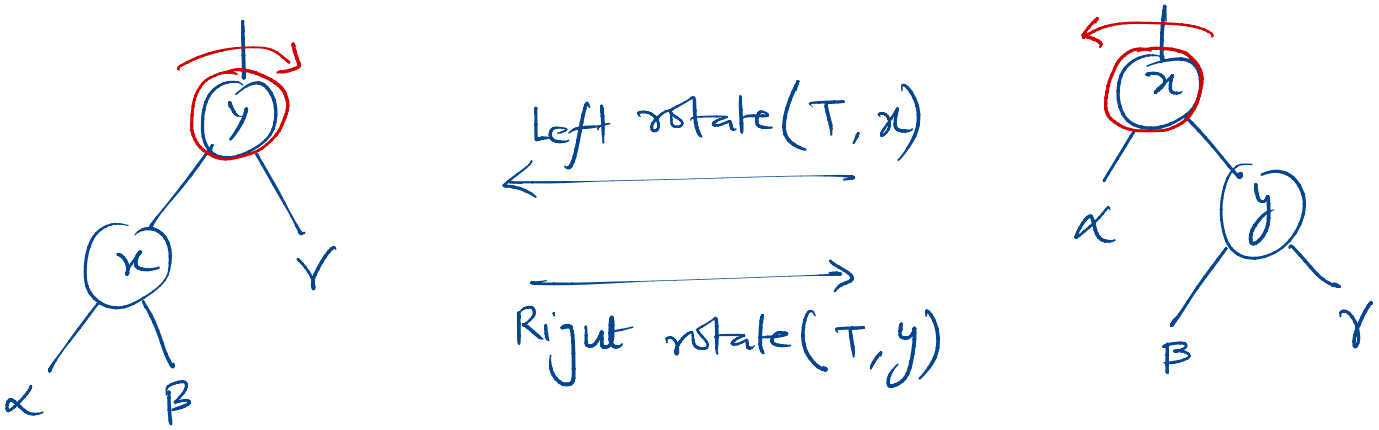
AVL Trees – II

Rotations: A local operation that preserves the BST property

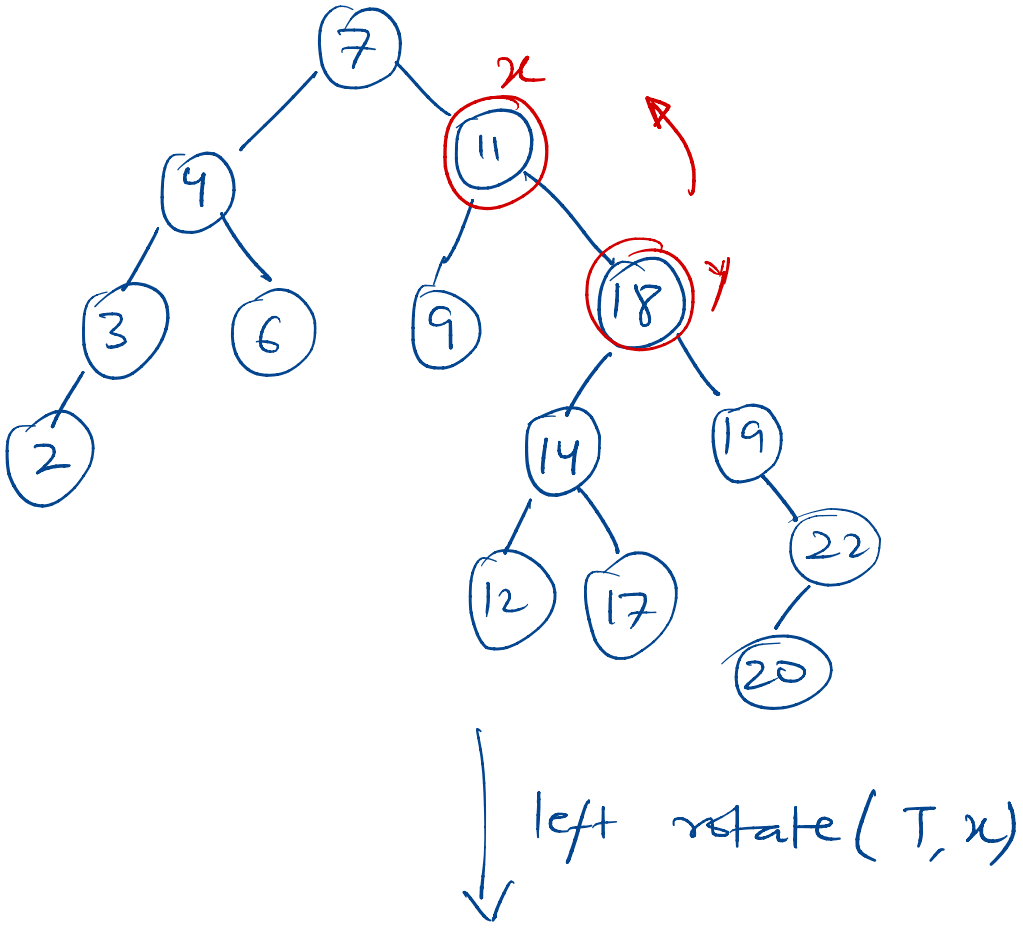
left rotation → right rotation.



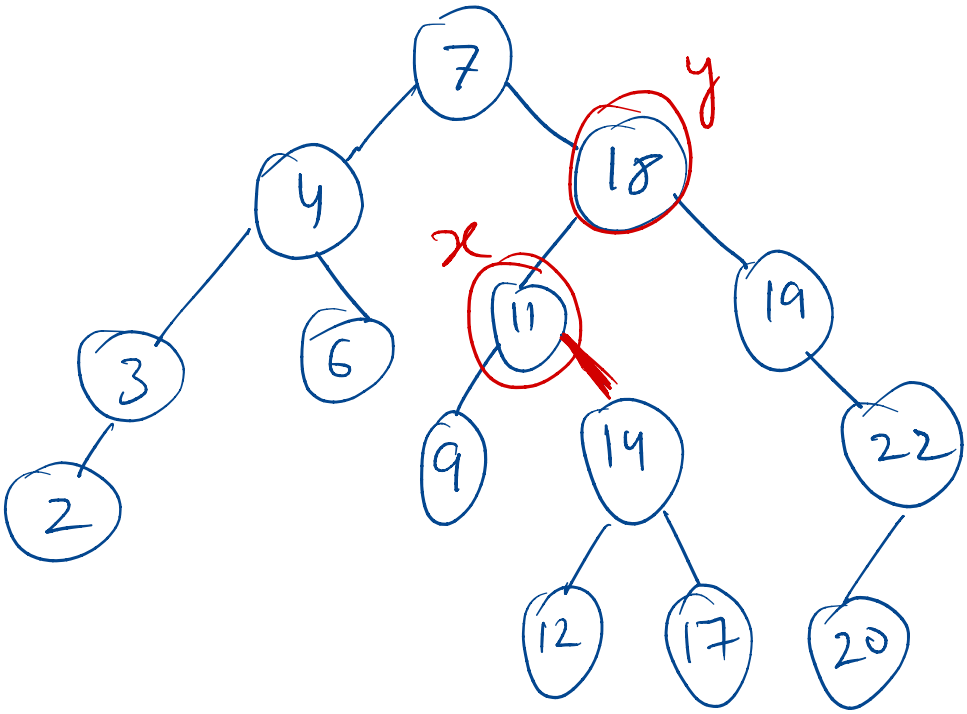
Left-Rotate(T, x)

```
y = x.right
x.right = y.left // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
else if x == x.p.left
    x.p.left = y
else
    x.p.right = y
y.left = x
x.p = y
```

Example :



left rotate (T, x)



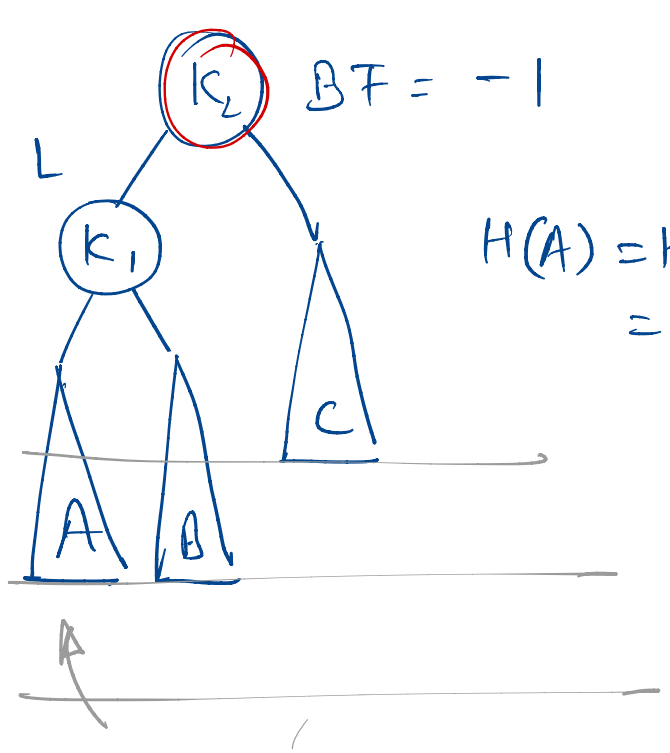
AVL Tree balancing

- Insert normally as in a BST
- If the tree becomes imbalanced then
  Fix the imbalance
  ⇓
  Only the ancestors of the newly inserted node are imbalanced
  ⇓
  Focus on the deepest node that is imbalanced

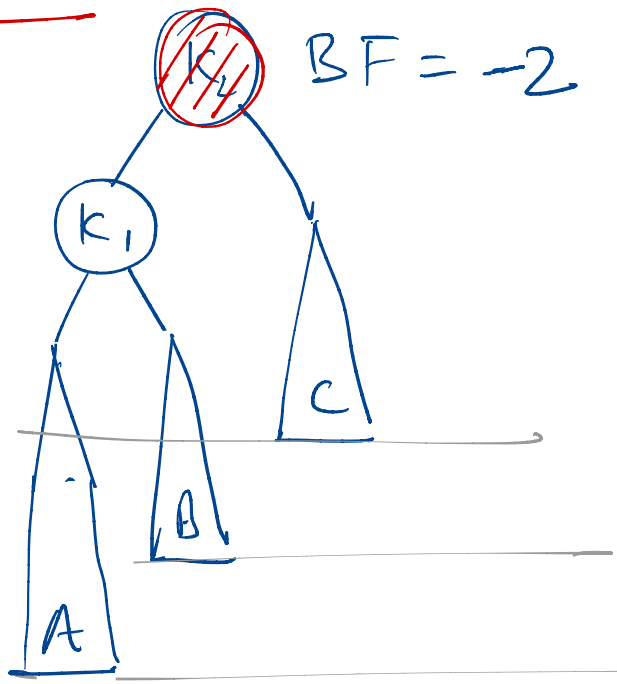
AVL add cases:

Case I: Left-Left (LL)

Insert new node in left subtree of left node



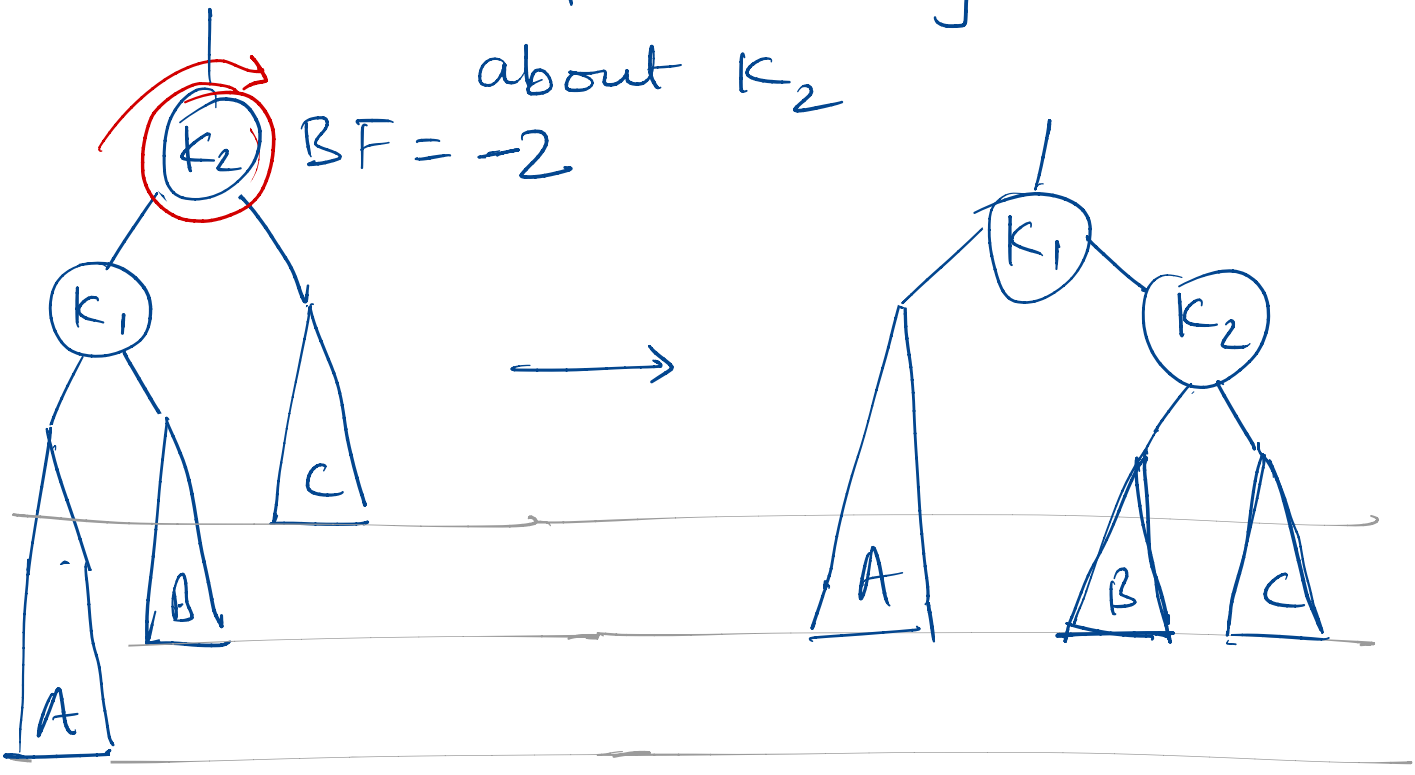
H(A) = H(B) = H(C)



insert in A

(K_2 is the deepest imbalanced node)

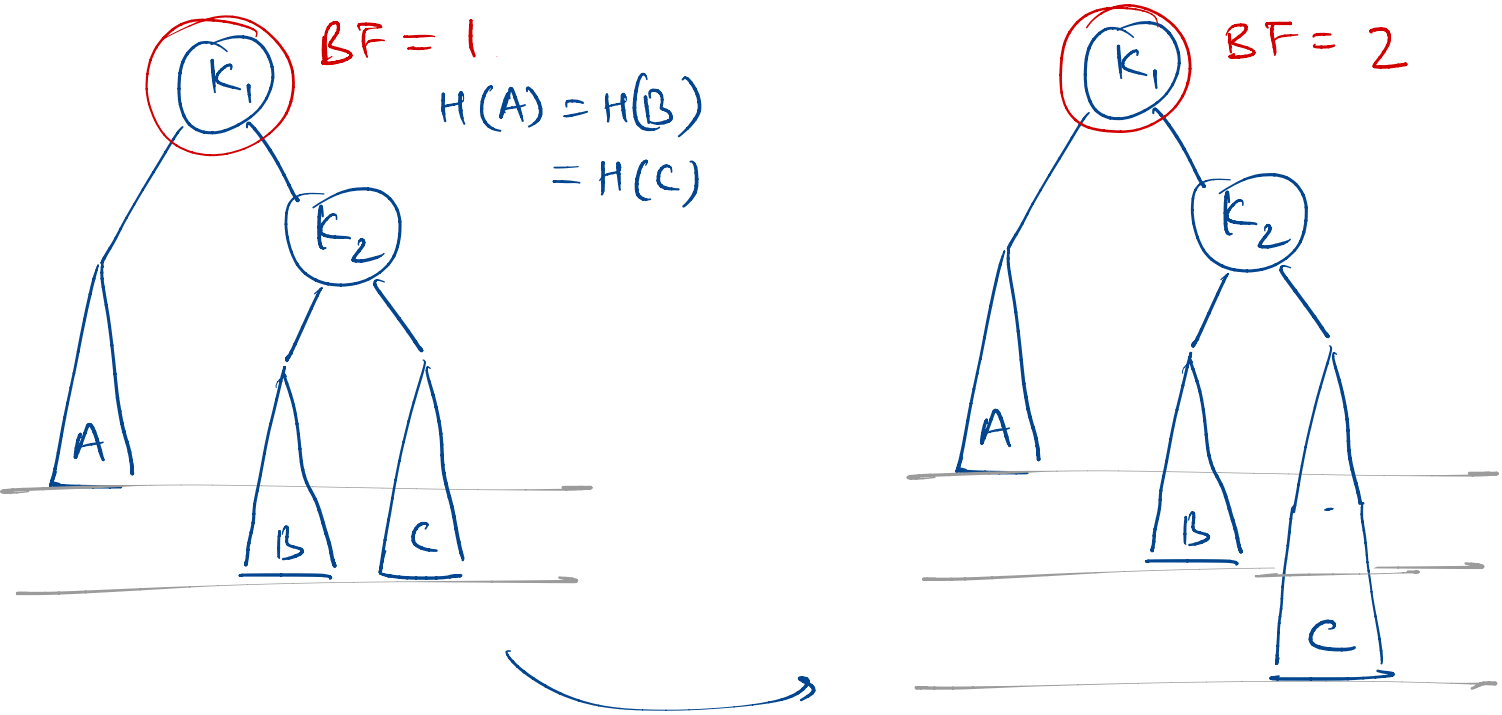
Resolution: Perform a right rotation about K_2



Case II : Right - Right (RR)

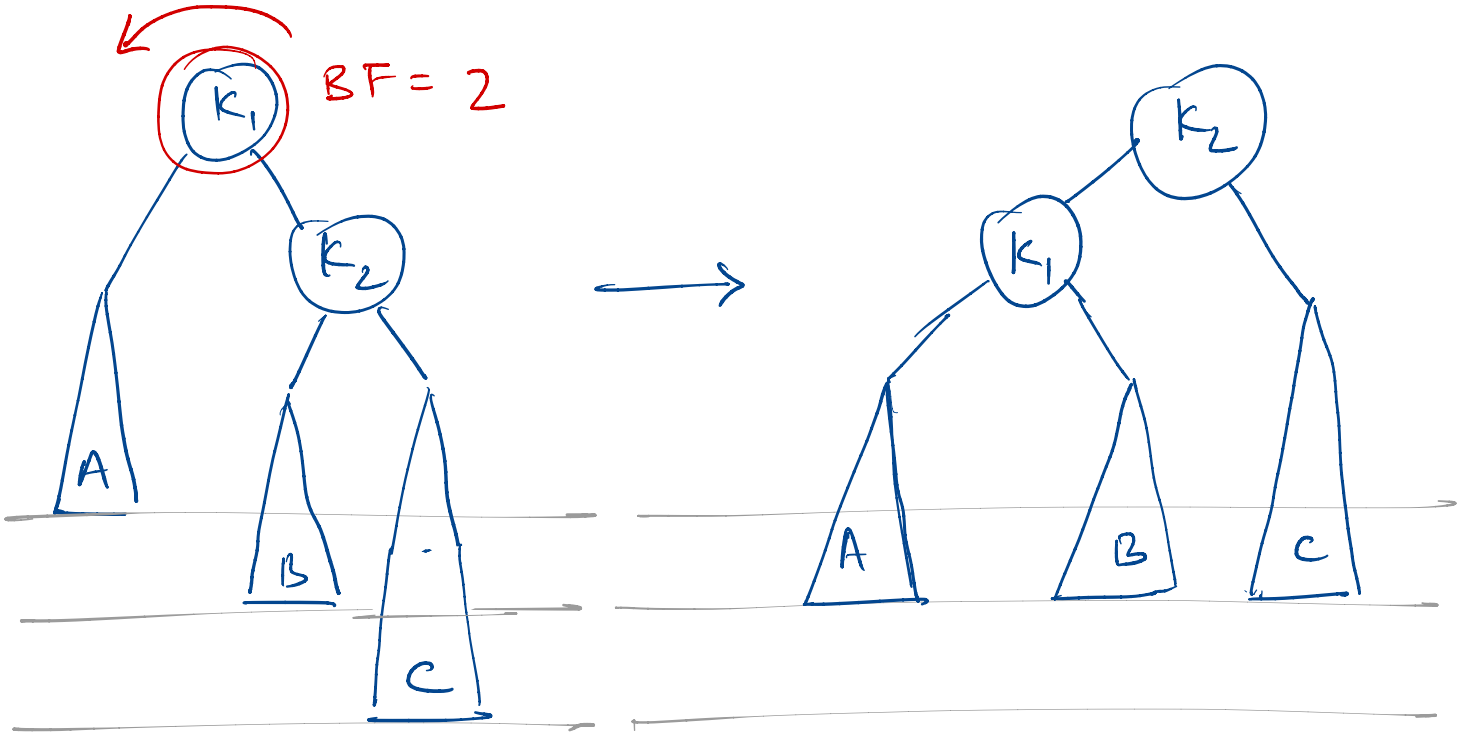
similar to LL case!

Insert new node in the right subtree of the right child of K_1

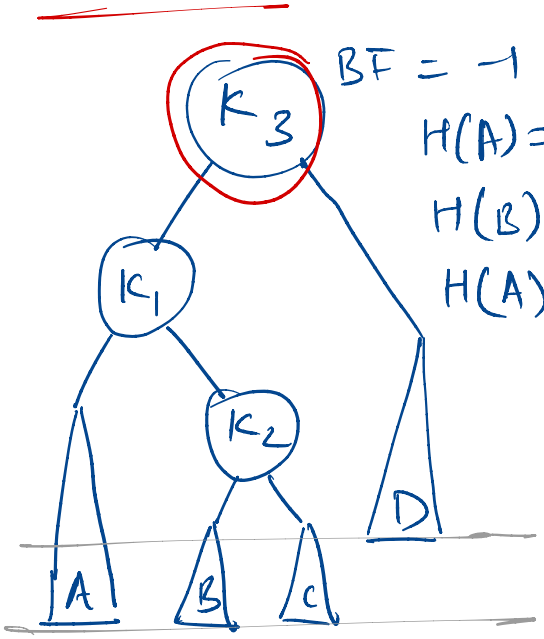


Insert in C

Resolution: Perform a single left rotation about K_1



Case III: Left - Right (LR)

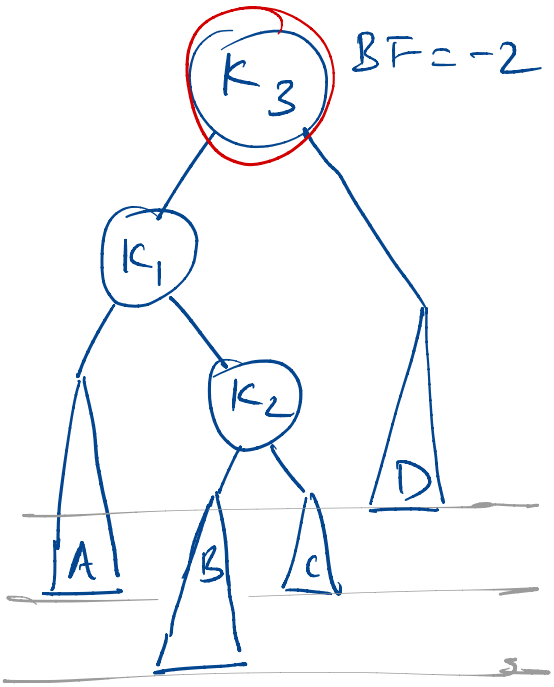


BF = -1

H(A) = H(D)

H(B) = H(C)

H(A) = H(B)+1



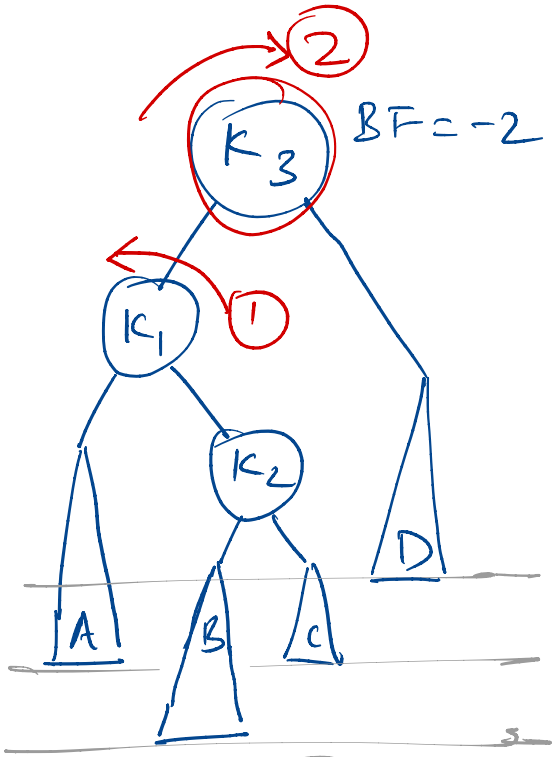
BF = -2

Insert in B (or C)

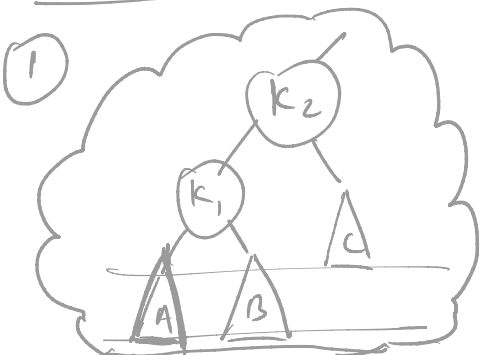
Resolution: 1. left rotate about K_3's left child K_1

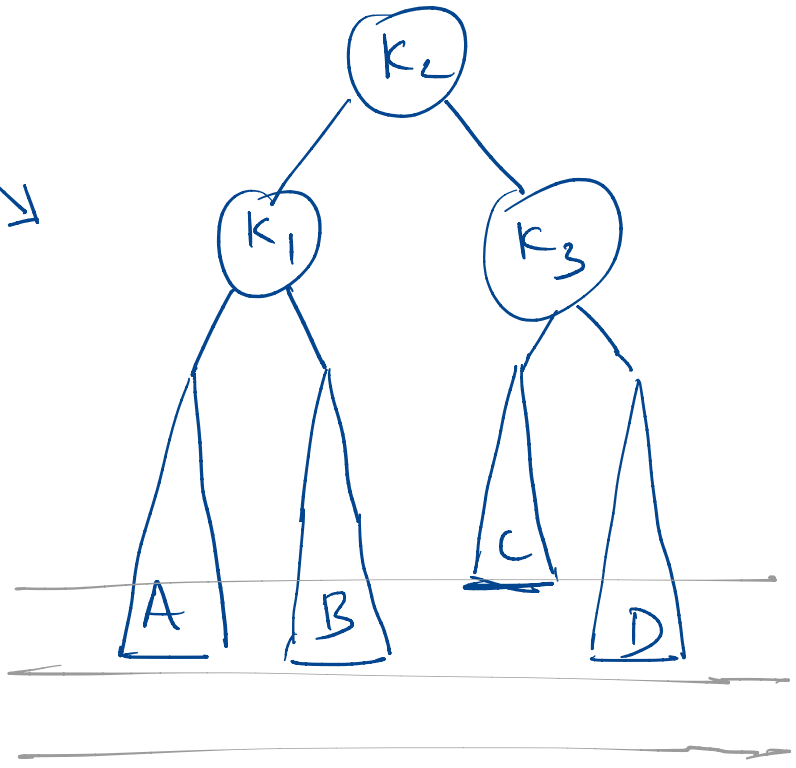
2. Right rotate about K_3

(reduce to Case I: LL)



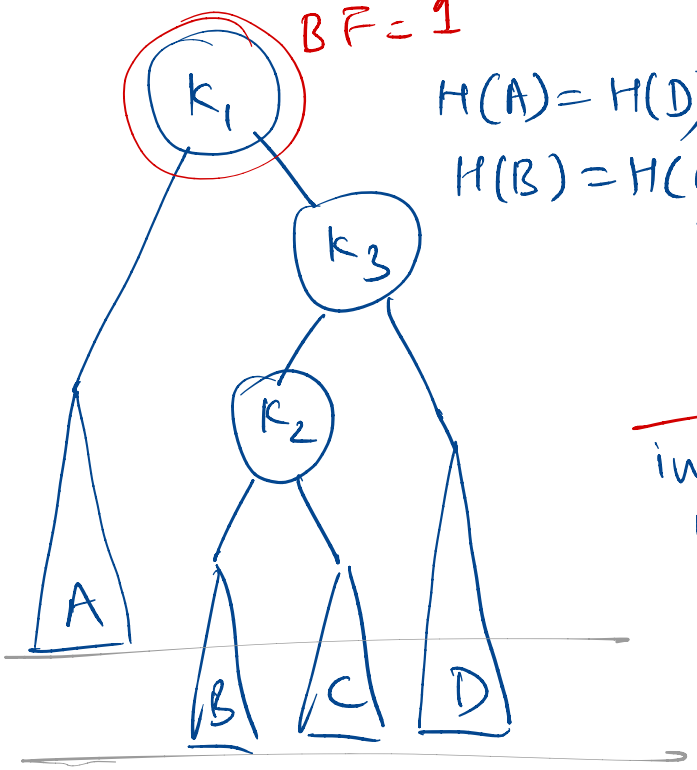
BF = -2



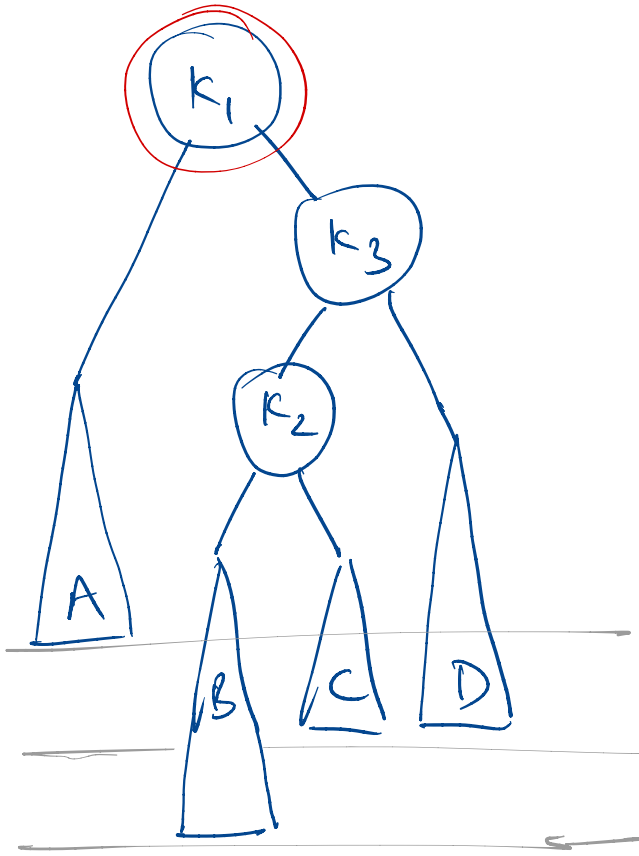


Case IV : Right - Left (RL)

Insert a new node into the left subtree of the right child of concerned node



insert in B (or C)



Resolution : 1. Right rotate at K's right child K_3

2. Left rotate at K_1