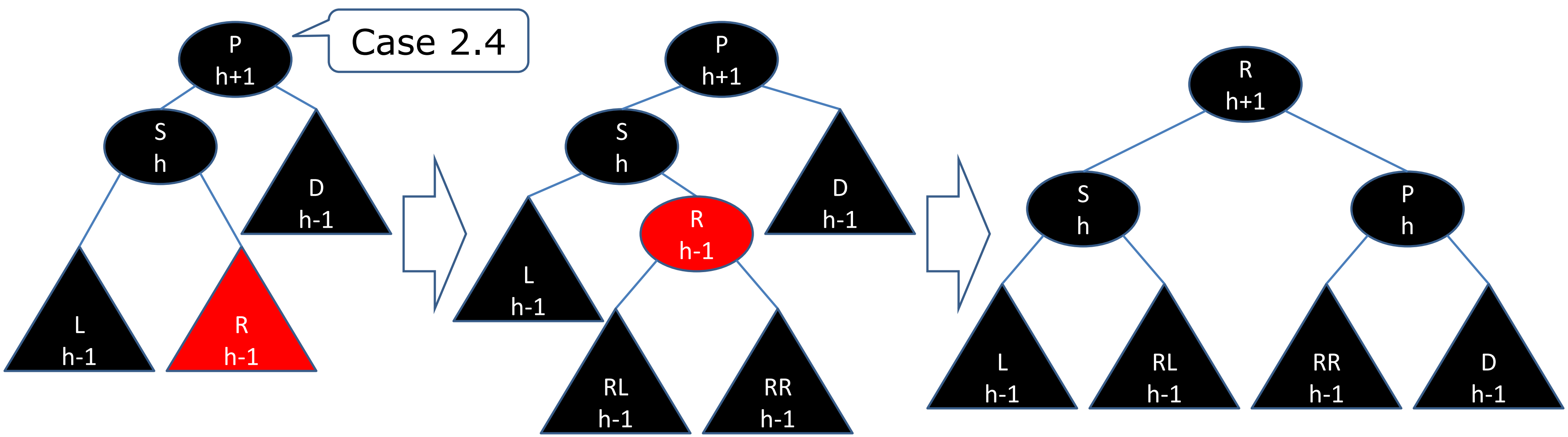
Remove and then rebalance (continue…)





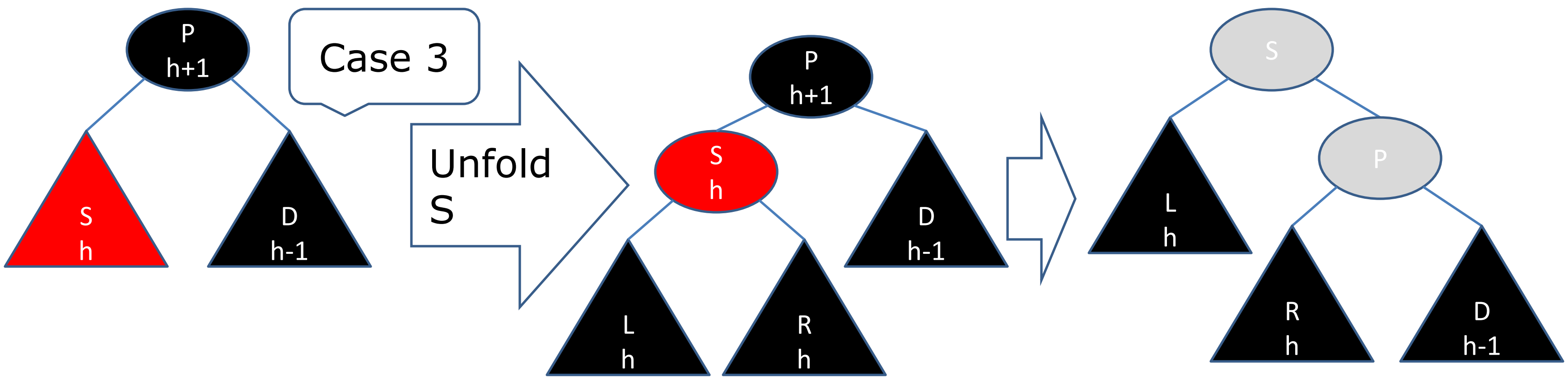
Remove and then rebalance
(continue…)





Remove and then rebalance (continue…)

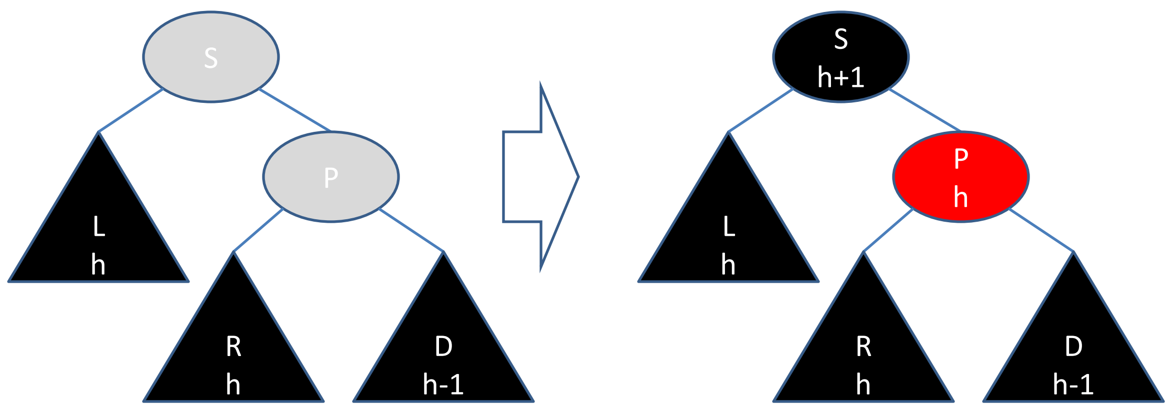
- The method to rebalance the sub-tree in the case 3:



- We unfold S and rotate the sub-tree, we still get a irregular red-black tree;
- But we can transform it into other case;

Remove and then rebalance (continue…)

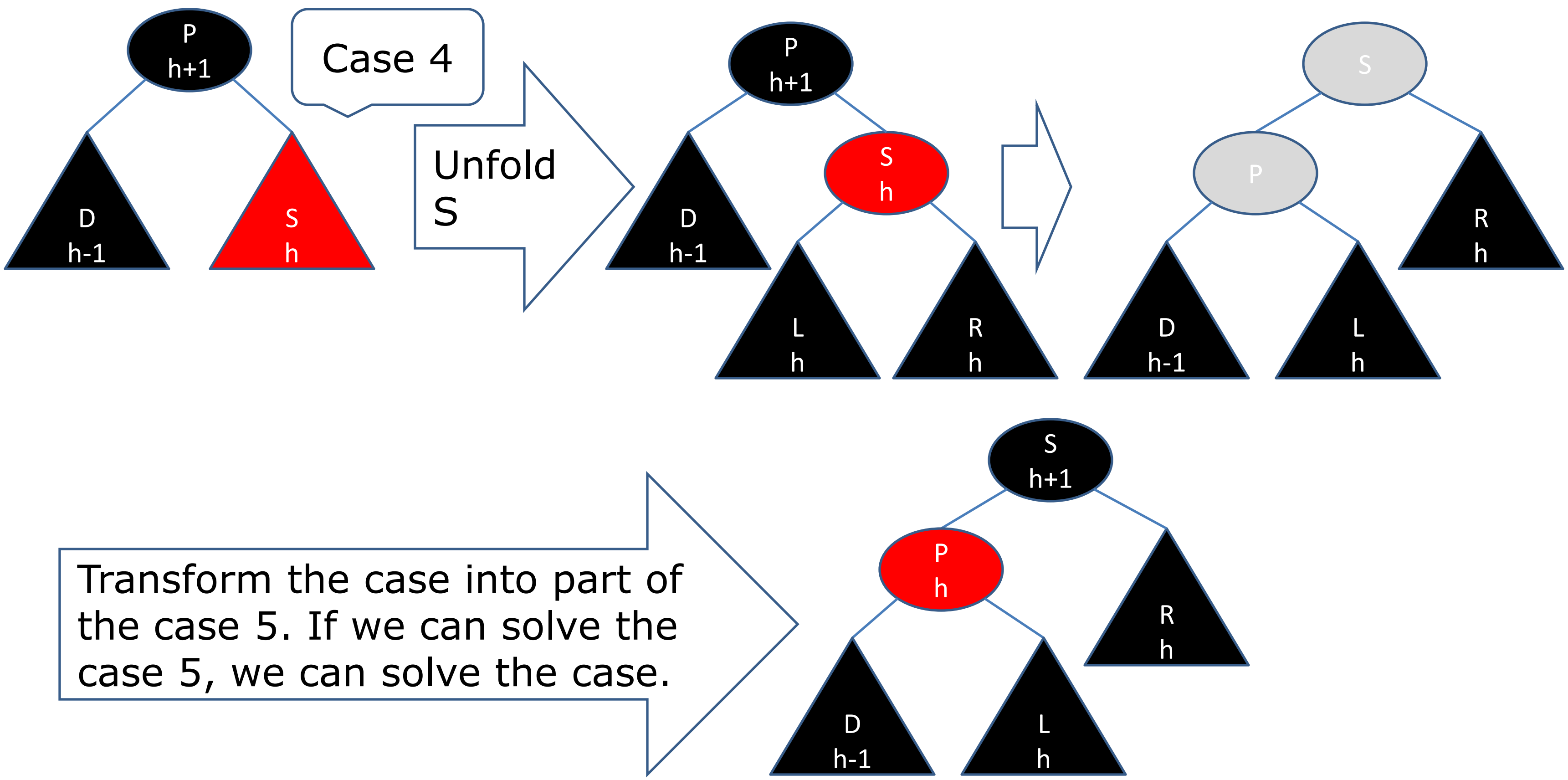
- If we dye P red, suppose that P's black depth is h; dye S black, S's black depth is h+1:



- And then it becomes part of the case 6. If we can resolve the case 6, we can resolve this case.

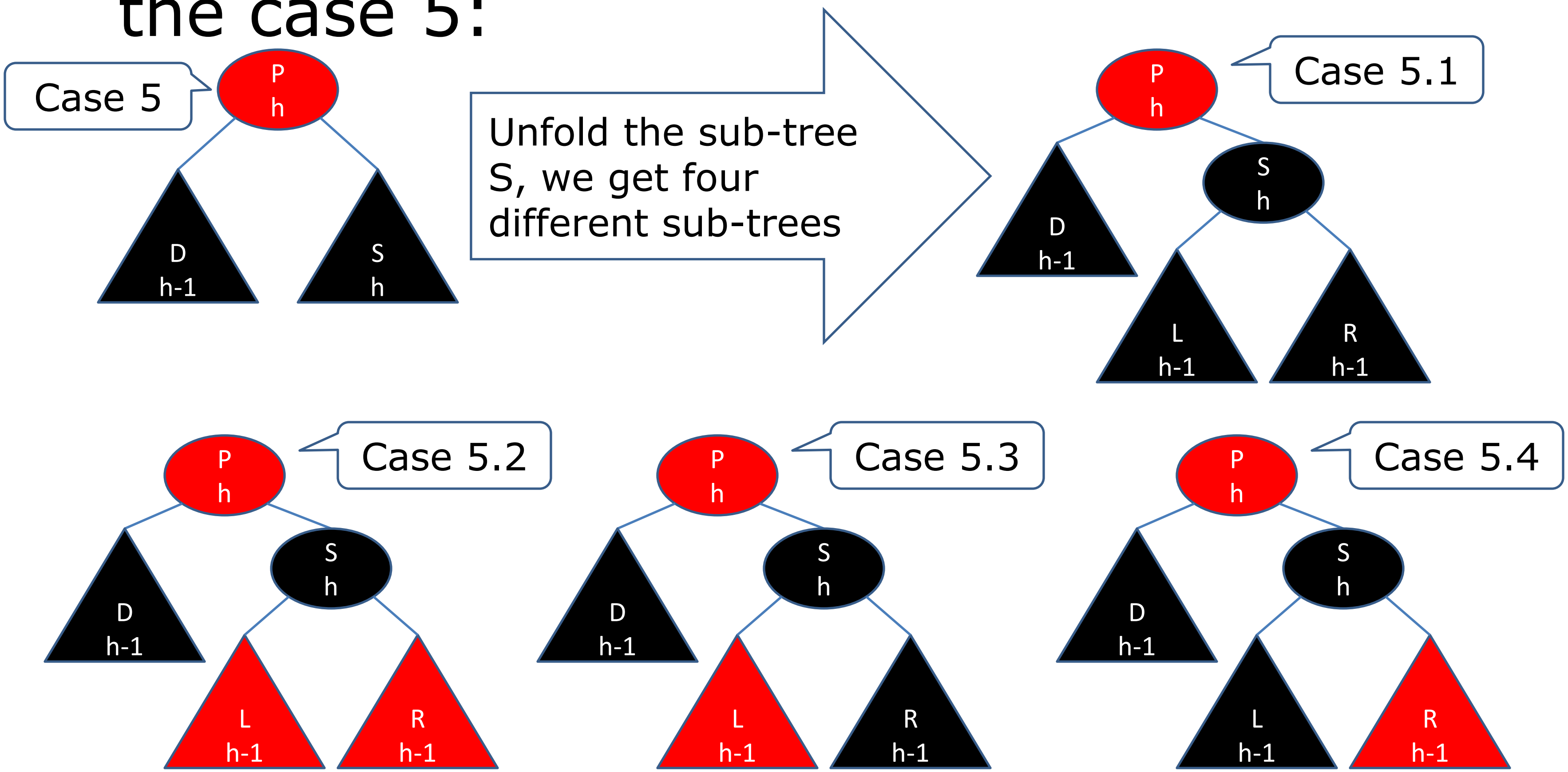
Remove and then rebalance (continue…)

- The method to rebalance the sub-tree in the case 4:



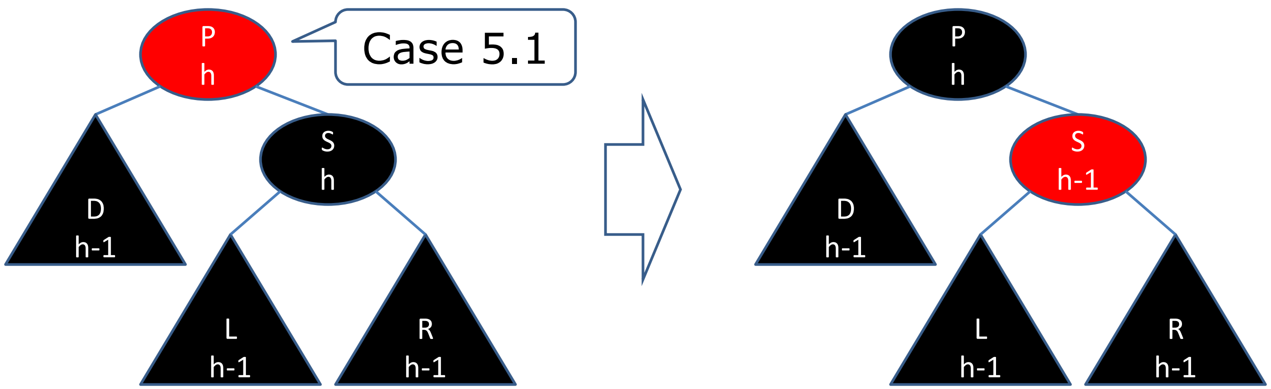
Remove and then rebalance (continue…)

- The method to rebalance the sub-tree in the case 5:



Remove and then rebalance (continue…)

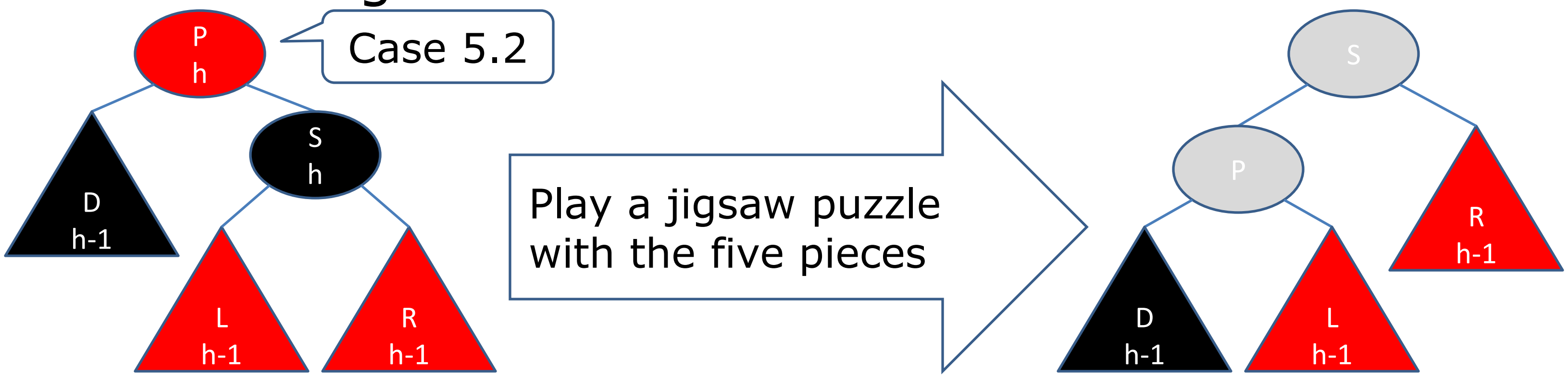
- For the irregular red-black tree 5.1, we can dye S red, dye P black:



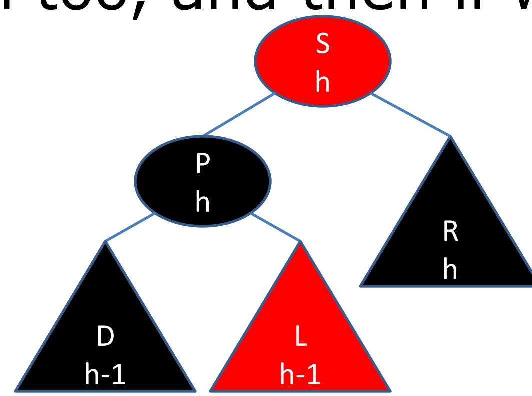
- And then we get a regular sub-tree where no rule is broken, the black depth of the sub-tree is still h, and the color change of the root node of the sub-tree will not cause that some rules may be broken in any upper level sub-trees, so the process is finished.

Remove and then rebalance (continue…)

- For the irregular red-black tree 1.2:

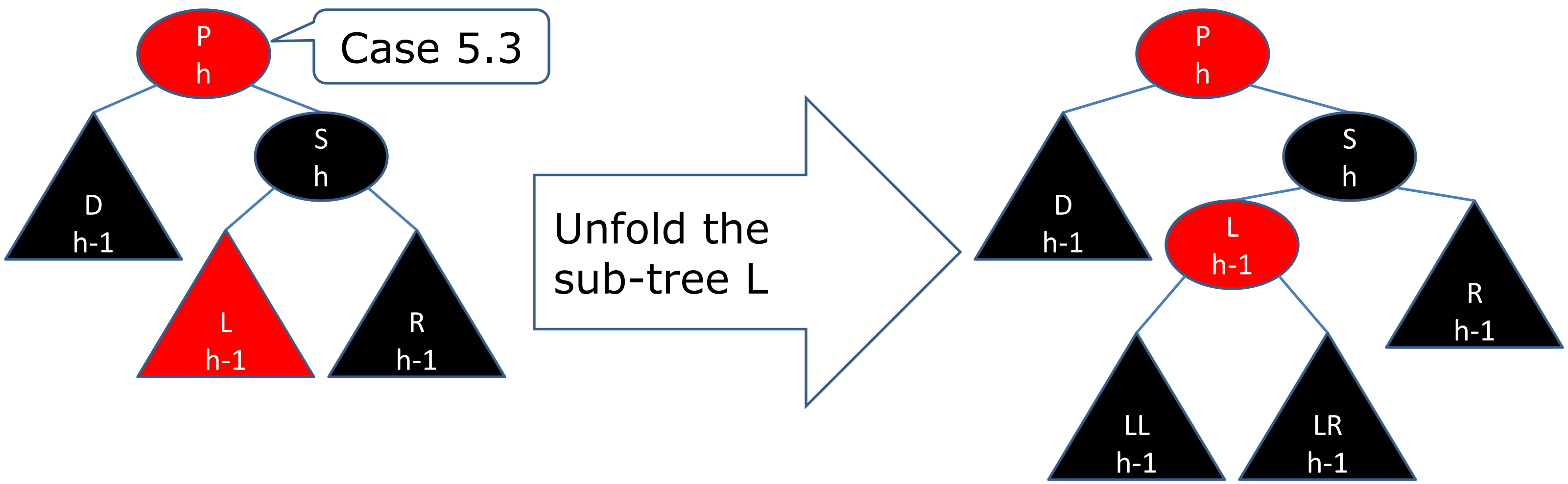


- Because L is red, so P must be black, then P’s black depth will be h, it causes that we must dye R black, so R’s black depth will be h too, and then if we dye S red, S’s black depth is h:

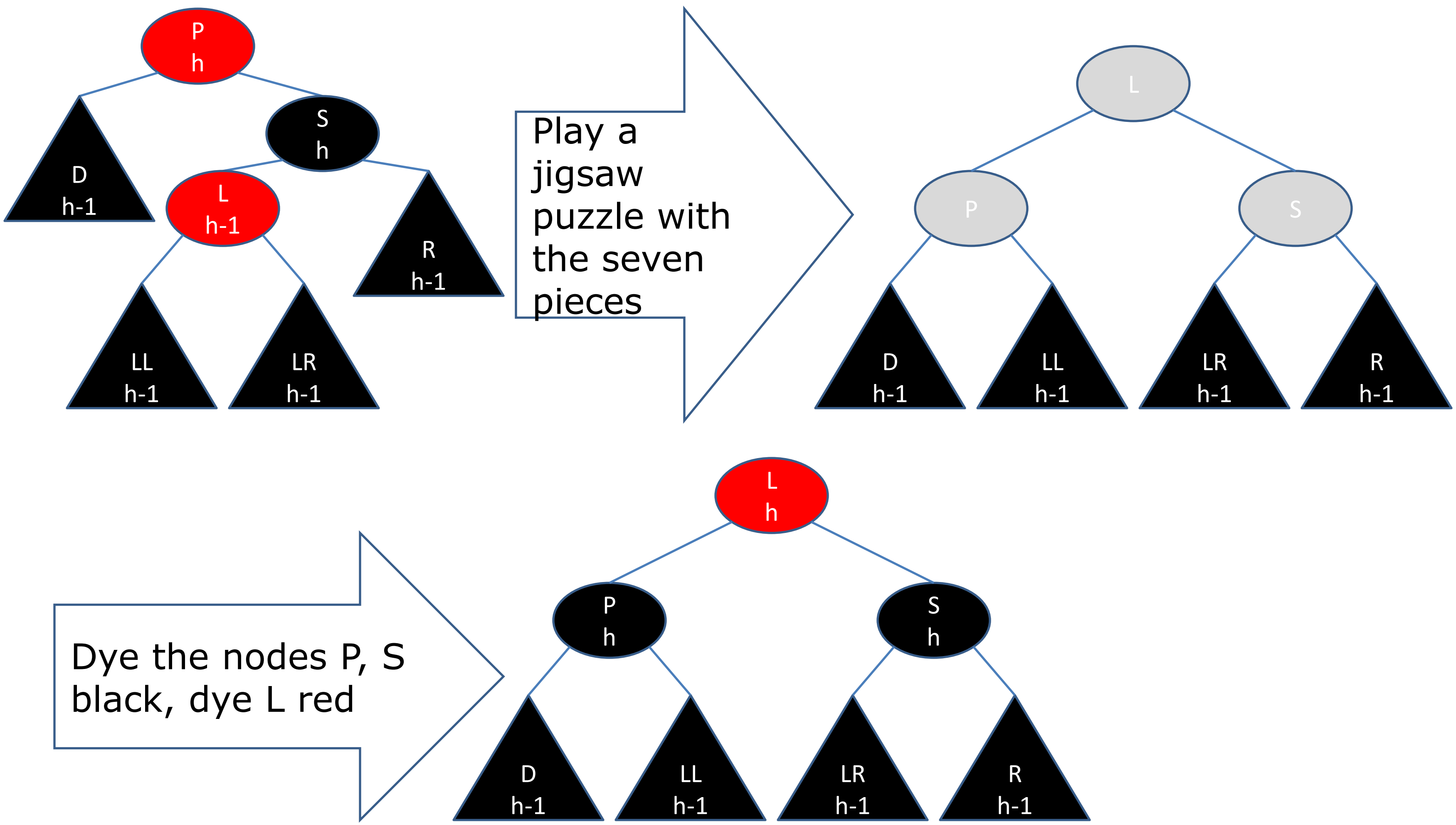


Remove and then rebalance (continue…)

- Our method resolves the case 5.2, no more action is required;
- For the irregular red-black tree 5.3:

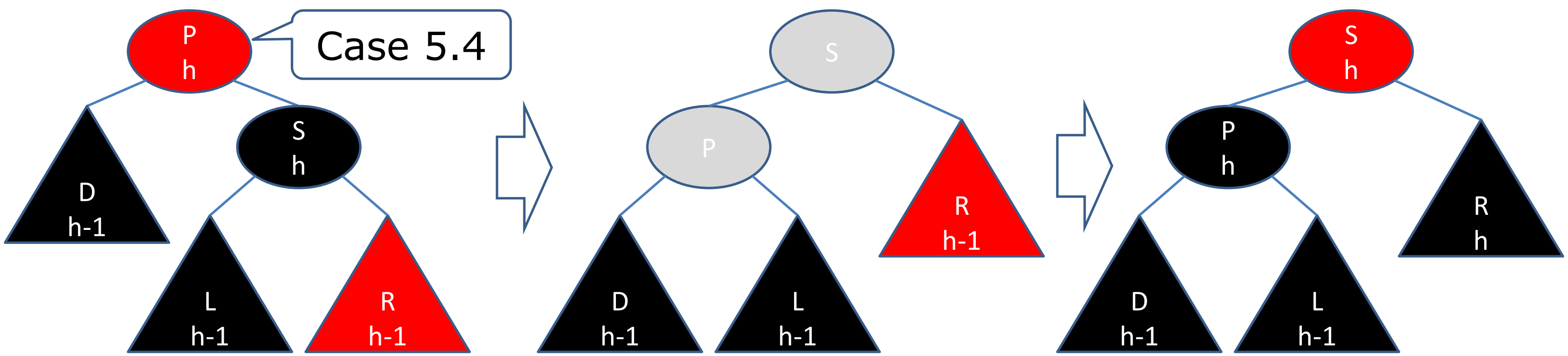


Remove and then rebalance
(continue…)



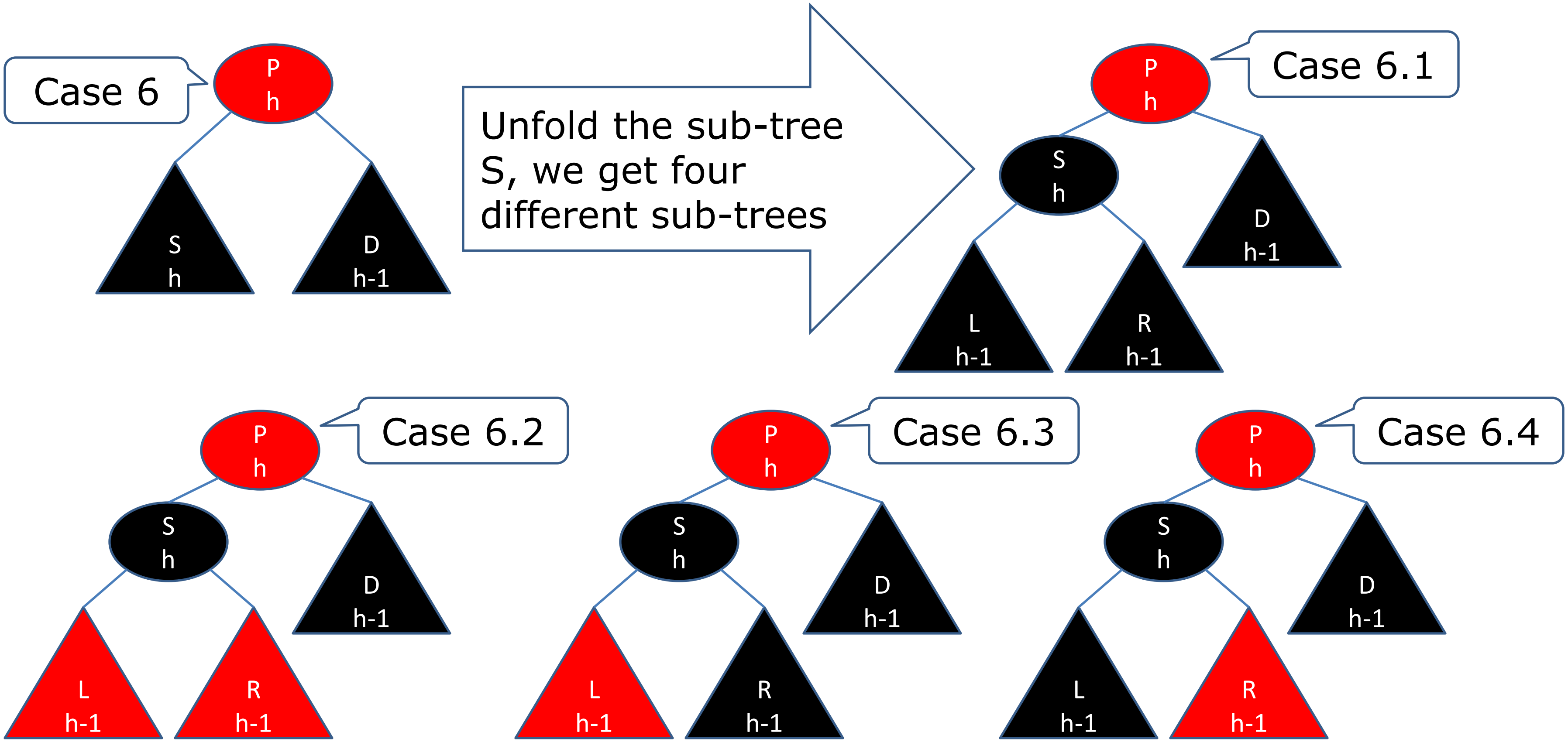
Remove and then rebalance
(continue…)

- Our method resolves the case 5.3, no more action is required;
- For the irregular red-black tree 5.4, we can use the method which is similar to the method 5.2 to resolve it:

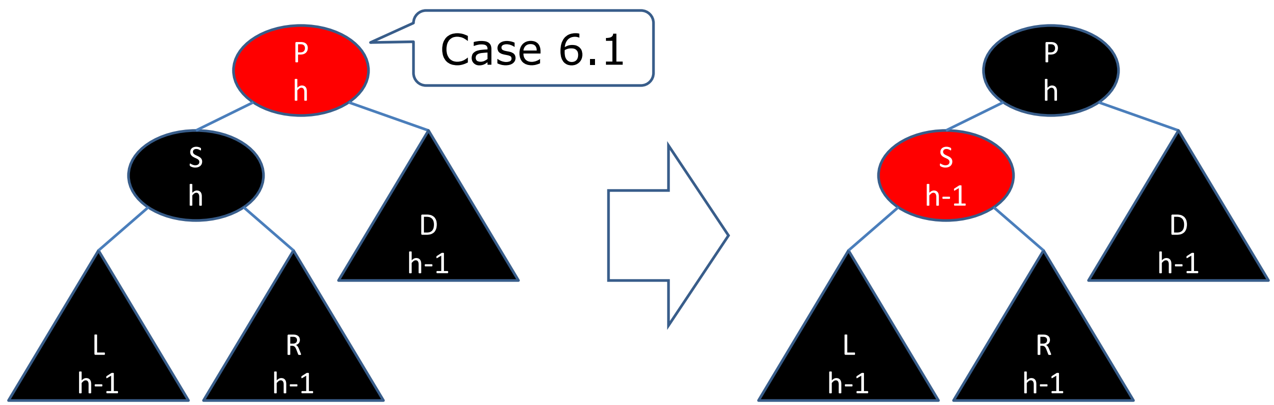


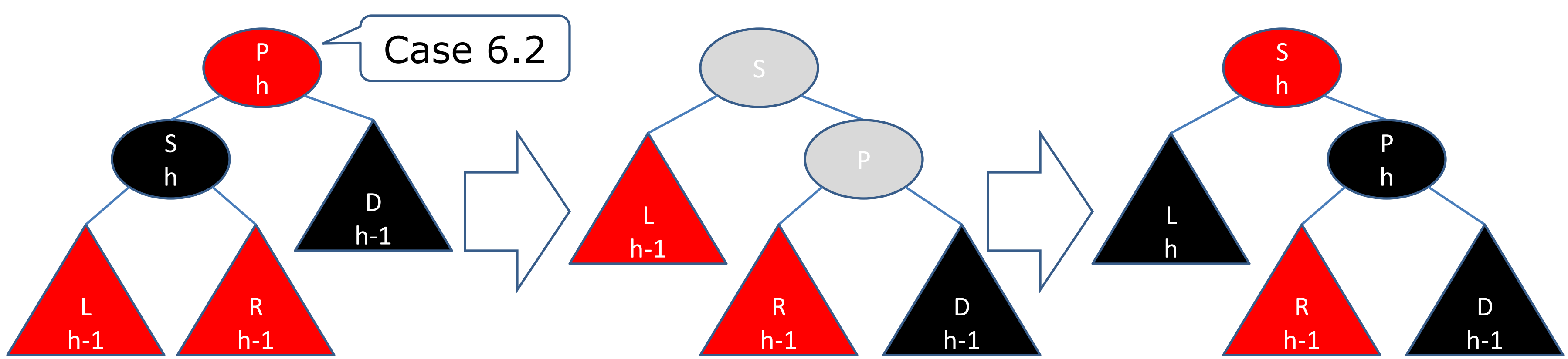
Remove and then rebalance (continue…)

- We can use the similar method to rebalance the sub-tree in the case 6:



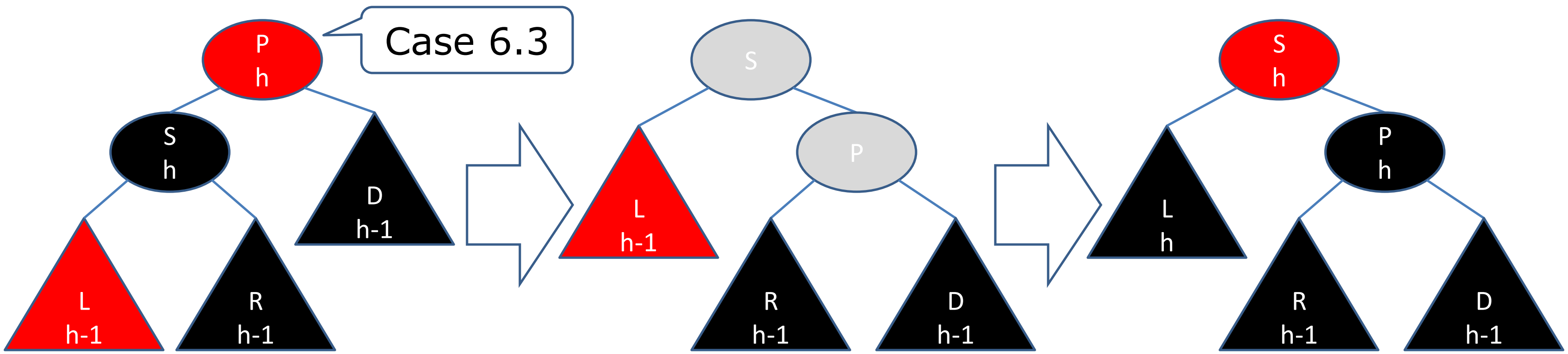
Remove and then rebalance
(continue…)

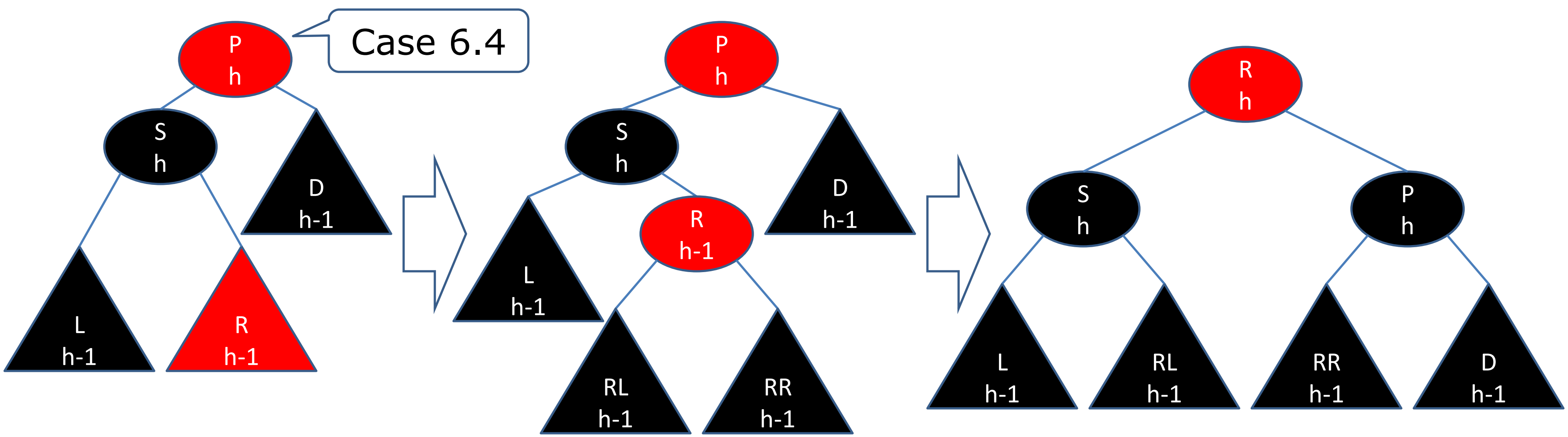




Remove and then rebalance

(continue…)





Remove and then rebalance (continue…)

- In summary, the removing and then rebalancing process is recursive:
  - the step 1: given a regular sub-tree D whose black depth is decreased by one. Before the depth change its root node was black and after that its root node is black too. The sub-tree D is always a regular sub-tree before and after the change. We call the root node of the sub-tree the **black node D**;
  - For brevity, we say that the black node D's black depth has been decreased by one, but strictly speaking, the root node of the sub-tree before the change may be replaced with another one after the change;
  - The invariant is: the sub-tree D is always the same child of a specific parent node, or it is the whole tree before and after the change;

Remove and then rebalance (continue…)

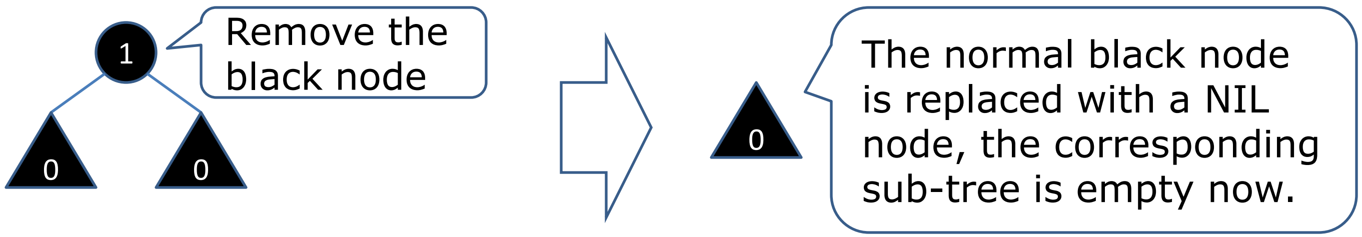
- the step 2: if the node D is the root node of the whole tree, the process is finished;
- the step 3: if D has parent, we get eighteen different irregular sub-trees to rebalance:
  - for the sub-trees in the cases 1.1, 2.1, we dye D's sibling red and then the black depth of the parent of the node D is decreased by one. The parent of the node D becomes the new black node D, we return to the step 1;
  - for the sub-trees in the cases 1.2, 1.3, 1.4, 2.2, 2.3, 2.4, we use the foregoing method to rebalance them to get regular sub-trees, and then the process is finished;

Remove and then rebalance (continue…)

- the sub-trees in the case 3 can be transformed into the sub-trees in the case 6 (strictly speaking, after the transformation, the resulted sub-trees are part of the sub-trees of the case 6);
- the sub-trees in the case 4 can be transformed into the sub-trees in the case 5 (strictly speaking, after the transformation, the resulted sub-tree is part of the sub-trees of the case 5);
- for the sub-trees in the cases 5.1, 6.1, we exchange the color of D’s sibling and D’s parent to finish the recursive process;
- for the sub-trees in the cases 5.2, 5.3, 5.4, 6.2, 6.3, 6.4, we use the foregoing method to rebalance them to get regular sub-trees, and then the process is finished.

Remove and then rebalance (continue…)

- The base case is: we select such a sub-tree which consists of a normal black node and its two child NIL nodes and replace the black node with a NIL node. Then the NIL node is the black node **D** of the sub-tree.



Code

- In this website https://github.com/cyril-gao/wheel/tree/master/Algorithms/BST, you can find C++ code and Python code which implements the algorithm;
- The code looks very complicated, but its performance is as good as the performance of the class std::set.