

The rebalancing process of AVL 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

- In a binary search tree the balance factor of a node is defined to be the height difference of its two child sub-trees:
 - $\text{BalanceFactor}(\text{node}) := \text{Height}(\text{RightSubtree}(\text{node})) - \text{Height}(\text{LeftSubtree}(\text{node}))$
- A binary search tree is defined to be an AVL tree if the invariant $\text{BalanceFactor}(\text{node}) \in \{-1, 0, 1\}$ holds for every node in the tree;
- The balance factor of a NIL node is 0.


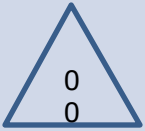
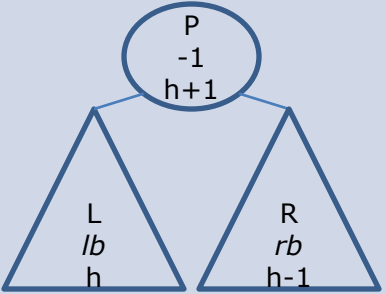
Some conventions

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

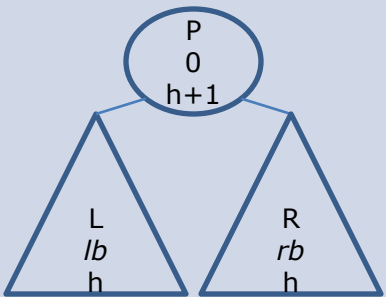
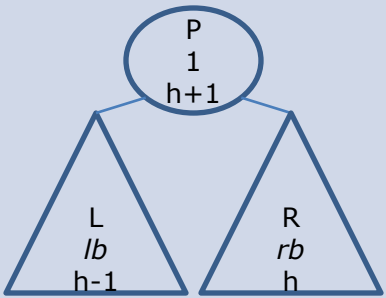
Figure	Meaning
	A node in a binary tree.
	<p>A binary tree, sub-tree, or empty tree. Note: it can be used to represent a tree or sub-tree in which there is only one node.</p> <p>Note: if it is not empty, we can unfold it to get the figure below:</p>
	A binary tree or sub-tree which has a root node and two sub-trees (a non-empty tree).

Some conventions (continue...)

- Each node in an AVL tree has a balance factor, which must belong to $\{-1, 0, 1\}$, so:

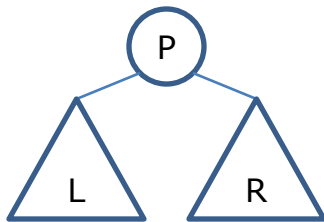
Figure	Meaning
	A normal (non-NIL) node in an AVL tree.
	An empty AVL tree (or sub-tree). Its height is 0, the balance factor of the root node of the tree is 0.
	An AVL tree or sub-tree whose root node P's balance factor is -1, P's height is $h+1$. The height of P's left sub-tree L is h , the root node of the left sub-tree L's balance factor $lb \in \{-1, 0, 1\}$. The height of P's right sub-tree R is $h-1$ ($h-1 \geq 0$), the root node of the right sub-tree R's balance factor $rb \in \{-1, 0, 1\}$.

Some conventions (continue...)

Figure	Meaning
	An AVL tree or sub-tree whose root node P's balance factor is 0, P's height is $h+1$. The height of P's left sub-tree L is h ($h \geq 0$), the root node of the left sub-tree L's balance factor $lb \in \{-1, 0, 1\}$. The height of P's right sub-tree R is h , the root node of the right sub-tree R's balance factor $rb \in \{-1, 0, 1\}$.
	An AVL tree or sub-tree whose root node P's balance factor is 1, P's height is $h+1$. The height of P's left sub-tree L is $h-1$ ($h-1 \geq 0$), the root node of the left sub-tree L's balance factor $lb \in \{-1, 0, 1\}$. The height of P's right sub-tree R is h , the root node of the right sub-tree R's balance factor $rb \in \{-1, 0, 1\}$.

Some conventions (continue...)

- 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;
- So you should understand what they mean when we say the node P, the sub-tree P, the node L, the sub-tree L, ..., etc:



Insert and then rebalance

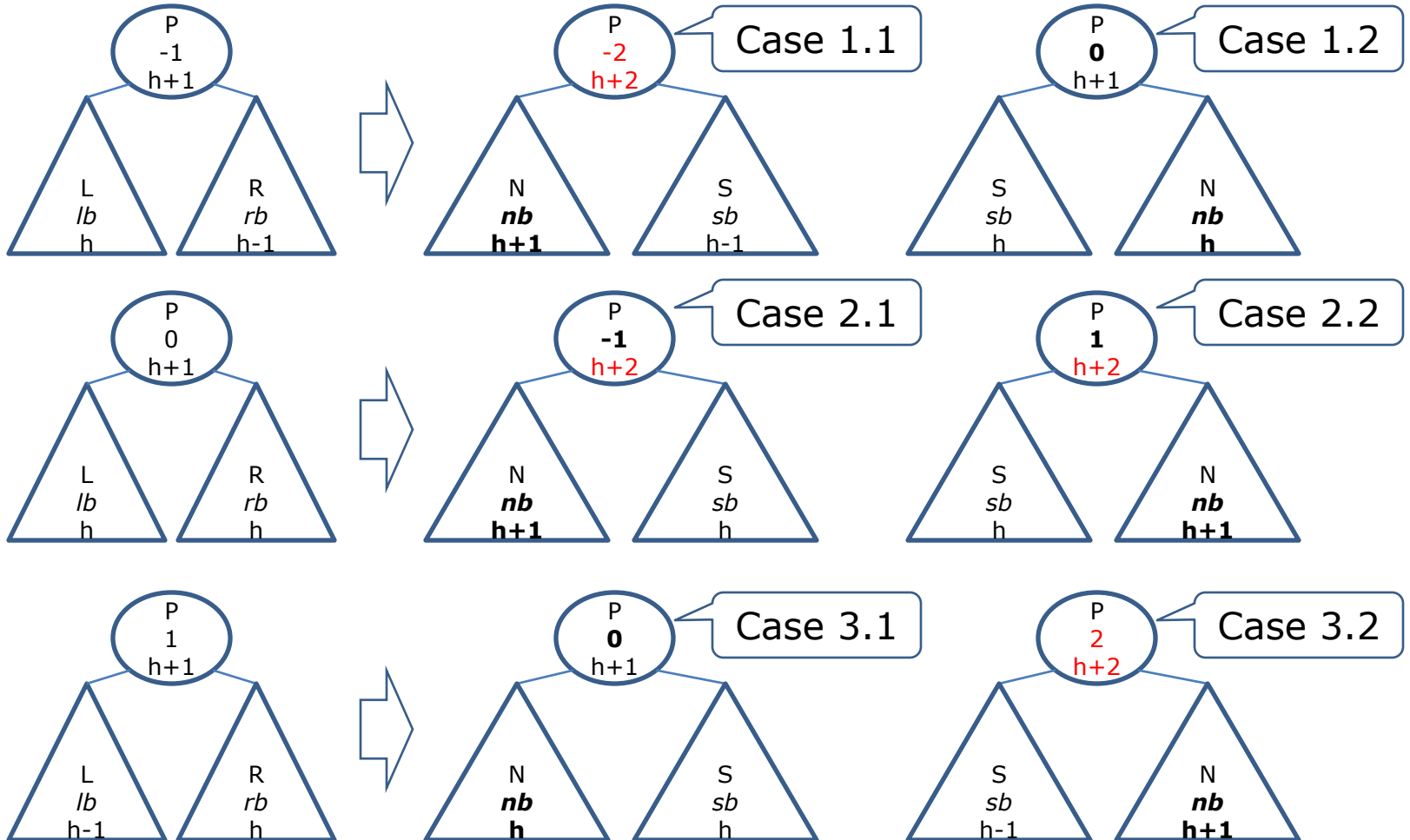
- We always insert such a node whose balance factor is 0 and height is 1 into an AVL tree (or we replace an empty sub-tree with a sub-tree whose root node has two traits: its balance factor is 0 and its height is 1);
- After we replace an empty sub-tree with such a sub-tree **N** (its height is 1 and its root node **N**'s balance factor is 0), we can say at the very place the height of the sub-tree is increased by one, it is a regular sub-tree, but the upper level sub-tree **P** where **N** is a child may not be regular.

Insert and then rebalance

(continue...)

- Why? because N's height is increased by one, and it cause that the balance factor of its parent P is increased (N is the right child) or decreased (N is the left child) by one;
- At first P's balance factor belonged to $\{-1, 0, 1\}$, after that it belongs to $\{-2, -1, 0, 1, 2\}$. More specifically, if P's balance factor was -1, it may be -2 or 0; if it was 0, it may be -1 or 1; if it was 1, it may be 0 or 2.

Insert and then rebalance (continue...)



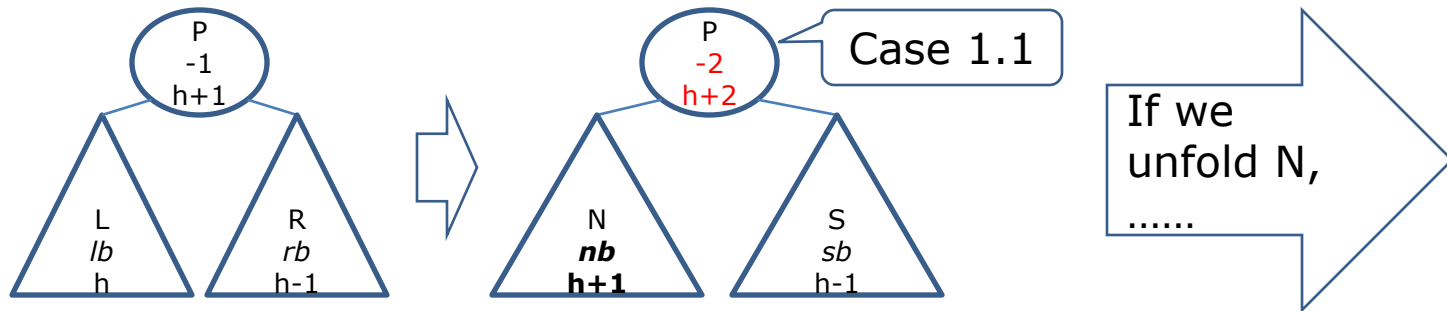
Insert and then rebalance

(continue...)

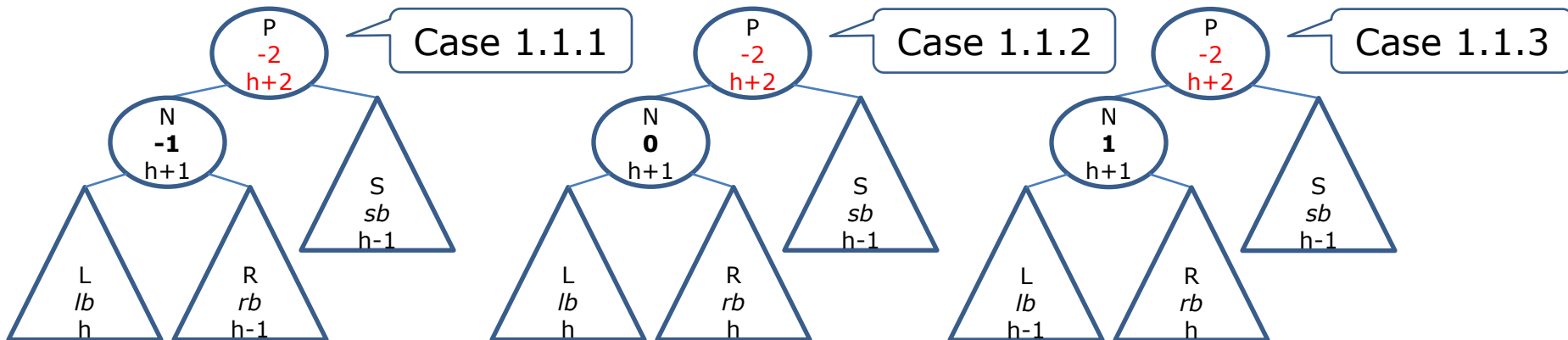
- In the cases 1.1 and 3.2 the sub-trees P are irregular, because P 's balance factor is -2 or 2 ;
- In the cases 1.2 and 3.1 the sub-trees P are regular, P 's height is not changed, and P 's parent is regular too (if there is). The insertion does not cause that any rule is broken. No more action is required;
- In the cases 2.1 and 2.2 the sub-trees P are regular, but P 's height is increased by one, so it demonstrates that the inserting and then rebalancing process is recursive.

Insert and then rebalance (continue...)

- For the case 1.1:

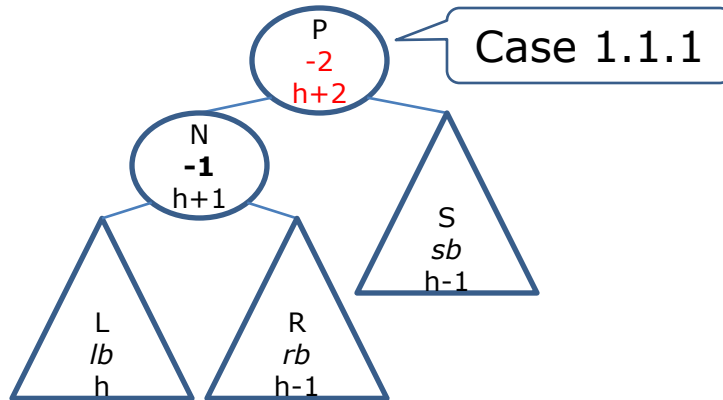


- We must remember: N is a regular sub-tree and N's height is increased by one;
- If we unfold N, we get three different sub-trees:



Insert and then rebalance (continue...)

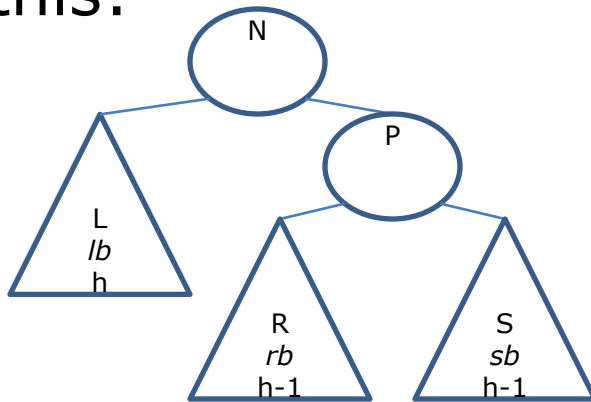
- For the case 1.1.1:



- 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 AVL sub-tree?

Insert and then rebalance (continue...)

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

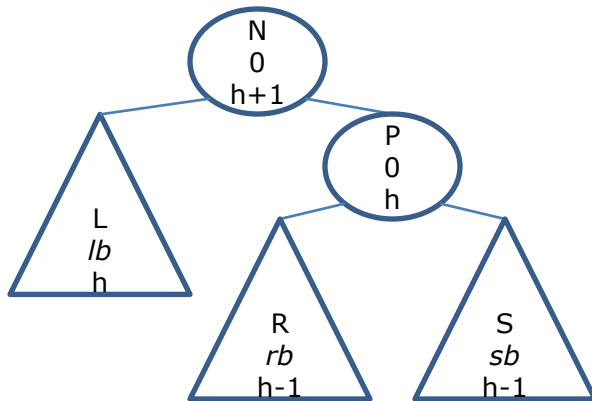


- The binary search tree has the following traits:
 - all the sub-trees L, R and S are regular AVL sub-trees (no rule is broken in them);
 - L's height is h , the height of R and S is $h-1$;
 - until now the balance factor and height of the nodes P and N are not determined.

Insert and then rebalance

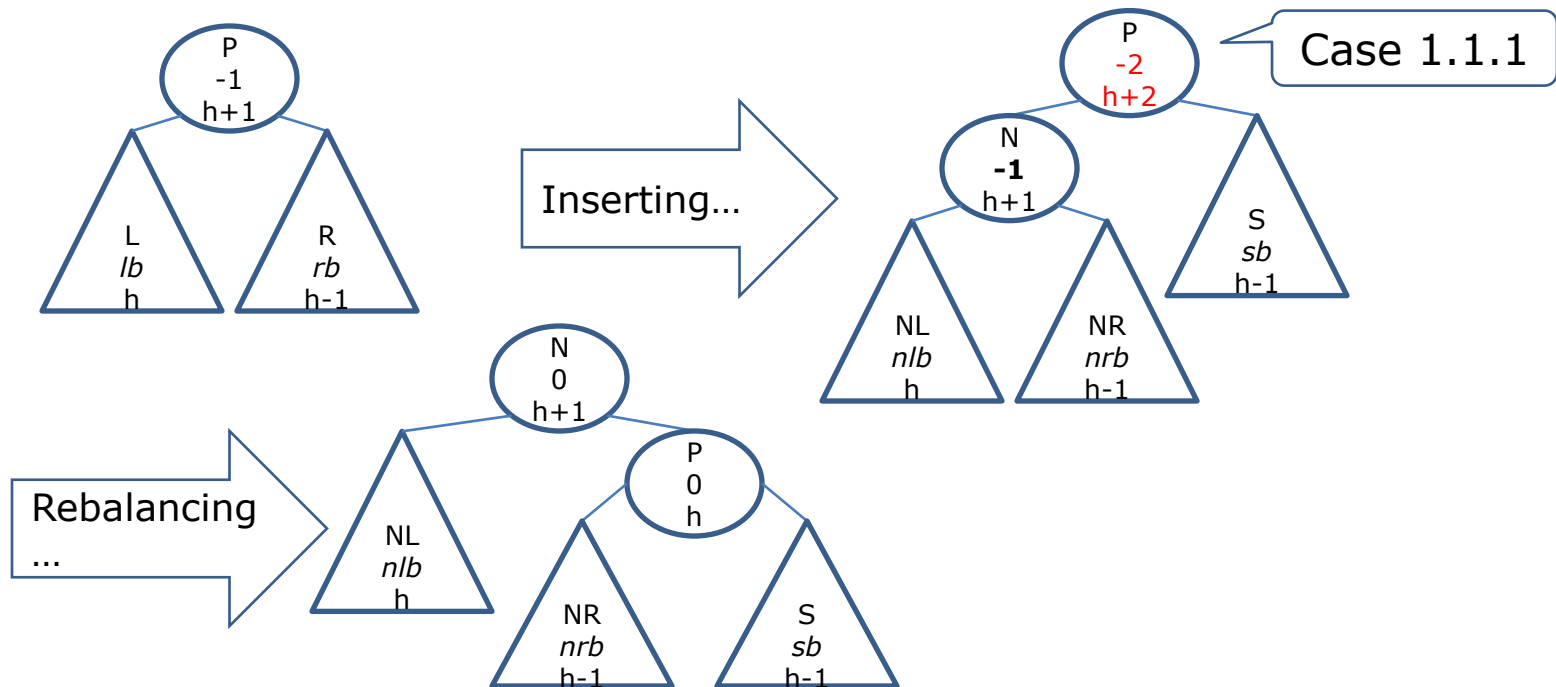
(continue...)

- Since R and S are the children of P, so P's height is h , P's balance factor is 0;
- L and P are the children of N, so N's height is $h+1$, N's balance factor is 0.



Insert and then rebalance (continue...)

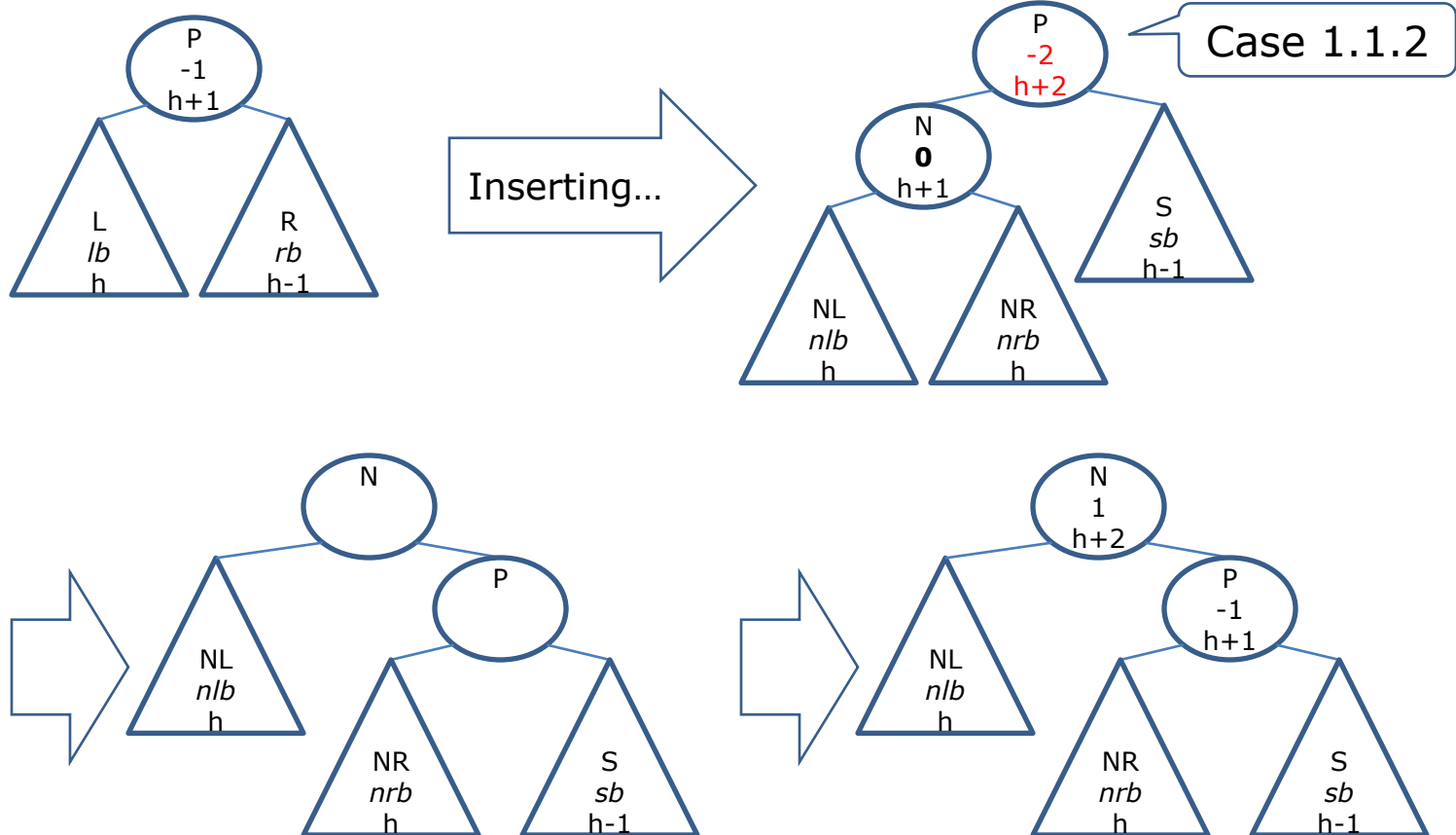
- The state changes of the sub-tree are as below:



- So for the case the inserting and then rebalancing process is finished.

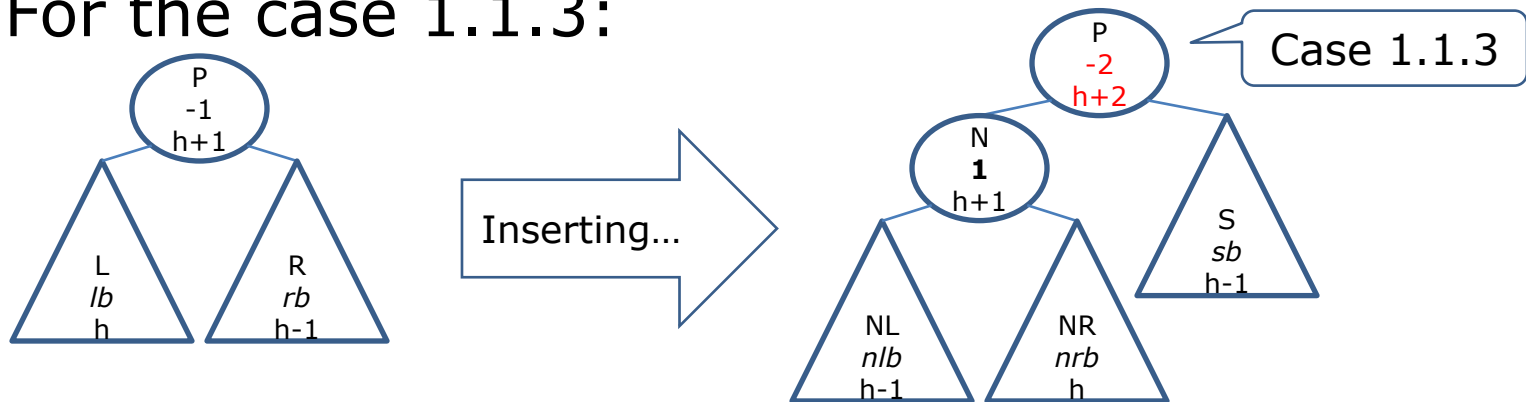
Insert and then rebalance (continue...)

- For the case 1.1.2, we can use the similar method to rebalance the sub-tree:



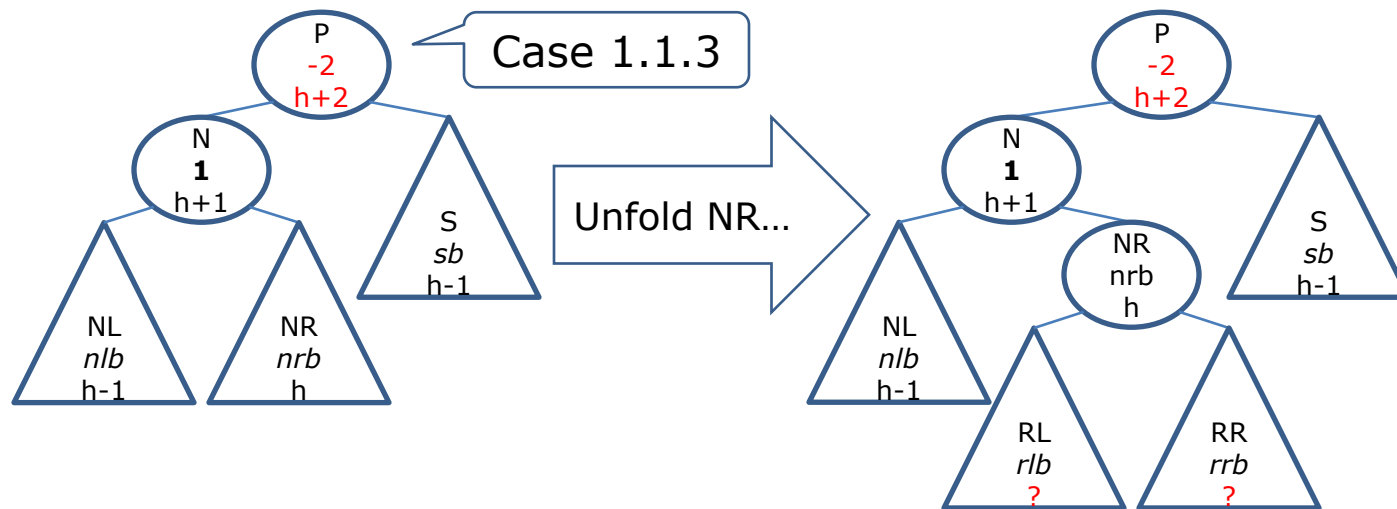
Insert and then rebalance (continue...)

- The sub-tree is regular now, but the height of the sub-tree is increased by one, so the recursive process continues;
- For the case 1.1.3:



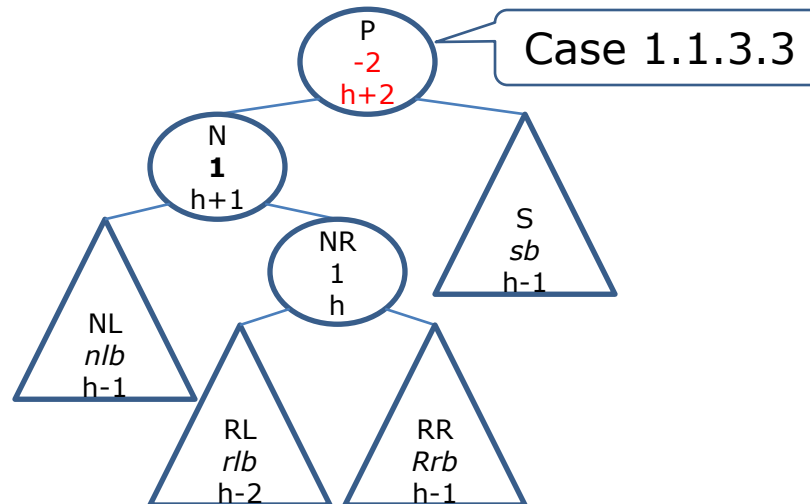
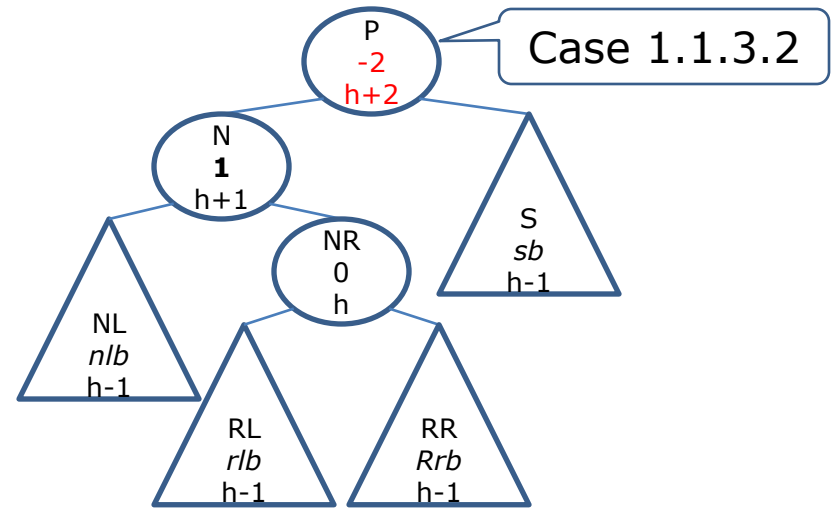
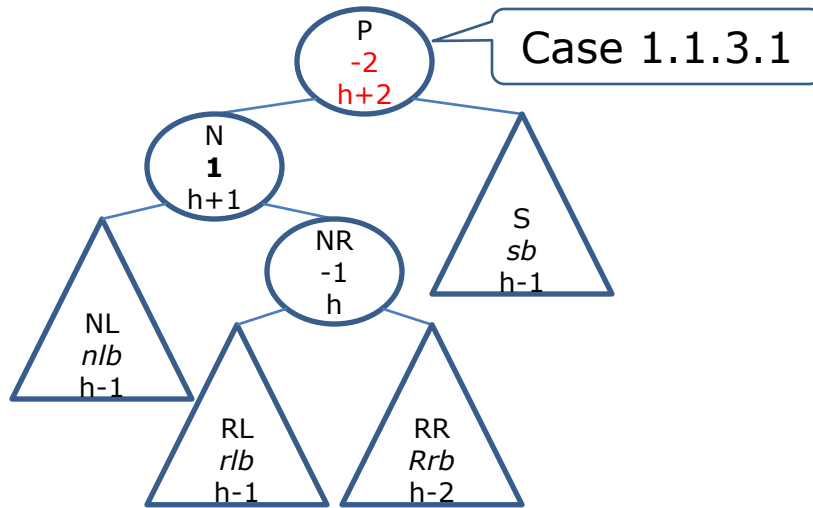
- We unfold the sub-tree NR first (we can do this because its sibling NL 's height $h-1 \geq 0$, we may not be able to do this on NL because NL 's height may be 0).

Insert and then rebalance (continue...)



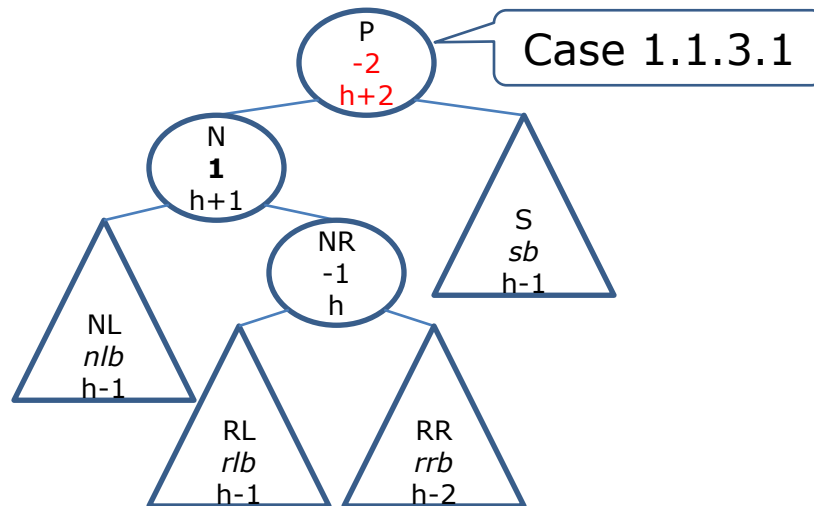
- We must know NR's balance factor first and then we know the height of its two children RL and RR, so we get other three sub-cases:

Insert and then rebalance (continue...)



Insert and then rebalance (continue...)

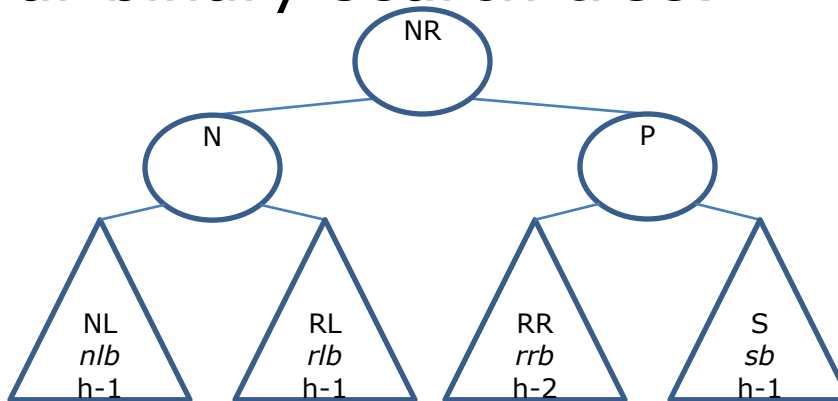
- For the case 1.1.3.1:



- 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 AVL sub-tree?

Insert and then rebalance (continue...)

- We can reorganize the seven pieces to get such a regular binary search tree:

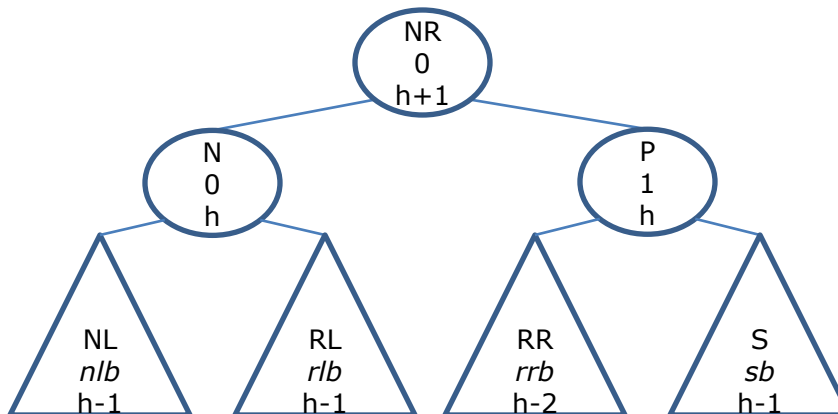


- The binary search tree has the following traits:
 - all the sub-trees NL, RL, RR and S are regular AVL sub-trees (no rule is broken in them);
 - the height of NL, RL and S is $h-1$, RR's height is $h-2$;
 - until now the balance factor and the height of the nodes N, P and NR are not determined.

Insert and then rebalance

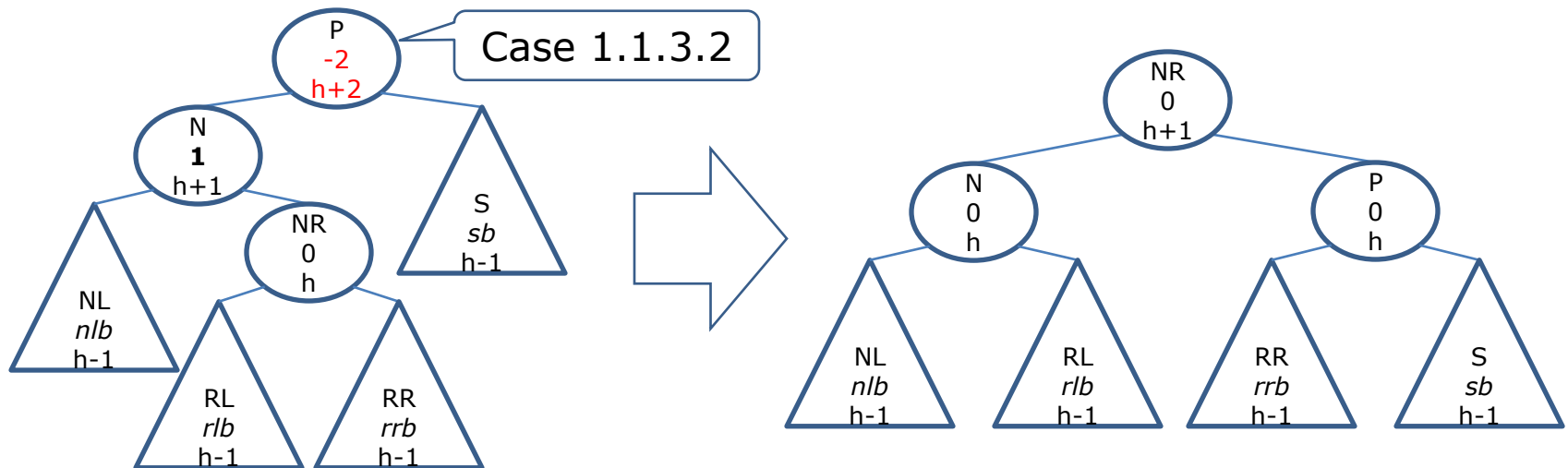
(continue...)

- NL and RL are N's children, their height is $h-1$, so N's height is h , N's balance factor is 0;
- RR and S are P's children, RR's height is $h-2$, S's height is $h-1$, so P's height is h , P's balance factor is 1;
- N and P are NR's children, NR's height is $h+1$, NR's balance factor is 0:

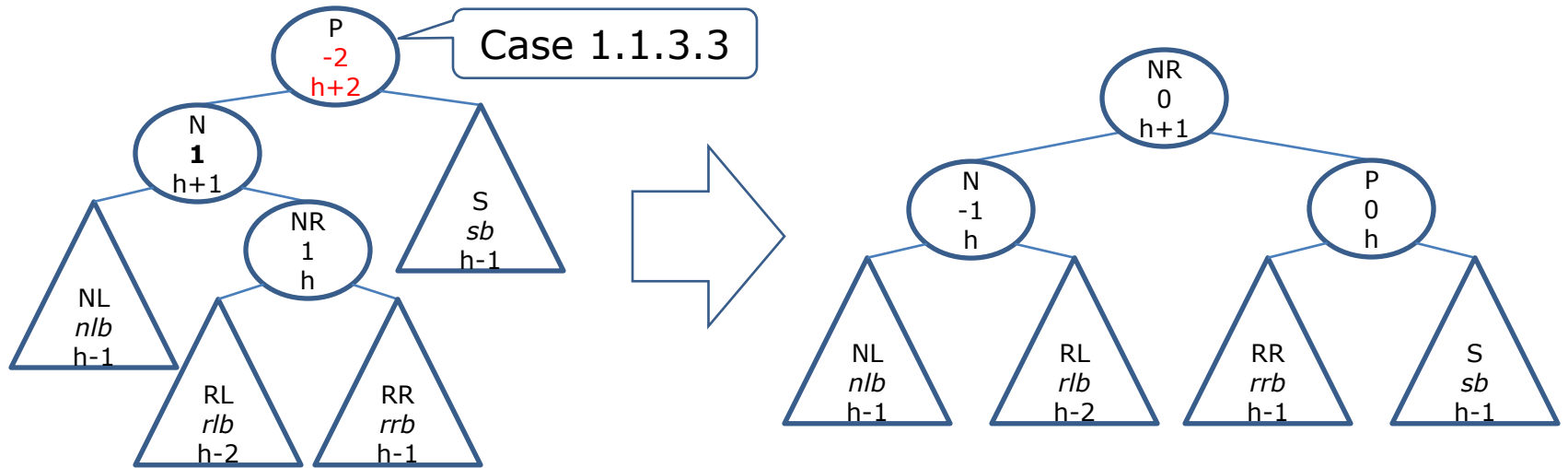


Insert and then rebalance (continue...)

- We rebuild a regular AVL sub-tree and the height of the sub-tree is $h+1$, which is the same as before. So the recursive process is finished;
- We can use the similar method to rebalance the sub-trees in other sub-cases 1.1.3.2 and 1.1.3.3:

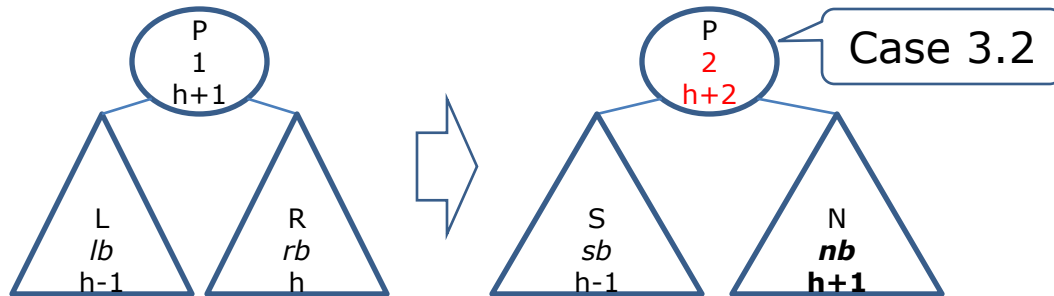


Insert and then rebalance (continue...)

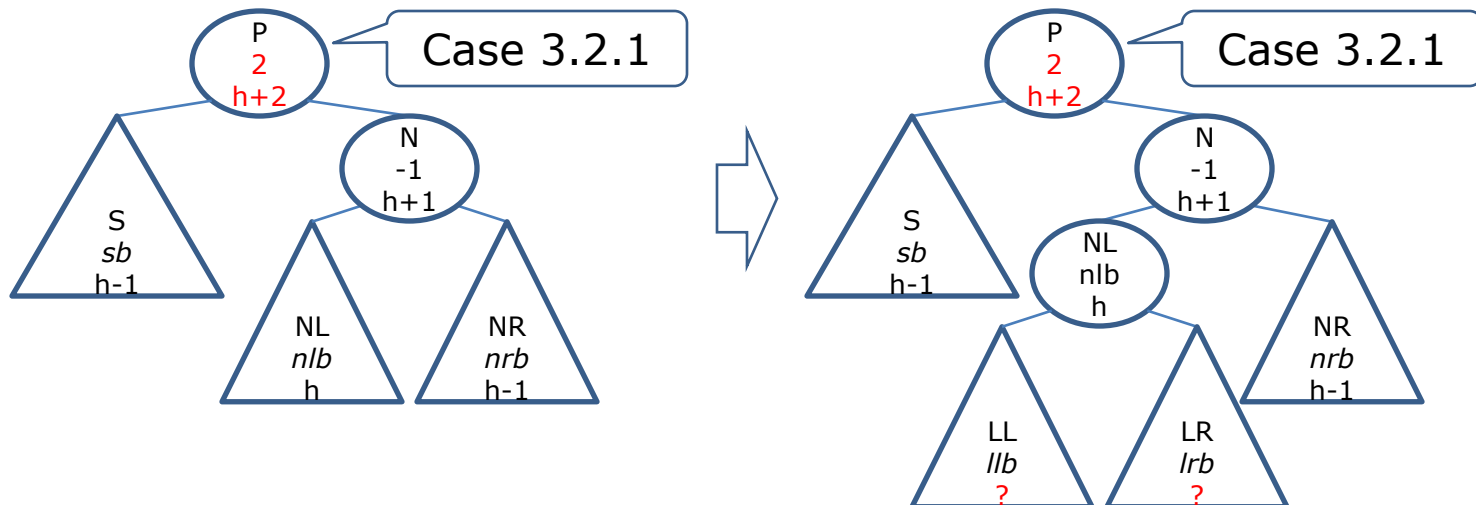


Insert and then rebalance (continue...)

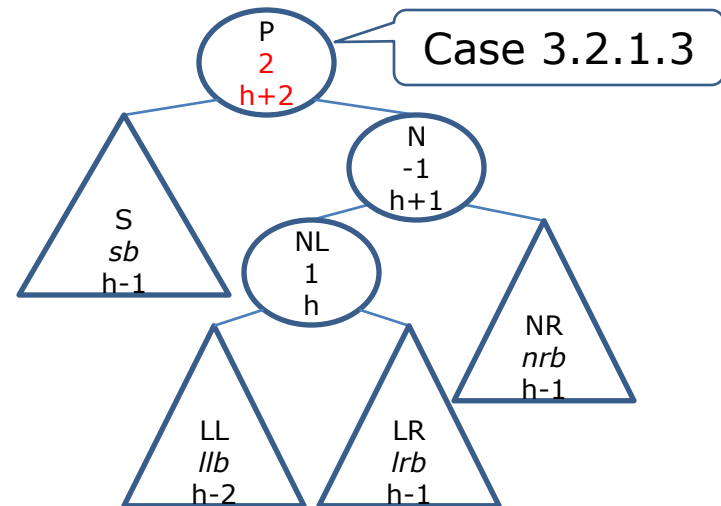
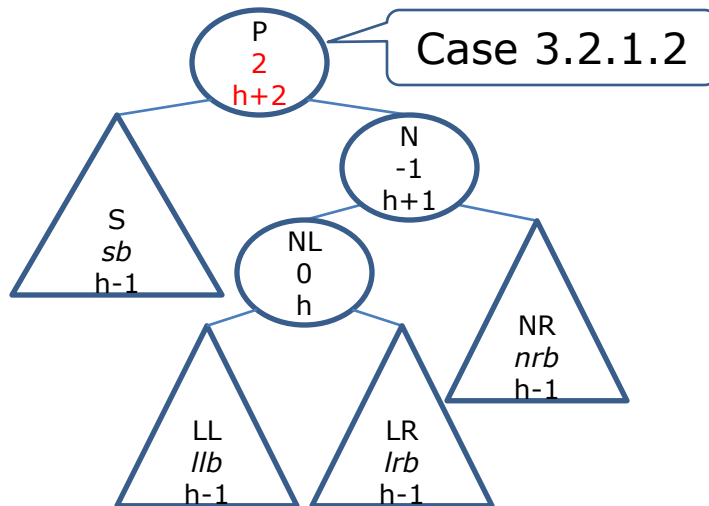
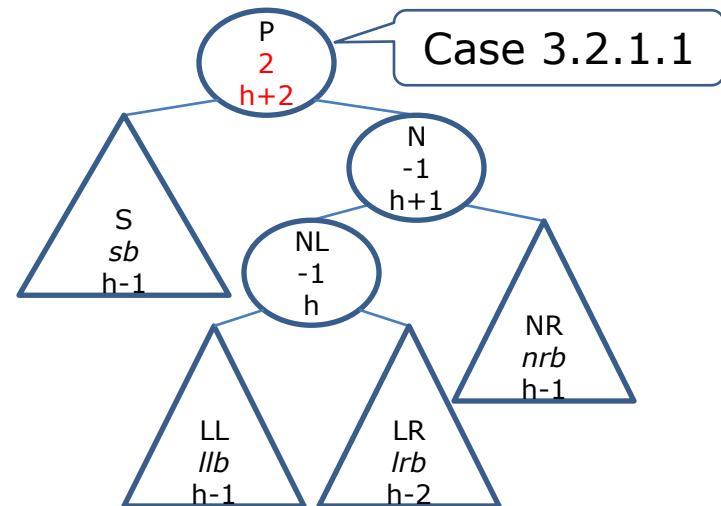
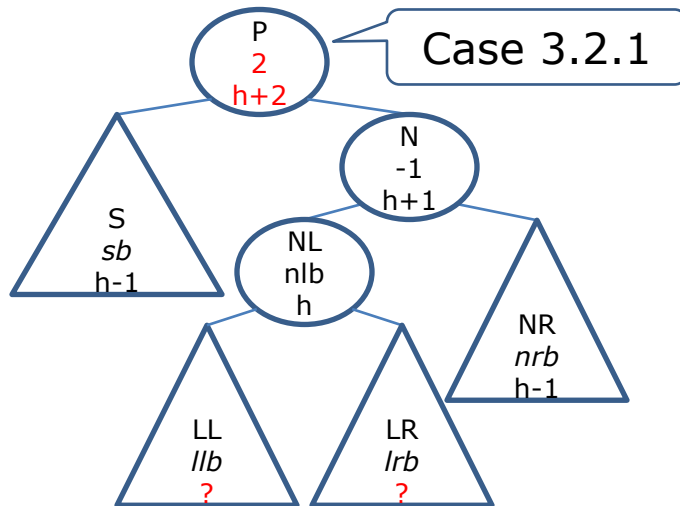
- For the case 3.2 :



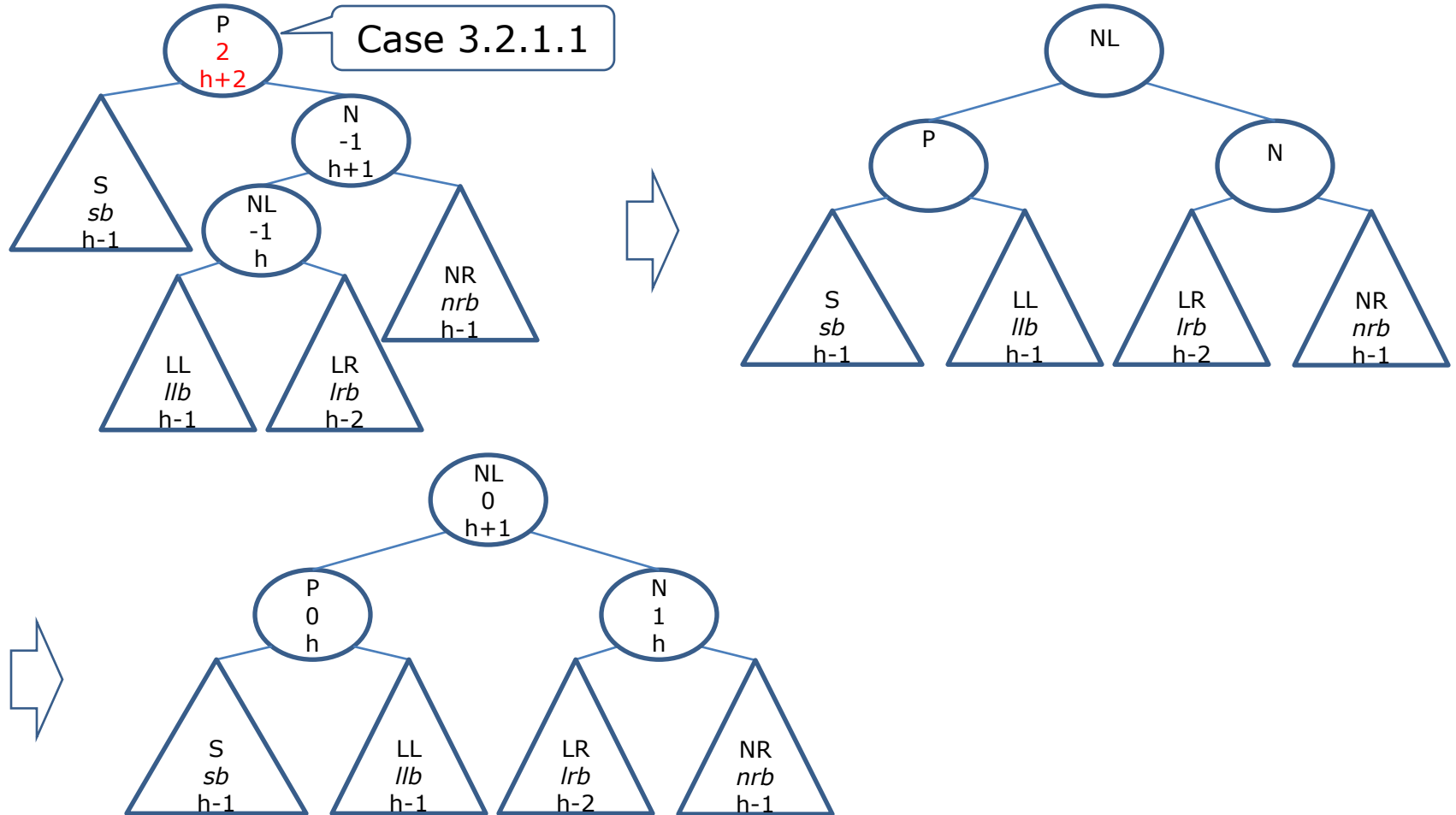
- We use the similar method to rebalance it:



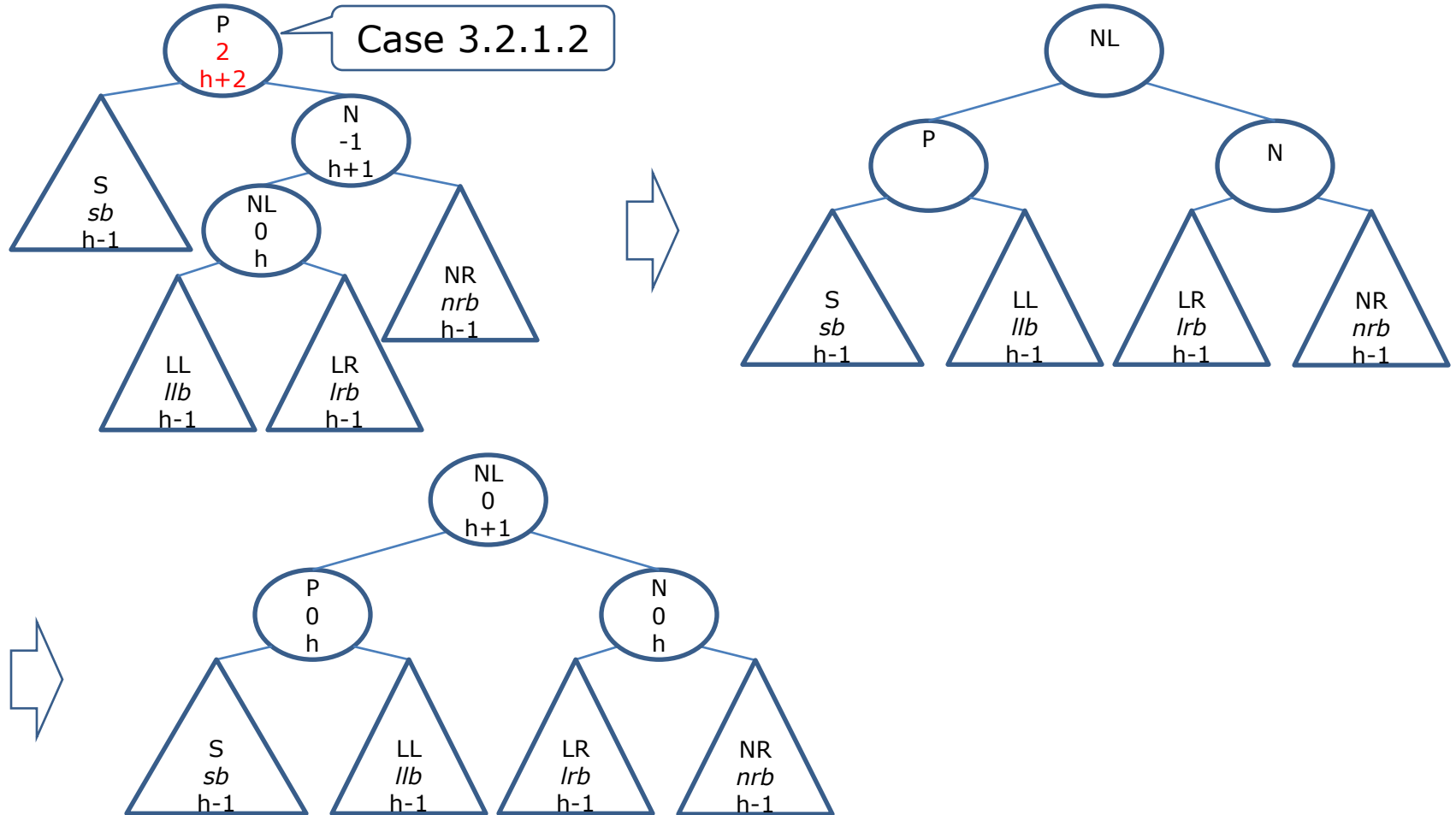
Insert and then rebalance (continue...)



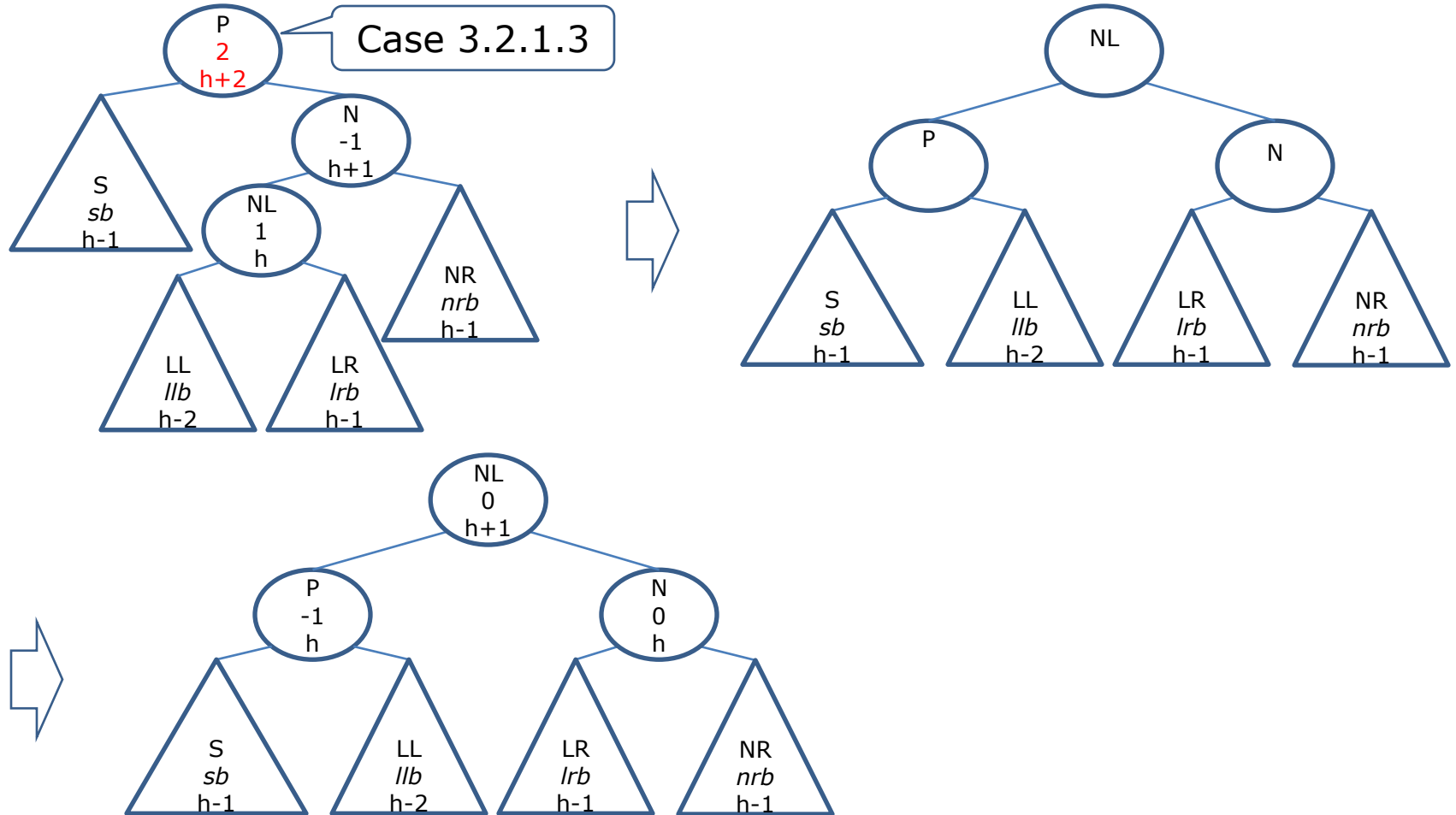
Insert and then rebalance (continue...)



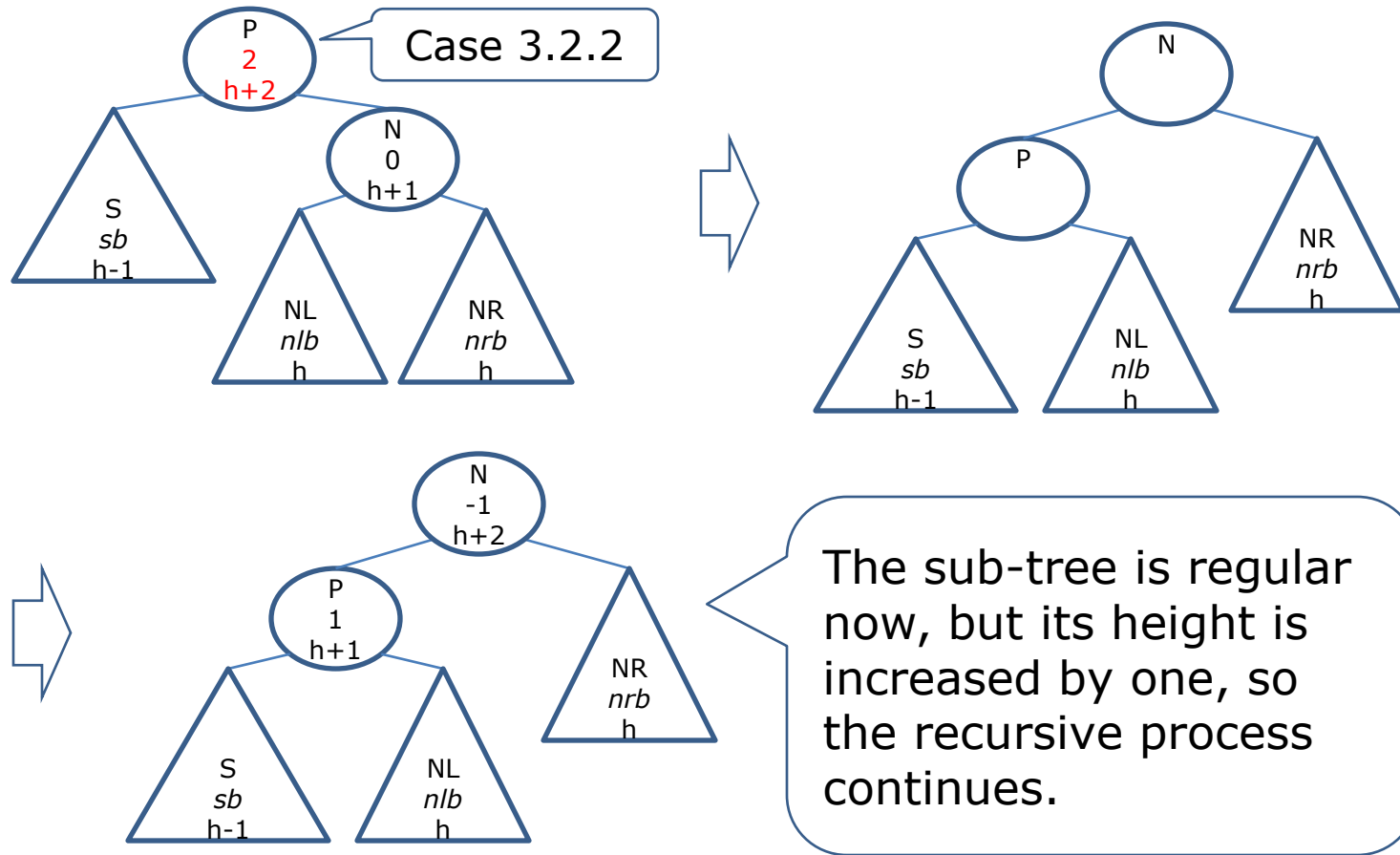
Insert and then rebalance (continue...)



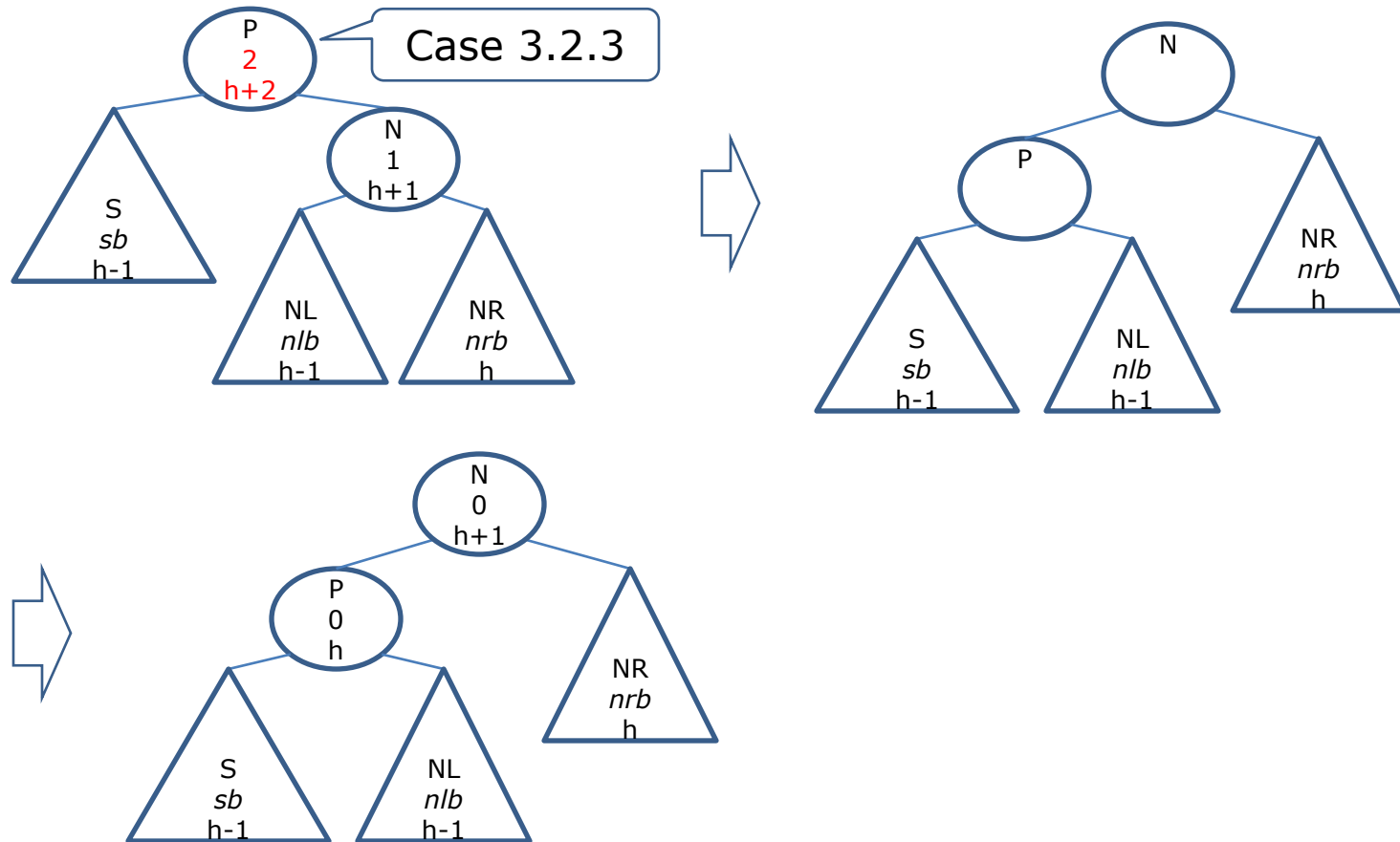
Insert and then rebalance (continue...)



Insert and then rebalance (continue...)

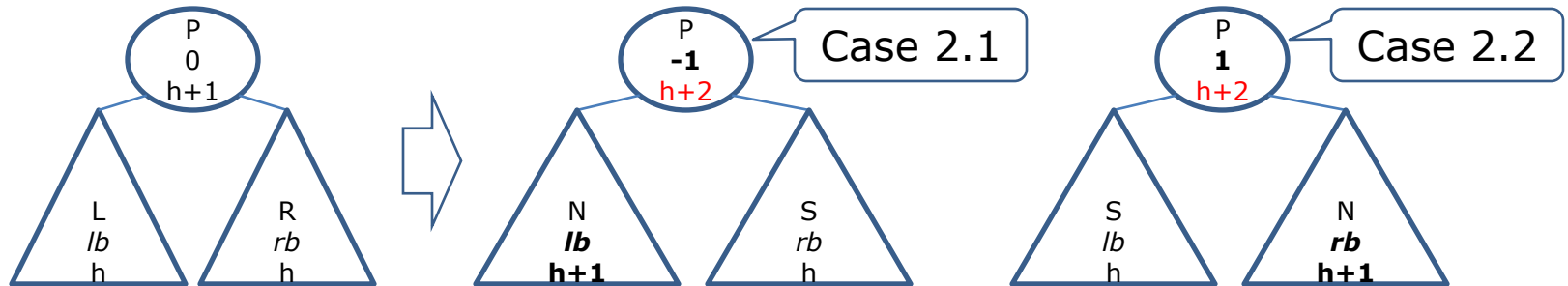


Insert and then rebalance (continue...)



Insert and then rebalance (continue...)

- For the cases 2.1 and 2.2 :



- The upper level sub-trees P are regular but its height is increased by one, it means that the recursive process continues.

Insert and then rebalance

(continue...)

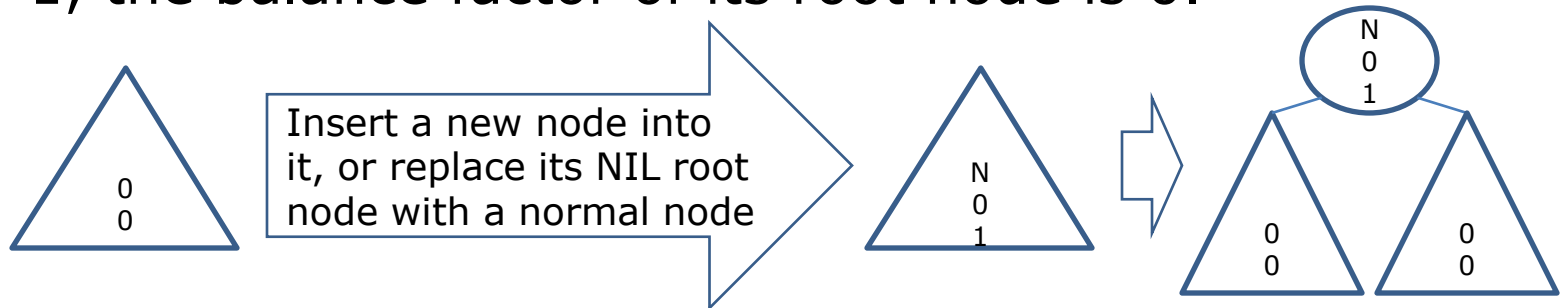
- In summary, the inserting and then rebalancing process is recursive:
 - the step 1: given a sub-tree **N** whose root node **N**'s height is increased by one. Note: this sub-tree is always regular before and after the change;
 - the step 2: if N is the root node of the whole tree, the process is finished;
 - the step 3: if N has parent, the height change causes that the balance factor of its parent P is increased (N is the right child) or decreased (N is the left child) by one, and then we may get twelve different irregular AVL sub-trees to rebalance:

Insert and then rebalance (continue...)

- For the sub-trees in the cases 1.1.3 (includes 1.1.3.1, 1.1.3.2, 1.1.3.3) and 3.2.1 (includes 3.2.1.1, 3.2.1.2, 3.2.1.3), we use the foregoing method to rebalance the sub-trees and then the process is finished;
- For the sub-trees in the cases 1.1.1 and 3.2.3, we use the foregoing method to rebalance the sub-trees and then the process is finished;
- For the sub-trees in the cases 1.1.2 and 3.2.2, we can rebalance them (which are N's parent) to get regular sub-trees, but their height is increased by one, so N's parent becomes **N**, we return to the step 1;
- For the sub-trees in the cases 2.1 and 2.2, N's parent becomes **N**, we return to the step 1.

Insert and then rebalance (continue...)

- the base case is: we select an empty sub-tree (or tree) and replace it with such a sub-tree: its height is 1, the balance factor of its root node is 0.



Remove and then rebalance

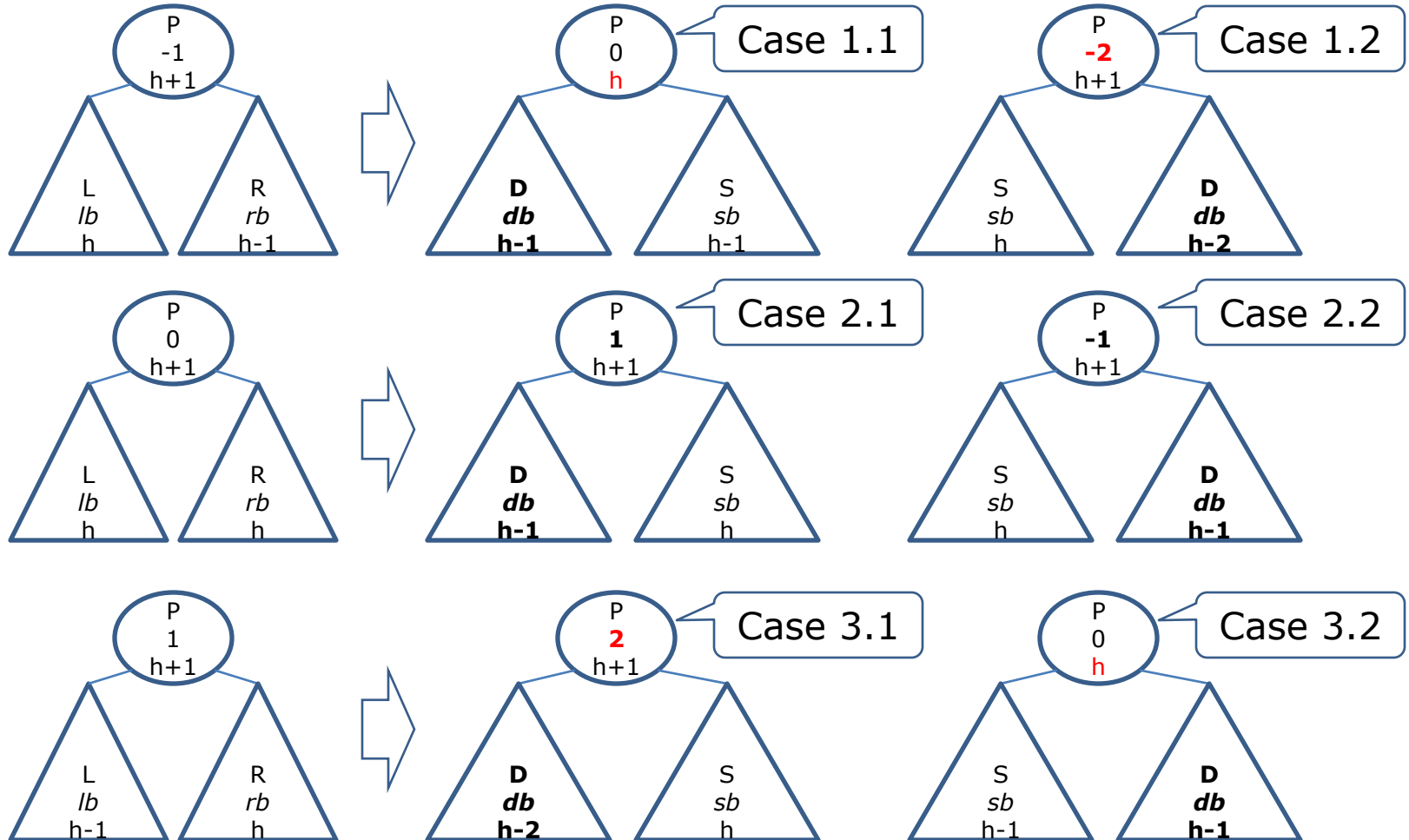
- We always remove such a node from a non-empty tree: its children are two NIL nodes;
- Obviously the node **D** itself forms a sub-tree. After it has been removed, the corresponding sub-tree becomes an empty sub-tree, its height is decreased from h (1) to $h-1$ (0);
- If D has parent, because D's height is decreased by one, its parent P's balance factor will be increased by one (if N is P's left child) or be decreased by one (if N is P's right child).

Remove and then rebalance

(continue...)

- D itself is still a regular sub-tree, but its parent P may be irregular. We may need to rebalance the sub-tree P;
- At first P's balance factor belonged to $\{-1, 0, 1\}$, after that it belongs to $\{-2, -1, 0, 1, 2\}$. More specifically, if P's balance factor was -1, it may be -2 or 0; if it was 0, it may be -1 or 1; if it was 1, it may be 0 or 2;
- We should know how many irregular sub-trees may be generated from the foregoing change.

Remove and then rebalance (continue...)



Remove and then rebalance

(continue...)

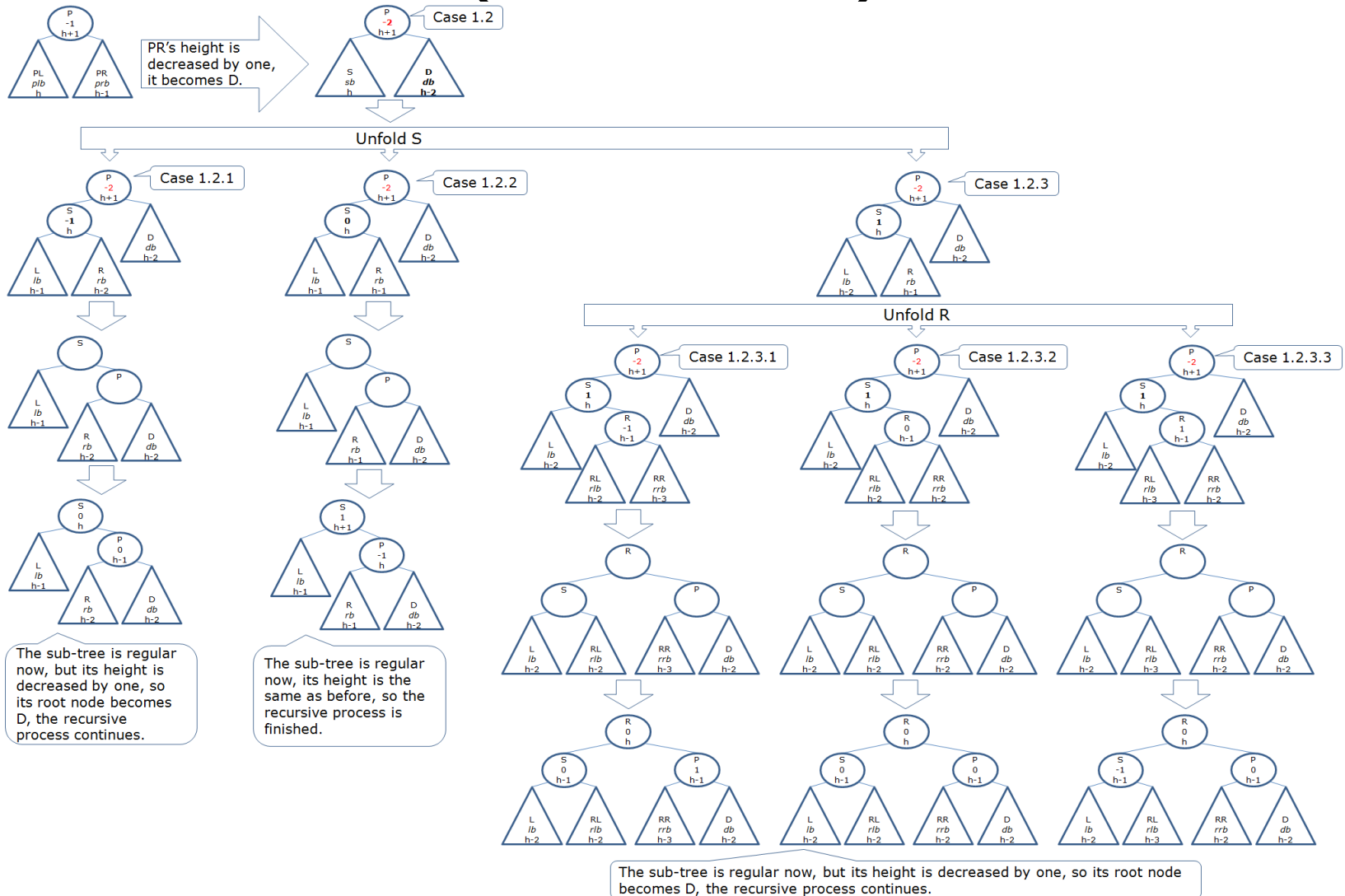
- In the cases 1.2 and 3.1 the sub-trees P are irregular, because P 's balance factor is -2 or 2 ;
- In the cases 2.1 and 2.2 the sub-trees P are regular, P 's height is not changed, and P 's parent is regular too (if there is). The removal does not cause that any rule is broken. No more action is required;
- In the cases 1.1 and 3.2 the sub-trees P are regular, but P 's height is decreased by one, so it demonstrates that the removing and then rebalancing process is recursive.

Remove and then rebalance

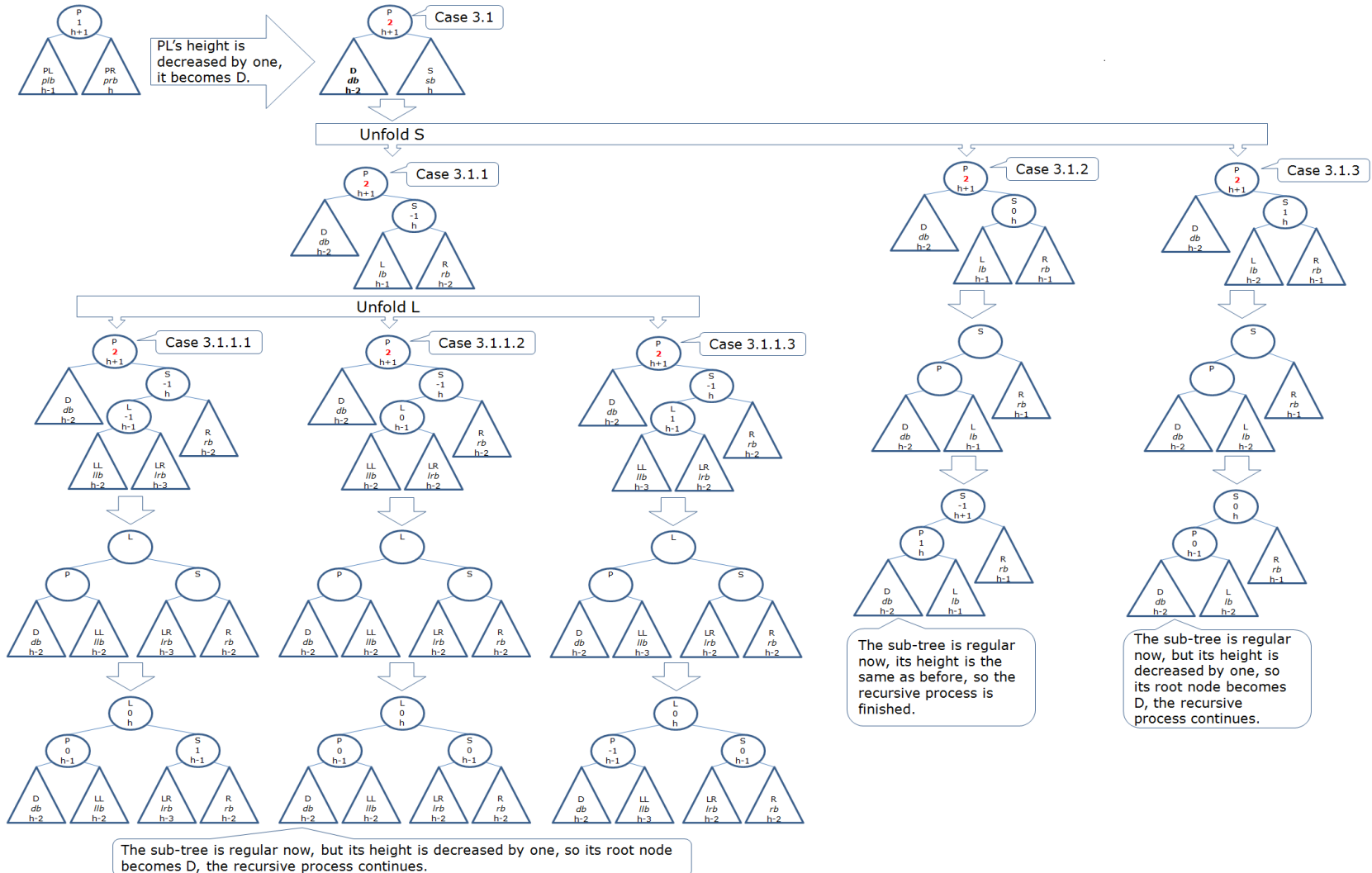
(continue...)

- If you have understood the inserting and then rebalancing process, you will find that the removing and then rebalancing process is extraordinarily similar;
- For brevity, I use pictures to describe the process without more explanation.

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 sub-tree **D** whose root node **D**'s height is decreased by one. Note: this sub-tree is always regular before and after the change;
 - the step 2: if D is the root node of the whole tree, the process is finished;
 - the step 3: if D has parent, the height change causes that the balance factor of its parent P is increased (N is the left child) or decreased (N is the right child) by one, and then we may get twelve different irregular AVL sub-trees to rebalance:

Remove and then rebalance (continue...)

- For the sub-trees in the cases 1.2.3 (includes 1.2.3.1, 1.2.3.2, 1.2.3.3) and 3.1.1 (includes 3.1.1.1, 3.1.1.2, 3.1.1.3), we use the foregoing method to rebalance the sub-trees and then the process is finished;
- For the sub-trees in the cases 1.2.2 and 3.1.2, we use the foregoing method to rebalance the sub-trees and then the process is finished;
- For the sub-trees in the cases 1.2.1 and 3.1.3, we can rebalance them (which are D's parent) to get regular sub-trees, but their height is increased by one, so D's parent becomes **D**, we return to the step 1;
- For the sub-trees in the cases 1.1 and 3.2, D's parent becomes **D**, we return to the step 1.

Remove and then rebalance

(continue...)

- the base case is: we select a sub-tree whose height is 1 and whose root node's balance factor is 0 to remove, and then the sub-tree D becomes an empty sub-tree, its root (NIL) node is the node D.

Code

- In this website <https://github.com/cyril-gao/wheel/tree/master/Algorithms/BST>, you can find C++ code and Python code which implement the algorithm.