빅 데 이 터 혁 신 공 유 대 학

파이썬으로 배우는 데이터 구조

한동대학교 전산전자공학부

김영섭 교수

교육부 NRF 한국연구재단

Data Structures in Python
Chapter 7 - 2

- Binary Search Tree(BST)
- BST Algorithms
- AVL Tree
- **AVL Algorithms**

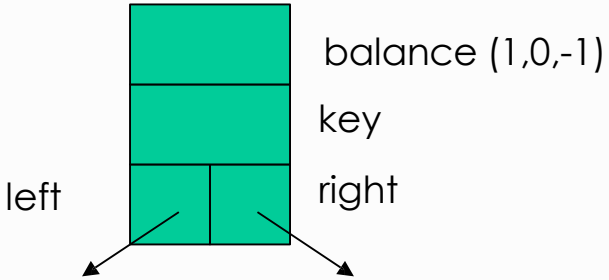
Agenda & Readings

- AVL Tree
  - Balance Factor
  - Rotations:
    - Single Rotation - LL and RR cases (Outside case)
    - Double Rotation - LR and RL cases (Inside case)
  - Define AVL Class
  - Coding:
    - height(), balanceFactor(), _add() & _delete(), rebalance()

- Reference:
  - Problem Solving with Algorithms and Data Structures
    Chapter 6 - Tree
  - Wikipedia: AVL tree

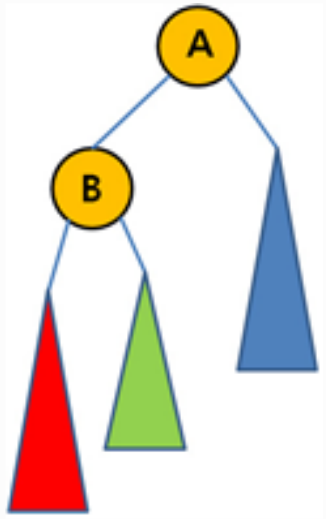
Balance Factor

- You can either keep the height or just the difference in height, i.e., the balance factor; this must be modified on the path of insertion or deletion even if you do not perform rotations.
  - Once you have performed a rotation (single or double) you won't need to go back up the tree for the computation.
- You may compute the balance factor **on the fly** after the insert is done during the recursion.

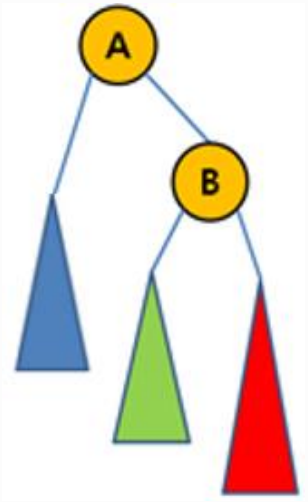


Single Rotation(Outside Case)

LL Case



RR Case

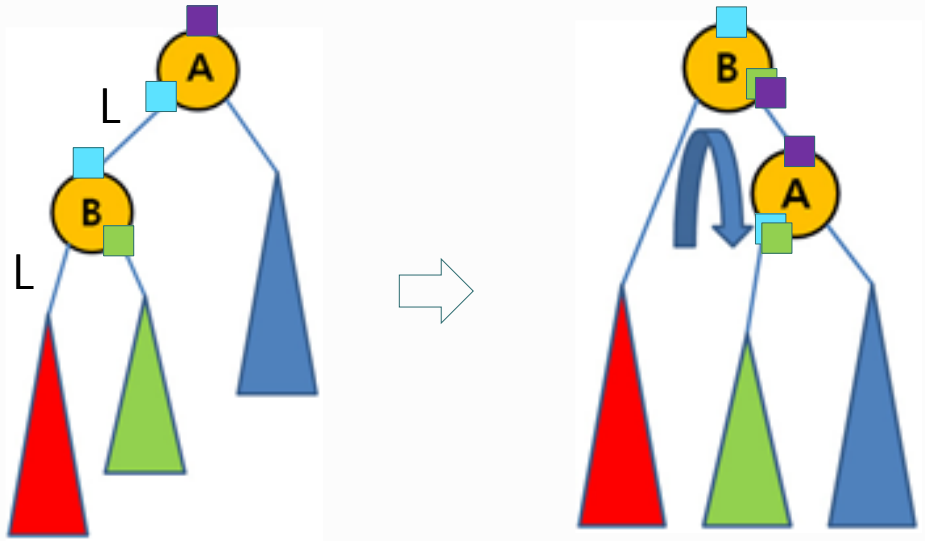


```
def LLcase(A):

    return
```

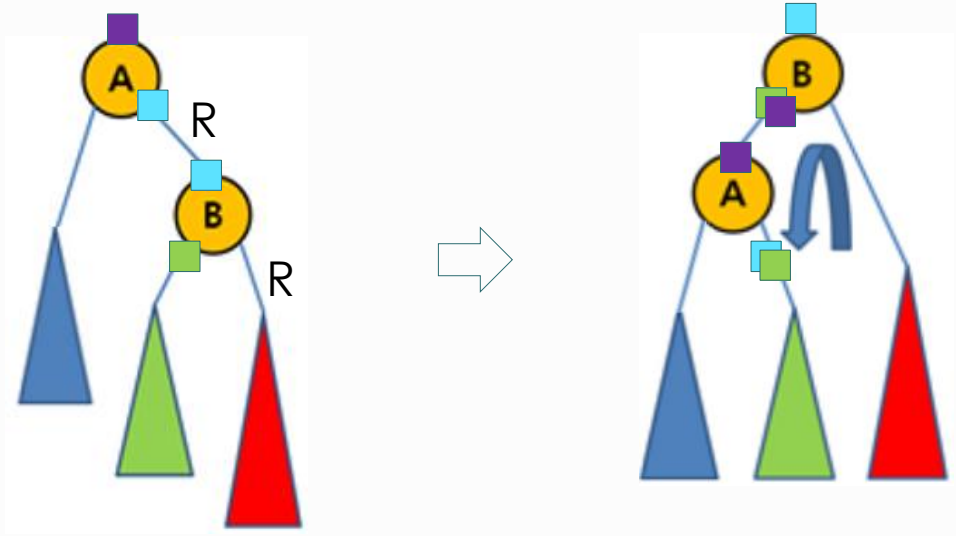
```
def RRcase(A):

    return
```

Single Rotation(Outside Case)

LL Case



RR Case

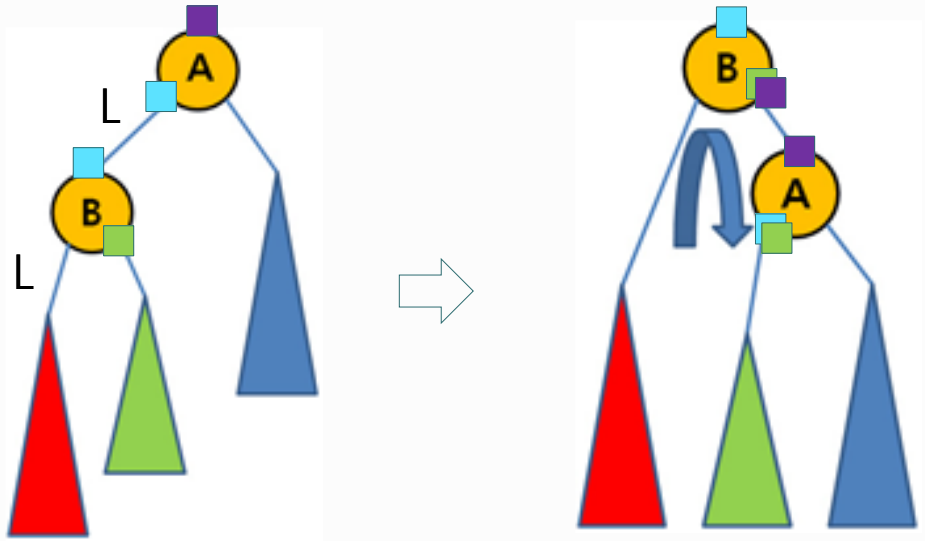



```
def LLcase(A):
    
    
    
    return
```

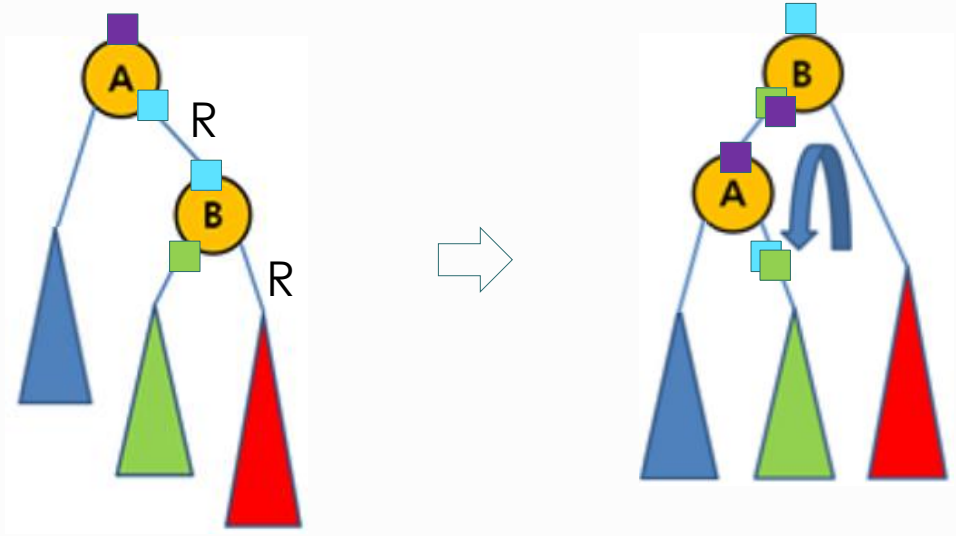
```
def RRcase(A):
    
    
    
    return
```

Single Rotation(Outside Case)

LL Case



RR Case

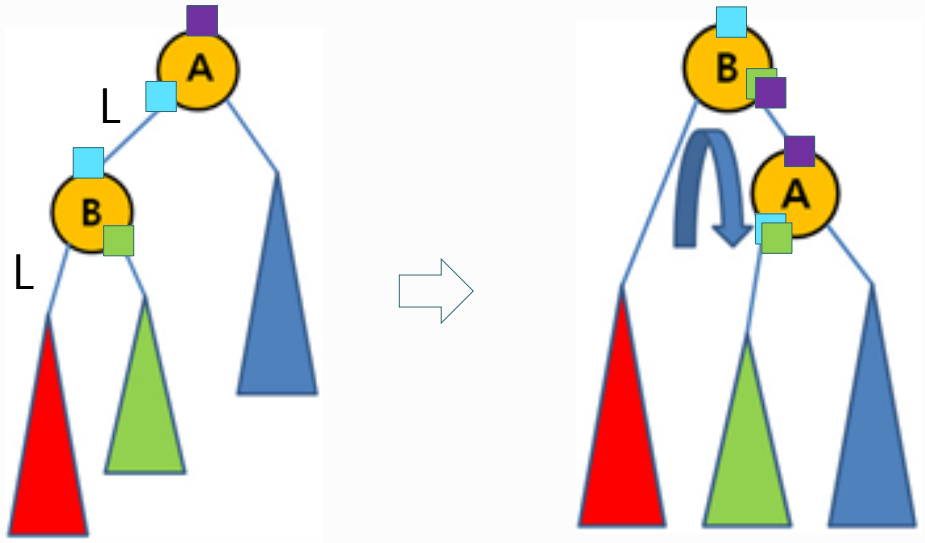



```
def LLcase(A):
    B = A.left
    return
```

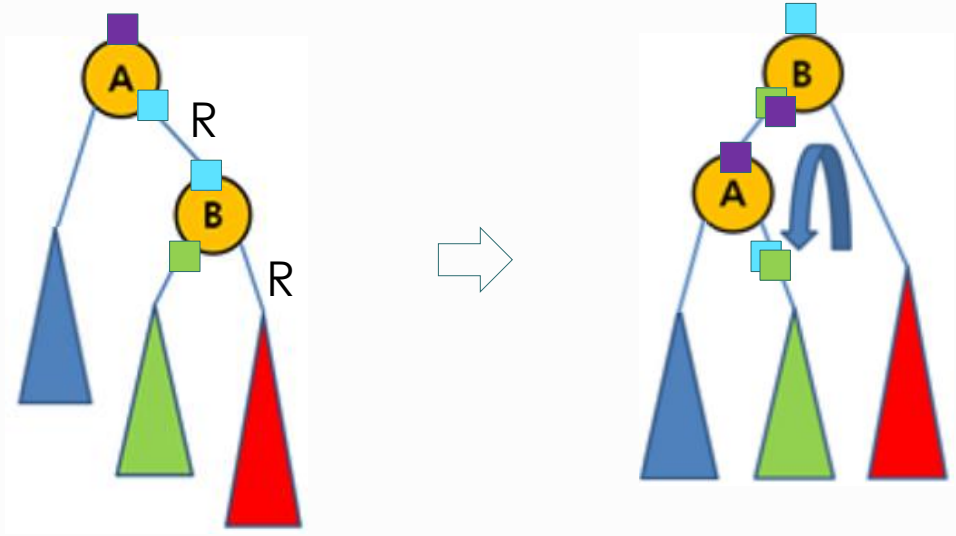
```
def RRcase(A):
    return
```

Single Rotation(Outside Case)

LL Case



RR Case

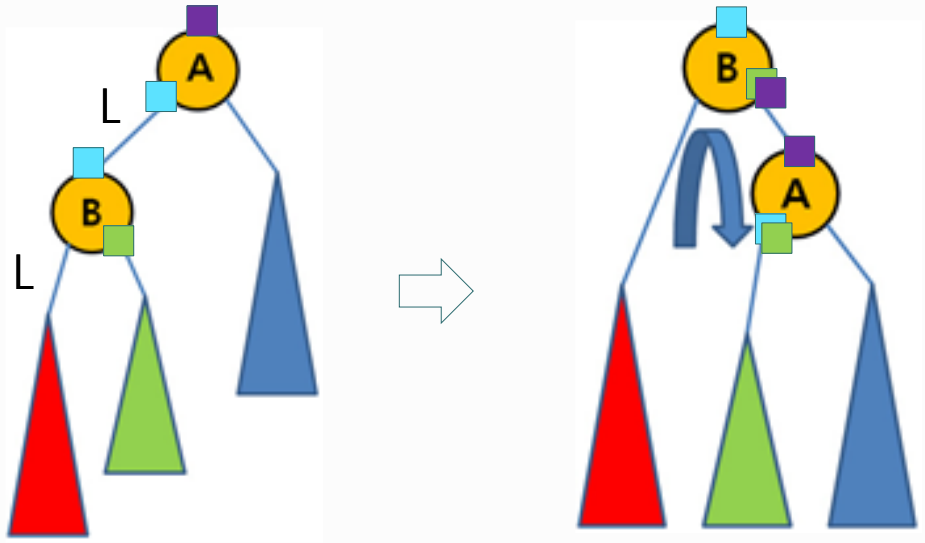



```
def LLcase(A):
    B = A.left
    A.left  = B.right

    return
```

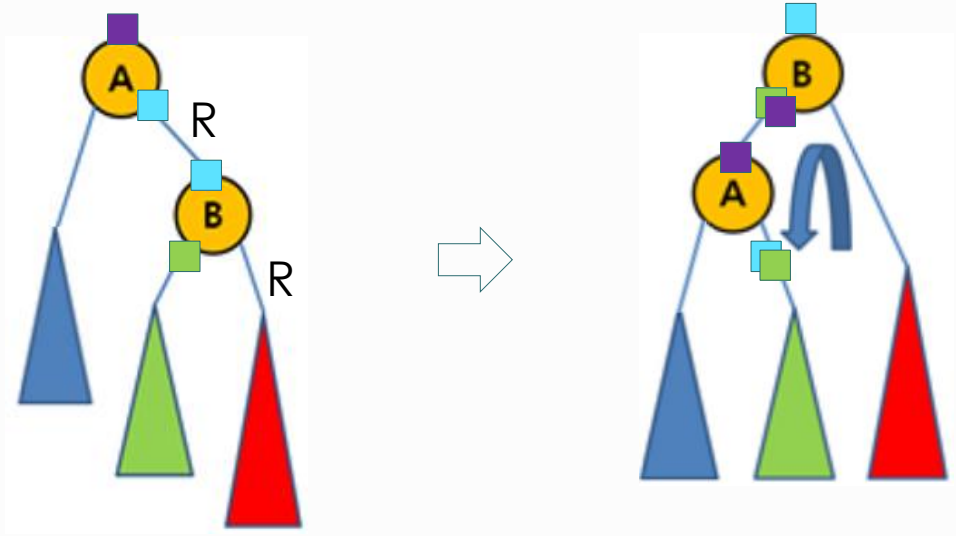
```
def RRcase(A):

    return
```

Single Rotation(Outside Case)

LL Case



RR Case

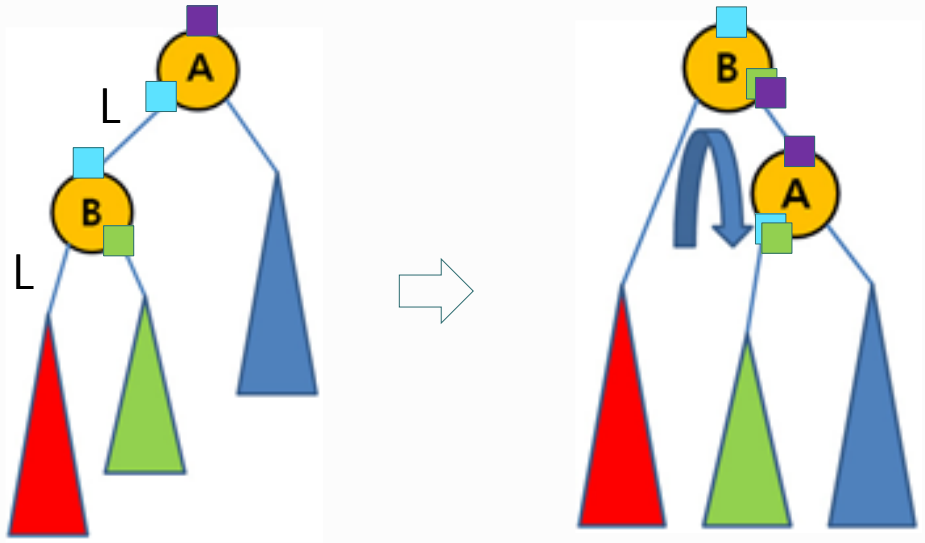



```
def LLcase(A):
    B = A.left
    A.left  = B.right
    B.right = A
    return
```

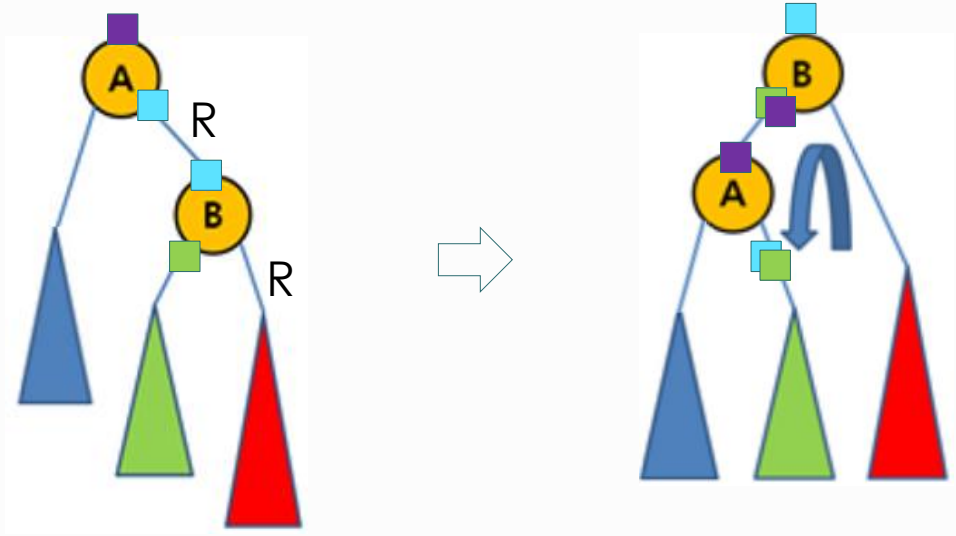
```
def RRcase(A):
    return
```

Single Rotation(Outside Case)

LL Case



RR Case

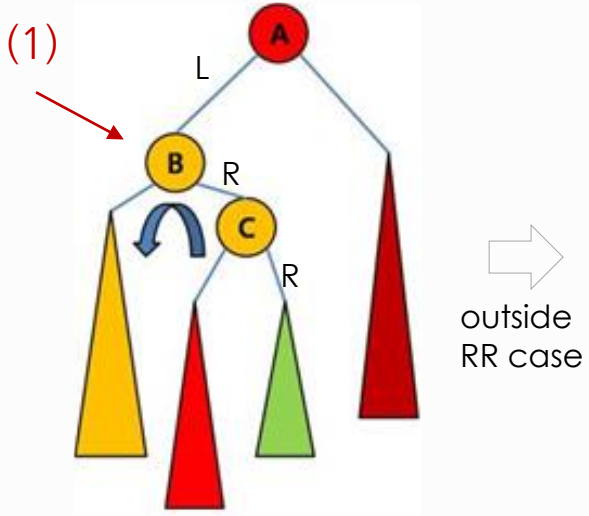



```
def LLcase(A):
    B = A.left
    A.left  = B.right
    B.right = A
    return B
```

```
def RRcase(A):
    
    
    
    return
```

Double Rotation(Inside Case)

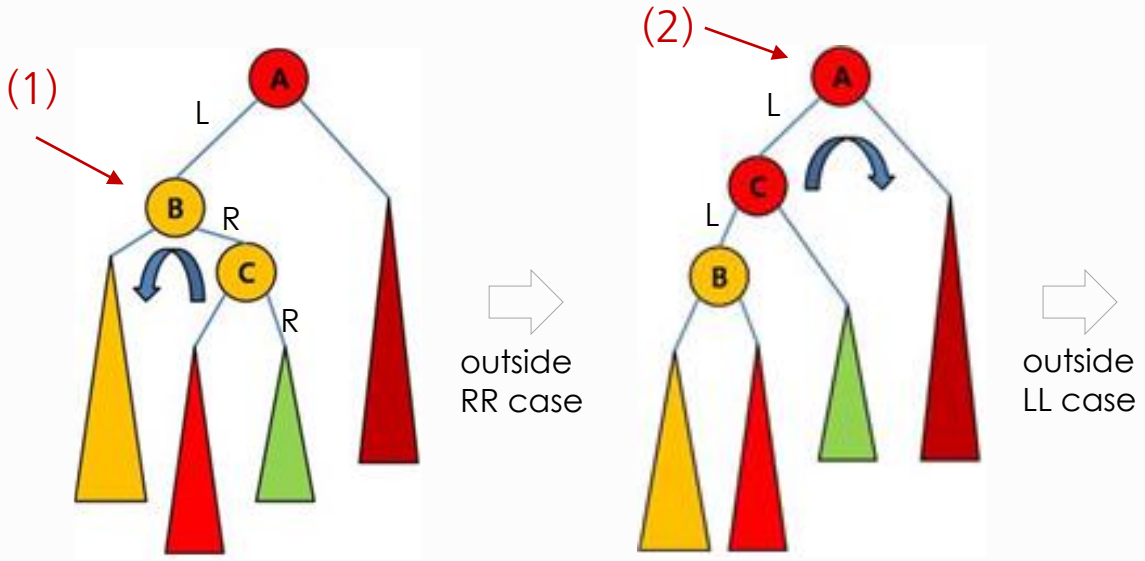
LR case




```
def LRcase(A):
```

Double Rotation(Inside Case)

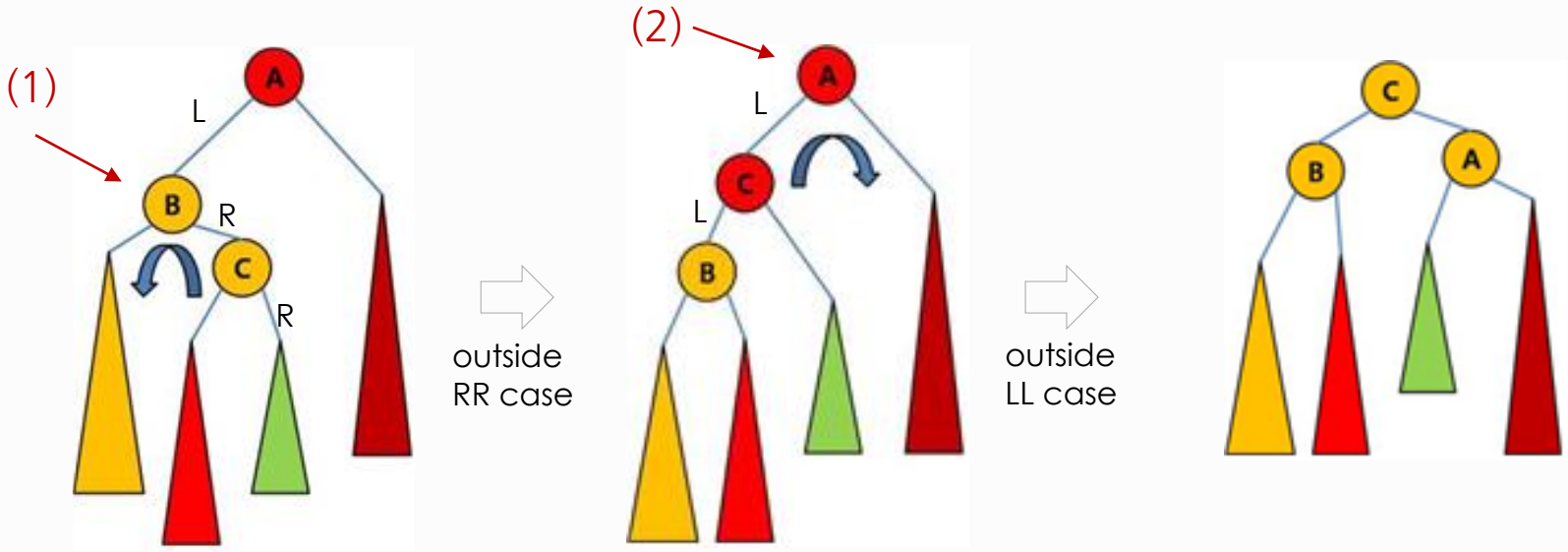
LR case




```
def LRcase(A):
```

Double Rotation(Inside Case)

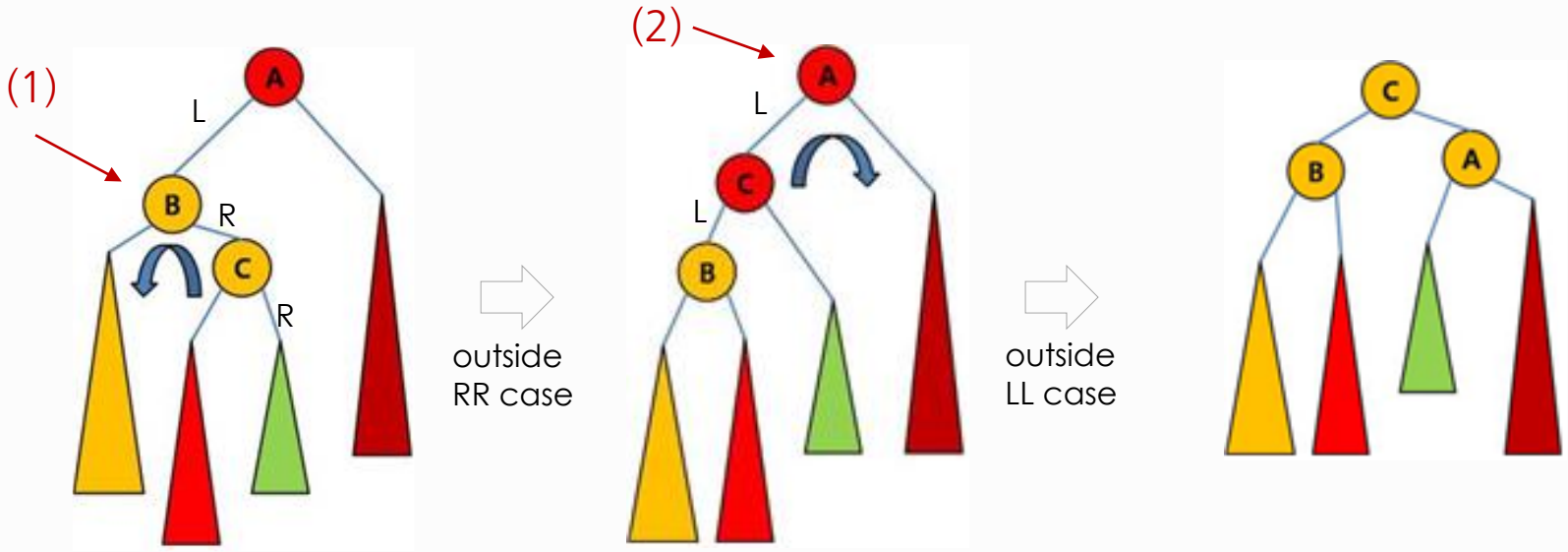
LR case




```
def LRcase(A):
```

Double Rotation(Inside Case)

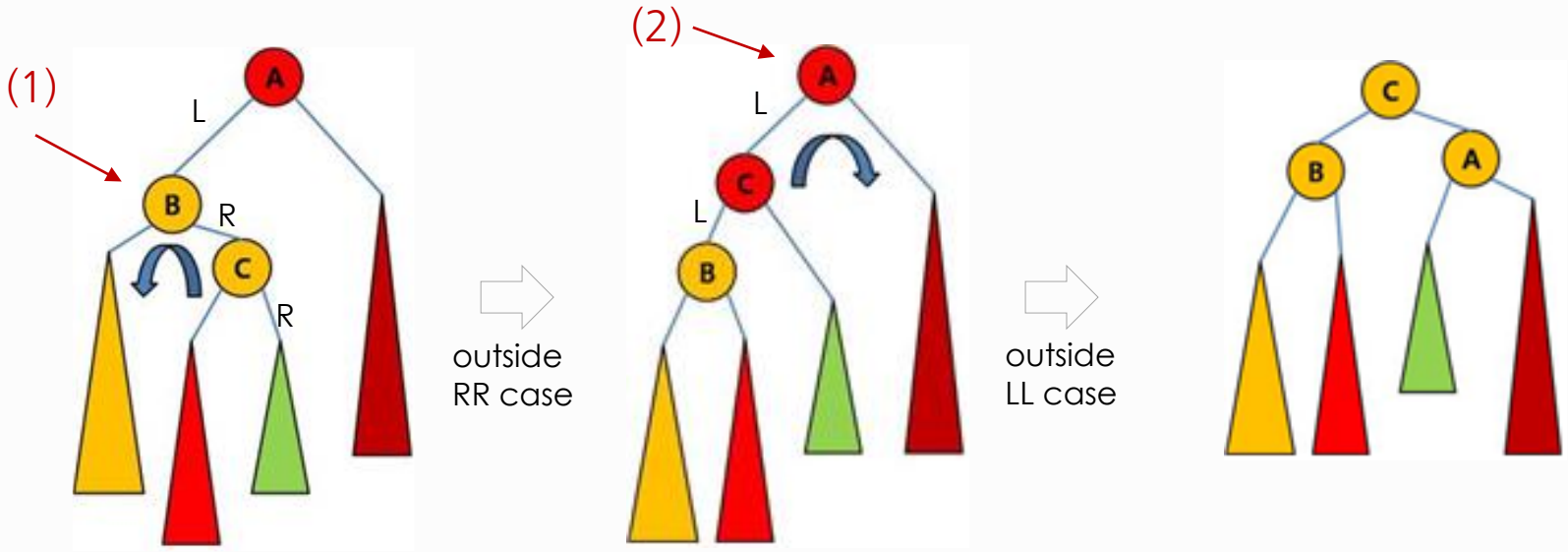
LR case




```
def LRcase(A):
    B = A.left
```

Double Rotation(Inside Case)

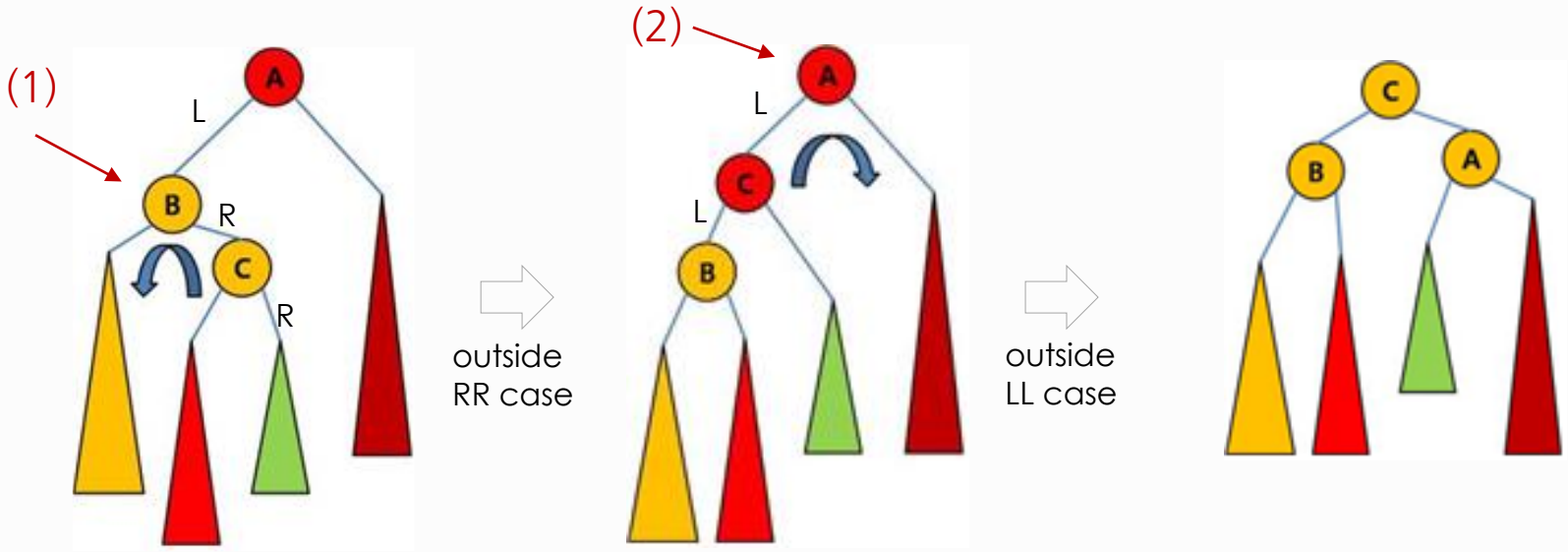
LR case




```
def LRcase(A):
    B = A.left
    A.left = RRcase(B)
```

Double Rotation(Inside Case)

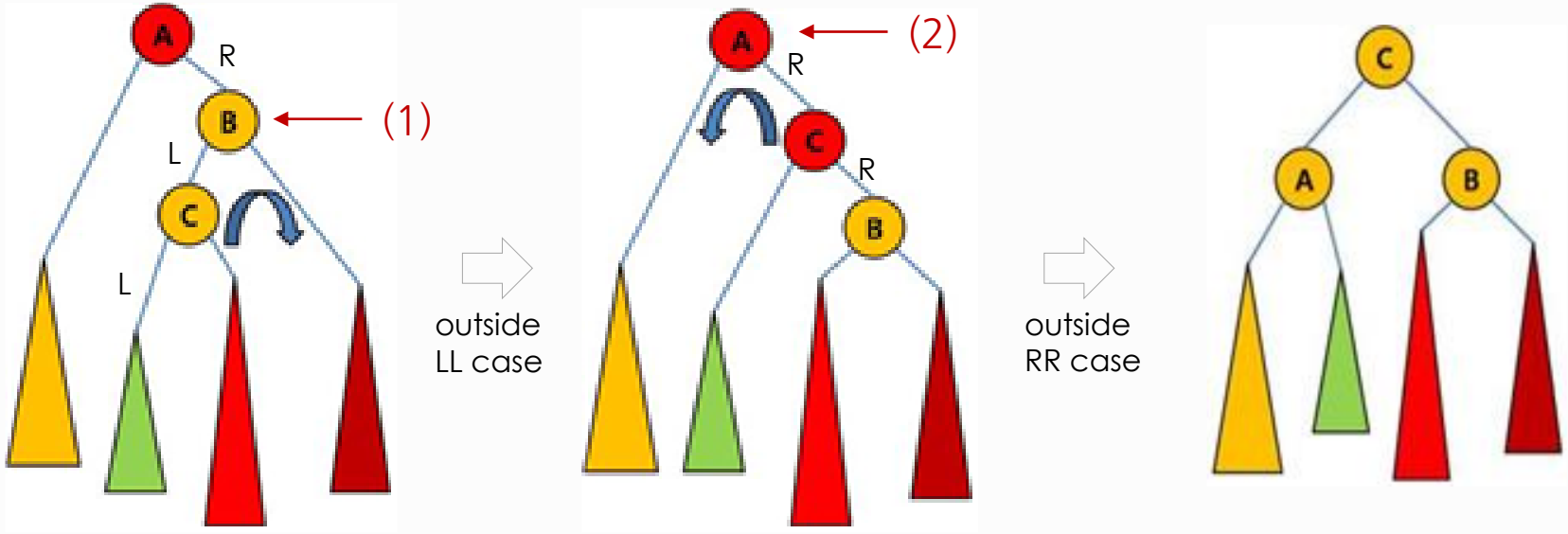
LR case




```
def LRcase(A):
    B = A.left
    A.left = RRcase(B)
    return LLcase(A)
```

Double Rotation(Inside Case)

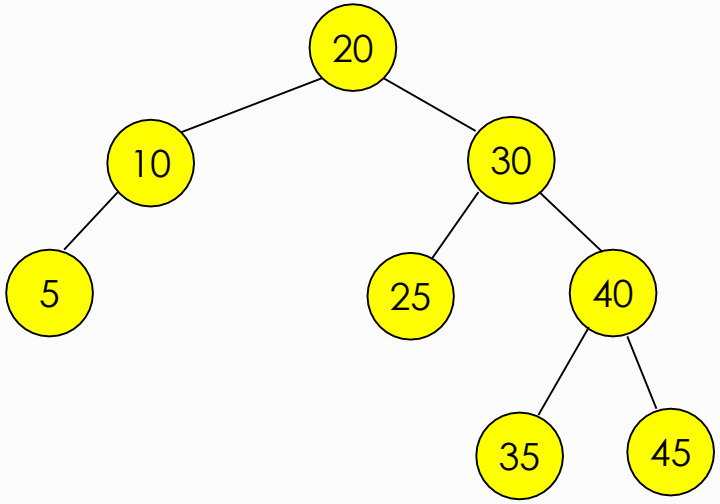
RL case




```
def RLcase(A):
```

Double Rotation - ??case

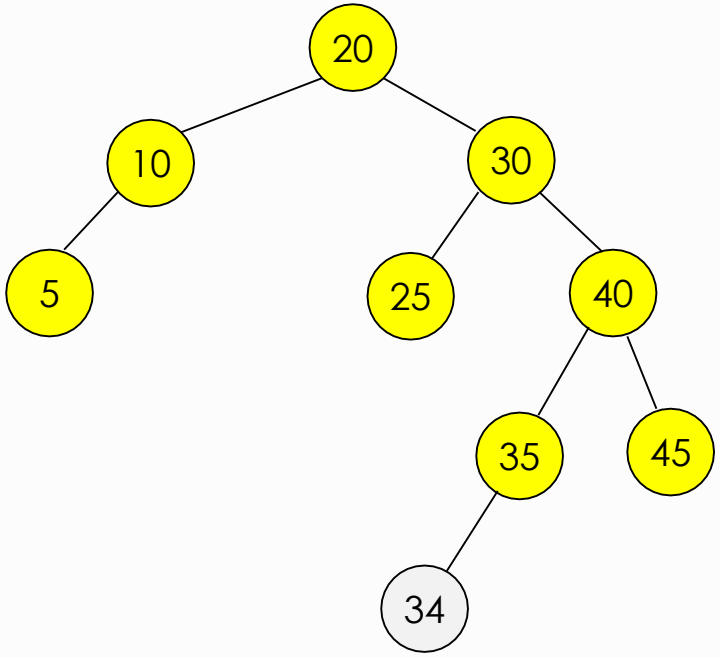
- **Insertion of 34**
- Imbalance at ?
- Balance factor ??
- Rotation ___??___ case



AVL balanced tree

Double Rotation - ??case

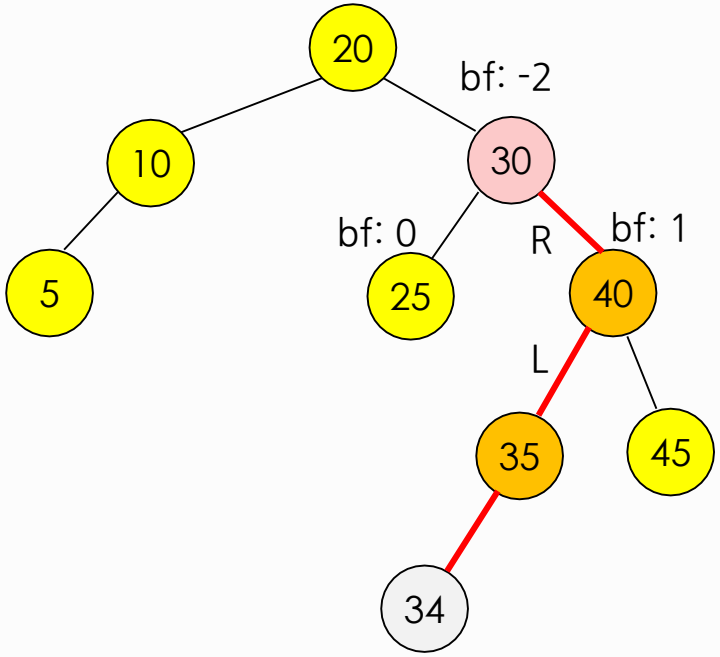
- Insertion of 34
- Imbalance at ?
- Balance factor ??
- Rotation ___??___ case



After insertion, it is AVL imbalanced at ?

Double Rotation - ??case

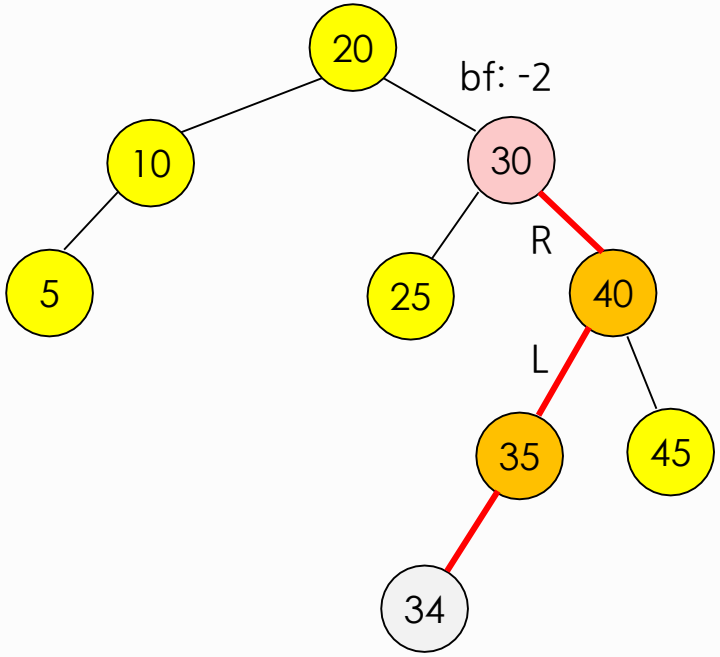
- Insertion of 34
- Imbalance at 30
- Balance factor **-2**
- Rotation ___??___ case



Double Rotation - RL case

- Insertion of 34
- Imbalance at 30
- Balance factor -2
- Rotation ___RL___ case

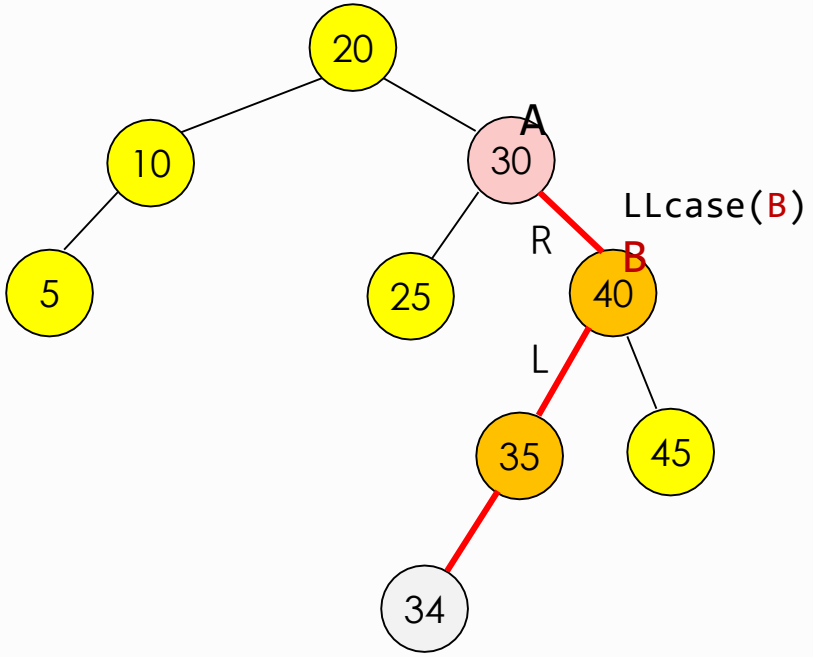
```
def RLcase(A):
  B = A.right
  A.right = LLcase(B)
  return RRcase(A)
```



Double Rotation - RL case

- Insertion of 34
- Imbalance at 30
- Balance factor -2
- Rotation ___RL___ case

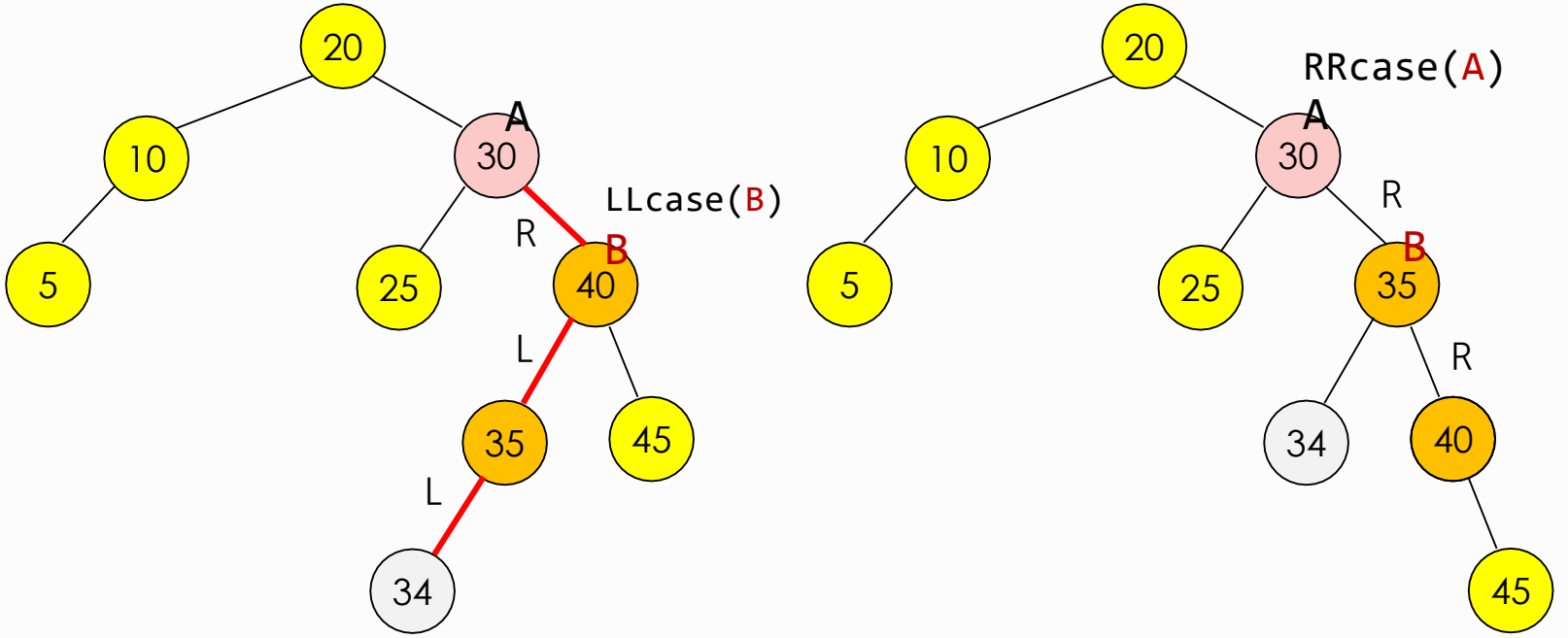
```
def RLcase(A):
  B = A.right
  A.right = LLcase(B)
  return RRcase(A)
```



Double Rotation - RL case

- Insertion of 34
- Imbalance at 30
- Balance factor -2
- Rotation ___RL___ case

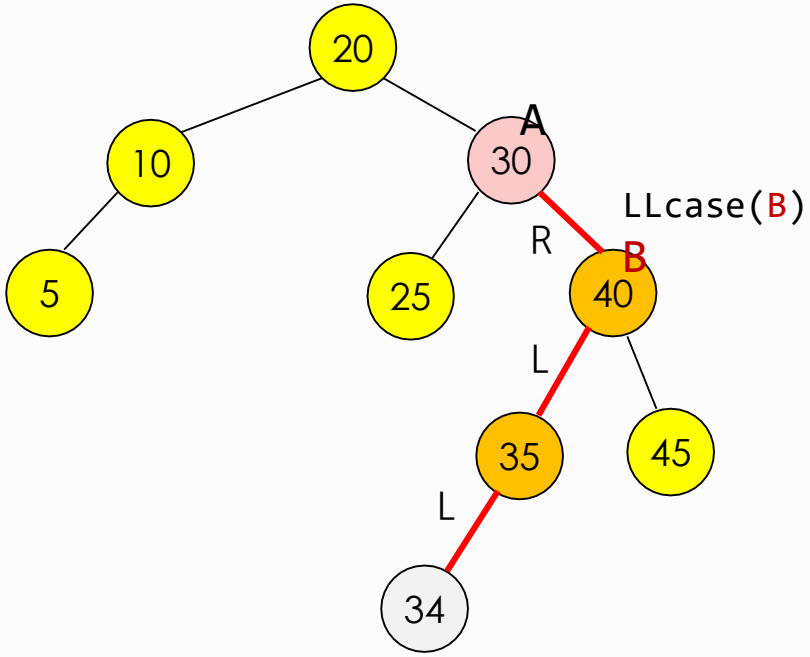
```
def RLcase(A):
  B = A.right
  A.right = LLcase(B)
  return RRcase(A)
```

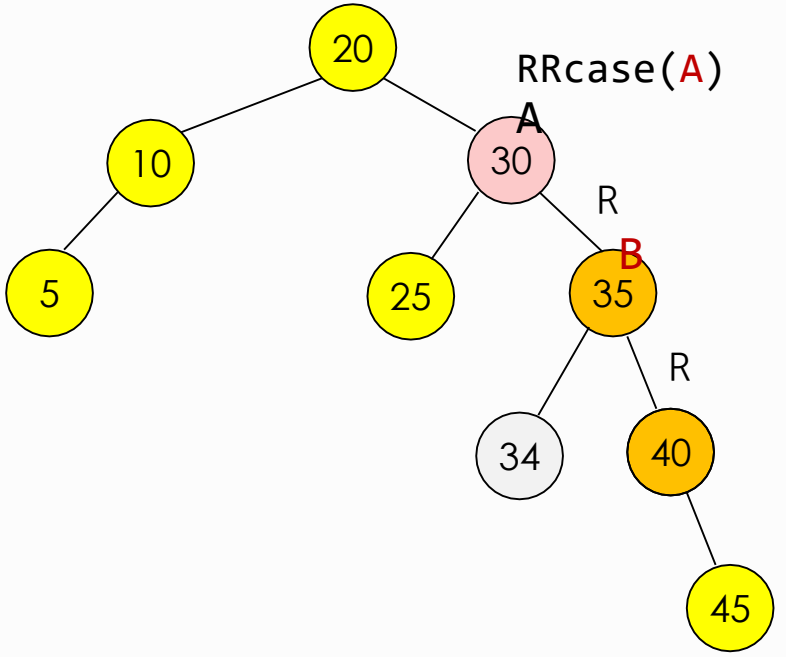


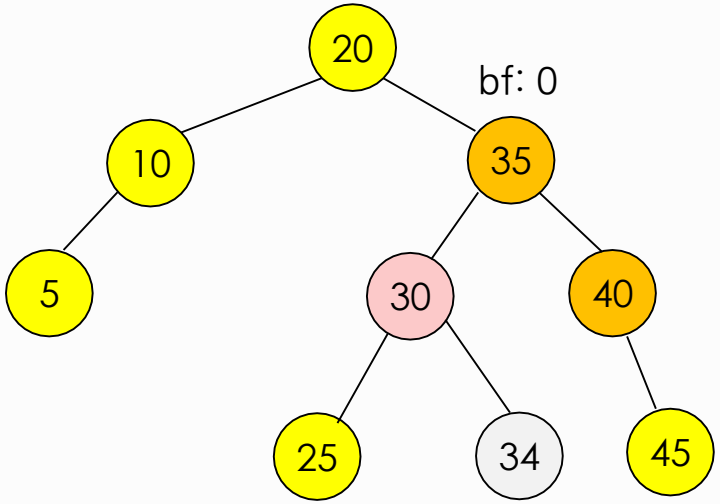
Double Rotation - RL case

- Insertion of 34
- Imbalance at 30
- Balance factor -2
- Rotation ___RL___ case

```
def RLcase(A):
  B = A.right
  A.right = LLcase(B)
  return RRcase(A)
```







After insertion, AVL imbalanced tree

After insertion, AVL balanced tree

height() and _height()

```
def height(self):
    return self._height(self.root)
```

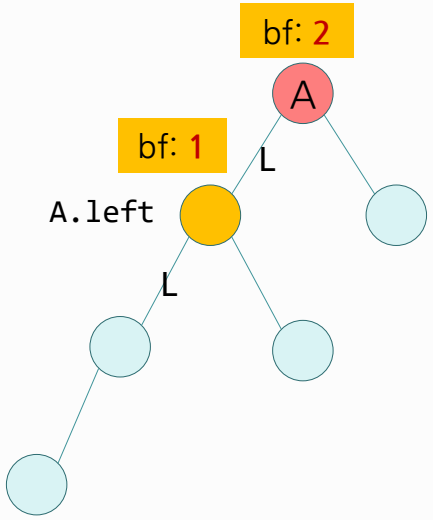
```
def _height(self, node):
    if node is None: return -1  # if empty

    l = self._height(node.left)
    r = self._height(node.right)
    return max(l, r) + 1
```

_add() & _delete()

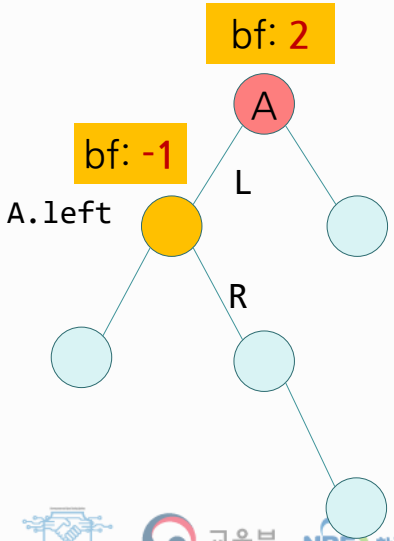
```
class AVL(BST):
    def _add(self, node, key):
        curr = super()._add(node, key)
        return self.rebalance(curr)

    def _delete(self, node, key):
        curr = super()._delete(node, key)
        return self.rebalance(curr)

    def rebalance(self, A):
        def balanceFactor(A):
            if A == None: return
            l = self._height(A.left)
            r = self._height(A.right)
            return l - r
        # your code here: define LLcase(), RRcase(), LRcase(), RLcase()
        # your code here: invoke rotate functions depending on balance factor.
        pass
```

rebalance()

LL Case



LR case




```
def rebalance(A):
    ...
    bf = balanceFactor(A)
    if bf == 2:
        bf:1, 0
        bf: -1

    elif bf == -2:



    return A          # no rebalanced needed
```

checking single or double rotation

LL Case

LR case

Observation: If A and its child have the same sign in bf's, a single rotation is needed, a double rotation otherwise.
If the bf of A's child is 0, treat it like the same sign of A.

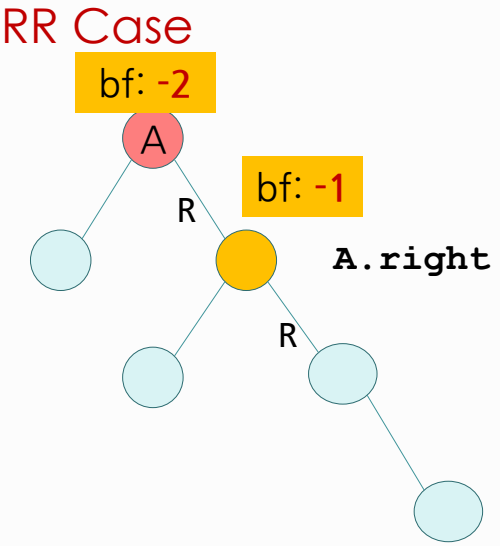
rebalance()

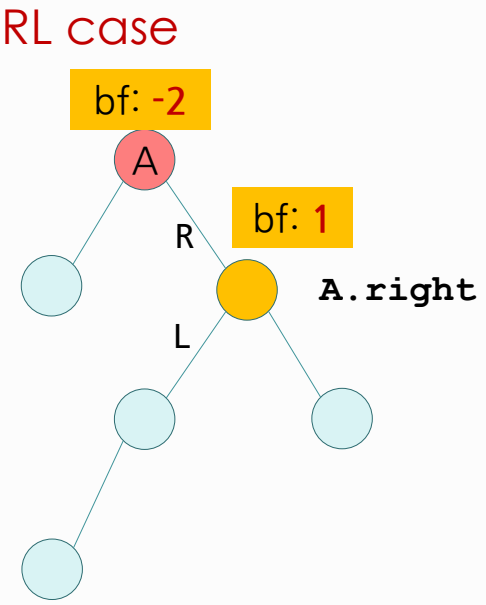
```
def rebalance(A):
    ...
    bf = balanceFactor(A)
    if bf == 2:



    elif bf == -2:
        bf:-1, 0
        bf: 1

    return A          # no rebalanced needed
```

checking single or double rotation

RR Case

RL case

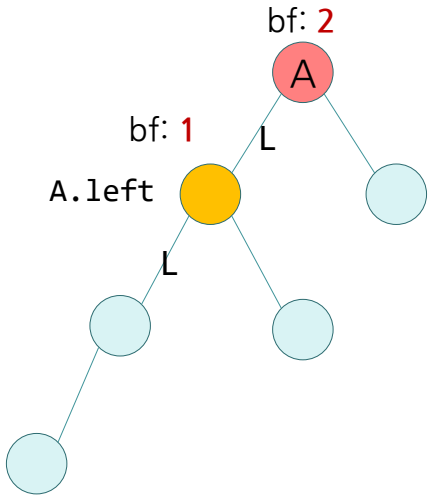




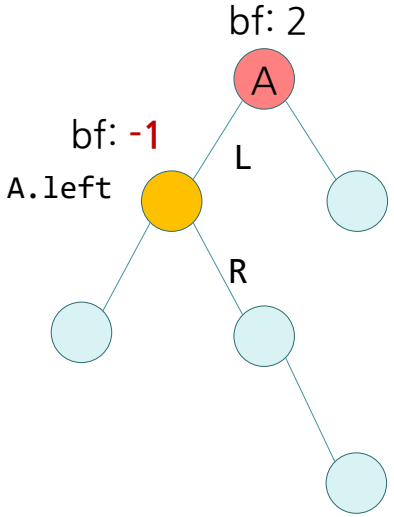
Observation: If A and its child have the same sign in bf's, a single rotation is needed, a double rotation otherwise.
If the bf of A's child is 0, treat it like the same sign of A.

rebalance()

LL Case



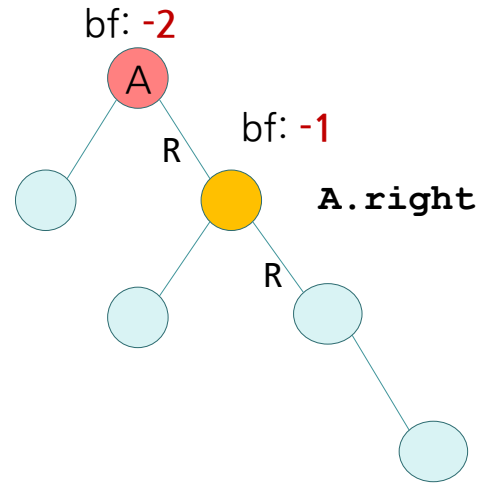
LR case



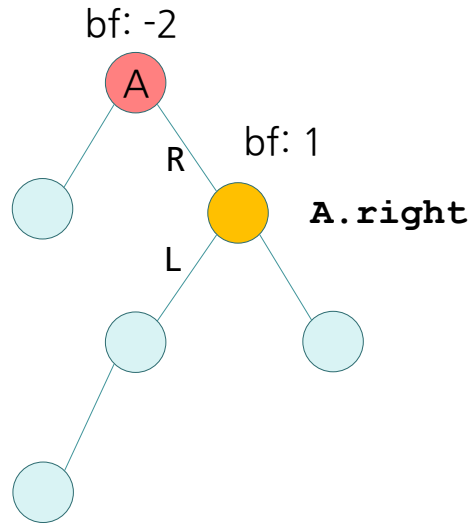

```
def rebalance(A):
    ...
    bf = balanceFactor(A)
    if bf == 2:

                                        LL Case

                                        LR case

    elif bf == -2:

                                        RR Case

                                        RL case

    return A          # no rebalance needed
```

Observation: If A and its child have the same sign in bf's, a single rotation is needed, a double rotation otherwise.
If the bf of A's child is 0, treat it like the same sign of A.

RR Case

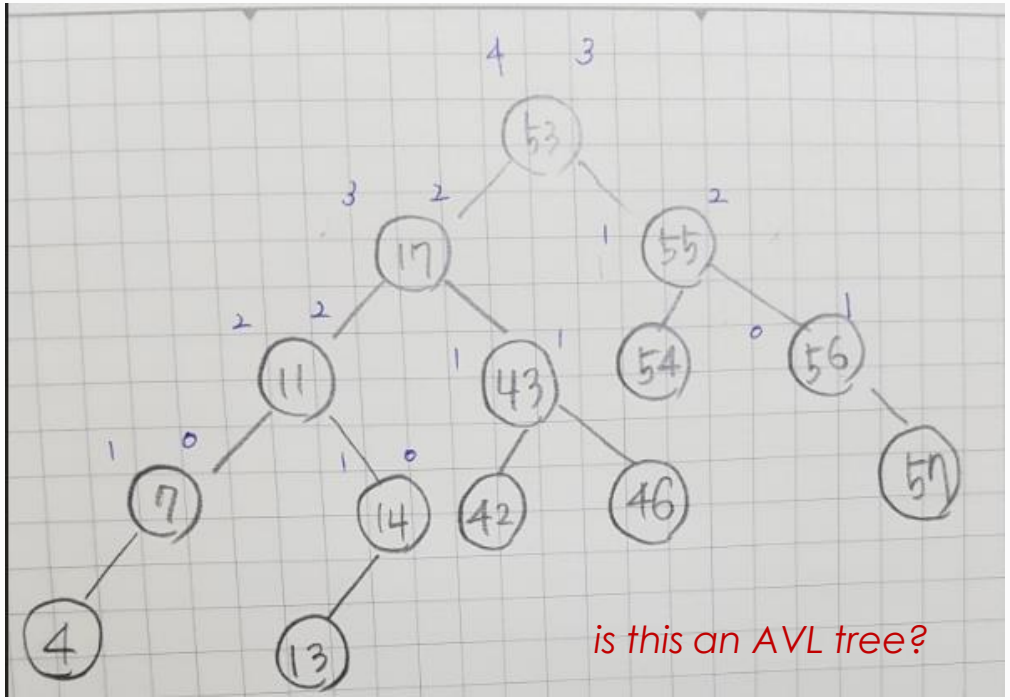


RL case



Summary

- AVL tree is binary search tree(BST) that balances every time a node is **inserted or deleted**.
- **Each node** of an AVL tree has the property that the heights of the sub-tree rooted at its children differ **by at most one**.
- The maximum height of an AVL tree having exactly n nodes is $\mathbf{h} \leq \mathbf{1.44} * \boldsymbol{log_2 n}$.
  For proof, refer to Wikipedia AVL tree.



Data Structures in Python
Chapter 7 - 2

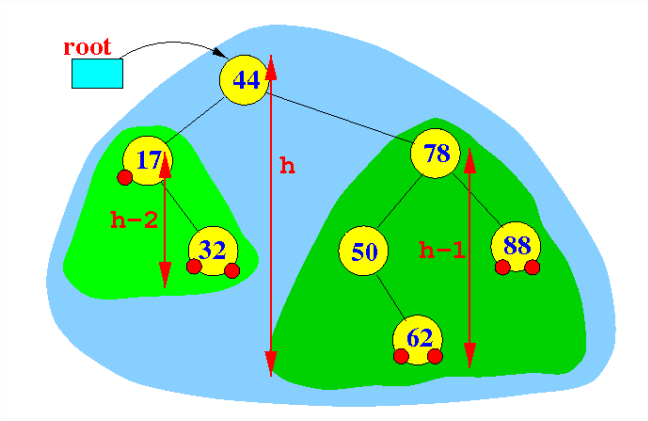
- Binary Search Tree(BST)
- BST Algorithms
- AVL Tree
- **AVL Algorithms**

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University

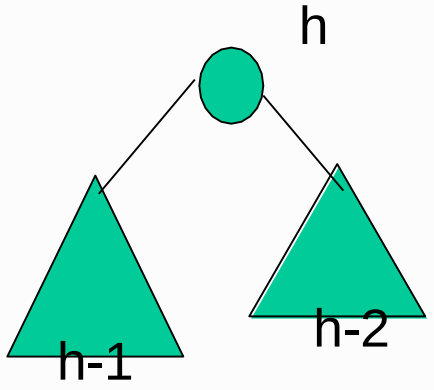
Height of an AVL Tree

- What is the maximum height of an AVL tree having exactly n nodes?
  - To answer this question, we must ask this question first:
    What is the minimum number of nodes (sparsest possible AVL tree) an AVL tree of height h?

- Consider **the minimum number of nodes** in an AVL tree of height h:
  We can get the recurrence relationship:
  $n(0) = 1, n(1) = 2, n(2) = 4, \cdots$
  $n(h) = n(h-1) + n(h-2) + 1$, where $h > 1$
  This approximate solution of the recurrence is known as $n(h) \cong 1.618^h$



- **Solve the equation** above for h to get **the max height of an AVL tree** with n
  $\log_2 n \geq h * \log_2 1.62$
  $h \leq 1/\log_2 1.618 * \log_2 n$
  $\mathbf{h \leq 1.44 * log_2 n}$



Height of an AVL Tree

- AVL trees are binary search trees that balances itself every time an element is **inserted or deleted**. **Each node** of an AVL tree has the property that the heights of the sub-tree rooted at its children differ **by at most one**.

- If there are n nodes in AVL tree, minimum height of AVL tree is floor$(\log_2 n)$.
- If there are n nodes in AVL tree, maximum height can't exceed $1.44 * \log_2 n$.
- If height of AVL tree is h, maximum number of nodes can be $2^{h+1} - 1$.
- Minimum number of nodes in a tree with height h can be represented as:
  N(h) = N(h-1) + N(h-2) + 1, where N(0) = 1 and N(1) = 2.
- The complexity of searching, inserting and deletion in AVL tree is O$(\log_2 n)$.
- The cost of balancing AVL tree is O(1).
  What is the time complexity of adding N elements to an empty AVL tree?
  Time complexity: log(1) + log(2) + …. + log(n) <= log(n) + log(n) + … + log(n) = n log (n)