The rebalancing process of AVL trees

Yuanming Gao
yuanming.gao@gmail.com

Agenda

- Definition
- Basic assumptions
- Insert a new node and then rebalance the tree
- Remove a node and then rebalance the tree

Note

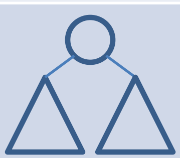
- Make sure that you have read the chapter 12 of the book 《Introduction to Algorithm》(third edition) before you continue reading this article;
- I do not talk about how to insert/remove a node into/from a binary tree, you may find details in the foregoing chapter (12.3 Insertion and deletion);
- The 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:
  - BalanceFactor(*node*) := Height(RightSubtree(*node*)) - Height(LeftSubtree(*node*))
- A binary search tree is defined to be an AVL tree if the invariant BalanceFactor(*node*) ∈ {-1, 0, 1} holds for every node in the tree;
- The balance factor of a NIL node is 0.

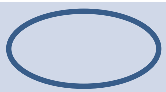
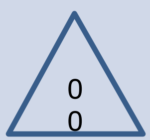
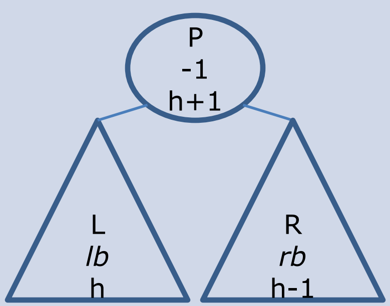
Basic assumptions

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

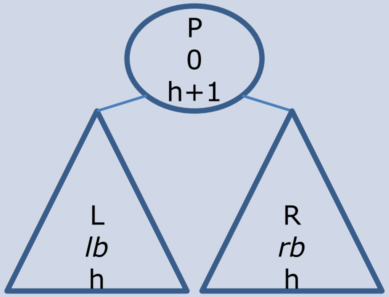
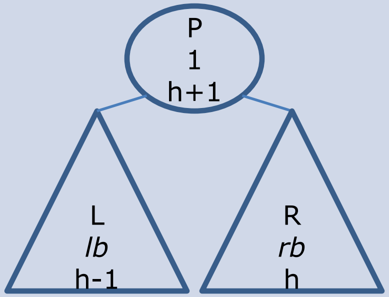
Figure	Meaning
	A node in a binary tree.
	A binary tree, sub-tree, or empty tree. Note: it can be used to represent a tree or sub-tree in which there is only one node.
	A binary tree or sub-tree which has a root node and two sub-trees (a non-empty tree).

Basic assumptions (continue…)

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

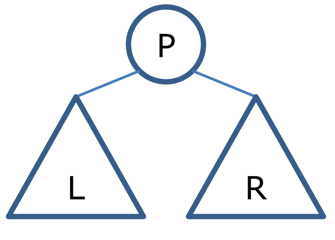
Figure	Meaning
	A normal (non-NIL) node in an AVL tree.
	An empty AVL tree. Its height is 0, the balance factor of the root node of the tree is 0.
	An AVL tree or sub-tree whose root node P's balance factor is -1, P's height is h+1 (h>=0). The height of P's left sub-tree L is h, the root node of the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

Basic assumptions (continue…)

Figure	Meaning
	An AVL tree or sub-tree whose root node P's balance factor is 0, P's height is h+1 (h>=0). The height of P's left sub-tree L is h, the root node of the left sub-tree L's balance factor lb ∈ {-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 ∈ {-1, 0, 1}.
	An AVL tree or sub-tree whose root node P's balance factor is 1, P's height is h+1 (h>=0). The height of P's left sub-tree L is h-1, the root node of the left sub-tree L's balance factor lb ∈ {-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 ∈ {-1, 0, 1}.

Basic assumptions

- Naming convention: if we name a tree (sub-tree or empty tree) **X**, usually we name its root node **X** too. The reverse is also the same;
- So you should understand what do they mean when we say the node P, the sub-tree P, the node L, the sub-tree L, …, etc:



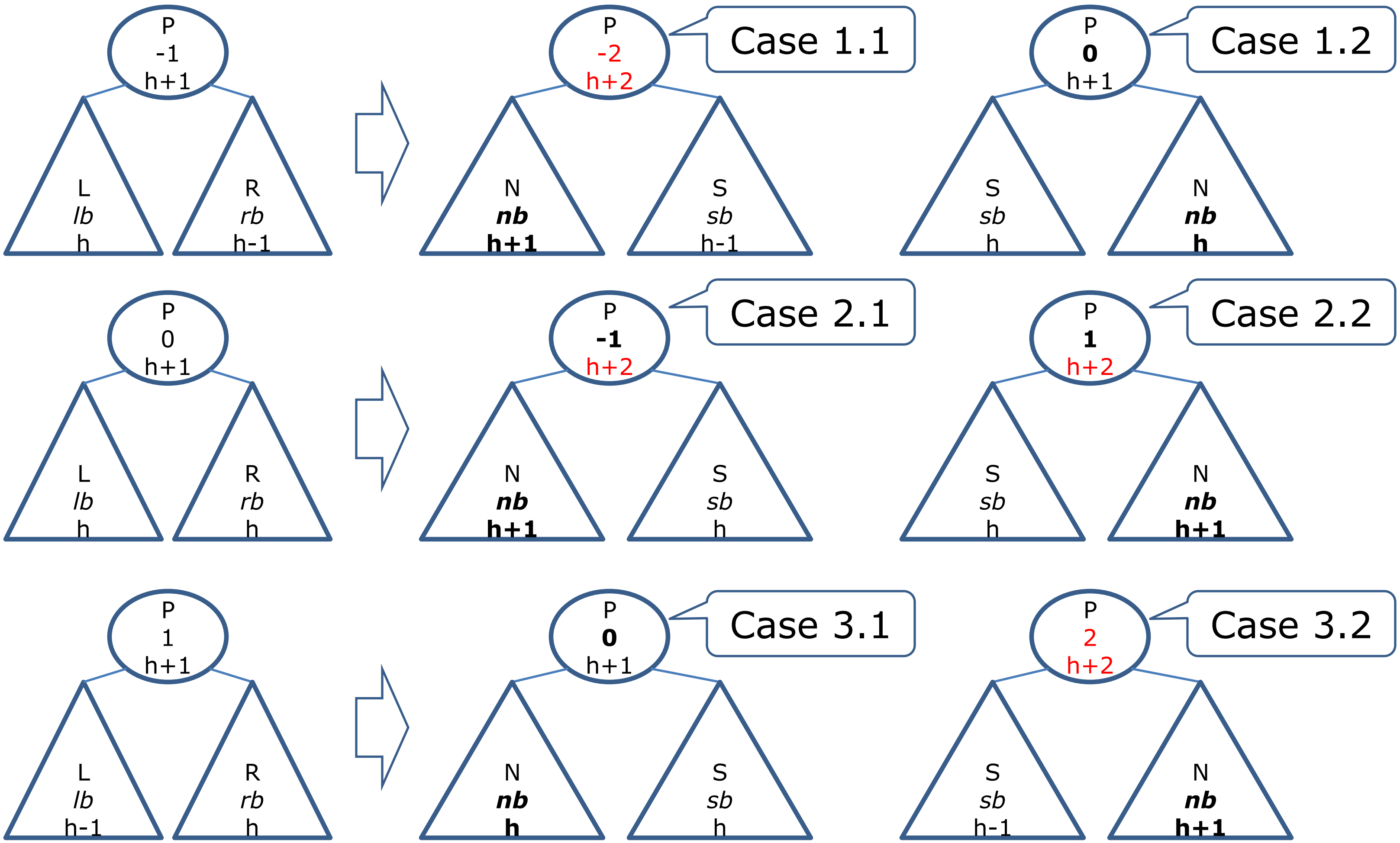
Insert and then rebalance

- 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 lever 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…)

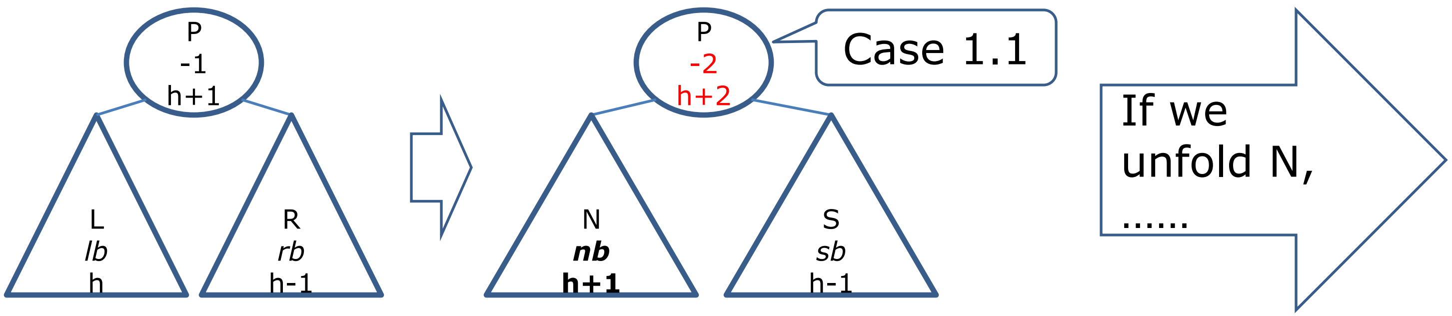


Insert and then rebalance (continue…)

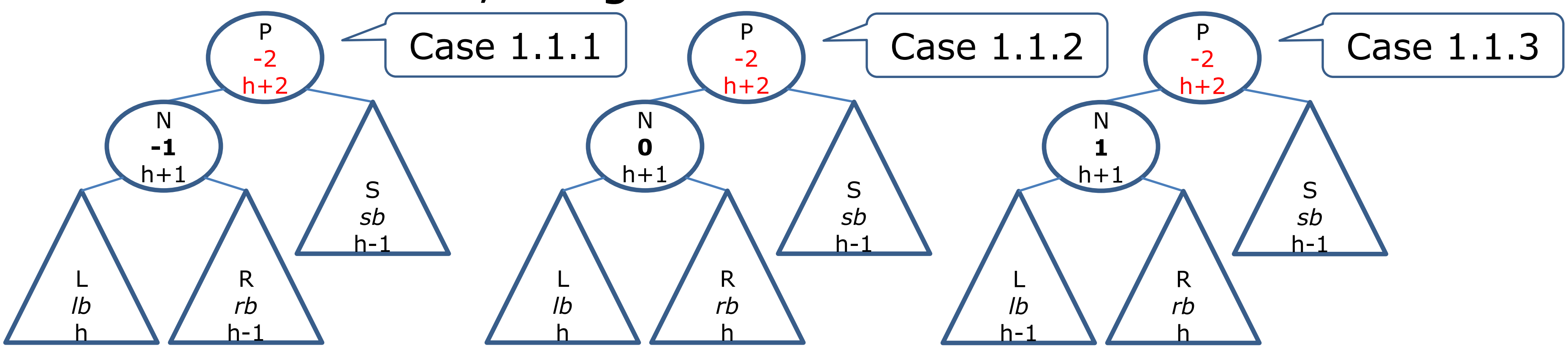
- In the cases 1.1 and 3.2 the sub-trees P are irregular, because P’s balance factor is -2 or 2;
- In the cases 1.2 and 3.1 the sub-trees P are regular, P’s height is not changed, and P’s parent is regular too (if there is). The insertion does not cause that any rule is broken. No more action is required;
- 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:

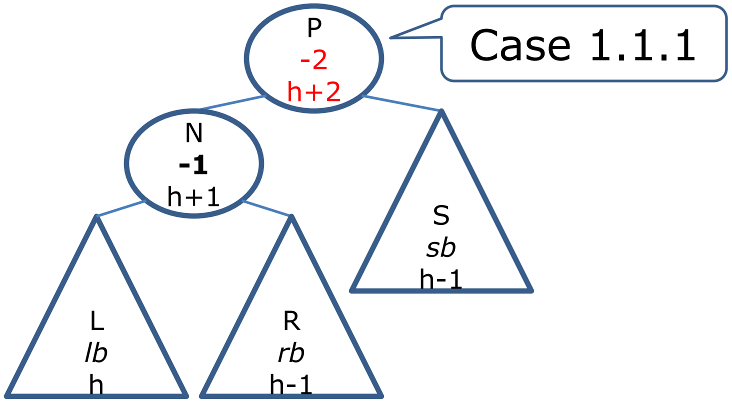


- We must remember: N is a regular sub-tree and N’s height is increased by one;
- If we unfold N, we get three different sub-trees:



Insert and then rebalance (continue…)

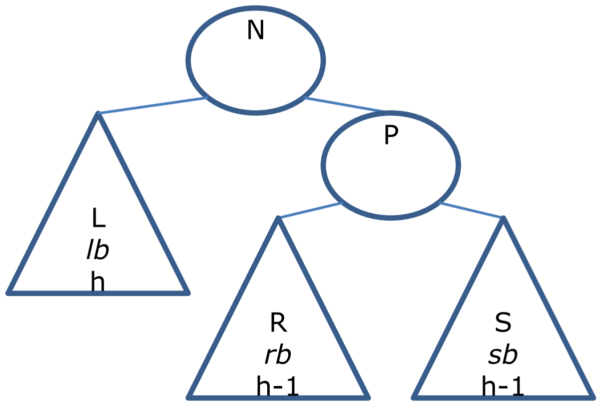
- For the case 1.1.1:



- 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…)

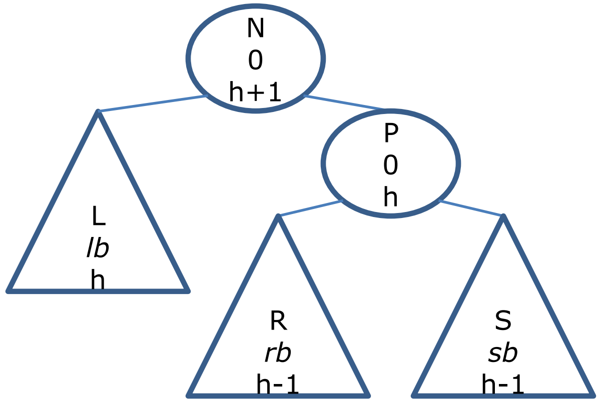
- First we can create a binary search tree like this:



- 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.

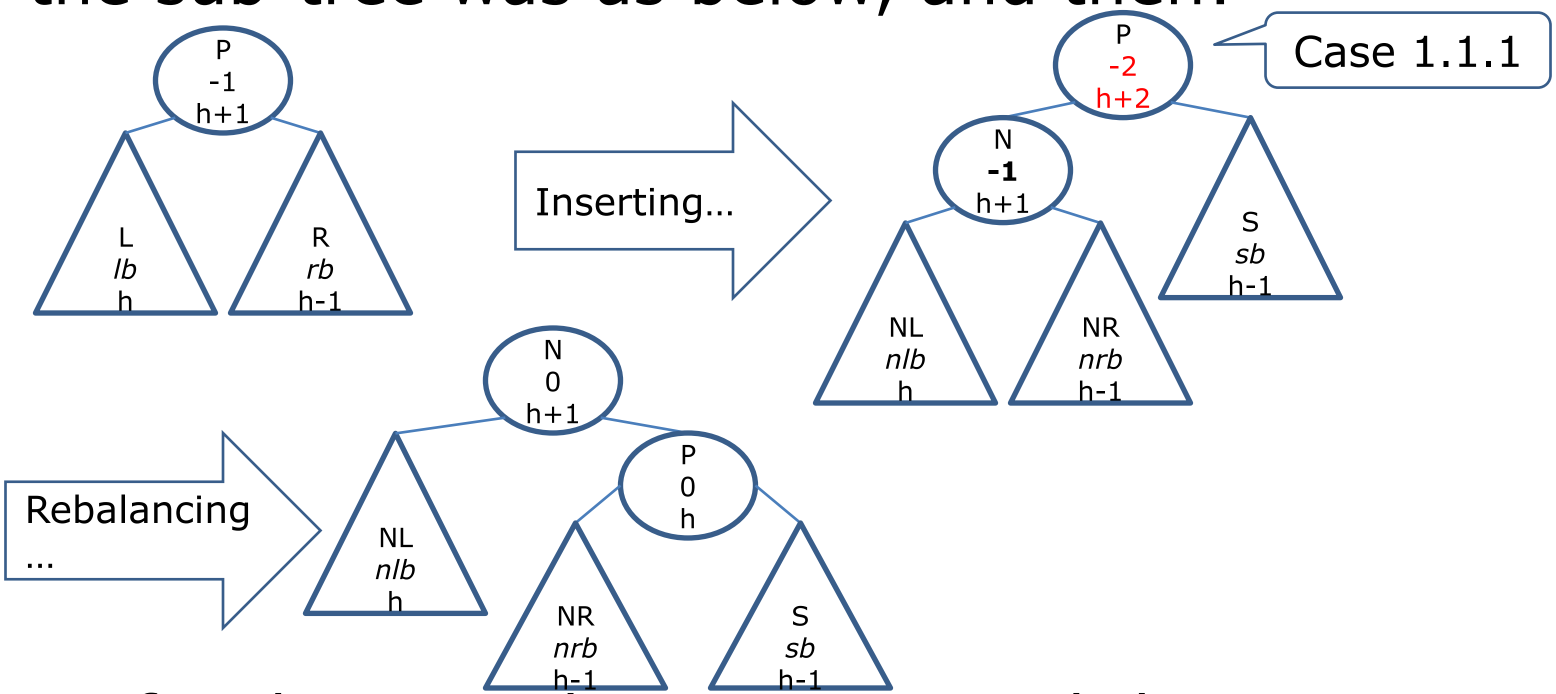
Insert and then rebalance (continue…)

- 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…)

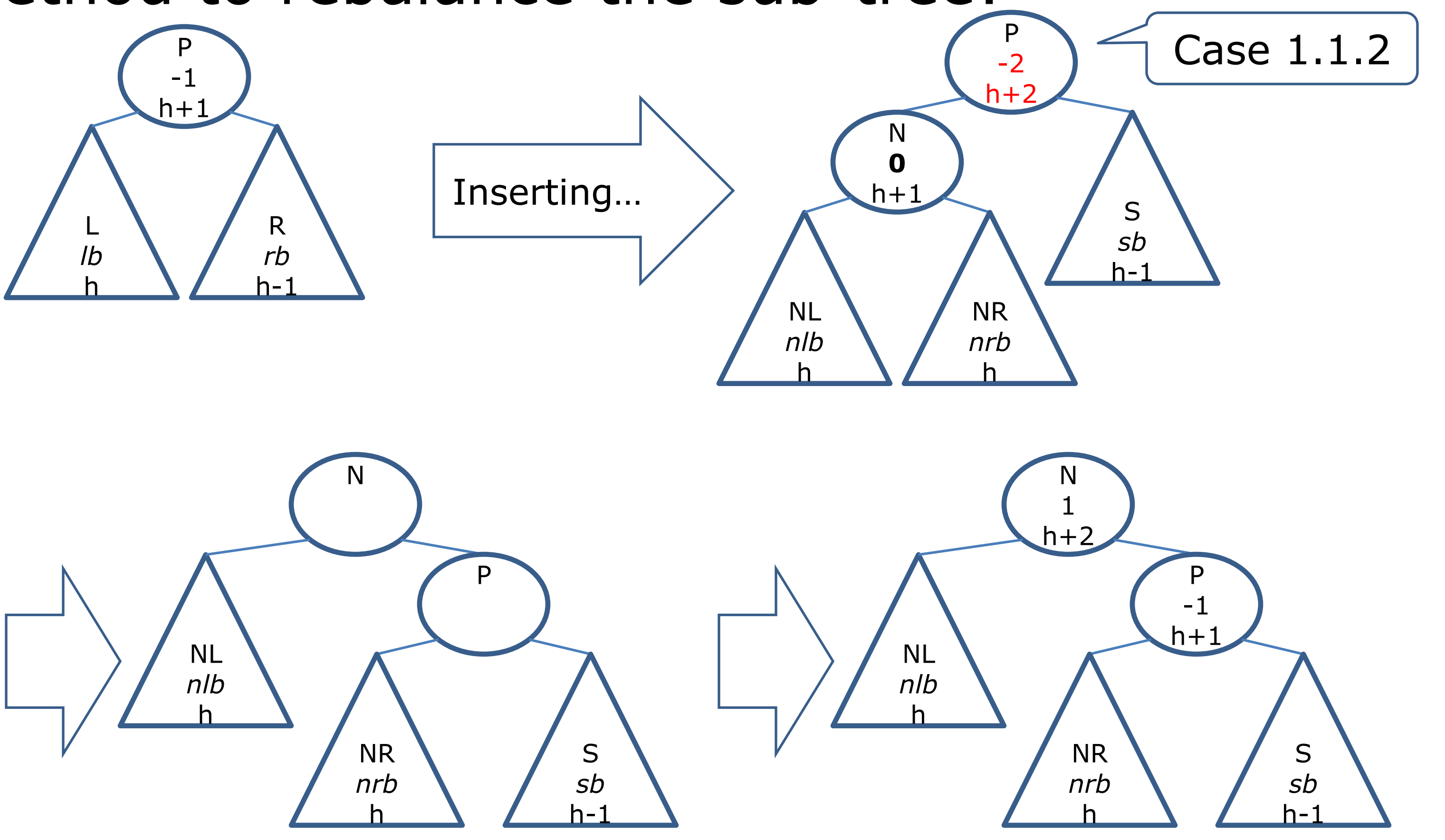
- Before the inserting and then rebalancing process, the sub-tree was as below, and then:



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

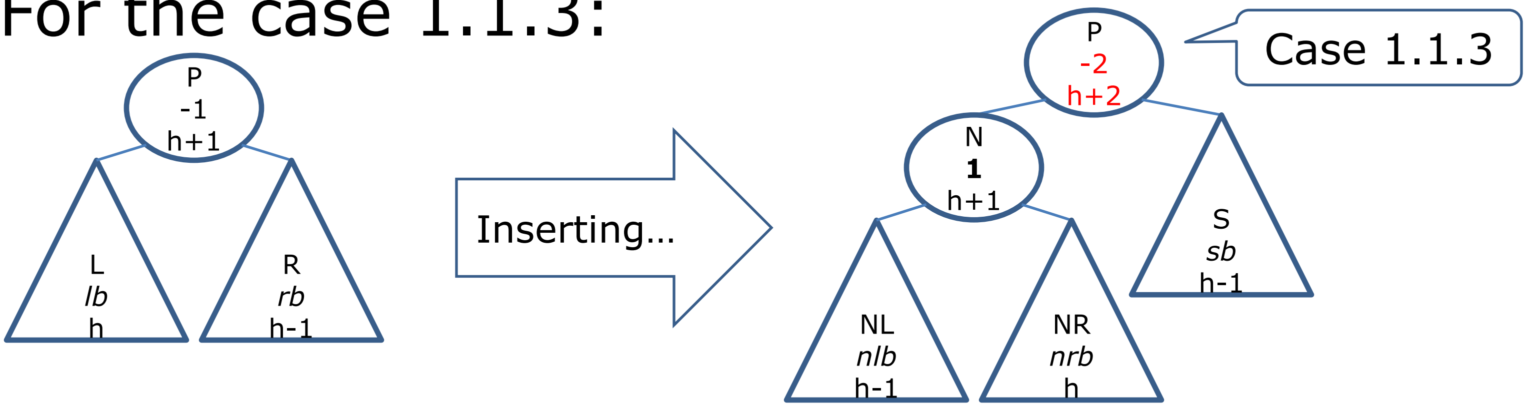
Insert and then rebalance (continue…)

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



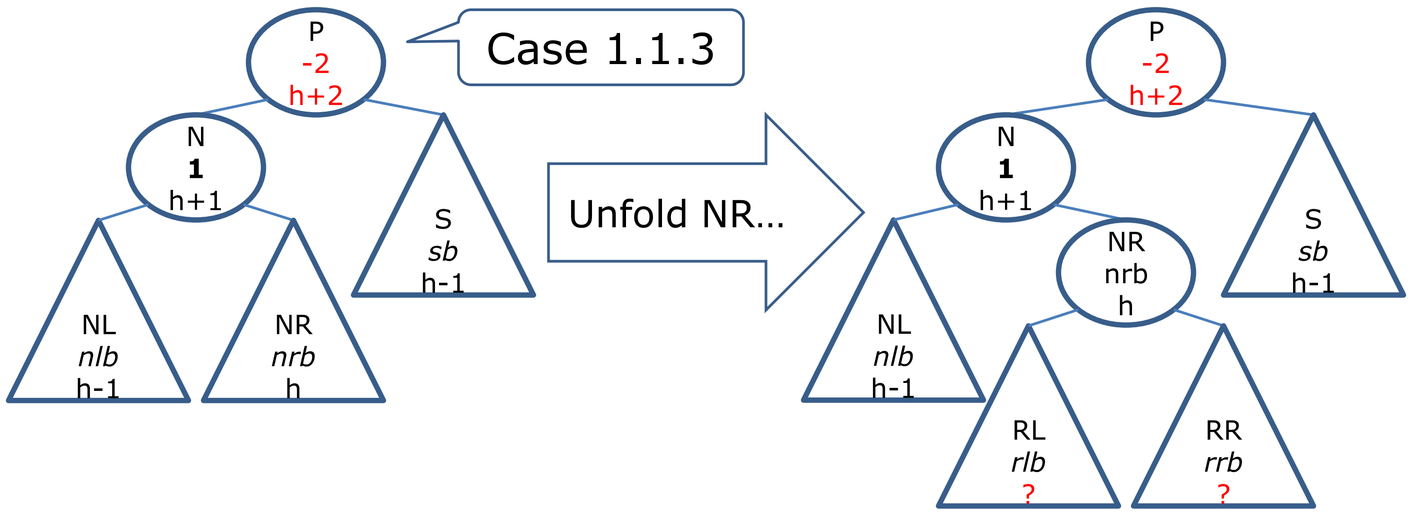
Insert and then rebalance (continue…)

- 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:



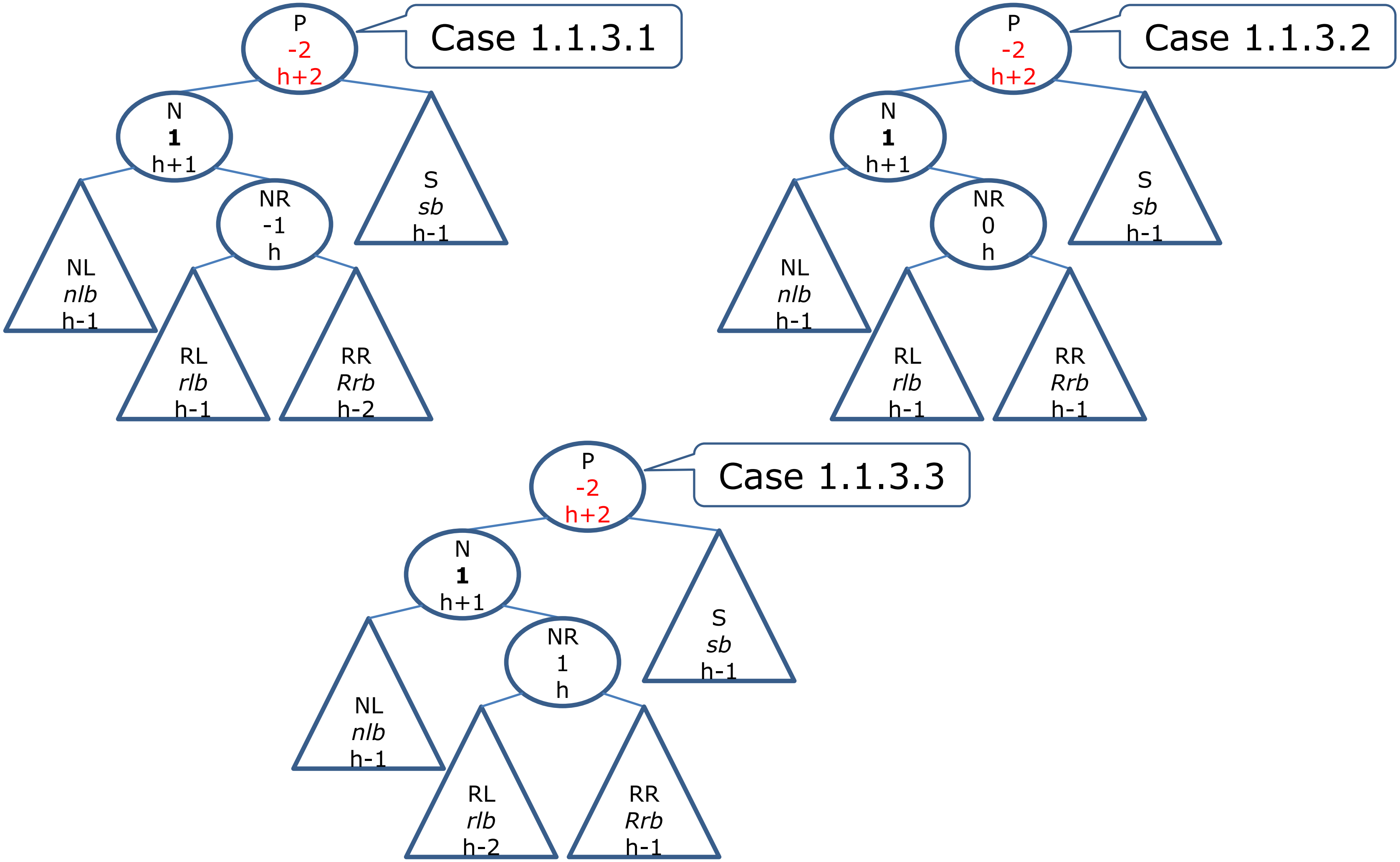
- We unfold the sub-tree NR first (we can do this because its sibling NL's height h-1 >= 0, we may not be able to do this on NL because NL's height may be 0).

Insert and then rebalance
(continue…)



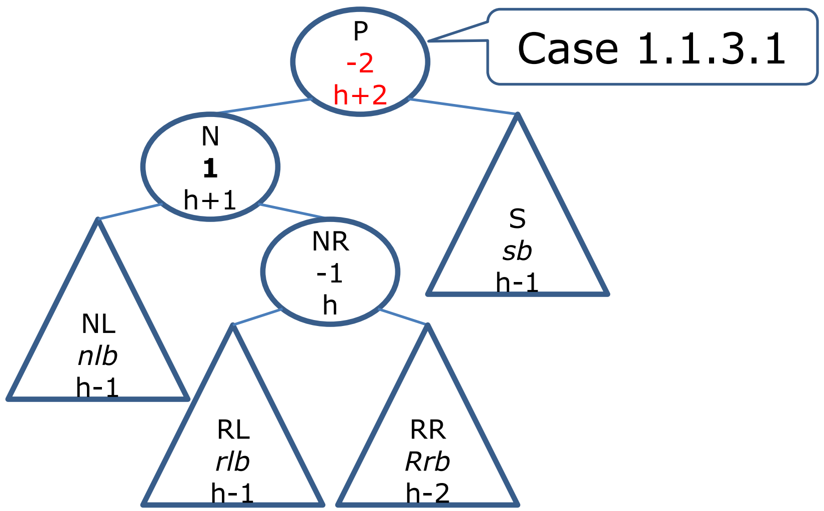
- We must know NR’s balance factor first and then we know the height of it tow children RL and RR, so we get other three sub-cases:

Insert and then rebalance
(continue...)



Insert and then rebalance (continue…)

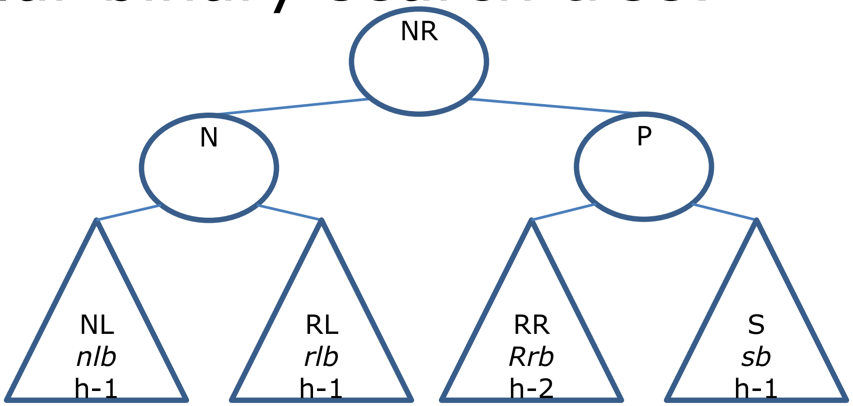
- For the case 1.1.3.1:



- 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…)

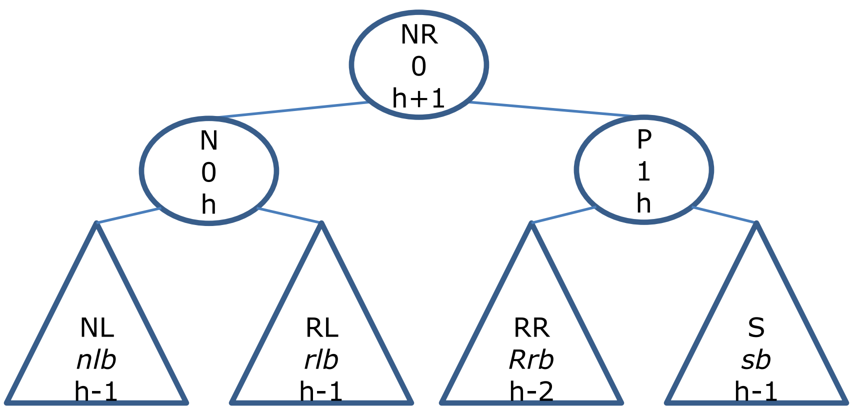
- We can reorganize the seven pieces to get such a regular binary search tree:



- 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.

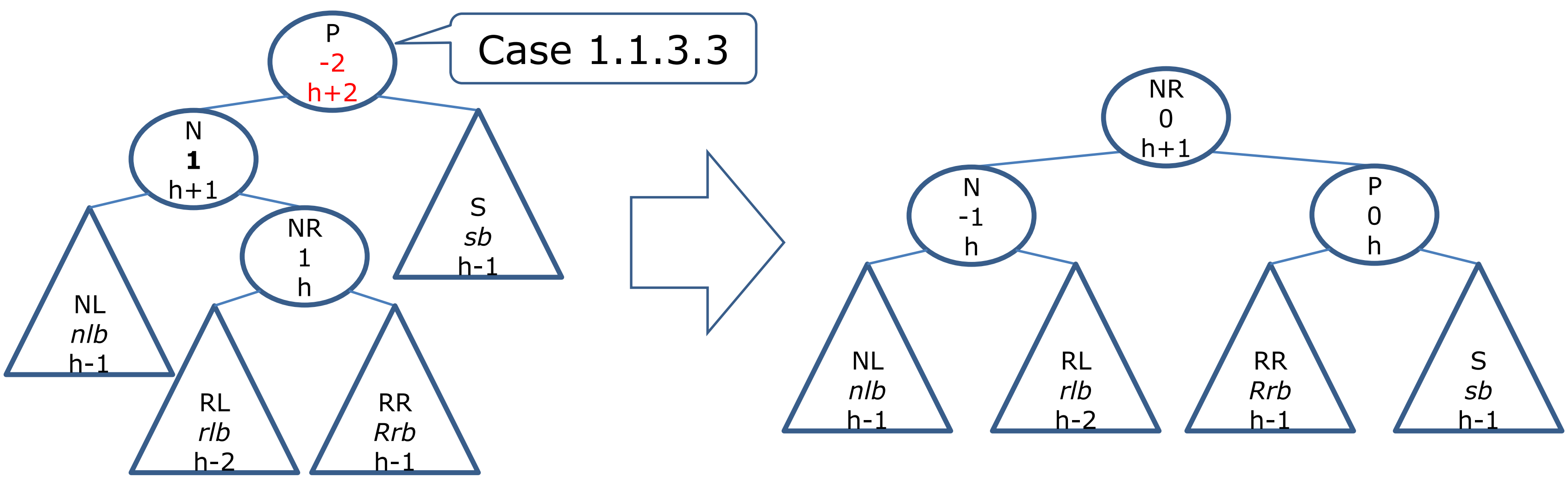
Insert and then rebalance (continue…)

- 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:



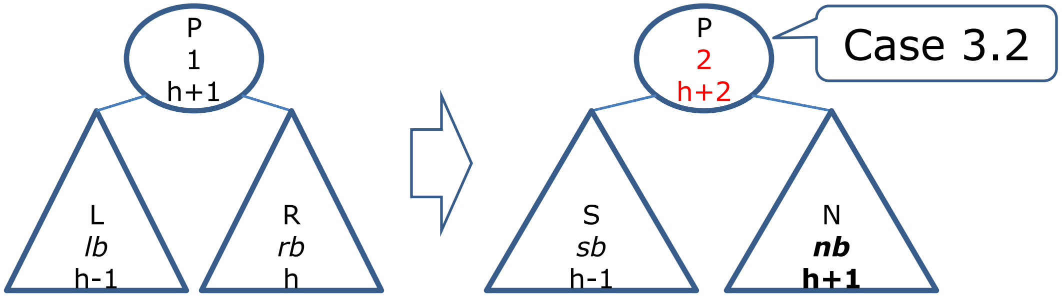
Insert and then rebalance (continue…)

- 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 is finished;
- We can use the similar method to rebalance the other sub-cases 1.1.3.2 and 1.1.3.3:



Insert and then rebalance (continue…)

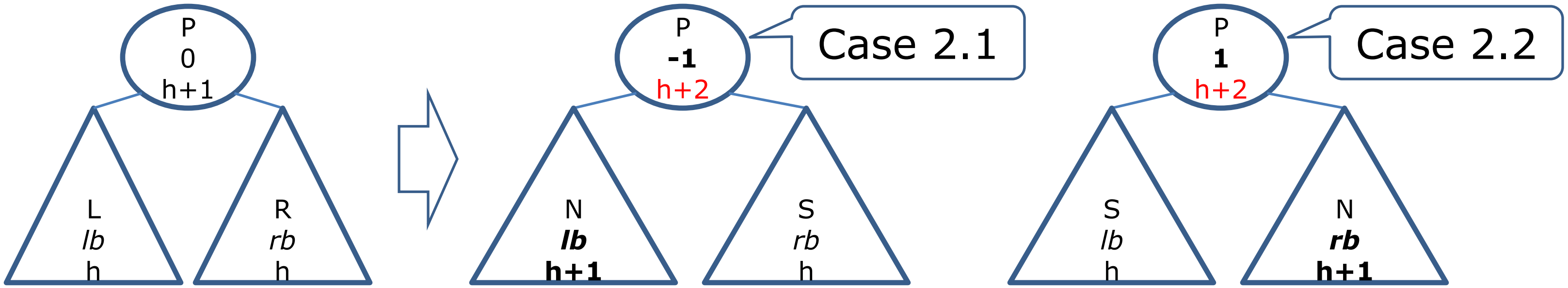
- For the case 3.2 :



- We use the similar method to rebalance it:

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

- 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, we call it the node N. 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 types of irregular AVL sub-trees to rebalance.

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 one of many sub-trees;
- How many?

Remove and then rebalance (continue…)

- for the sub-trees 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 1.2, 1.3, 1.4, 2.2, 2.3, 2.4, we use the foregoing method to rebalance them to get conforming sub-trees, and then the process is finished;
- the sub-tree 3 can be transformed into the sub-tree 6 (strictly speaking, after the transformation, the resulted sub-tree is part of the sub-tree 6);
- the sub-tree 4 can be transformed into the sub-tree 5 (strictly speaking, after the transformation, the resulted sub-tree is part of the sub-tree 5);

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.

Insert and then rebalance
(continue…)