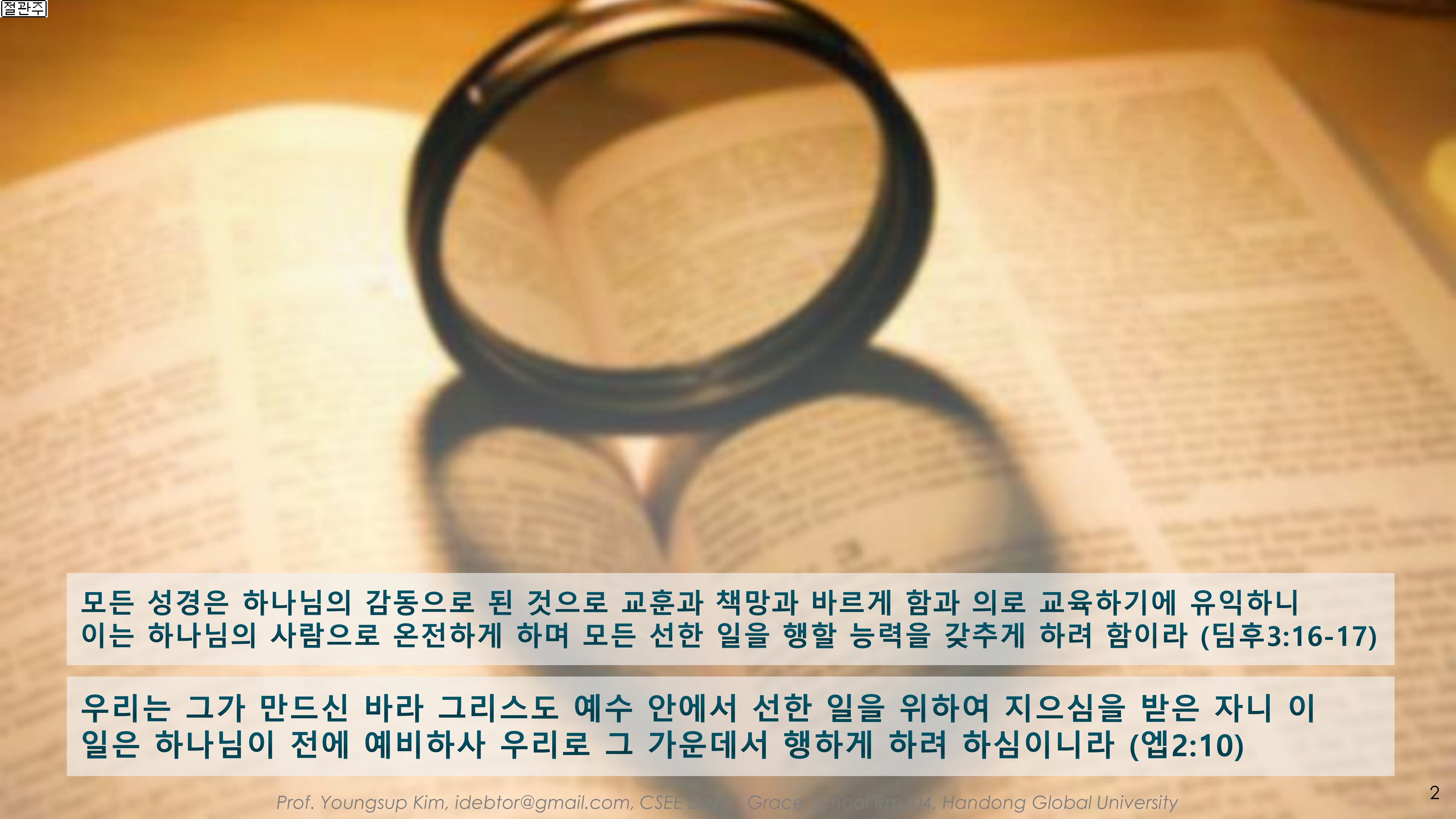
Data Structures
Chapter 5 Tree

1. Introduction
2. Binary Tree
3. Binary Search Tree
4. **Balancing Tree**
   - AVL Tree Introduction
   - **AVL Operations**
   - AVL Coding

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

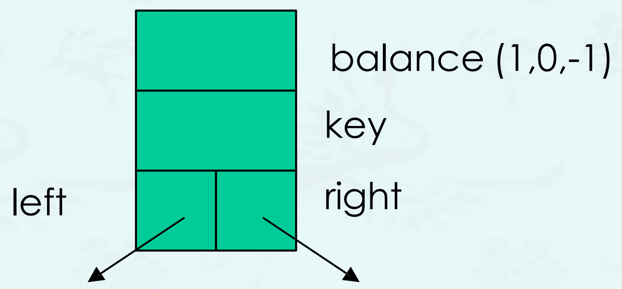


모든 성경은 하나님의 감동으로 된 것으로 교훈과 책망과 바르게 함과 의로 교육하기에 유익하니 이는 하나님의 사람으로 온전하게 하며 모든 선한 일을 행할 능력을 갖추게 하려 함이라 (딤후3:16-17)

우리는 그가 만드신 바라 그리스도 예수 안에서 선한 일을 위하여 지으심을 받은 자니 이 일은 하나님이 전에 예비하사 우리로 그 가운데서 행하게 하려 하심이니라 (엡2:10)

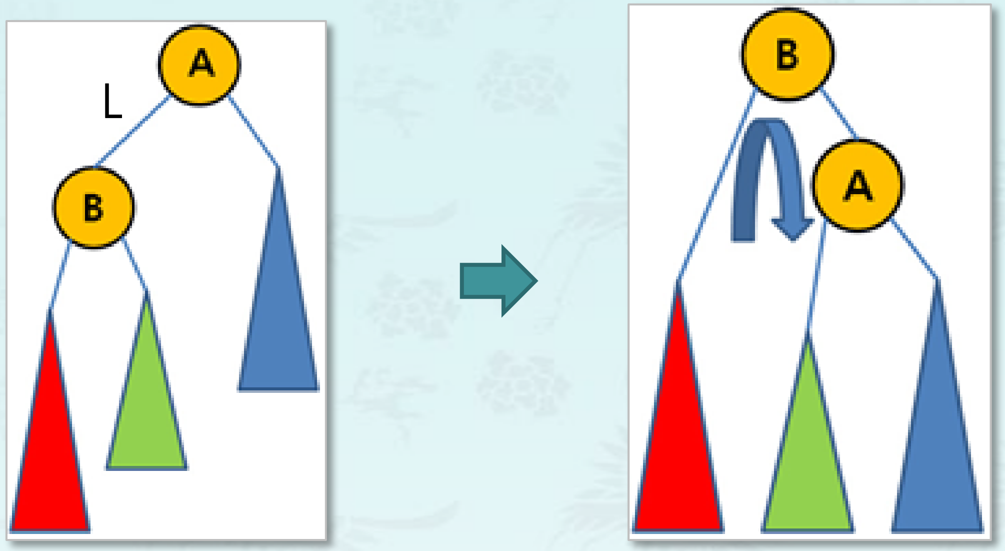
Coding

- You can either keep the height or just the difference in height, i.e. the balance factor; this has to be modified on the path of insertion even if you don't 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 - LL case

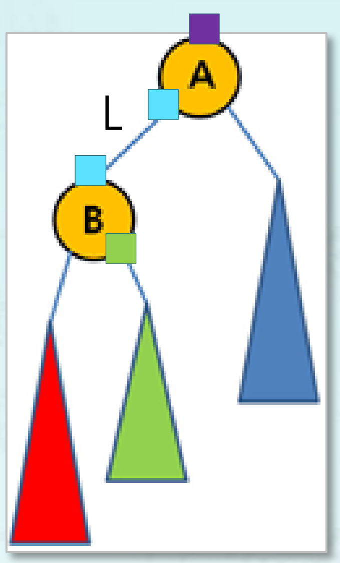
outside case

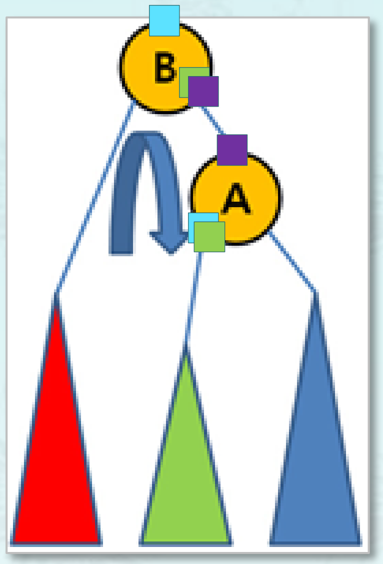


```
tree rotateLL(tree A)
{



  return
}
```

Single Rotation - LL case

outside case

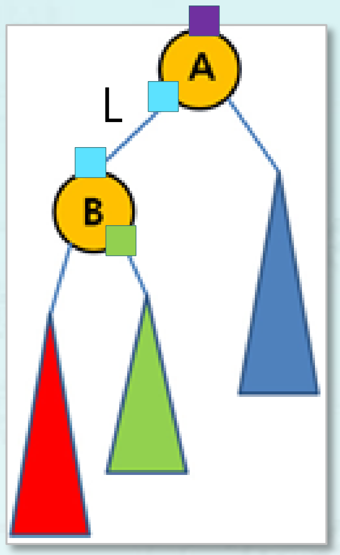


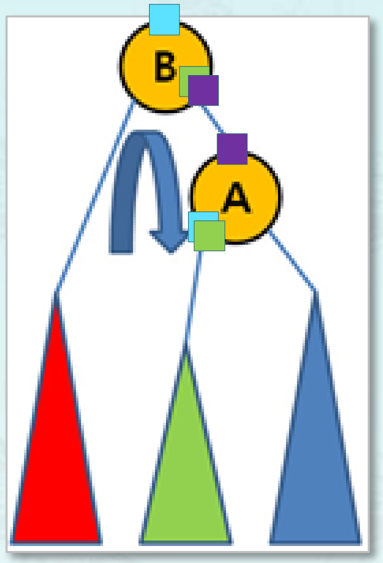


```
tree rotateLL(tree A)                  🟪
{
                ?                      🟦

                                       🟩

                                       🟪

    return      ?                      🟦
}
```

Single Rotation - LL case

outside case

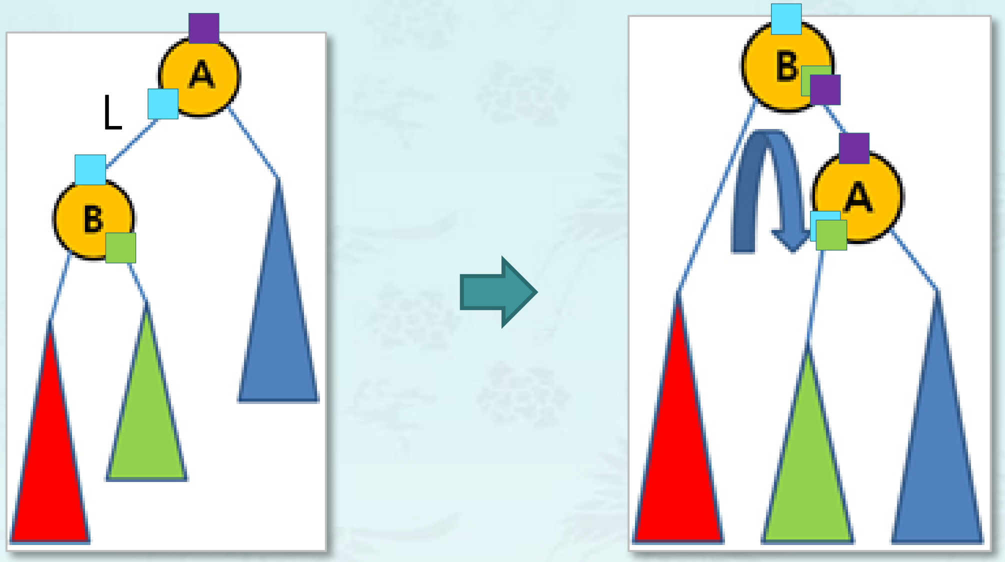




```
tree rotateLL(tree A)
{
   tree B   = A->left;


   return B;
}
```

Single Rotation - LL case

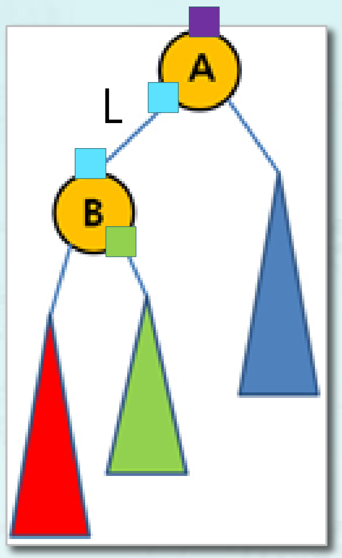
outside case

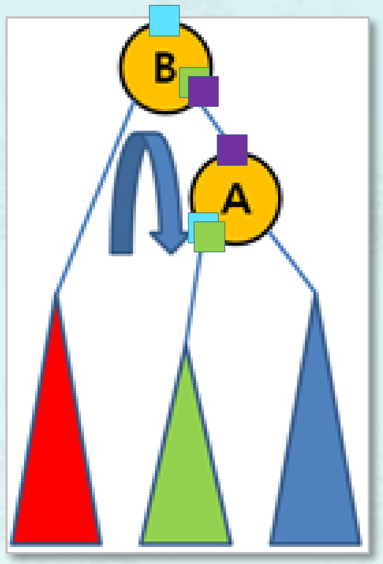


```
tree rotateLL(tree A)                🟪
{
  tree B    = A->left;               🟦
  A->left   = B->right;              🟩
                                     🟪
  return B;                          🟦
}
```

Single Rotation - LL case

outside case

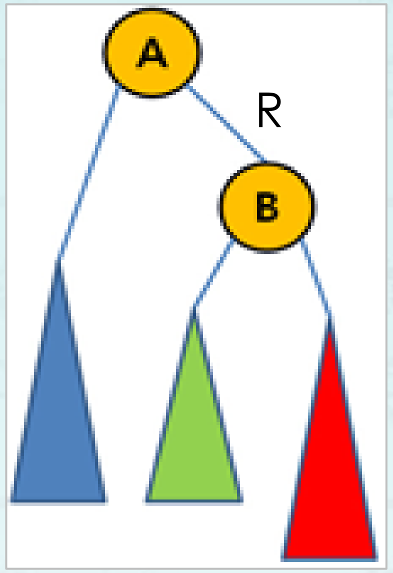


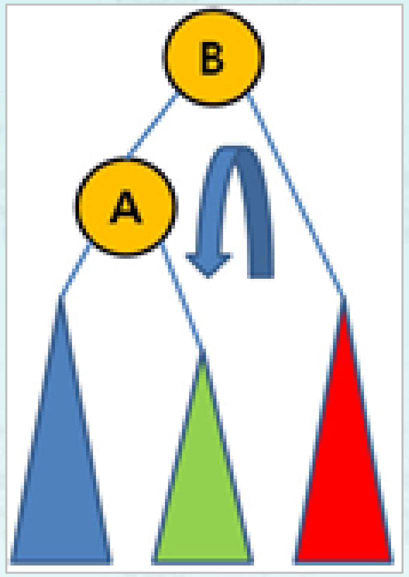


```
tree rotateLL(tree A)
{
  tree B    = A->left;
  A->left   = B->right;
  B->right = A;
  return B;
}
```

Single Rotation – RR case

outside case

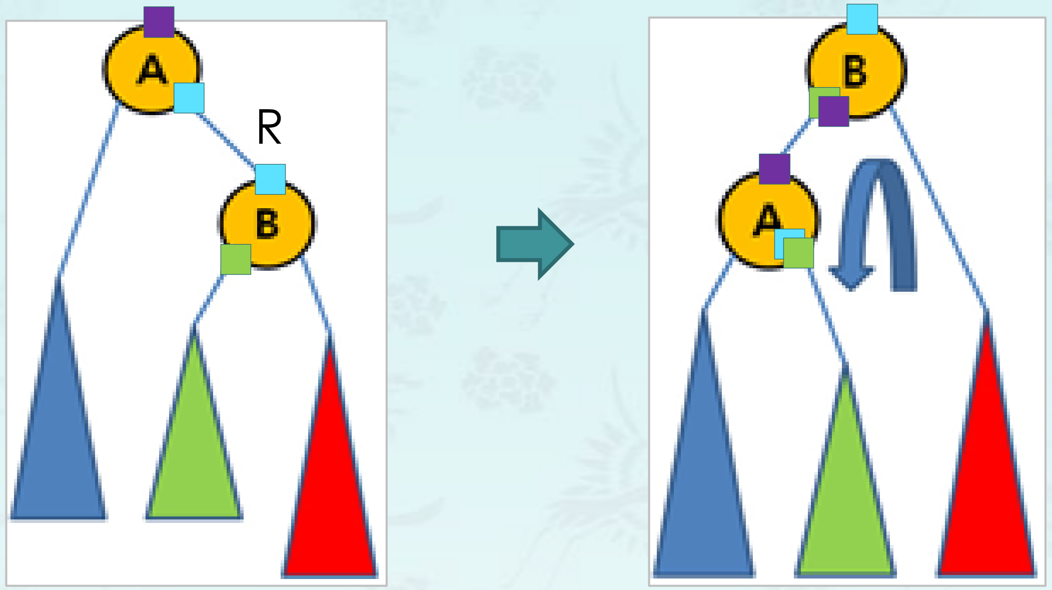


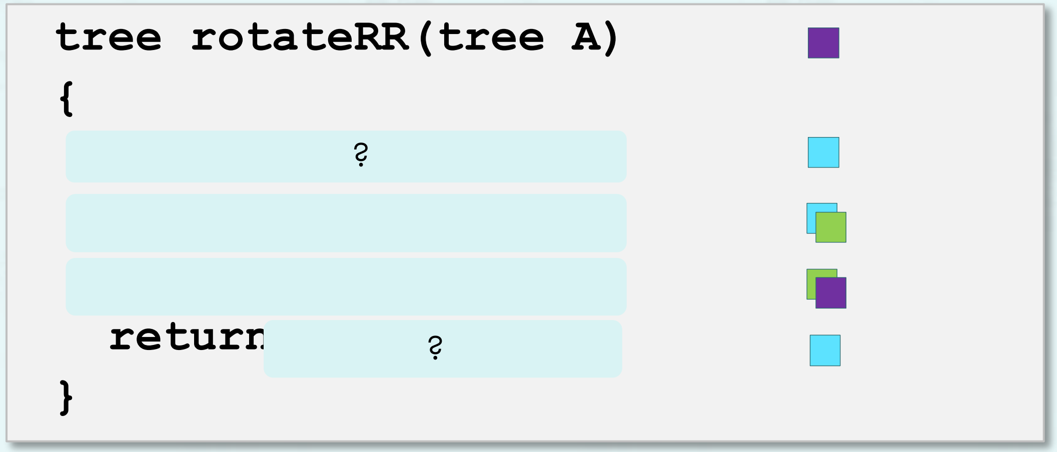


```
tree rotateRR(tree A)
{
                  ?
                                  ;

  return          ?
}
```

Single Rotation – RR case

outside case

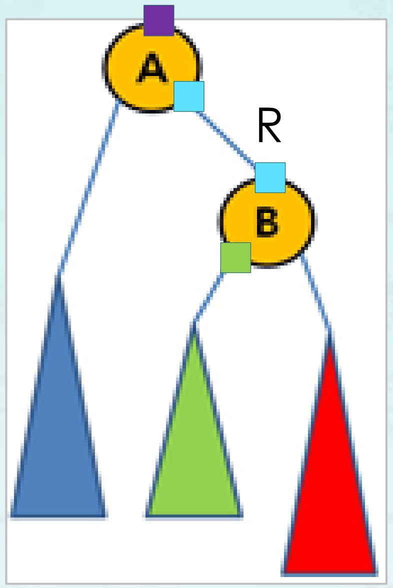


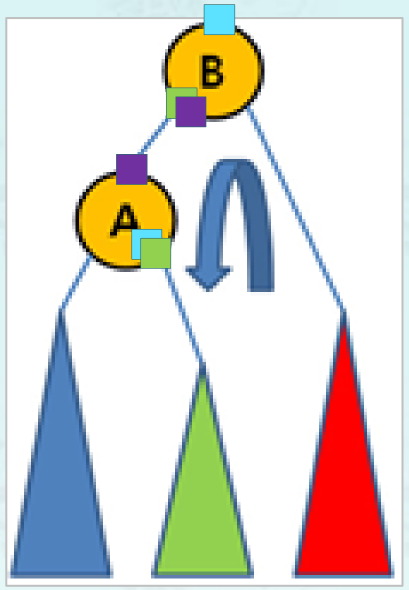


```
tree rotateRR(tree A)
{
                     ?


    return          ?
}
```

Single Rotation – RR case

outside case

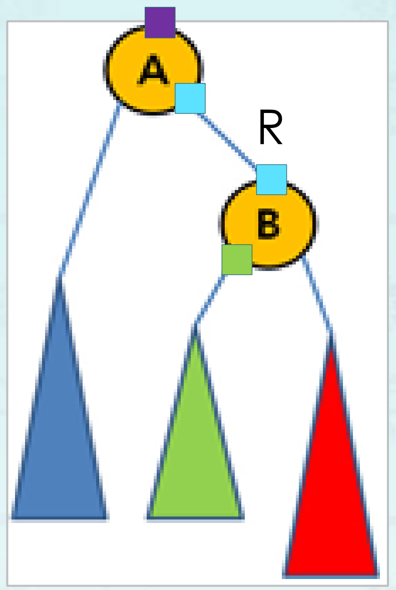


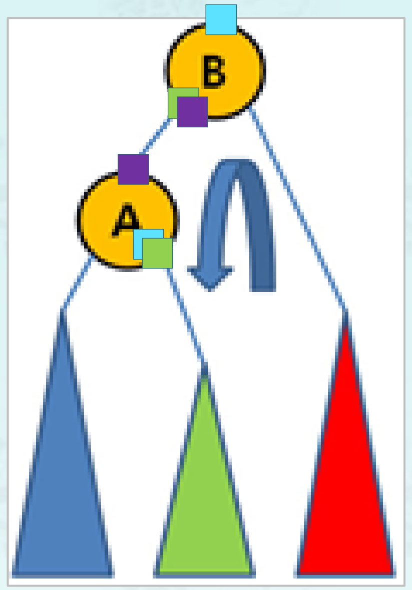


```
tree rotateRR(tree A)
{
  tree B   = A->right;


  return B;
}
```

Single Rotation – RR case

outside case

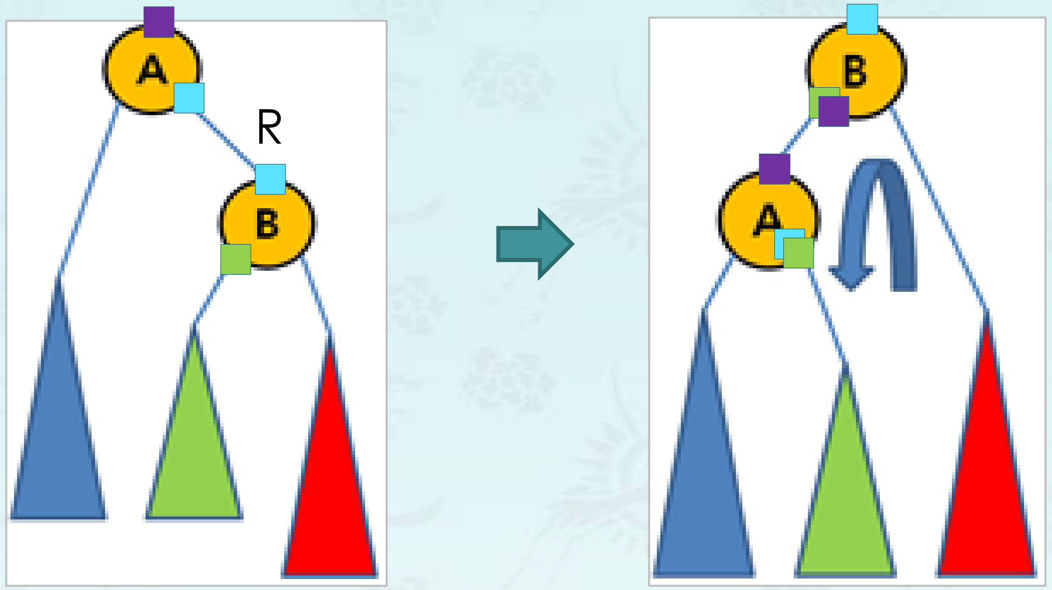




```
tree rotateRR(tree A)                         🟪
{
  tree B    = A->right;                       🟦
  A->right = B->left;                         🟩
                                              🟪
  return B;                                   🟦
}
```

Single Rotation – RR case

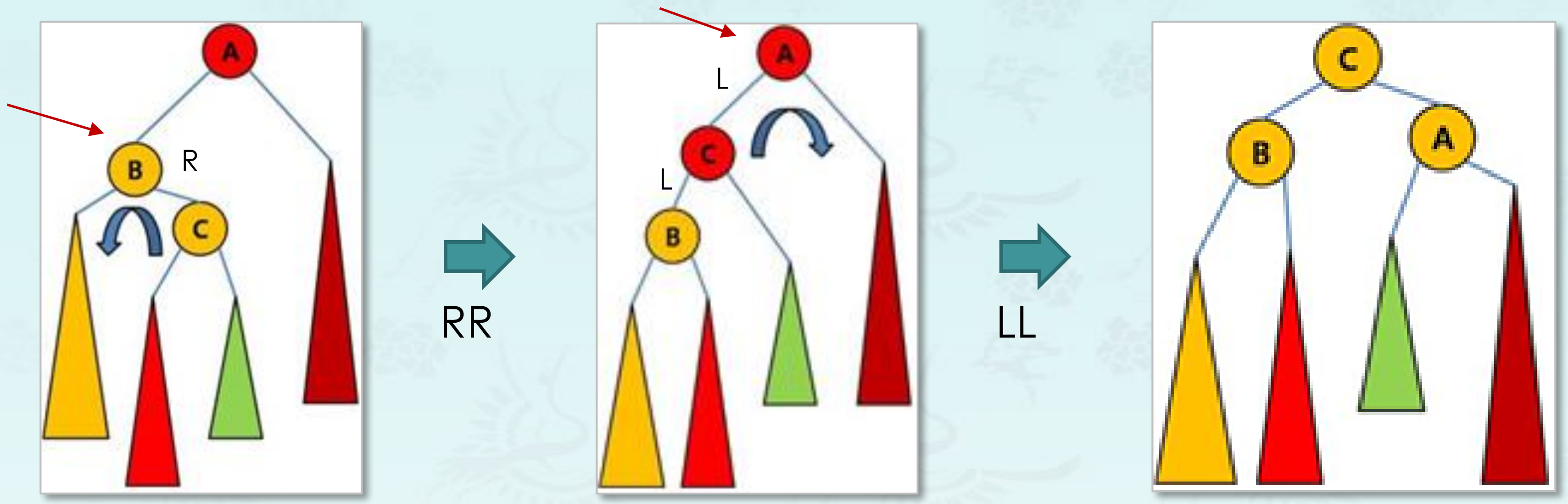
outside case



```
tree rotateRR(tree A)           🟪
{
  tree B    = A->right;         🟦
  A->right = B->left;           🟩
  B->left  = A;                 🟪
  return B;                     🟦
}
```

Double Rotation - LR case

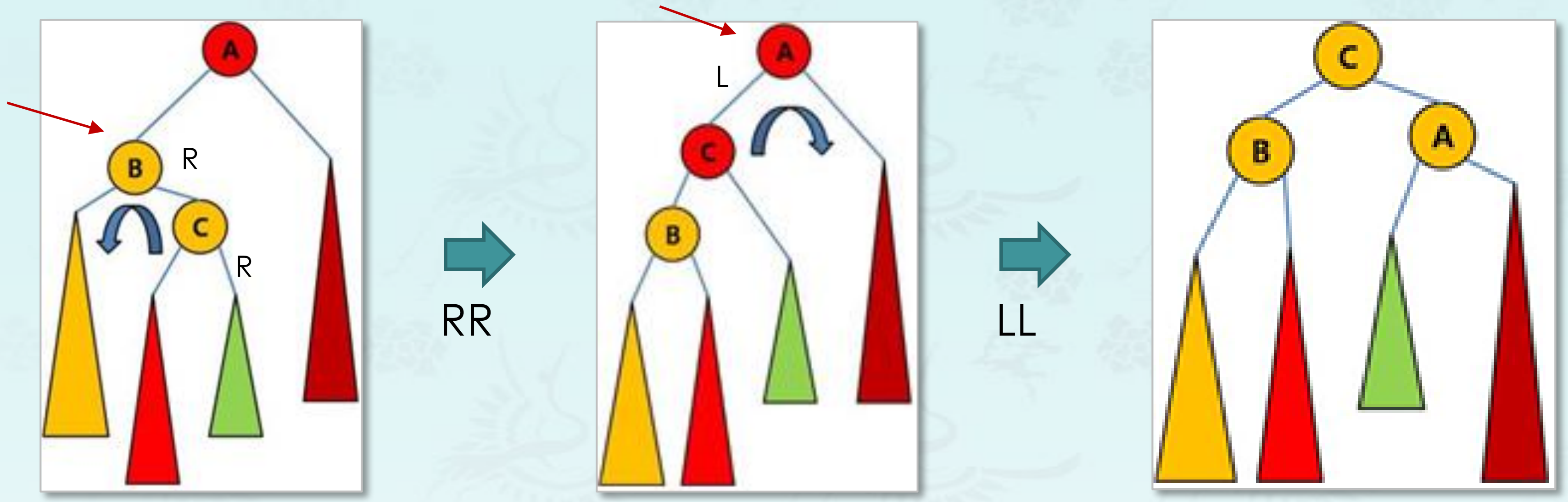
inside case



```
tree rotateLR(tree A) // RR and LL
{


}
```

Double Rotation - LR case

inside case

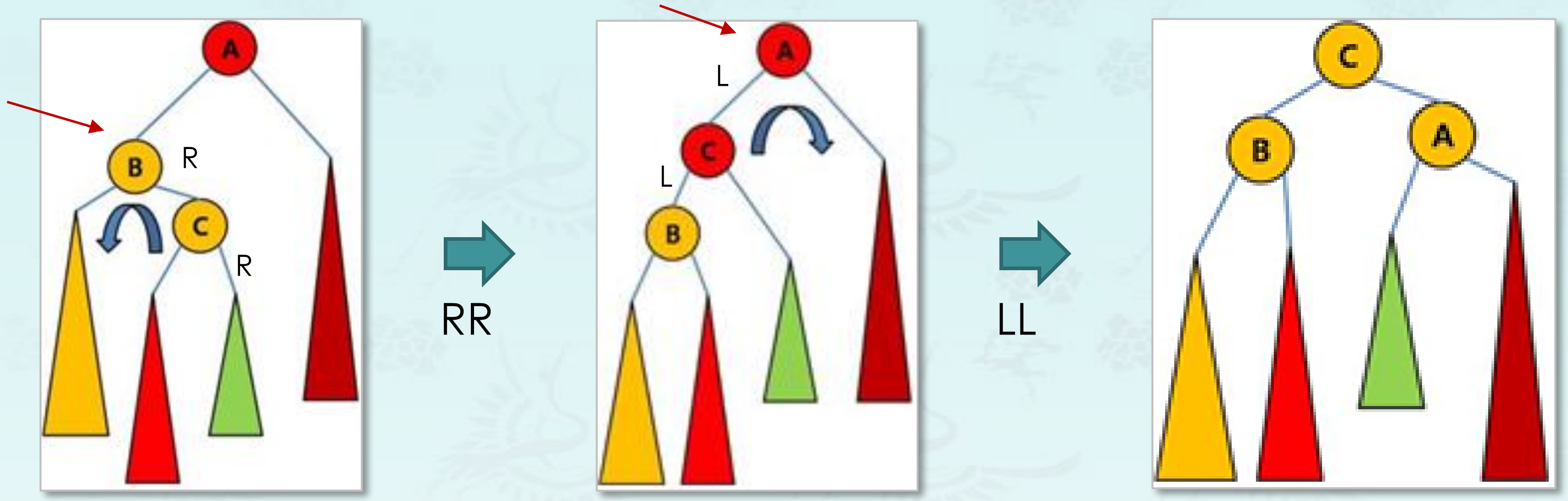


```
tree rotateLR(tree A) // RR and LL
{
  tree B  = A->left;


}
```

What will return eventually?

Double Rotation - LR case

inside case

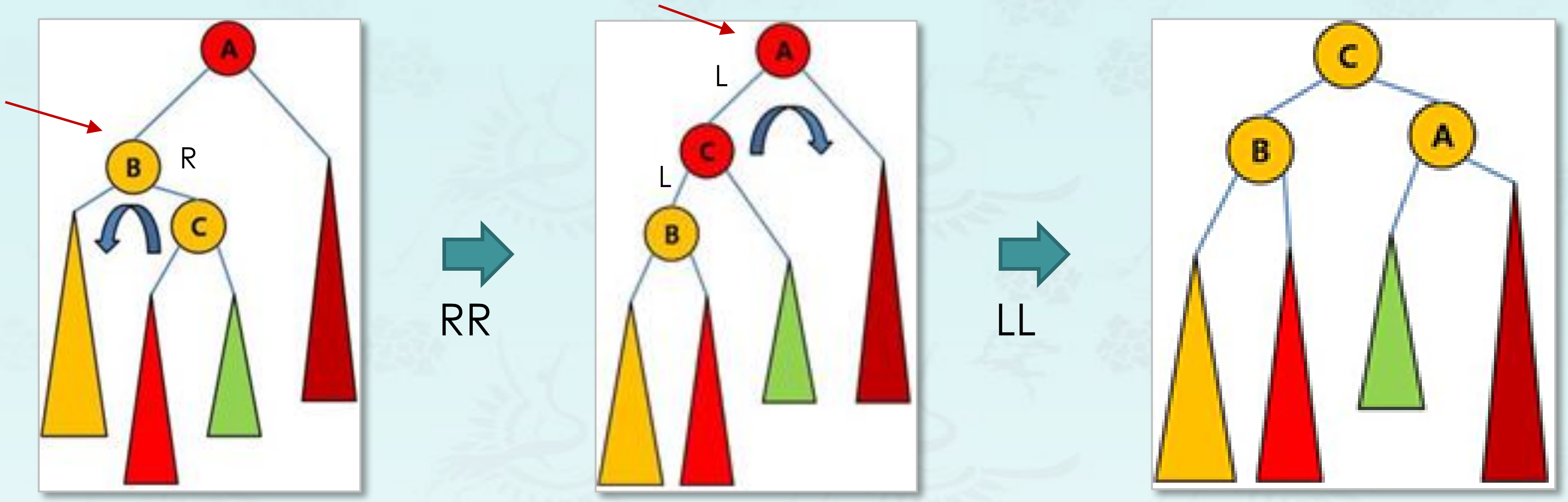


```
tree rotateLR(tree A) // RR and LL
{
   tree B  = A->left;
   A->left = rotateRR(B);

}
```

What will return eventually?

Double Rotation - LR case

inside case

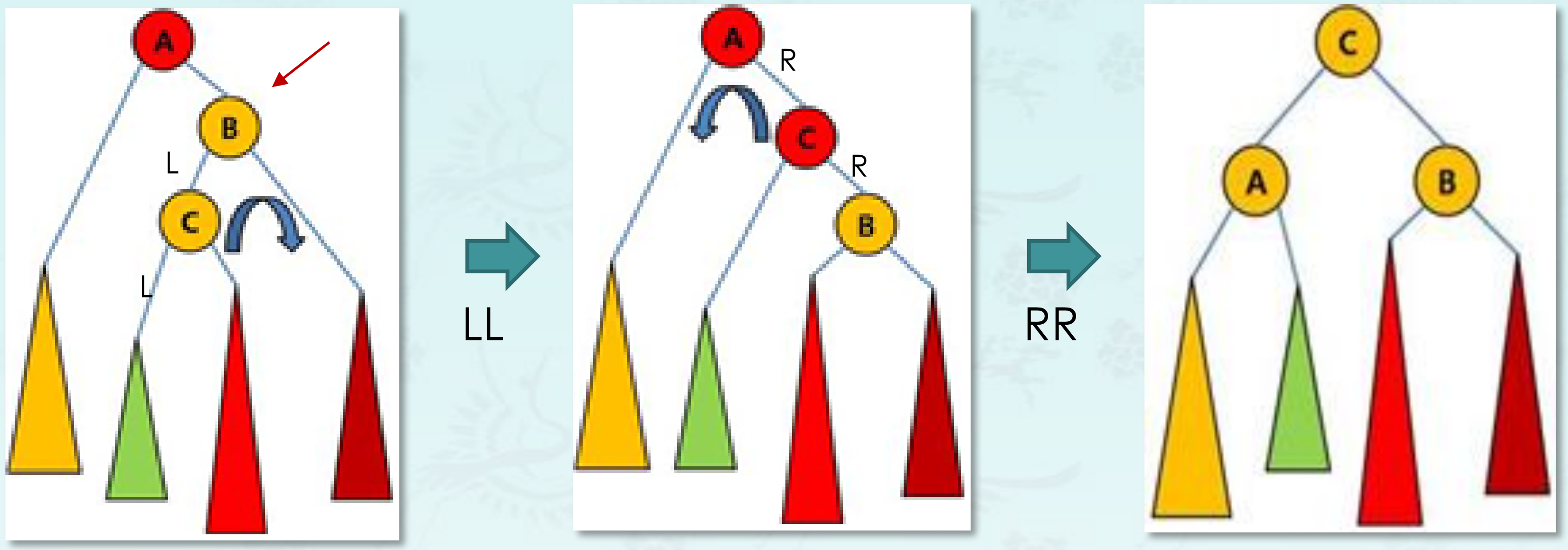


```
tree rotateLR(tree A) // RR and LL
{
  tree B  = A->left;
  A->left = rotateRR(B);
  return rotateLL(A);
}
```

What will return eventually?

Double Rotation - RL case

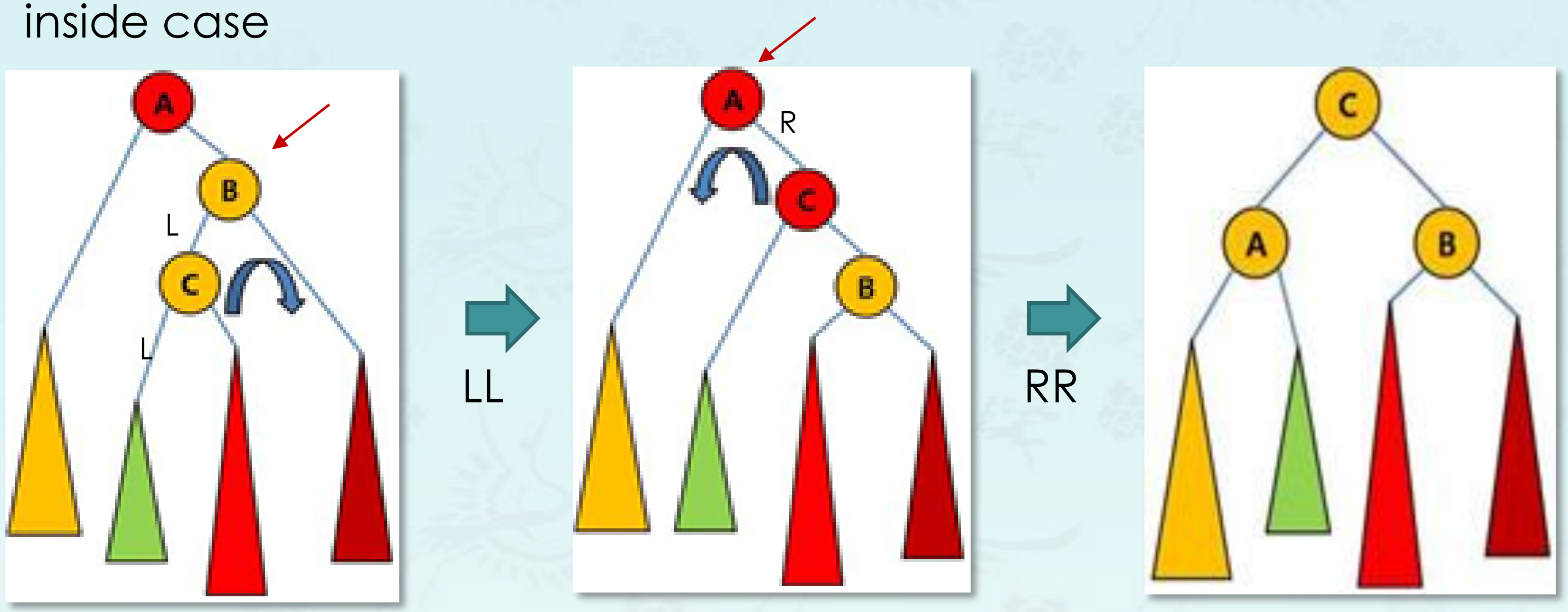
inside case



```
tree rotateRL(tree A) { // LL and RR
{

}
```

Double Rotation - RL case

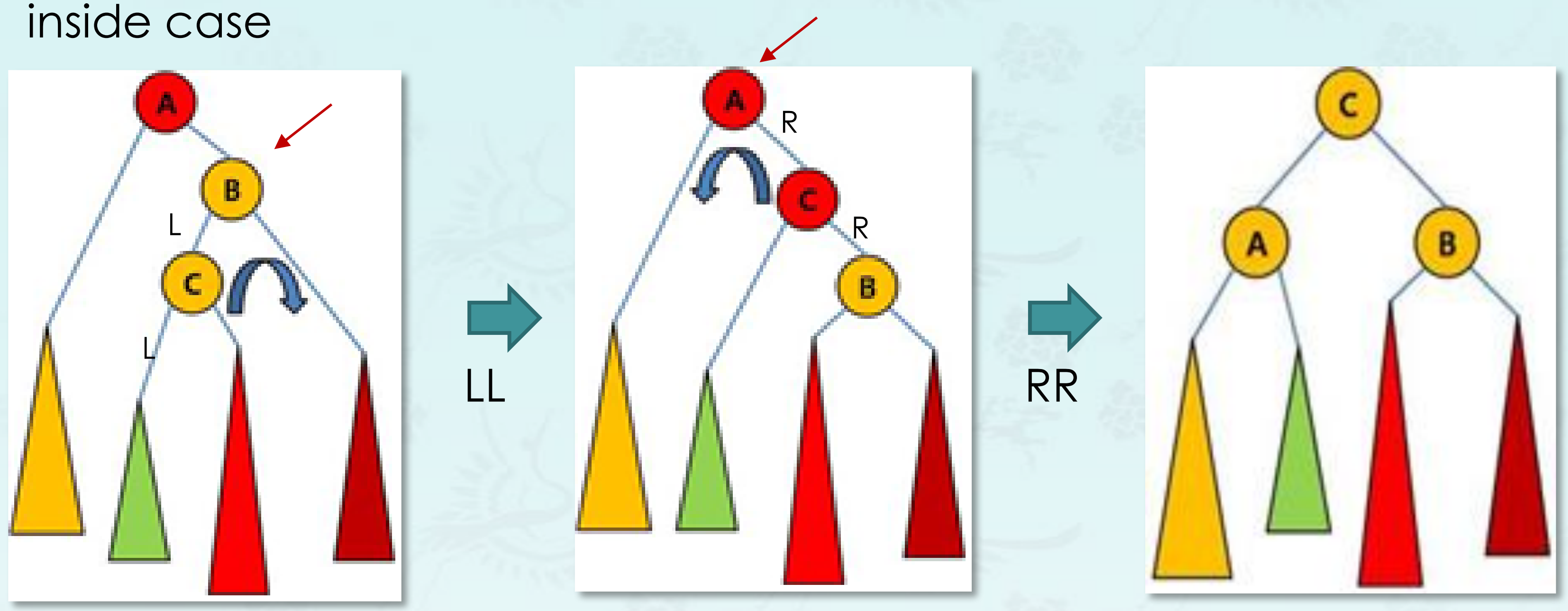
inside case



```
tree rotateRL(tree A) { // LL and RR
{
   tree B  = A->right;

}
```

Double Rotation - RL case

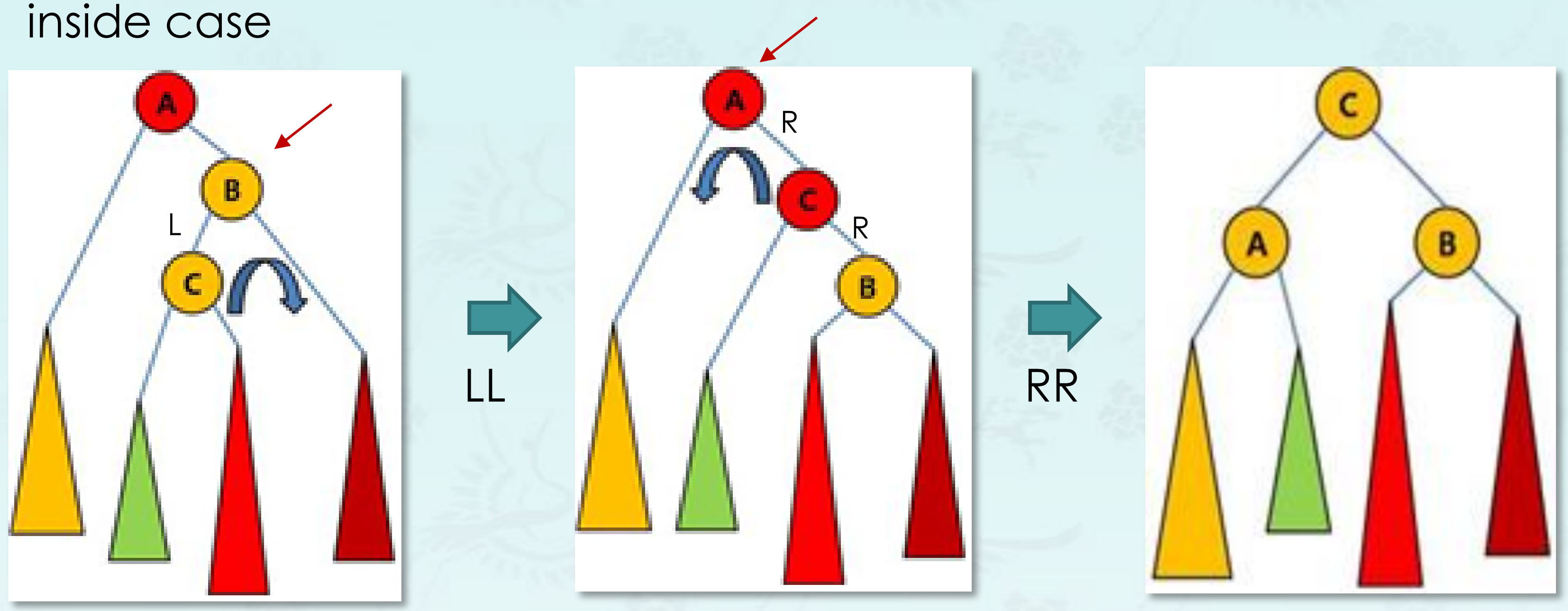
inside case



```
tree rotateRL(tree A) { // LL and RR
{
   tree B  = A->right;
   A->right = rotateLL(B);

}
```

Double Rotation - RL case

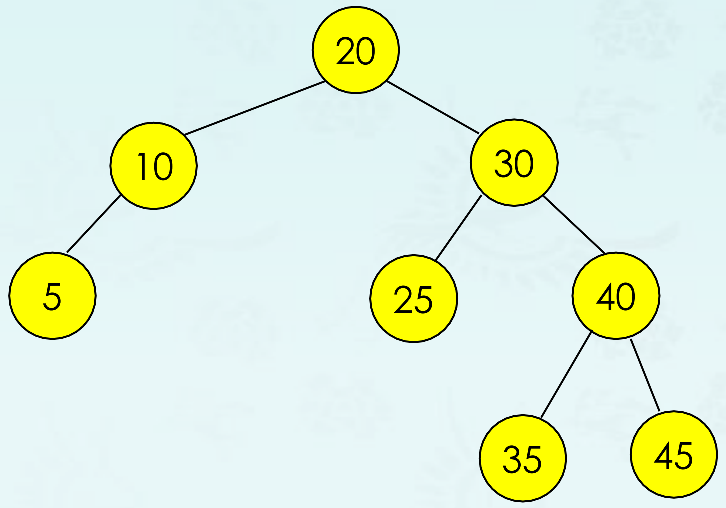
inside case



```
tree rotateRL(tree A) { // LL and RR
{
  tree B  = A->right;
  A->right = rotateLL(B);
  return rotateRR(A);
}
```

Double Rotation -??case

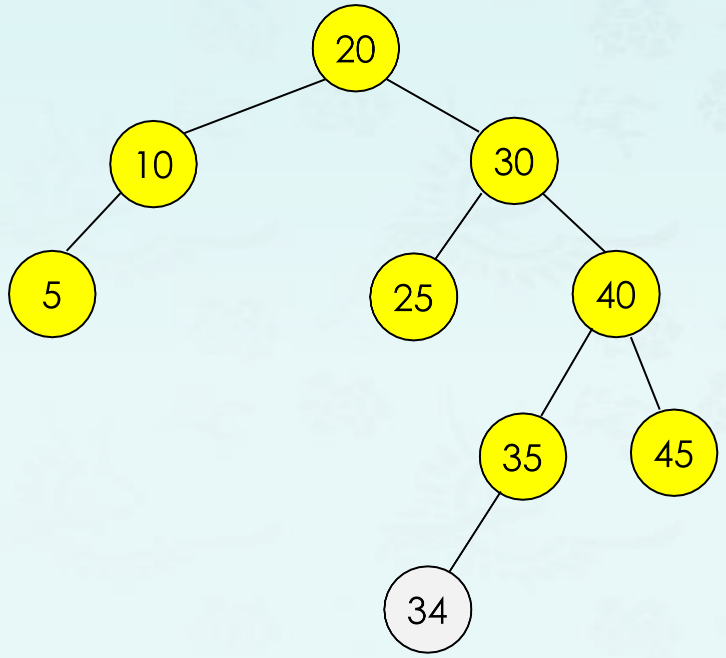
- **Insertion of 34**
- Imbalance at ?
- Balance factor ??
- Rotation ___??___ case



AVL balanced tree

Double Rotation -??case

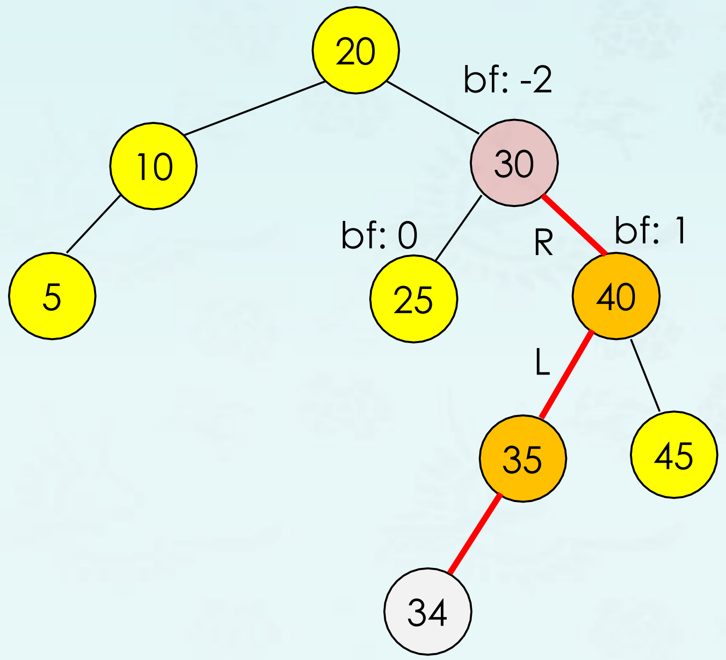
- Insertion of 34
- Imbalance at ?
- Balance factor ??
- Rotation ___??___ case



After insertion, AVL imbalanced tree

Double Rotation -??case

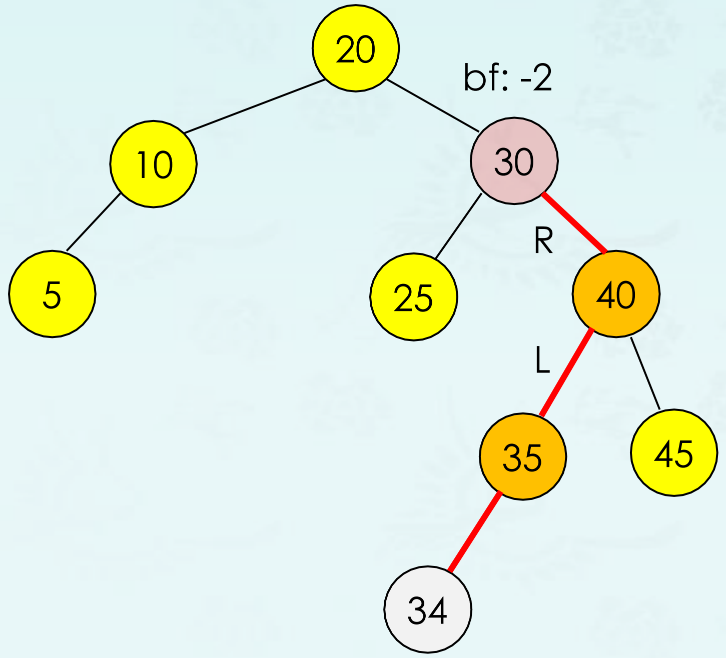
- Insertion of 34
- Imbalance at 30
- Balance factor **-2**
- Rotation ___??___ case



Double Rotation - RL case

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

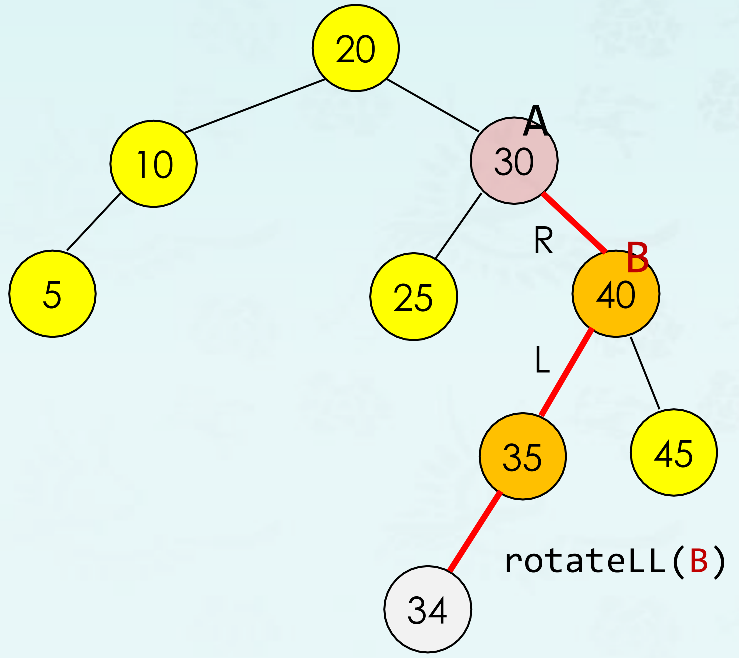
```
tree rotateRL(tree A) {
  tree B  = A->right;
  A->right = rotateLL(B);
  return rotateRR(A);
}
```



Double Rotation - RL case

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

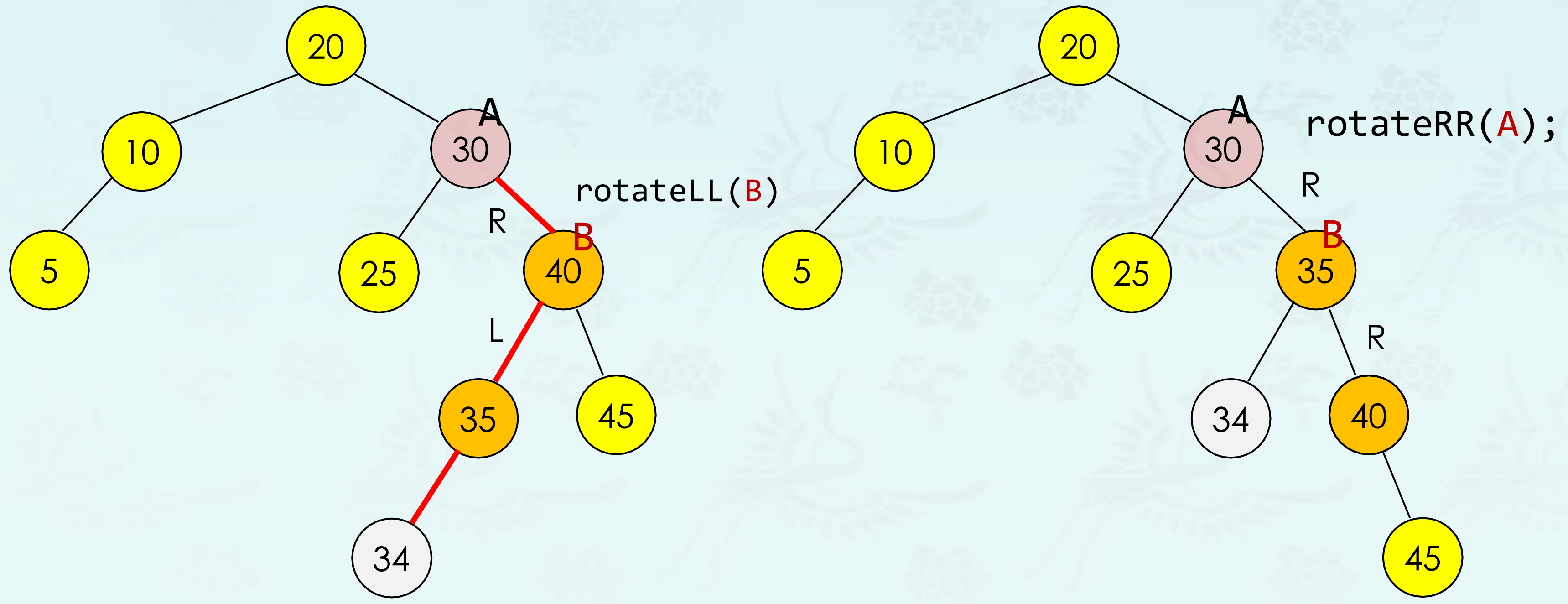
```
tree rotateRL(tree A) {
  tree B  = A->right;
  A->right = rotateLL(B);
  return rotateRR(A);
}
```



Double Rotation - RL case

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

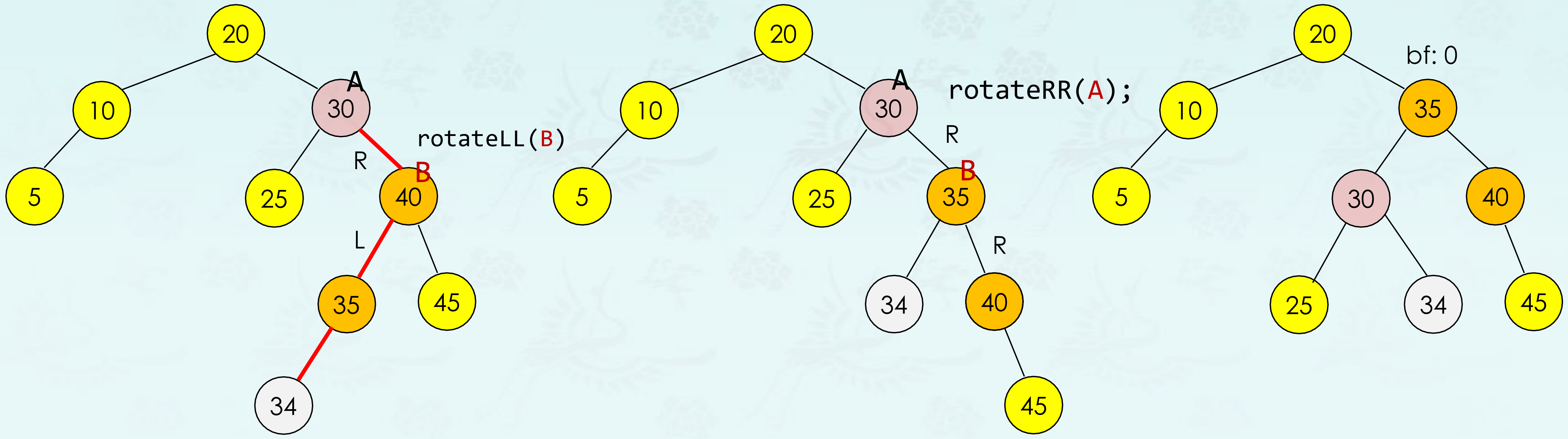
```
tree rotateRL(tree A) {
  tree B  = A->right;
  A->right = rotateLL(B);
  return rotateRR(A);
}
```



Double Rotation - RL case

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

```
tree rotateRL(tree A) {
  tree B  = A->right;
  A->right = rotateLL(B);
  return rotateRR(A);
}
```



After insertion, AVL imbalanced tree

After insertion, AVL balanced tree

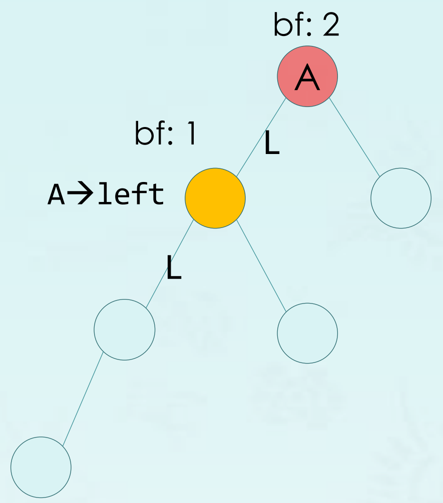
Balance Factor and Height

```
int height(tree node) {
  if (empty(node)) return -1;
  int left = height(node->left);
  int right = height(node->right);
  return max(left, right) + 1;
}
```

```
int balanceFactor(tree node) {
  if (node == NULL) return 0;
  int left  = height(node->left);
  int right = height(node->right);
  return left – right;
}
```

Rebalance

outside case

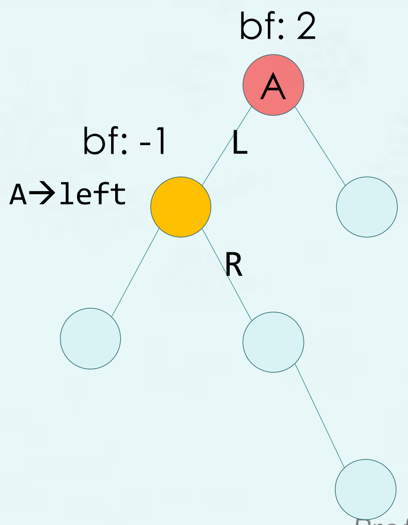


```
tree rebalance(tree A) {
  int bf = balanceFactor(A);
  if (bf == 2) {
    if (balanceFactor(A->left) == 1)


  }
```

checking single or double rotation

yet to fix to handle **A's child bf is 0**

inside 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.

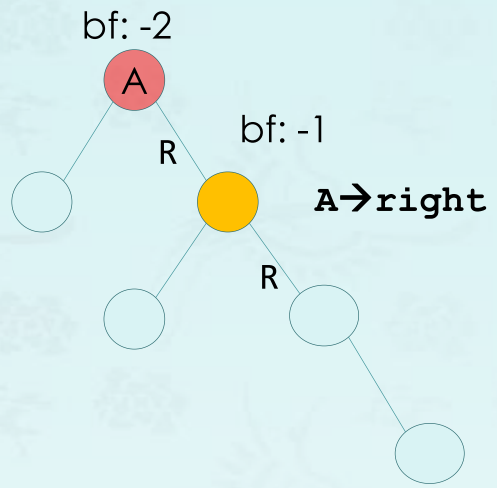
Rebalance

```
tree rebalance(tree A) {
  int bf = balanceFactor(A);
  if (bf == 2) {




  else if (bf == -2) {
    if (balanceFactor(A->right) == 1)



  }
  return A;
}
```

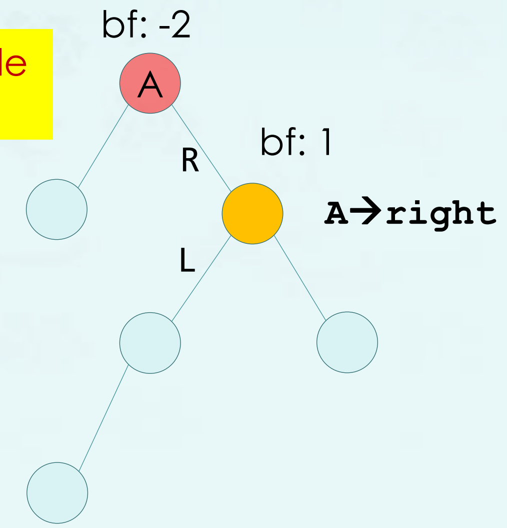
checking single or double rotation

yet to fix to handle **A's child bf is 0**

outside case



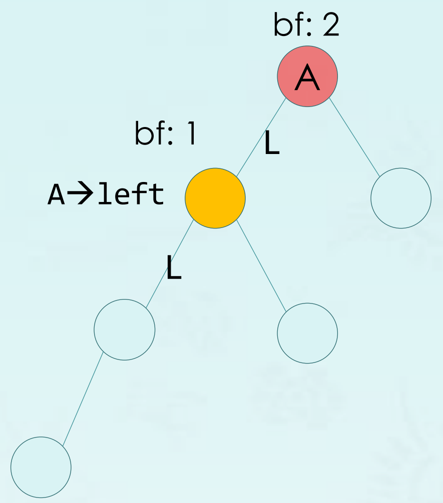
inside case



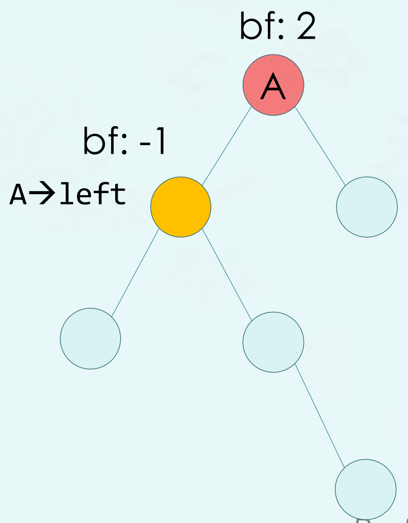
Observation: If A and its child have the same sign in bf's, a single rotation is needed, a double rotation otherwise.
If the bf of A's child is 0, treat it like the same sign of A.

Rebalance

outside case



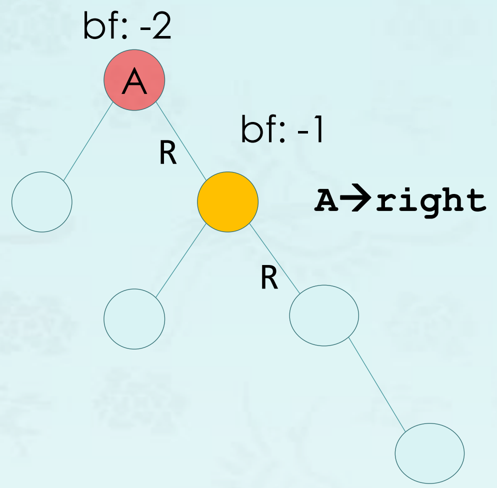
inside case



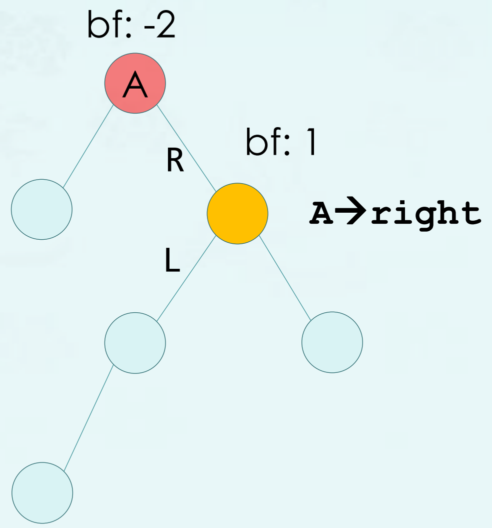
```
tree rebalance(tree A) {
  int bf = balanceFactor(A);
  if (bf == 2) {
    if (balanceFactor(A->left) == 1)


  }
  else if (bf == -2) {
    if (balanceFactor(A->right) == -1)


  }
  return A;  // no rebalanced needed
}
```

yet to fix to handle
A's child bf is 0

outside case



inside 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.

growAVL() & trimAVL()

```
// inserts a key into the AVL tree and rebalance it.
tree growAVL(tree node, int key) {
  if (node == nullptr) return new TreeNode(key);

  // your code here

  return rebalance(node);      // O(log n)
}
```

almost same as grow()

AVL rotation if necessary

```
// deletes a key into the AVL tree and rebalance it.
tree trimAVL(tree node, int key) {
  if (node == nullptr) return node;

  // your code here

  return rebalance(node);      // O(log n)
}
```

almost same as trim()

AVL rotation if necessary

Data Structures
Chapter 5 Tree

1. Introduction
2. Binary Tree
3. Binary Search Tree
4. **Balancing Tree**
   - AVL Tree Introduction
   - **AVL Operations**
   - AVL Coding