Data Structures and Algorithms

Lab 7
BST trees. AVL trees.

Agenda

- Recap: binary search trees
- AVL trees
- Coding exercise

Recap: binary search trees

- What is a BST?
- How to find a key in a BST?
- How to insert into BST?
- How to delete from BST?

Recap: binary search trees

- **What is a BST?**
- How to find a key in a BST?
- How to insert into BST?
- How to delete from BST?

Recap: binary search trees

- What is a BST?
- **How to find a key in a BST?**
- How to insert into BST?
- How to delete from BST?

Recap: binary search trees

- What is a BST?
- How to find a key in a BST?
- **How to insert into BST?**
- How to delete from BST?

Recap: binary search trees

- What is a BST?
- How to find a key in a BST?
- How to insert into BST?
- **How to delete from BST?**

AVL trees: invariants

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

An **AVL tree** is a binary search tree whose **height is balanced**:

> For each node, the heights of its subtrees differ by **at most 1**.

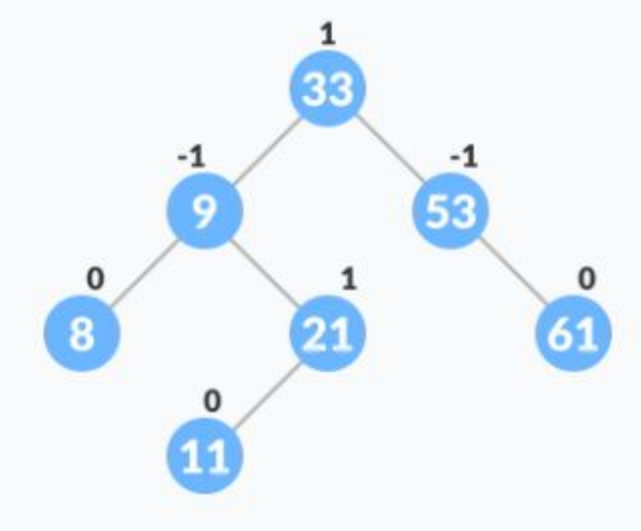
Note: implementations of AVL trees maintain an extra attribute in each node (e.g. x.h is the height of node x).

AVL trees: balance factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

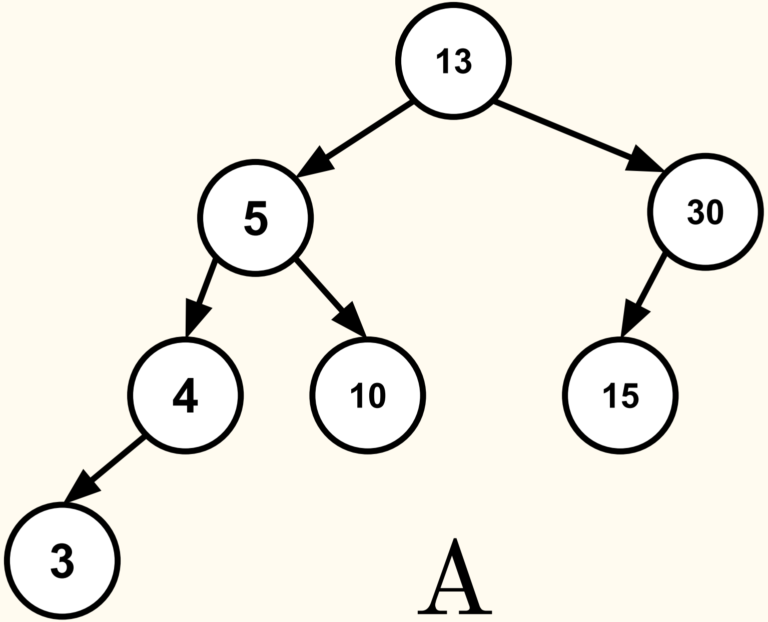
Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)

The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

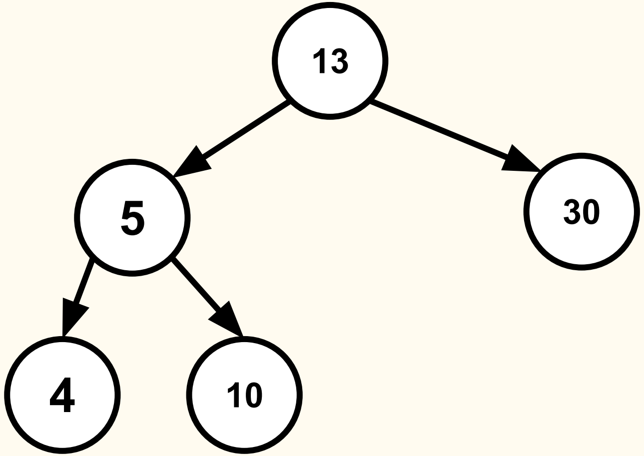


AVL trees: example

7.1. Which of the following are valid AVL trees?

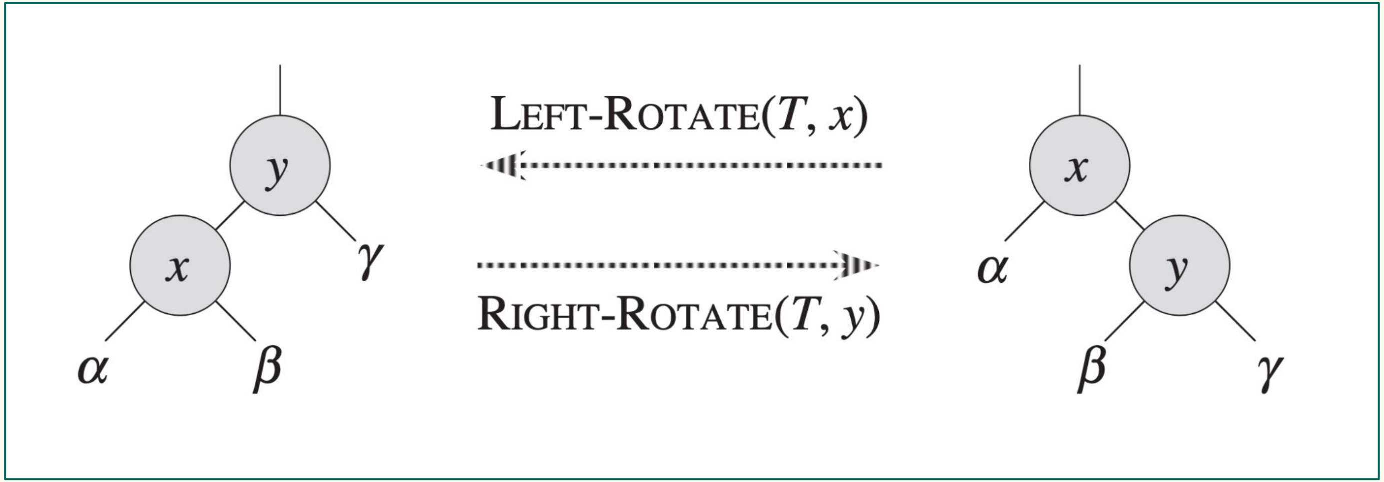


A



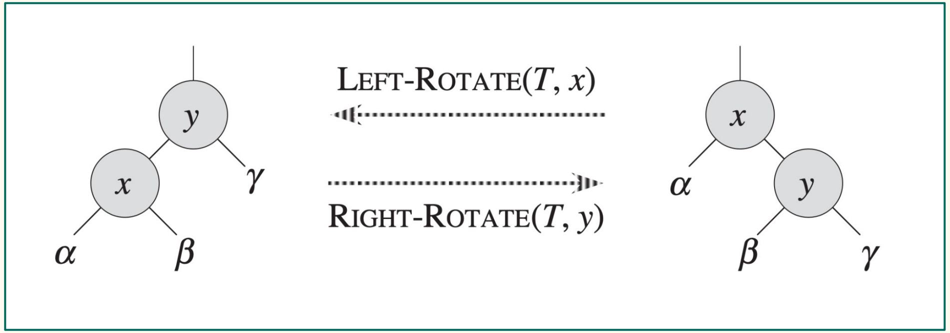
B

AVL trees: Operations



Idea: change the shape of the tree, preserving BST property.

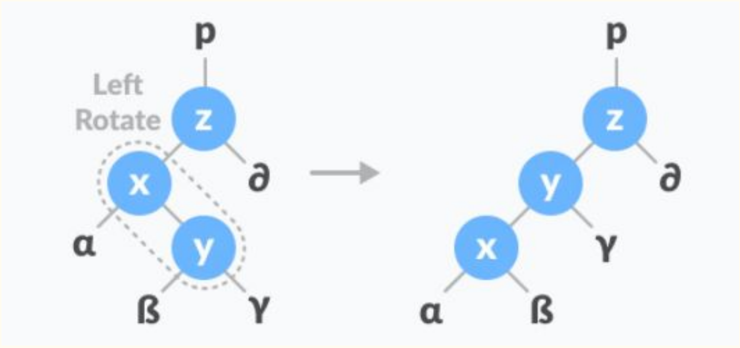
AVL trees: Operations

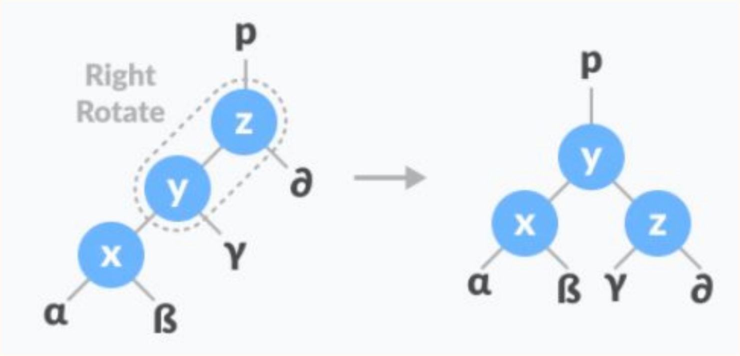


1. In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.
2. In left-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

AVL trees: Left-right operation

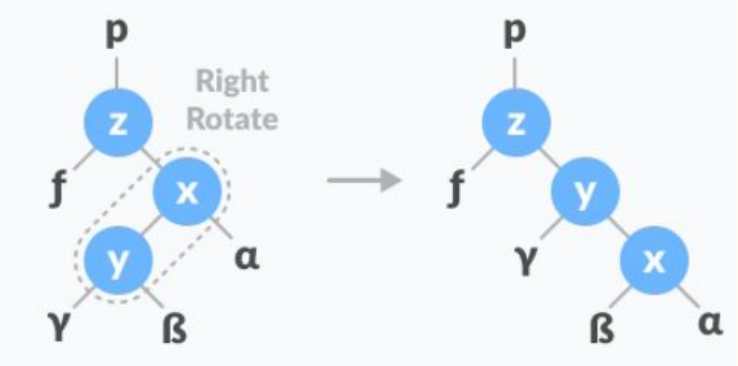
In left-right rotation, the arrangements are first shifted to the left and then to the right.

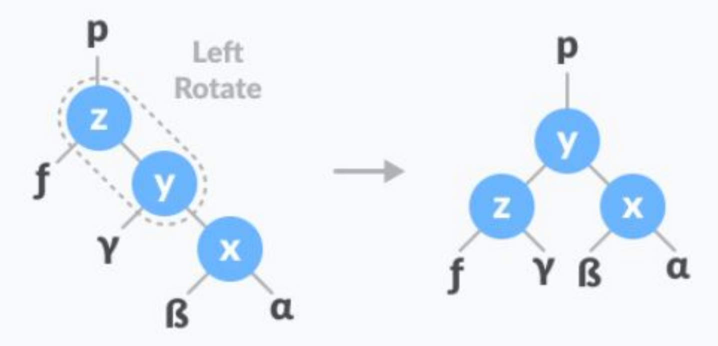




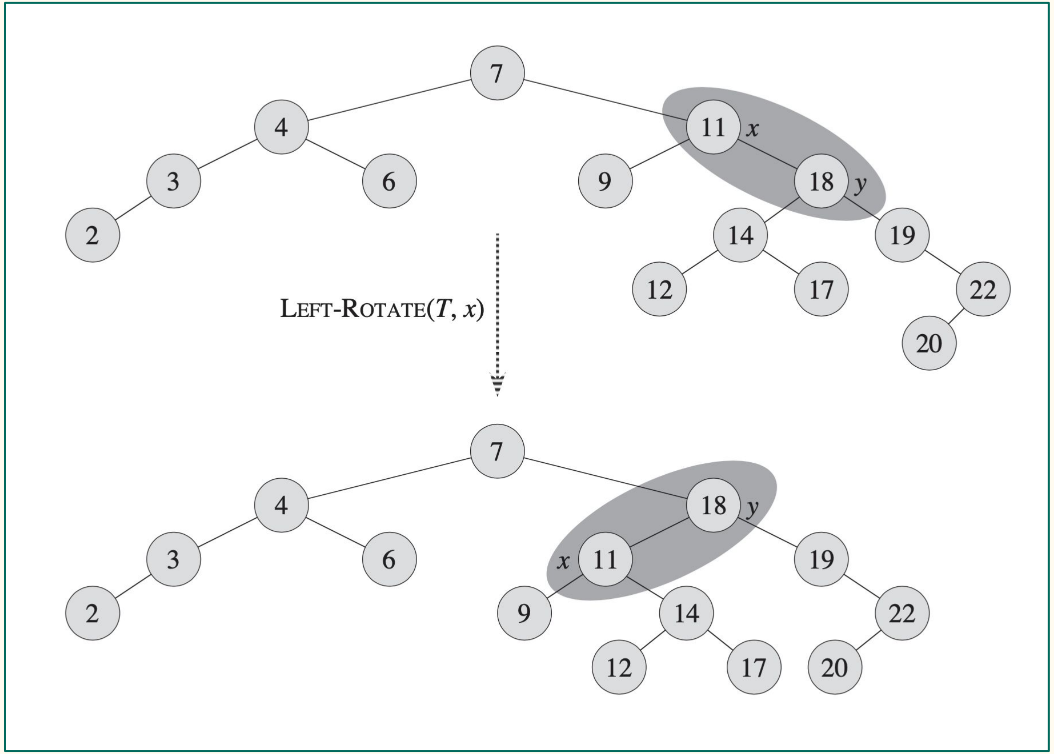
AVL trees: right-left operation

In left-right rotation, the arrangements are first shifted to the left and then to the right.





AVL trees: Operation example



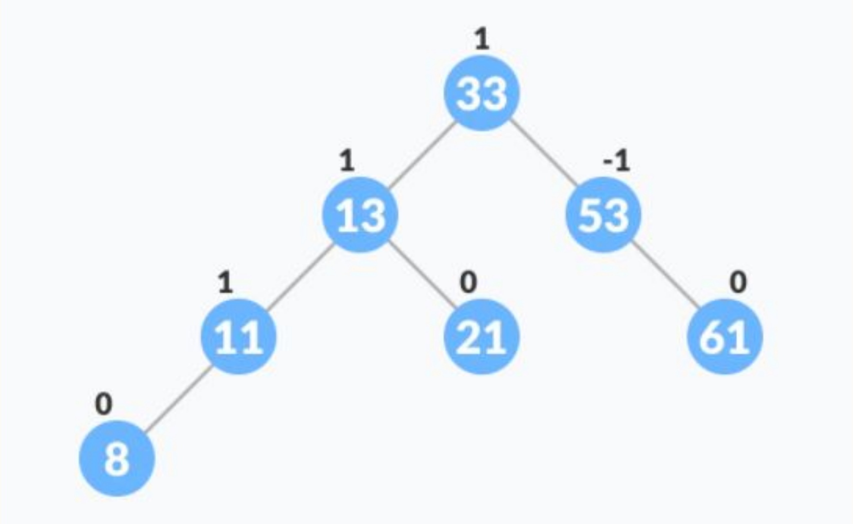
AVL trees: exercises

7.3. Describe a recursive procedure AVL-INSERT(x, z), that takes a node x within an AVL tree and a newly created node z, and adds z to the subtree rooted at x, maintaining the property that x is the root of an AVL tree.

7.4. Describe a recursive procedure AVL-DELETE(x, z), that takes a node x within an AVL tree and another node z, and removes z from the subtree rooted at x, maintaining the property that x is the root of an AVL tree.

AVL trees: Insert a new element

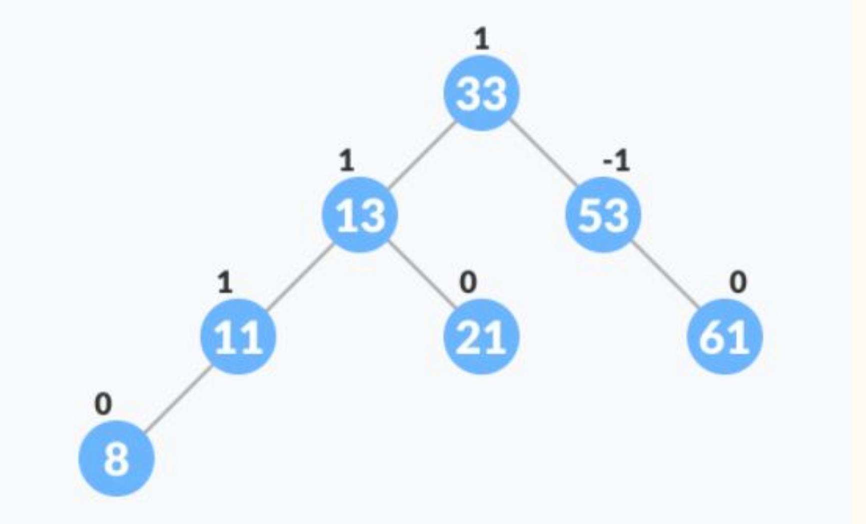
We want to insert the element 9



AVL trees: Insert a new element

we apply the recursive algorithm to search an item

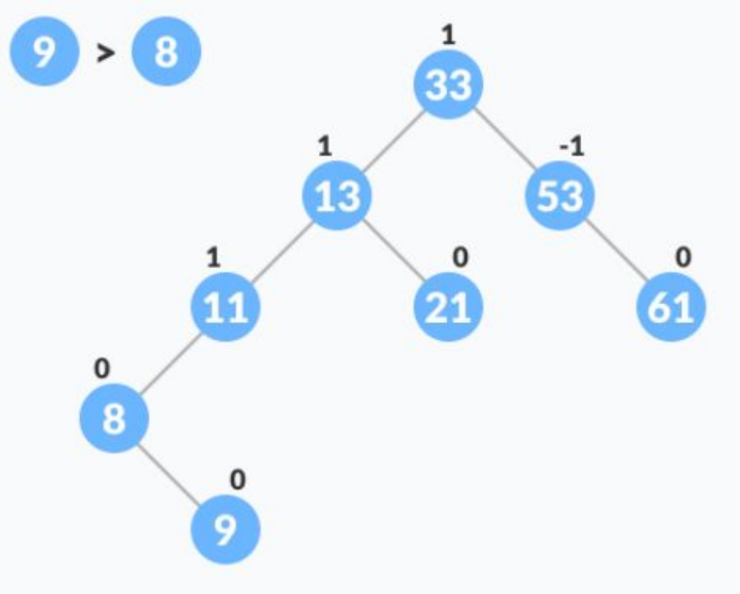
```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```



AVL trees: Insert a new element

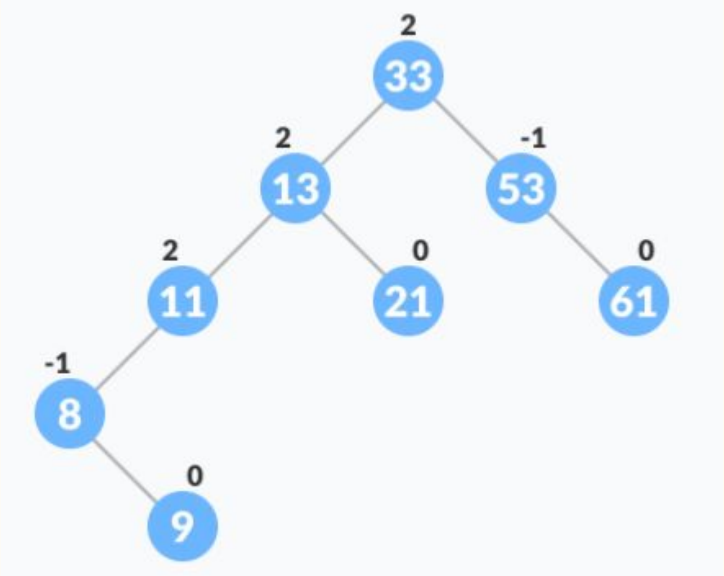
We want to insert the element 9= `newKey`

- Compare *`leafKey`* obtained from the above steps with `newKey`:
- If *`newKey < leafKey`*, make *newNode* as the *`leftChild`* of *`leafNode`*.
- Else, make *`newNode`* as *rightChild* of *`leafNode`*



AVL trees: Insert a new element

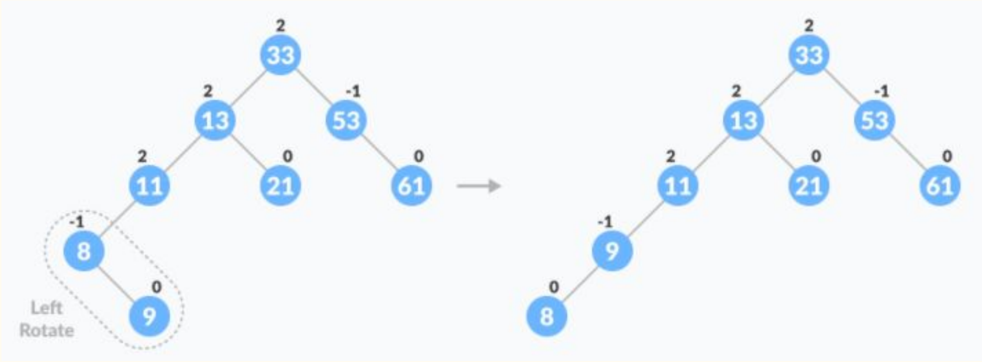
Update the *balanceFactor* of the nodes



AVL trees: Insert a new element

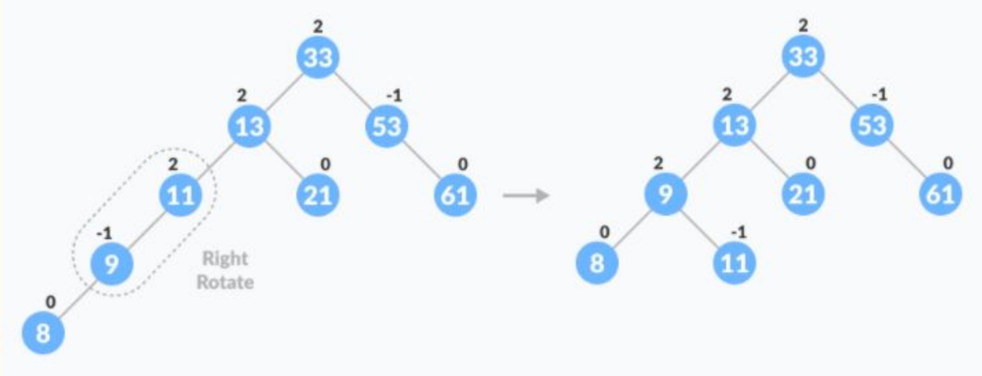
If `balanceFactor` > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation

1. If `newNodeKey` < `leftChildKey` do right rotation.
2. Else, do left-right rotation.



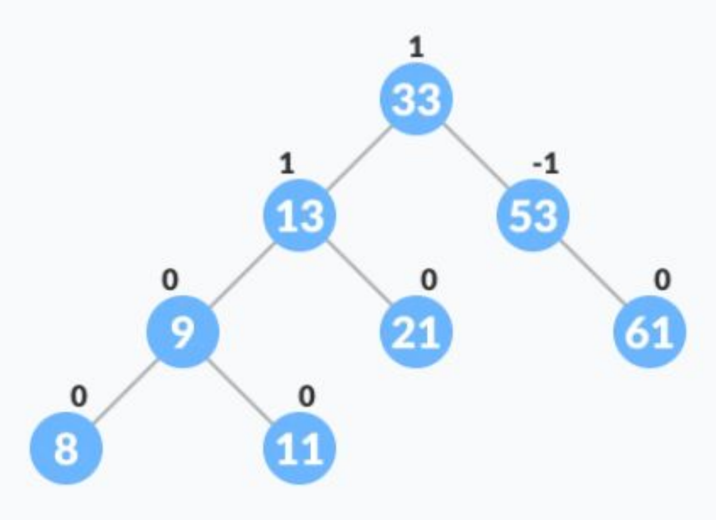
If *balanceFactor* < -1, it means the height of the right subtree is greater than that of the left subtree. So, do left rotation or right-left rotation

1. If *newNodeKey* > *rightChildKey* do left rotation.
2. Else, do right-left rotation



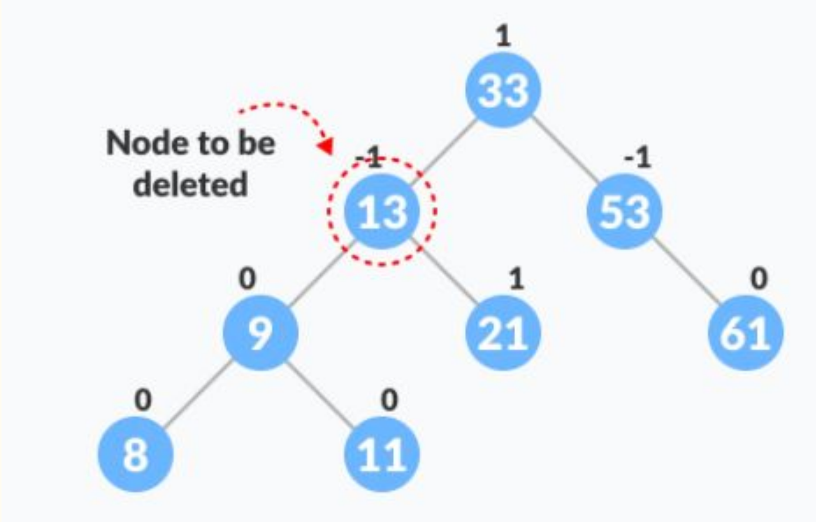
AVL trees: Insert a new element

the final tree is:



AVL trees: delete an element

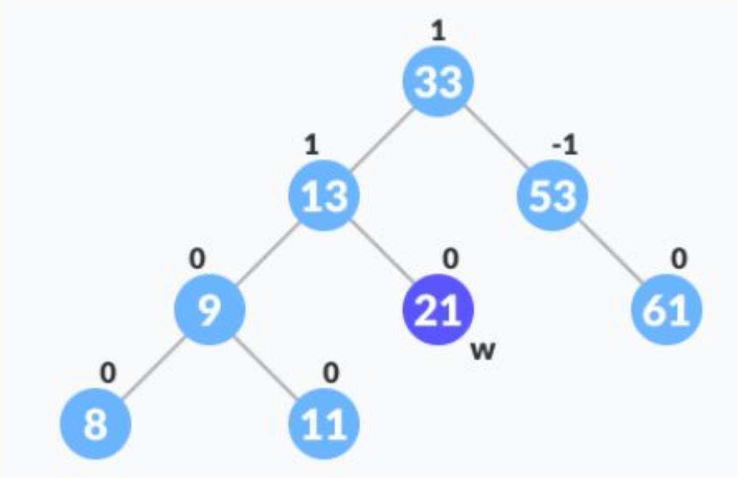
Locate *nodeToBeDeleted* (recursion is used to find *nodeToBeDeleted*).



AVL trees: delete an element

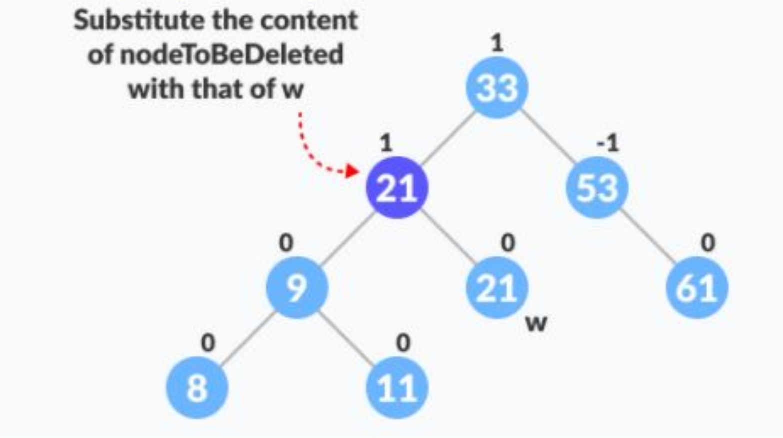
There are three cases for deleting a node:

1. If *nodeToBeDeleted* is the leaf node (ie. does not have any child), then remove *nodeToBeDeleted*.
2. If *nodeToBeDeleted* has one child, then substitute the contents of *nodeToBeDeleted* with that of the child. Remove the child.
3. If *nodeToBeDeleted* has two children, find the inorder successor w of *nodeToBeDeleted* (ie. node with a minimum value of key in the right subtree).



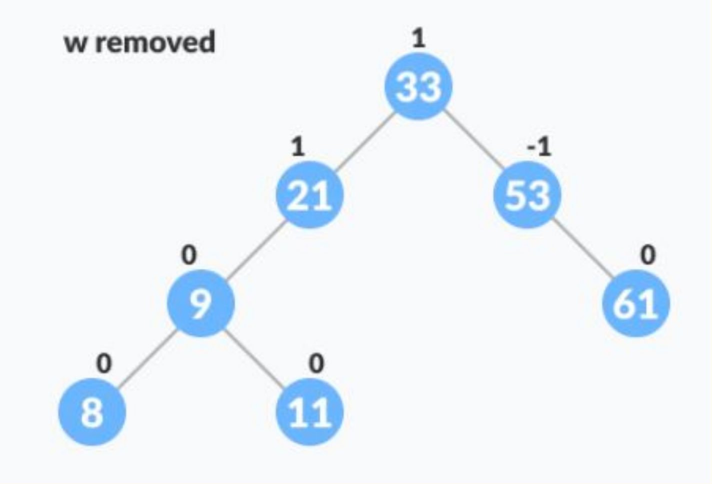
AVL trees: delete an element

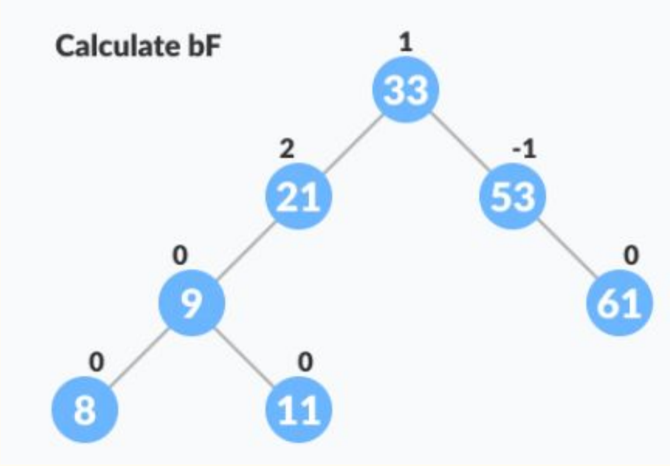
Substitute the contents of *nodeToBeDeleted* with that of w



AVL trees: delete an element

Remove the leaf node w and update the *balanceFactor*

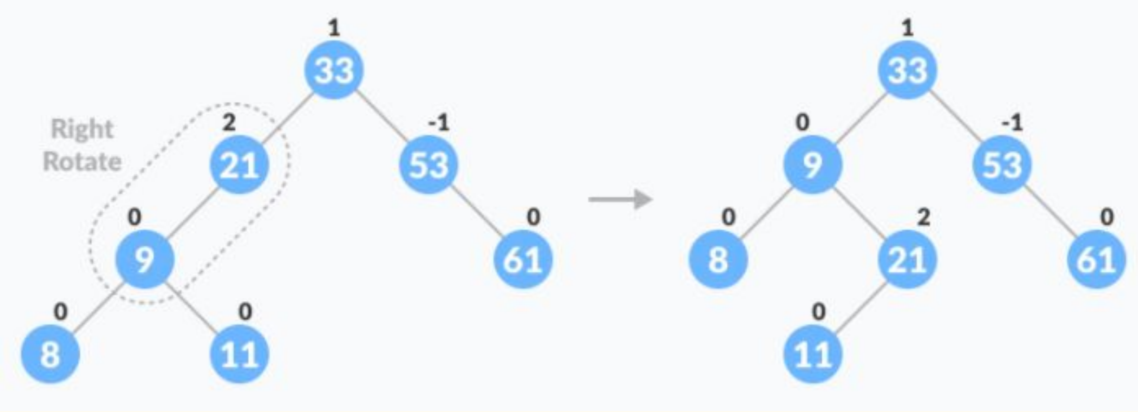




AVL trees: delete an element

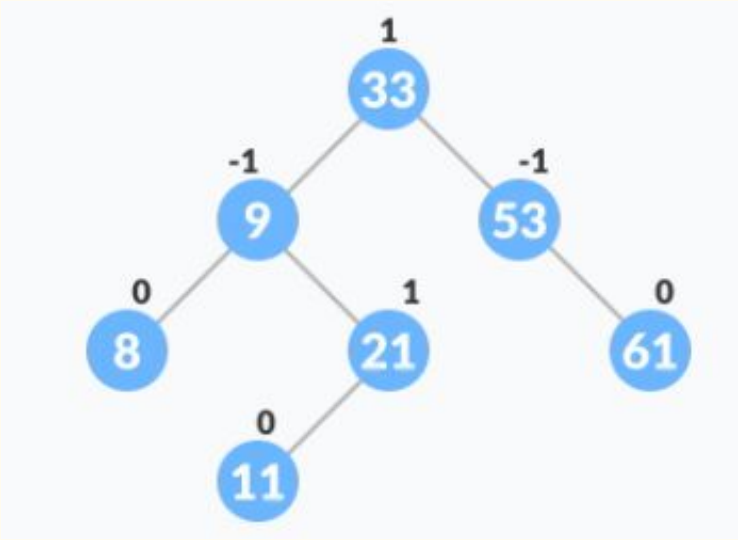
Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.

1. If *balanceFactor* of *currentNode* $>$ 1,
   a. If *balanceFactor* of *leftChild* $>=$ 0, do right rotation.
   b. Else do left-right rotation.
2. If *balanceFactor* of currentNode $<$ -1,
   a. If *balanceFactor* of *rightChild* $<=$ 0, do left rotation.
   b. Else do right-left rotation.



AVL trees: delete an element

The final tree is:



AVL trees: exercises

7.5. Build an AVL tree by inserting these keys in order:

$$8, \quad 12, \quad 19, \quad 31, \quad 38, \quad 41$$

7.5. delete the element :

$$19$$

Codeforce

Given some numbers, build a binary search tree (BST).

Input

Input starts with a line with one number N $(0 < N < = 10^5)$. The next line has N integer numbers.

Output

Start output with N — number of nodes in binary search tree.

In the next N lines output information about nodes (one node per line). For each node output integer value x_i at node i, l_i (index of the left node or -1) and r_i (index of the right node or -1).

In the final line output the index of the root node.

Node indexing starts with 1 and does not have to preserve input order.

Examples

input Copy

```
3
1 2 3
```

output Copy

```
3
2 2 3
1 -1 -1
3 -1 -1
1
```

https://codeforces.com/group/M5kRwzPJlU/contest/318782

See You next week!