AVL Trees

Data and File Structures Laboratory

http://www.isical.ac.in/~dfslab/2020/index.html

AVL tree nodes

```
class AVLNode() :
    """A single node in an AVL tree.
Attributes: as for binary tree Node, but with additional attribute
    height

Following Weiss, DS & AA in C++, 4ed., we store the height of the
subtree rooted at a node, rather than a balance factor.
"""
    def __init__(self, data, left, right, parent, height):
        self.data = data
        self.left = left
        self.right = right
        self.parent = parent
        self.height = height
```

Functions

Main API functions

- `AVLTree(n)` : create empty tree with a capacity of upto n nodes
- `search(nodeindex, data)` : *inherited with no change*
- `insert(parent_index, nodeindex, data)`, `delete(parent_index, nodeindex, data)`

Functions

Main API functions

- `AVLTree(n)` : create empty tree with a capacity of upto n nodes
- `search(nodeindex, data)` : *inherited with no change*
- `insert(parent_index, nodeindex, data)`, `delete(parent_index, nodeindex, data)`

Other functions: `height(nodeindex)`, `print_pstree(nodeindex)`

Functions

Main API functions

- `AVLTree(n)` : create empty tree with a capacity of upto n nodes
- `search(nodeindex, data)` : *inherited with no change*
- `insert(parent_index, nodeindex, data)`, `delete(parent_index, nodeindex, data)`

Other functions: `height(nodeindex)`, `print_pstree(nodeindex)`

Helper functions:

- `find_successor(parent_index, nodeindex)`
- `balance(parent_index, nodeindex)`

Functions

Main API functions

- `AVLTree(n)` : create empty tree with a capacity of upto n nodes
- `search(nodeindex, data)` : *inherited with no change*
- `insert(parent_index, nodeindex, data)`, `delete(parent_index, nodeindex, data)`

Other functions: `height(nodeindex)`, `print_pstree(nodeindex)`

Helper functions:

- `find_successor(parent_index, nodeindex)`
- `balance(parent_index, nodeindex)`
- `rotate_LL`, `rotate_RR`, `rotate_LR`, `rotate_LR`

balance()

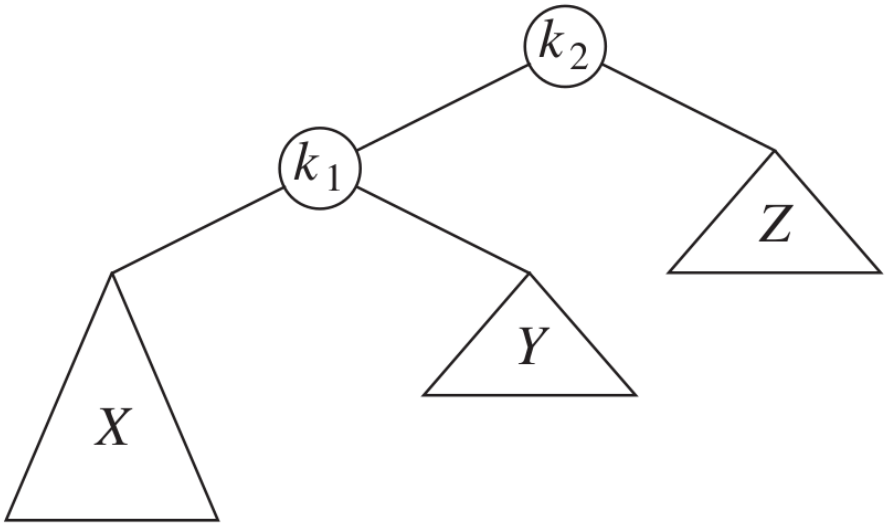
```
def balance(self, parent, start_index) :
    thisnode = start_index
    left = self.nodelist[thisnode].left
    right = self.nodelist[thisnode].right

    if self.height(left) - self.height(right) > 1 :
```

```
        if self.height(self.nodelist[left].left) >= self.height(self.
nodelist[left].right) :
            return self.rotate_LL(parent, start_index)
        else :
            return self.rotate_RL(parent, start_index)
```

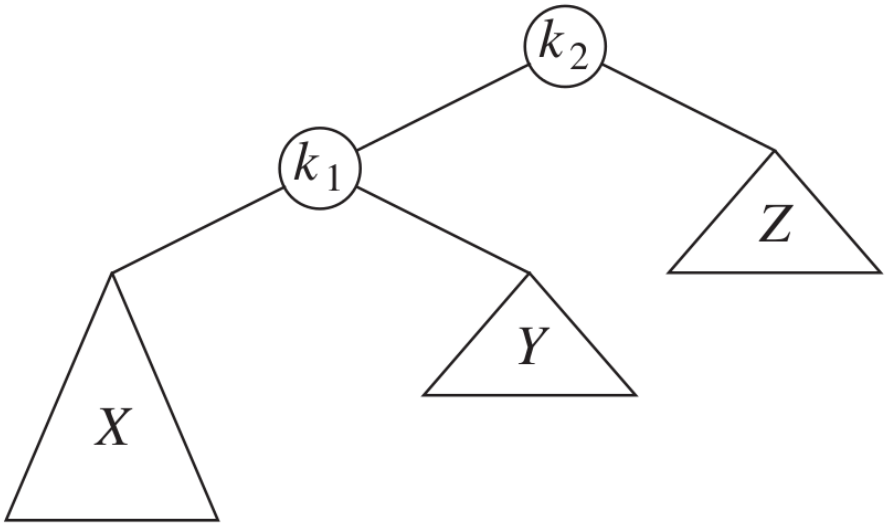
Rotate LL

- **L**eft subtree of **L**eft subtree causes imbalance
- Rotate right to restore balance



Rotate LL

- **L**eft subtree of **L**eft subtree causes imbalance
- Rotate right to restore balance

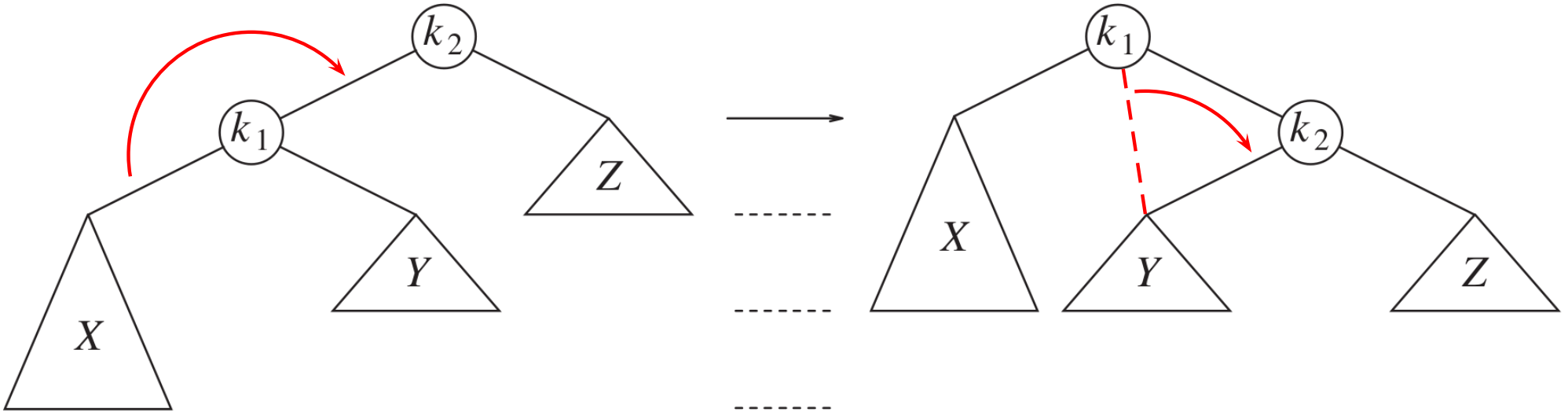


bf must have been +1 $\Rightarrow$ $h(k_1) = 1 + h(Z)$

$\Rightarrow$ $h_{old}(X) = h(Z)$ and $h(Y) = h(Z)$ or 1 less

Rotate LL

- **L**eft subtree of **L**eft subtree causes imbalance
- Rotate right to restore balance



bf must have been +1 $\Rightarrow$ $h(k_1) = 1 + h(Z)$

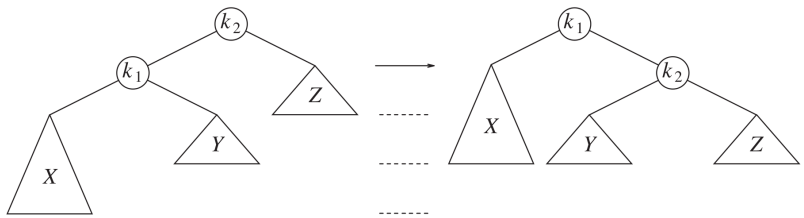
$\Rightarrow$ $h_{old}(X) = h(Z)$ and $h(Y) = h(Z)$ or 1 less

Rotate LL - code I

```
def rotate_LL(self, parent, start_index) :
    # See Weiss, DS & AA in C++, 4 ed., Section 4.4.1, Figure 4.34
    if DEBUG :
        print("LL (right) rotation at %d\n" % self.nodelist[start_index
].data)
    k2 = start_index
    k1 = self.nodelist[k2].left
    Z = self.nodelist[k2].right
    X = self.nodelist[k1].left
    Y = self.nodelist[k1].right

    # rotate
    self.nodelist[k2].left = Y
    self.nodelist[k1].right = k2

    # parents (optional)
    self.nodelist[k1].parent = parent
    self.nodelist[k2].parent = k1
```



Rotate LL - code II

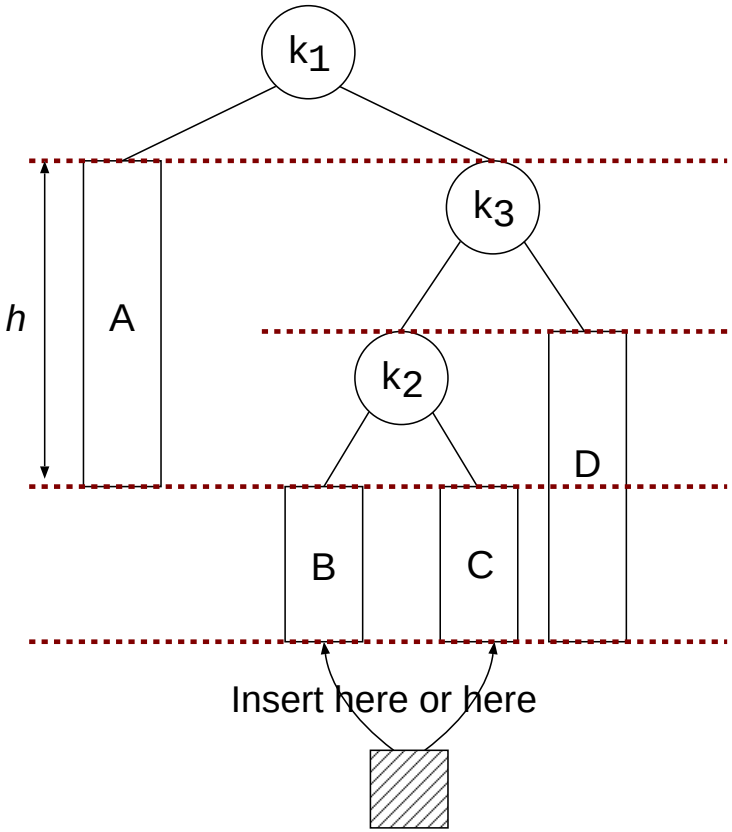
```
    if Y != -1 : self.nodelist[Y].parent = k2

    # update heights
    self.nodelist[k2].height = 1 + max(self.height(Y), self.height(Z))
    self.nodelist[k1].height = 1 + max(self.height(X), self.height(k2))

    return k1
```

Rotate LR

Left subtree of **R**ight subtree causes imbalance

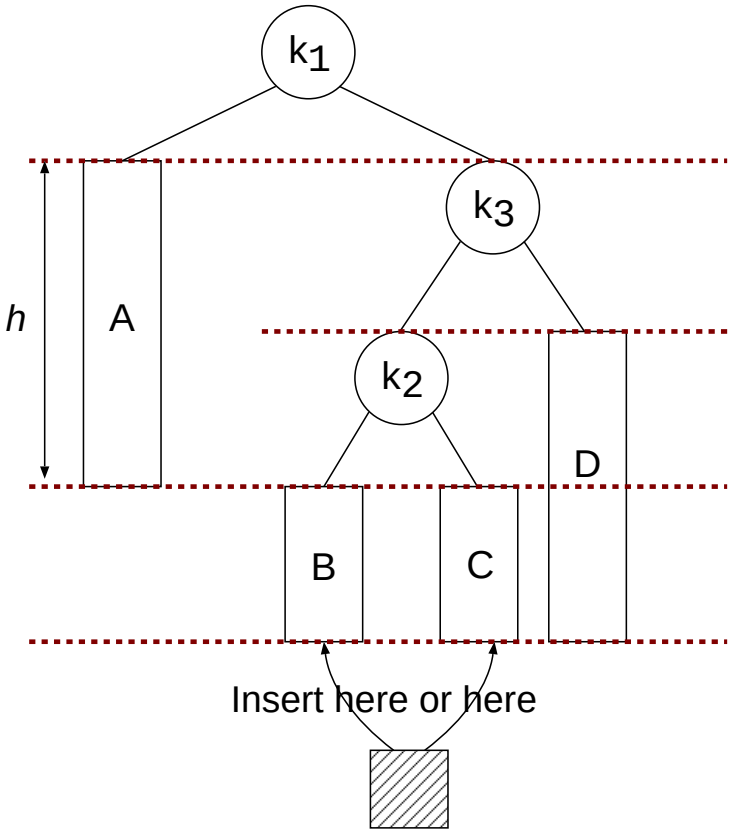


h - height

bf - balance factor

Rotate LR

Left subtree of **R**ight subtree causes imbalance



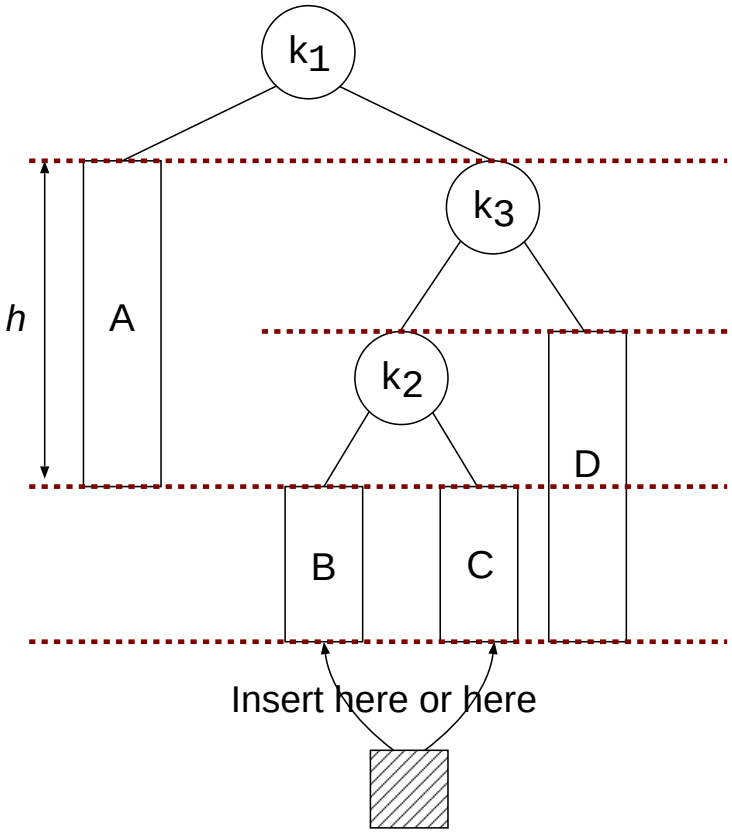
$$bf_{new}(k_1) = -2, \qquad bf_{old}(k_1) = -1$$

h - height

bf - balance factor

Rotate LR

Left subtree of **R**ight subtree causes imbalance



$$bf_{new}(k_1) = -2, \qquad bf_{old}(k_1) = -1$$

$$\Rightarrow \; h_{new}(k_3) = h + 2, \qquad h_{old}(k_3) = h + 1$$

$$\Rightarrow \; \max(h_{new}(k_2), h(D)) = h + 1$$

$$\max(h_{old}(k_2), h(D)) = h$$

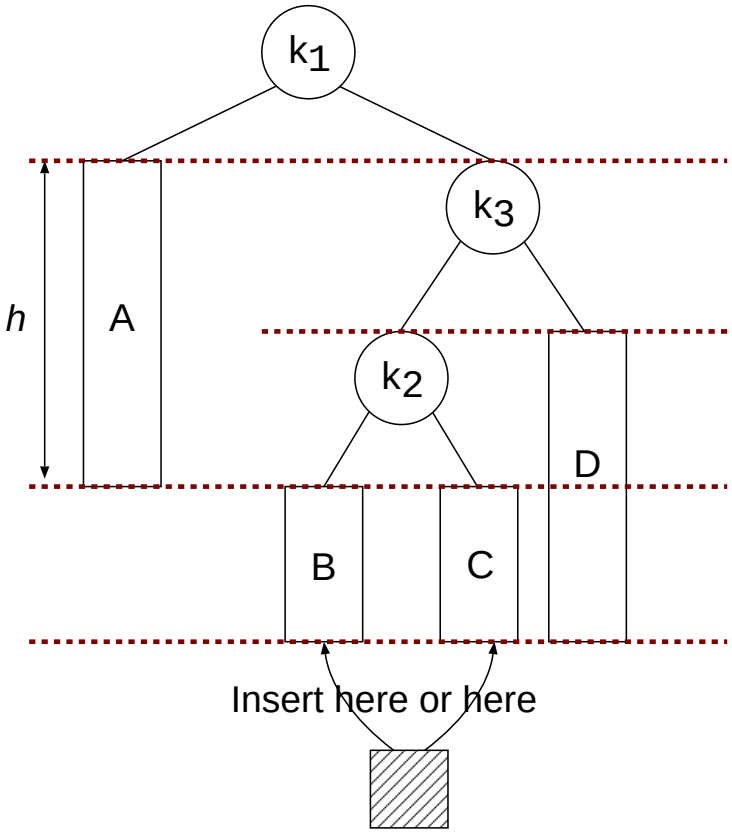
$$\Rightarrow \; h_{new}(k_2) = h + 1, \qquad h_{old}(k_2) = h$$

h - height

bf - balance factor

Rotate LR

Left subtree of **R**ight subtree causes imbalance



h - height

bf - balance factor

$$bf_{new}(k_1) = -2, \qquad bf_{old}(k_1) = -1$$

$$\Rightarrow \; h_{new}(k_3) = h + 2, \qquad h_{old}(k_3) = h + 1$$

$$\Rightarrow \; \max(h_{new}(k_2), h(D)) = h + 1$$

$$\max(h_{old}(k_2), h(D)) = h$$

$$\Rightarrow \; h_{new}(k_2) = h + 1, \qquad h_{old}(k_2) = h$$

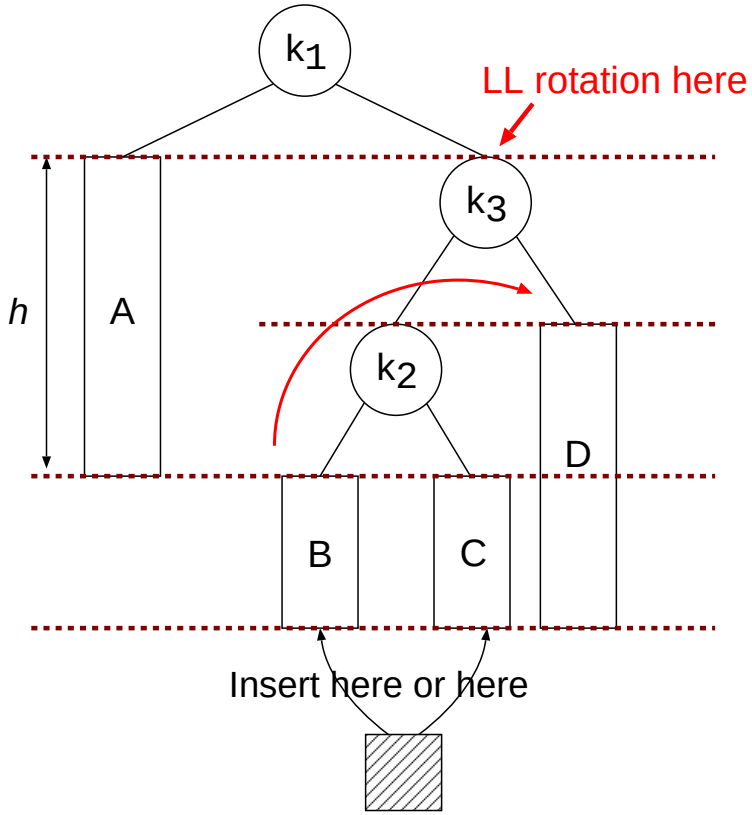
$h(D) \in \{h, h-1\}$

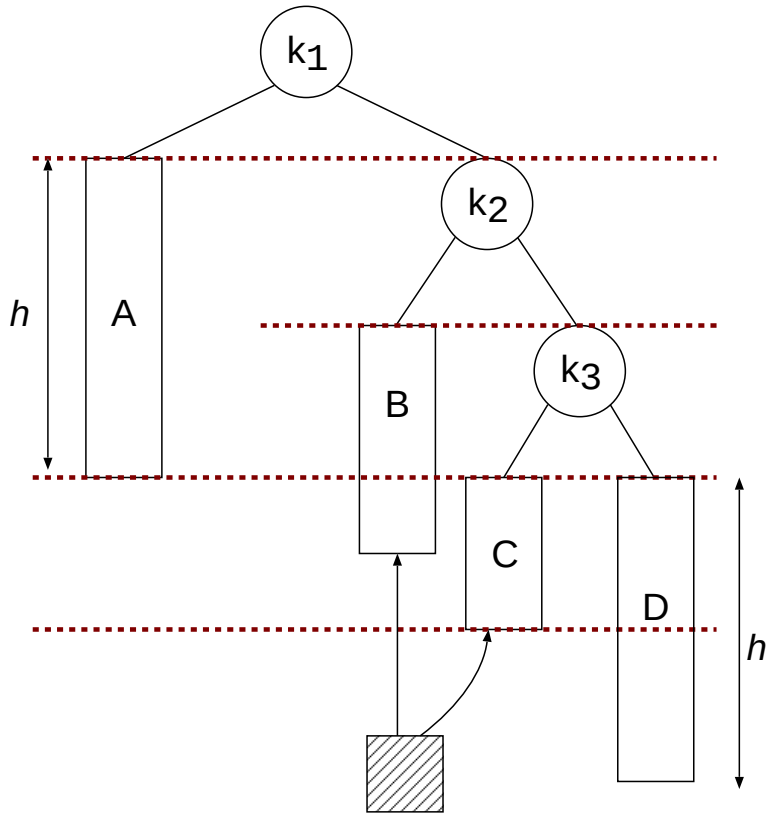
If $h(D) = h - 1$, imbalance would be observed

at k_3 before k_1

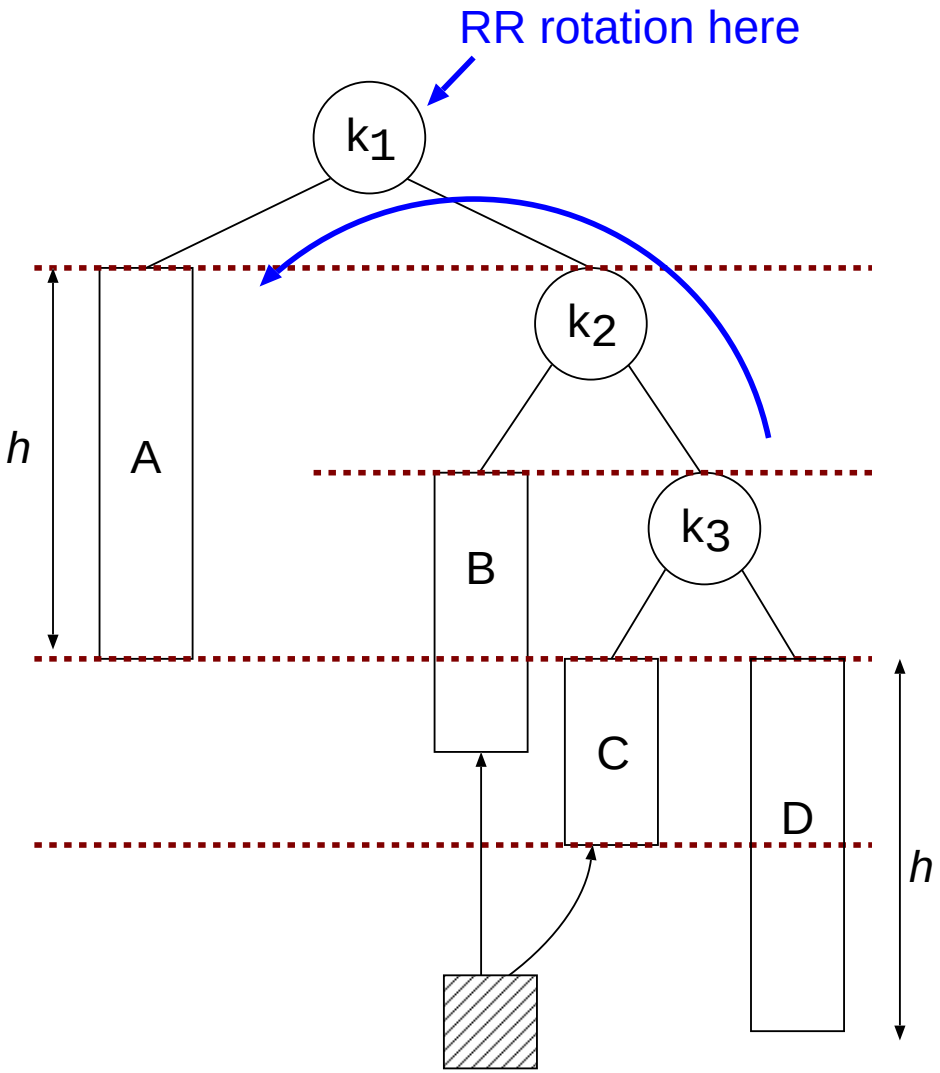
$\Rightarrow \; h(D) = h$

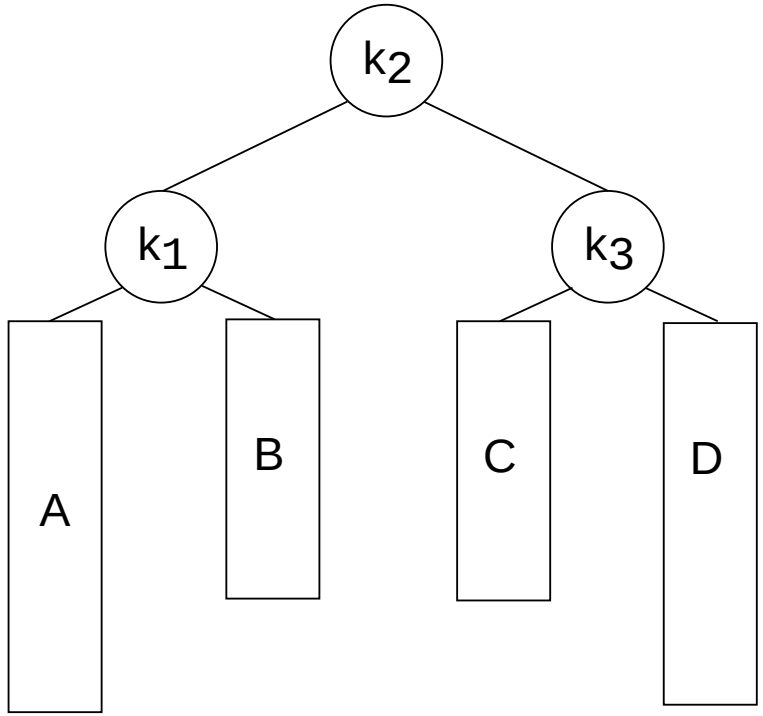
Rotate LR - I


k1
LL rotation here
k3
h
A
k2
D
B
C
Insert here or here


k1
k2
h
A
B
k3
C
D
h

Rotate LR - II





Rotate LR - code

```
def rotate_LR(self, parent, start_index) :
    # See CMSC 420 Lecture Notes by David M. Mount, UMCP, pg. 39.
    if DEBUG :
        print("LR (double) rotation at %d\n" % self.nodelist[
start_index].data)
    k1 = start_index
    self.nodelist[k1].right = self.rotate_LL(k1, self.nodelist[k1].
right)
    return self.rotate_RR(parent, start_index)
```

delete() I

```
def delete(self, parent, start_index, d) :
    thisnode = start_index
    if thisnode == -1 :
        return thisnode
    if d < self.nodelist[thisnode].data :
        if DDEBUG :
            print("Deleting %d recursively from left subtree of %d" %
                  (d, self.nodelist[thisnode].data))
        self.nodelist[thisnode].left = self.delete(thisnode, self.
nodelist[thisnode].left, d)
```

delete() II

```
        # DELETE THIS NODE
        if (self.nodelist[thisnode].left != -1 and
            self.nodelist[thisnode].right != -1) :
            successor = self.find_successor(thisnode)
            assert(successor != -1)
            if DDEBUG :
                print("Replacing", end=" ") ; self.print_node(thisnode)
                print("by successor", end=" ") ; self.print_node(
successor)
                print()
            self.nodelist[thisnode].data = self.nodelist[successor].
data
            self.nodelist[thisnode].right = self.delete(thisnode, self.
nodelist[thisnode].right, self.nodelist[successor].data)
        else :
            # EITHER LEAF or ONLY ONE CHILD
            if DDEBUG :
                print("Deleting", end=" ") ; self.print_node(thisnode)
```

delete() III

```
                print("\n")
            if self.nodelist[thisnode].left != -1 :
                thisnode = self.nodelist[thisnode].left
                self.nodelist[thisnode].parent = parent
            elif self.nodelist[thisnode].right != -1 :
                thisnode = self.nodelist[thisnode].right
                self.nodelist[thisnode].parent = parent
            else :
                thisnode = -1

        self.num_nodes -= 1
        return thisnode
```

delete() IV

```
    # Deletion done, now rebalance and update height
    start_index = self.balance(parent, start_index)
    left = self.nodelist[start_index].left
    right = self.nodelist[start_index].right
    self.nodelist[start_index].height = 1 + \
        max(self.height(left), self.height(right))
    return start_index
```

Problems I

1. Complete the missing parts (marked `TODO`) in the provided implementation of AVL trees.
2. Given a tree that is supposed to be an AVL tree, write a function to check whether it is indeed an AVL tree, and whether all fields have correct / consistent values.

   You will need to check whether the `left`, `right`, and `parent` fields are consistent, whether the `height` field is correct (if not, fill in the field with the correct value), and finally whether imbalances (if any) are within the permissible limit.

   For this problem, the tree will be given to you via an input file, using a format similar to the one used in Lab Test 2. The name of the input file will be given as a command-line argument.
3. Use the function in problem 2 to test (and debug if necessary) the AVL tree implementation in program 1.

Problems II

4. Consider an array A that contains elements from an ordered set. Two elements $A[i]$ and $A[j]$ of the array are said to form an *inversion* if $A[i] > A[j]$ and $i < j$. Write a program to count the number of inversions in a given array.
   Note that the inversion count indicates how far (or close) the array is from being sorted in ascending order. If the array is already sorted then the inversion count is 0. If the array is sorted in reverse order, then the inversion count is the maximum. For example, if the given array is $8\ 4\ 2\ 1$, your program should output 6 (the six inversions are $(8,4), (8,2), (8,1), (4,2), (4,1), (2,1)$).