In this question, you have to perform **rotate nodes** on AVL tree. Note that:

- When adding a node which has the same value as parent node, add it in the **right sub tree**.

Your task is to implement function: **rotateRight, rotateLeft**. You could define one or more functions to achieve this task.

In this question, you have to perform **add** on AVL tree. Note that:

- When adding a node which has the same value as parent node, add it in the **right sub tree**.

Your task is to implement function: **insert**. The function should cover at least these cases:

In this question, you have to perform add on AVL tree. Note that:

When adding a node which has the same value as parent node, add it in the right sub tree.

Your task is to implement function: `insert`. The function should cover at least these cases:

- Balanced tree
- Right of right unbalanced tree
- Left of right unbalanced tree

In this question, you have to perform **add** on AVL tree. Note that:

- When adding a node which has the same value as parent node, add it in the **right sub tree**.

Your task is to implement function: **insert**. You could define one or more functions to achieve this task.

In this question, you have to perform **delete in AVL tree - balanced, L-L, R-L, E-L**. Note that:

- Provided **insert** function already.

Your task is to implement function: **remove** to perform re-balancing (balanced, left of left, right of left, equal of left). You could define one or more functions to achieve this task.

In this question, you have to perform **delete on AVL tree**. Note that:

- Provided **insert** function already.

Your task is to implement two functions: **remove**. You could define one or more functions to achieve this task.

In this question, you have to search and print inorder on **AVL tree**. You have o implement functions: **search** and **printInorder** to complete the task. Note that:

- When the tree is null, don't print anything.

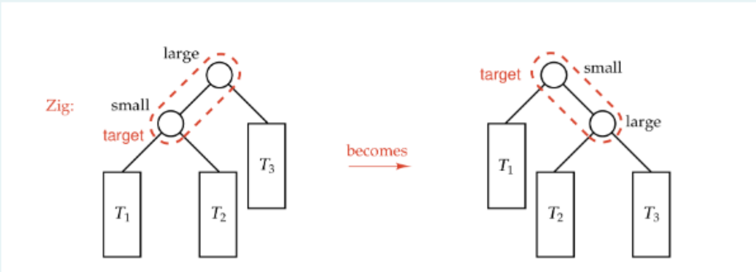
- There's a whitespace at the end when print the tree inorder in case the tree is not null.

1. void splay(Node* p): bottom-up splaying a Node

When a splay operation is performed on Node p, it will be moved to the root. To perform a splay operation we carry out a sequence of splay steps, each of which moves p closer to the root.

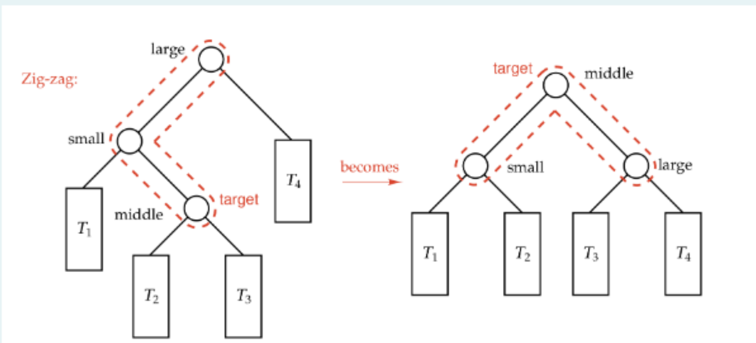
The three types of splay steps are:

- Zig step



- Zig-zig step:

- Zig-zag step:



Note: there are also zag, zag-zag and zag-zig step but we don't show them here

2. void insert(int val):

To insert a value val into a splay tree:

+ Insert val as with a normal binary search tree.

+ When the new value is inserted, a splay operation is performed. As a result, the newly inserted node becomes the root of the tree.

Note: In a splay tree, the values the in left subtree <= root's value <= the values in the right subtree. In this exercise, when inserting a duplicate value, you have to insert it to the right subtree to pass the testcases.

Method splay and insert are already implemented

You have to implement the following method:

bool search(int val): search for the value val in the tree.

The search operation in splay tree do the same thing as BST search. In addition, it also splays the node containing the value to the root.

+ If the search is successful, the node that is found will become the new root and the function return true.

+ Else, the last accessed node will be splayed and become the new root and the function return false.

The methods splay, insert and search are already implemented.

Implement the following method:

Node* remove(int val): remove the first Node with value equal to val from the tree and return it.

To perform remove operation on splay tree:

1. If root is NULL, return the root

2. Search for the first node containing the given value val and splay it. If val is present, the found node will become the root. Else the last accessed leaf node becomes the root.

3. If new root's value is not equal to val, return NULL as val is not present.

4. Else the value val is present, we remove root from the tree by the following steps:

   4.1 Split the tree into two tree: tree1 = root's left subtree and tree2 = root's right subtree

   4.2 If tree1 is NULL, tree2 is the new root

   4.3 Else, splay the leaf node with the largest value in tree1. tree1 will be a left skewed binary tree. Make tree2 the right subtree of tree1. tree1 becomes the new root

   4.4 Return the removed node.