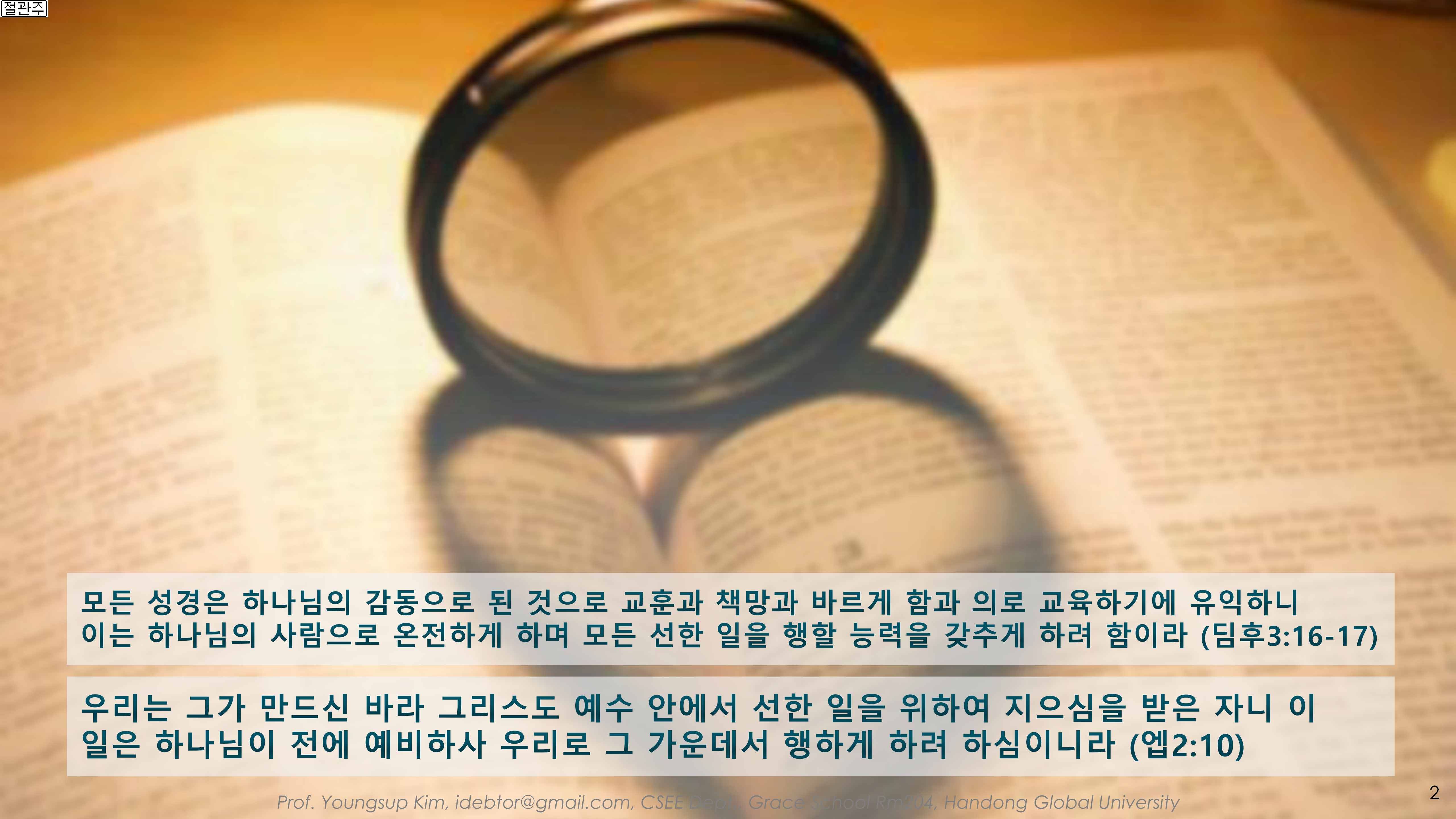
Data Structures
Chapter 5 Tree

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

절관주

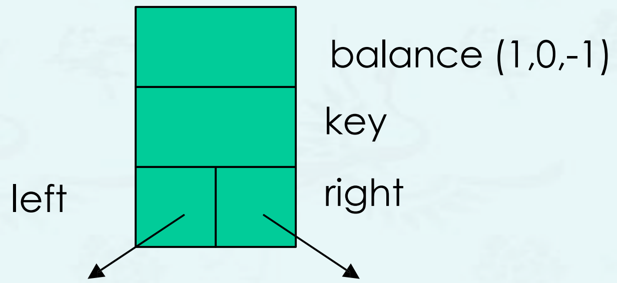


모든 성경은 하나님의 감동으로 된 것으로 교훈과 책망과 바르게 함과 의로 교육하기에 유익하니 이는 하나님의 사람으로 온전하게 하며 모든 선한 일을 행할 능력을 갖추게 하려 함이라 (딤후3:16-17)

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)

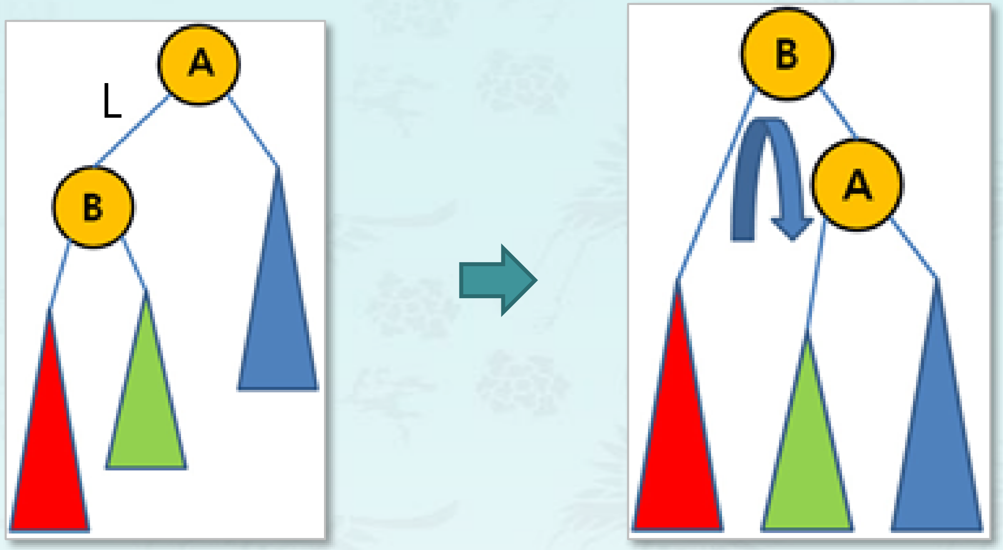
Coding

- 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

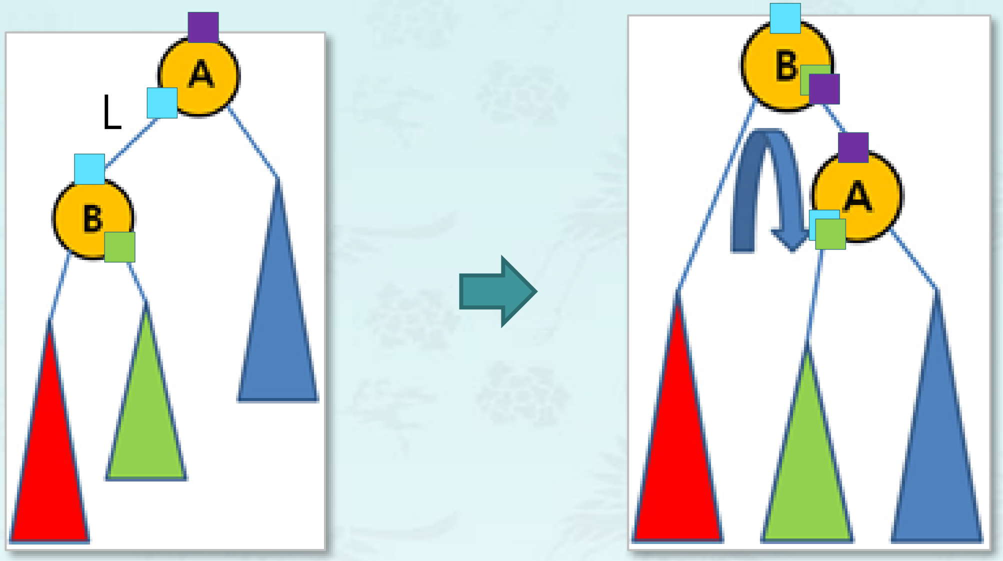
outside case



```
tree rotateLL(tree A)
{



   return
}
```

Single Rotation - LL case

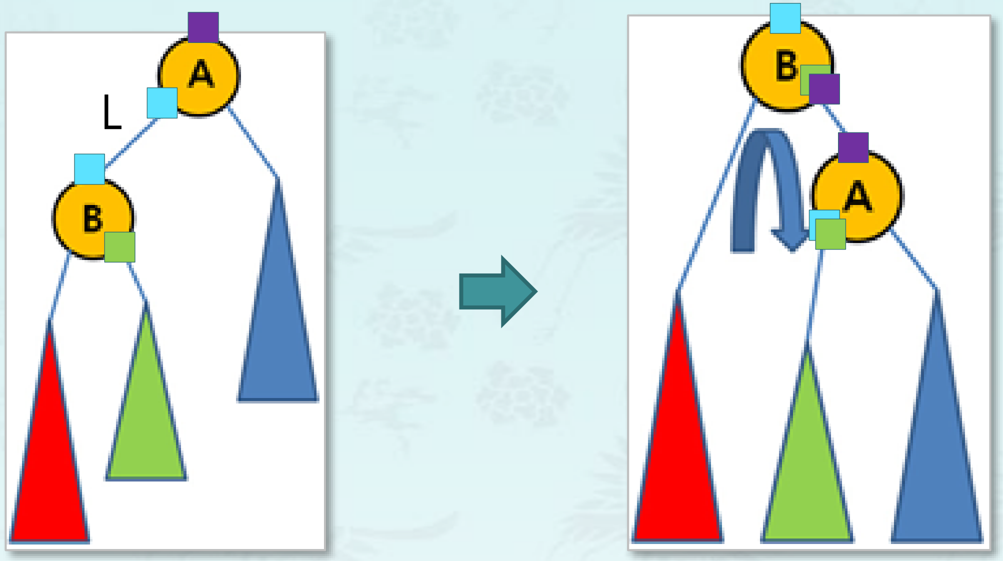
outside case



```
tree rotateLL(tree A)
{
        ?


   return    ?
}
```

Single Rotation - LL case

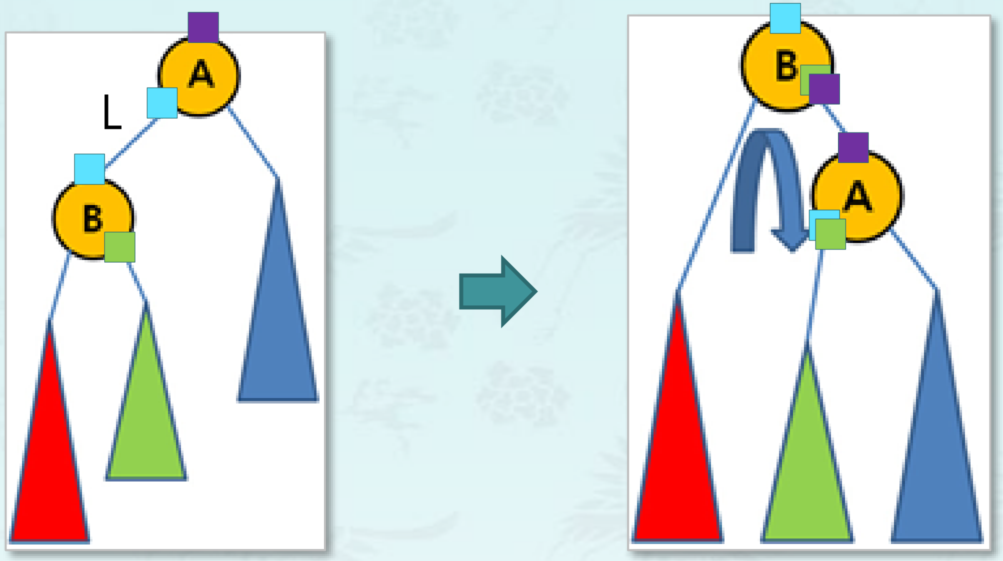
outside case



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


  return B;
}
```

Single Rotation - LL case

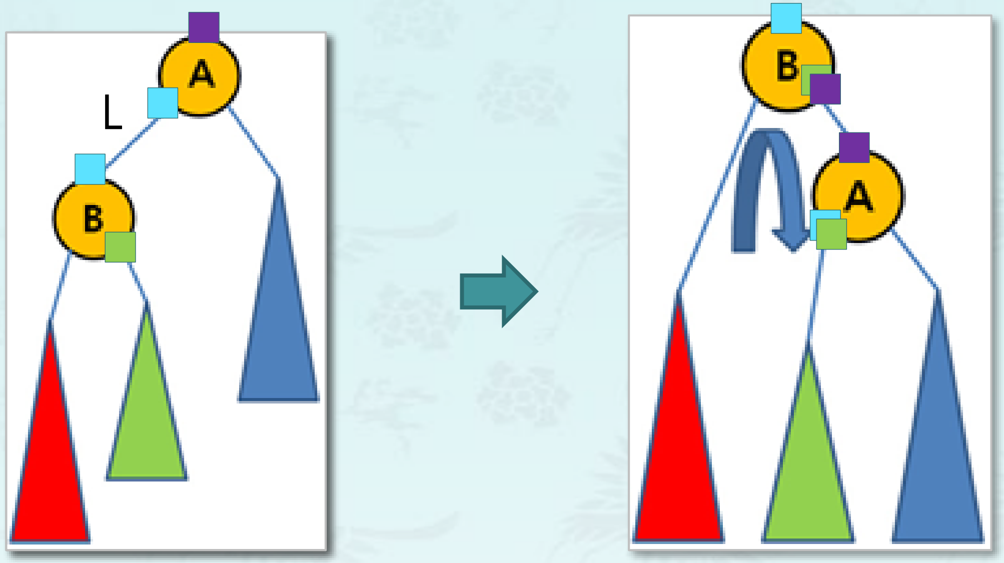
outside case



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

  return B;
}
```

Single Rotation - LL case

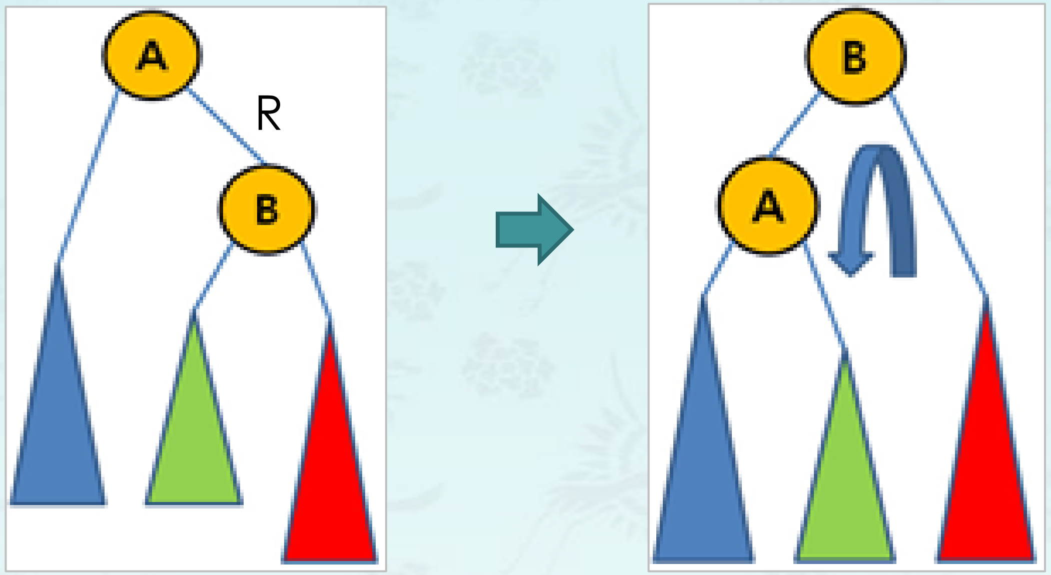
outside case



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

Single Rotation – RR case

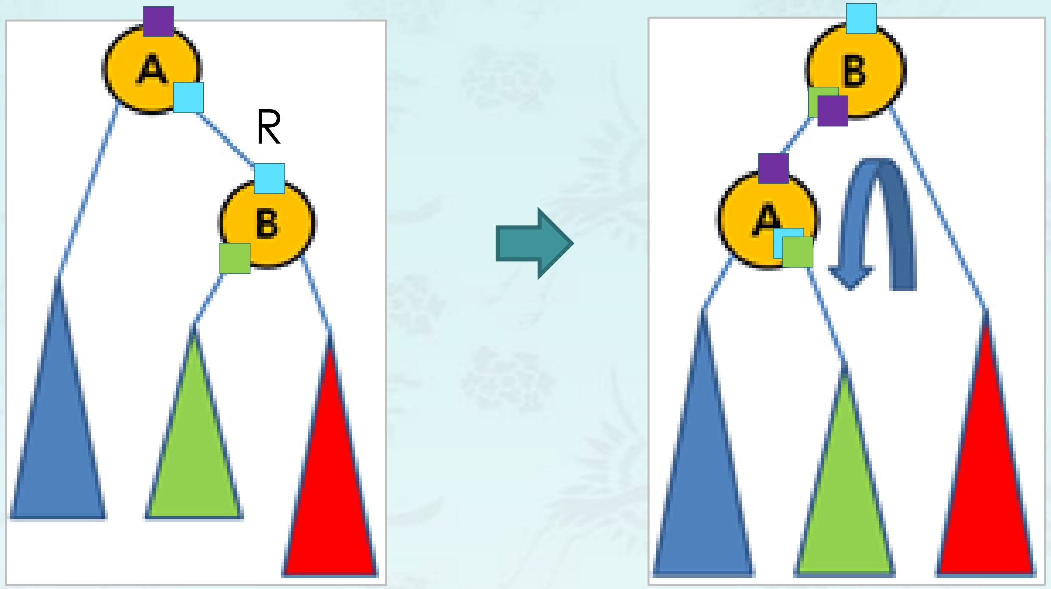
outside case



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

  return      ?
}
```

Single Rotation – RR case

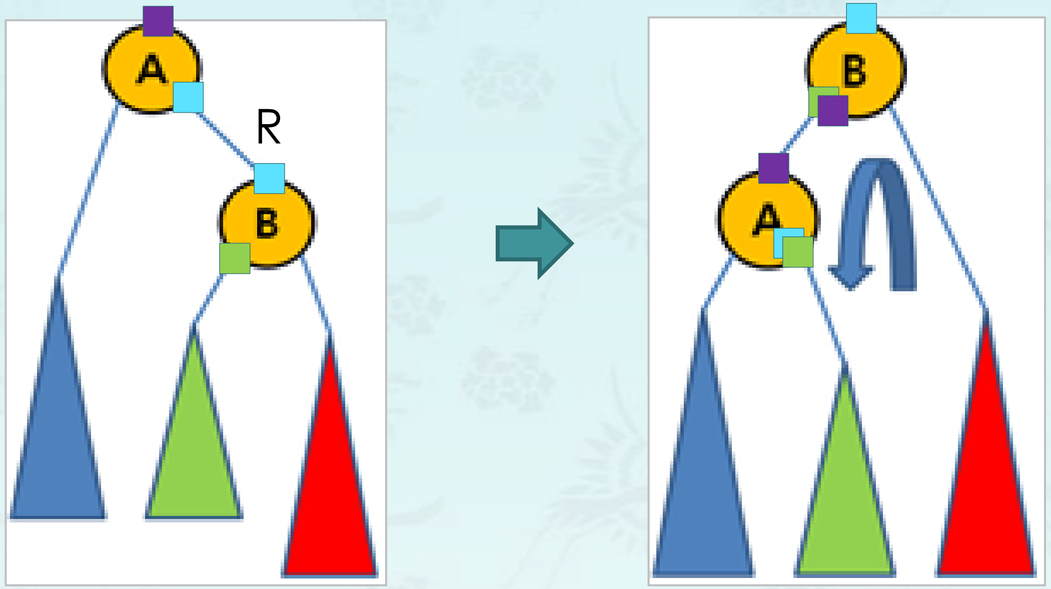
outside case



```
tree rotateRR(tree A)              🟪
{
                ?                  🟦
                                   🟦🟩
                                   🟩🟪
   return             ?            🟦
}
```

Single Rotation – RR case

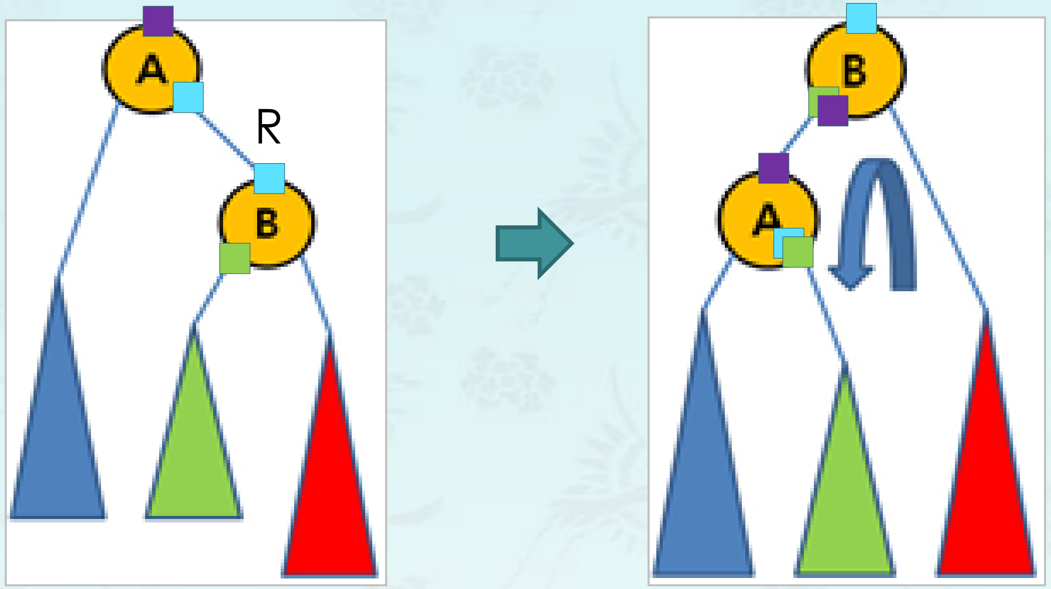
outside case



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


  return B;
}
```

Single Rotation – RR case

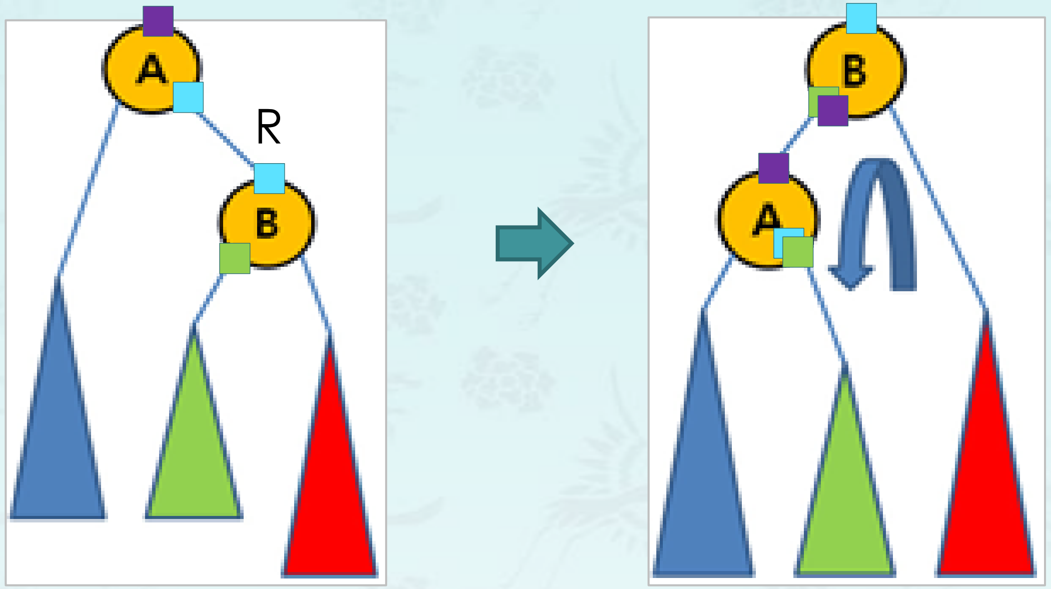
outside case



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

  return B;
}
```

Single Rotation – RR case

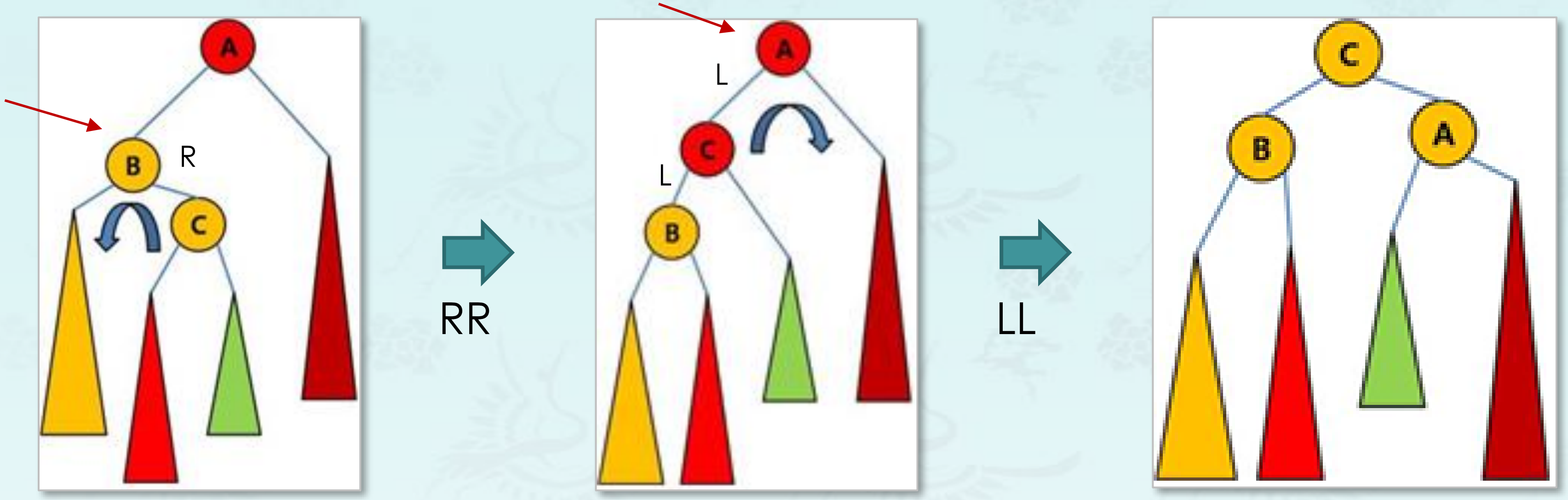
outside case



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

Double Rotation - LR case

inside case



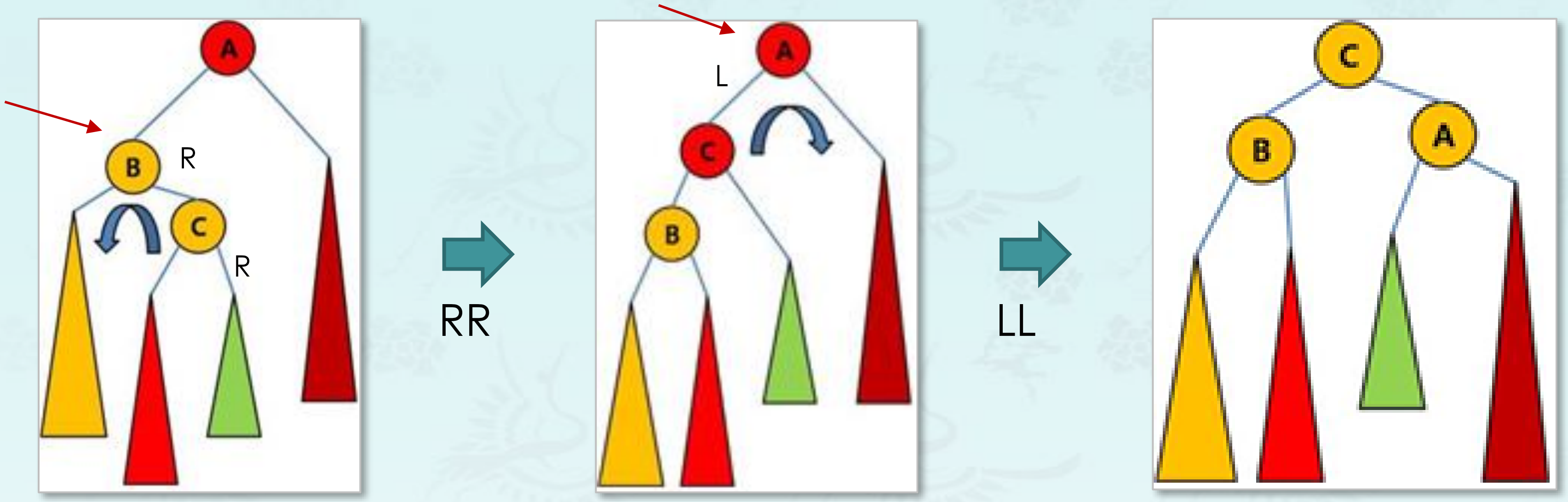
RR

LL

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


}
```

Double Rotation - LR case

inside case

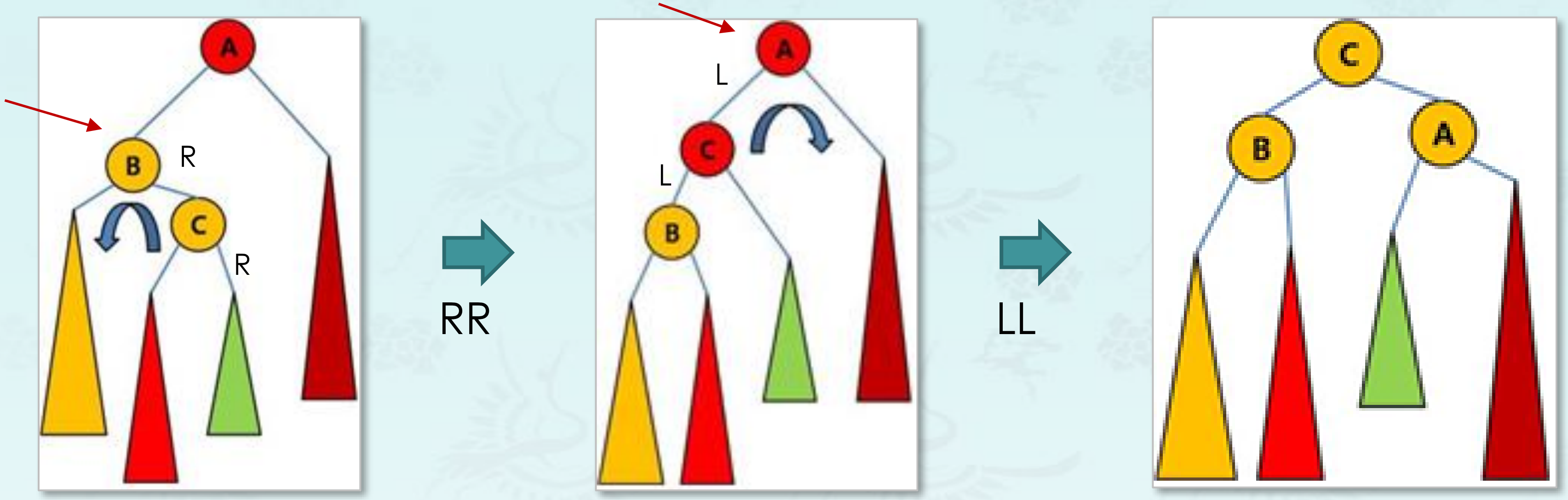


```
tree rotateLR(tree A) // RR and LL
{
  tree B  = A->left;


}
```

What will return eventually?

Double Rotation - LR case

inside case

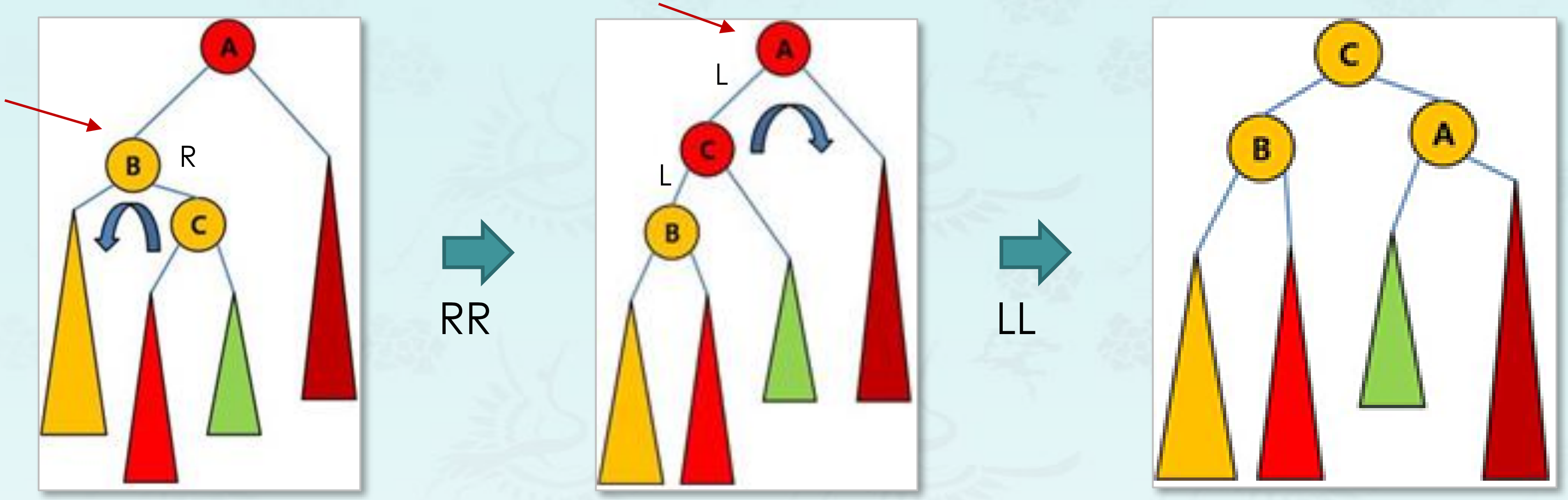


```
tree rotateLR(tree A) // RR and LL
{
  tree B  = A->left;
  A->left = rotateRR(B);

}
```

What will return eventually?

Double Rotation - LR case

inside case

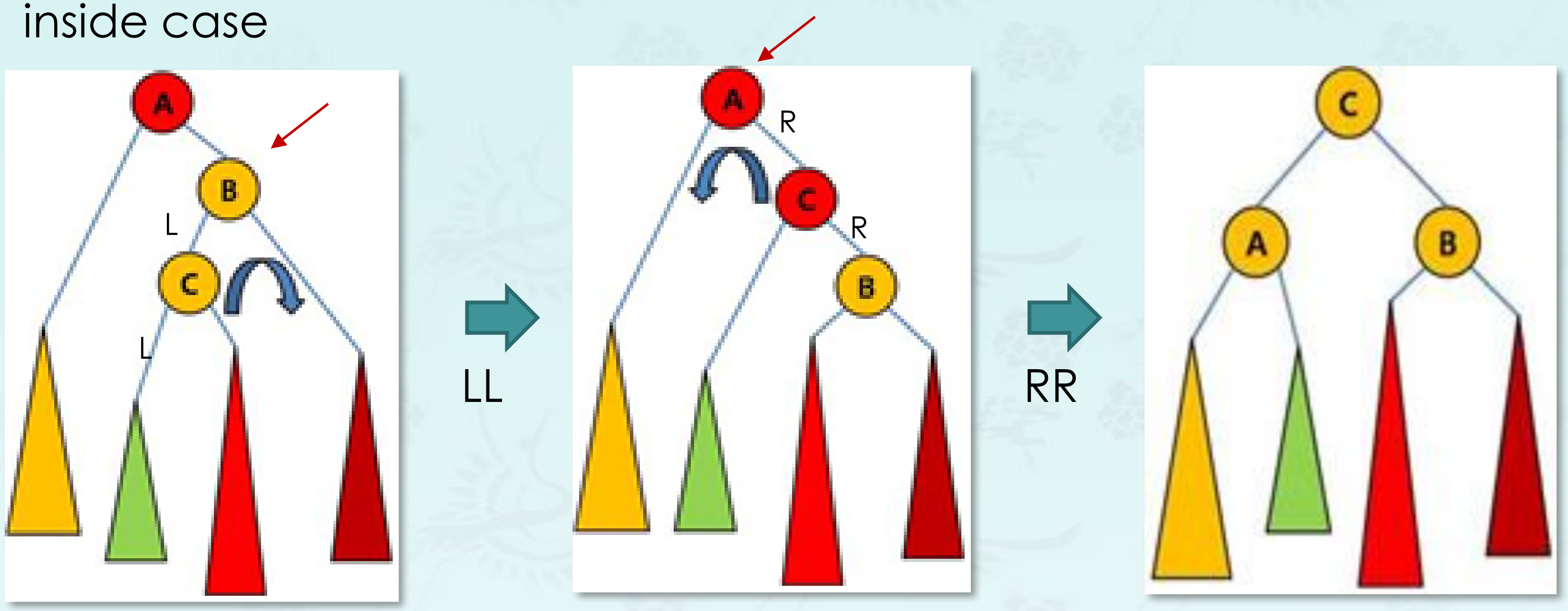


```
tree rotateLR(tree A) // RR and LL
{
  tree B  = A->left;
  A->left = rotateRR(B);
  return rotateLL(A);
}
```

What will return eventually?

Double Rotation - RL case

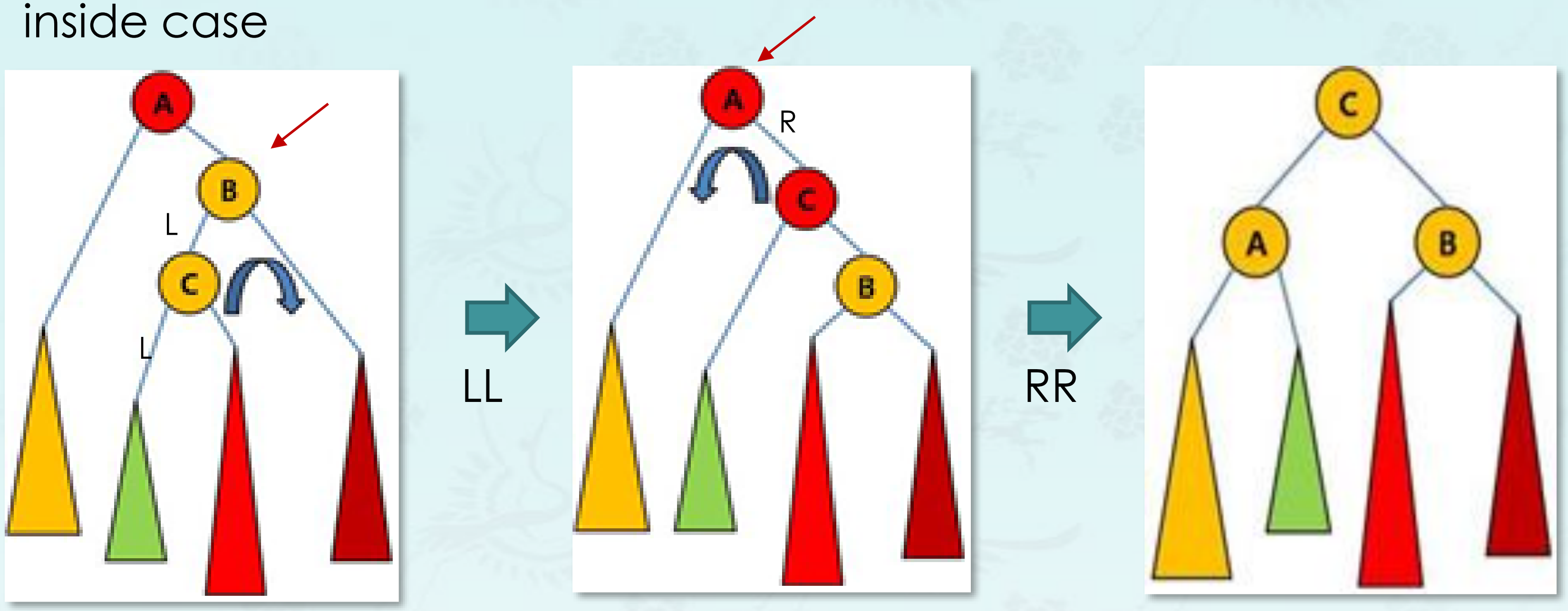
inside case



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

}
```

Double Rotation - RL case

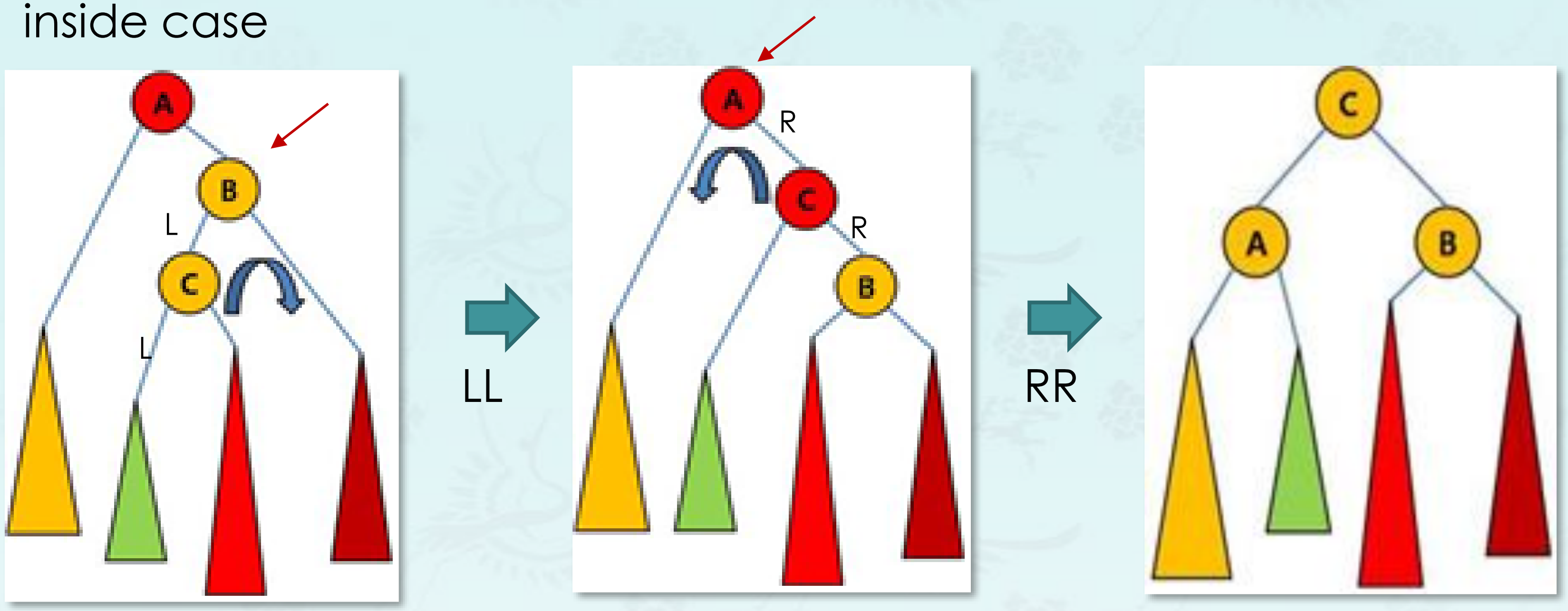
inside case



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

}
```

Double Rotation - RL case

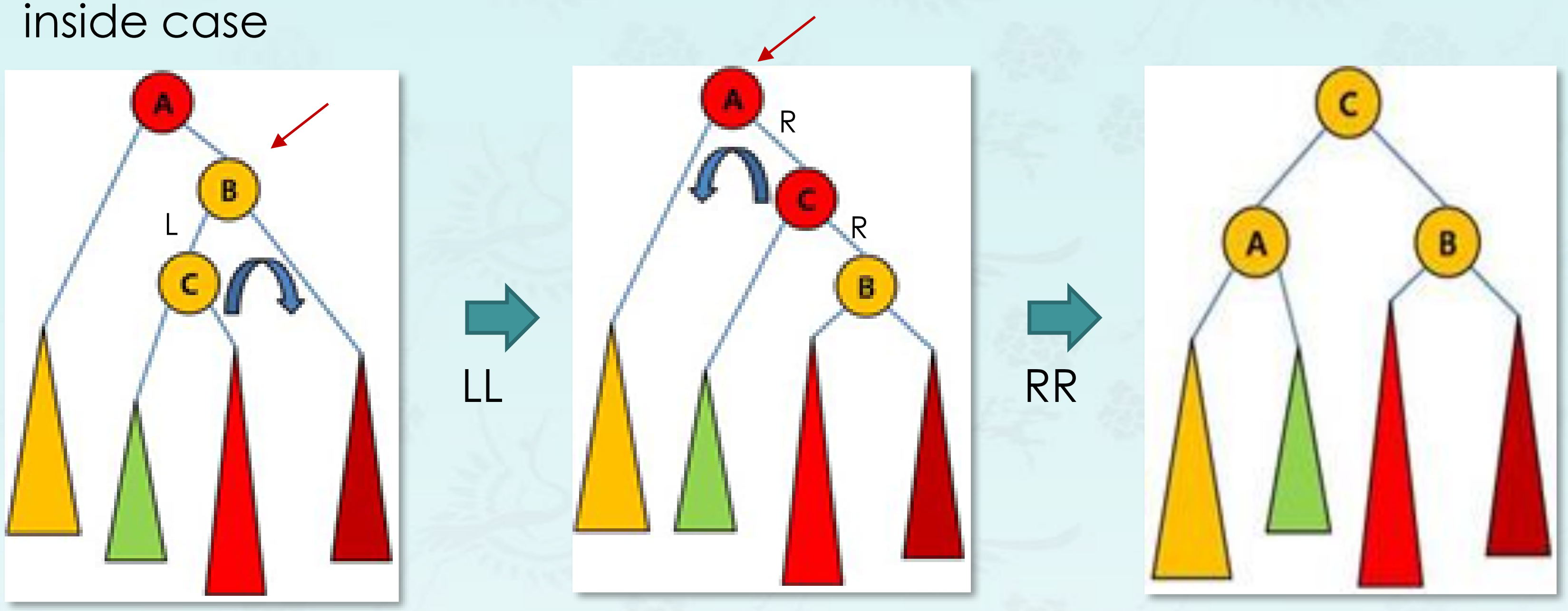
inside case



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

}
```

Double Rotation - RL case

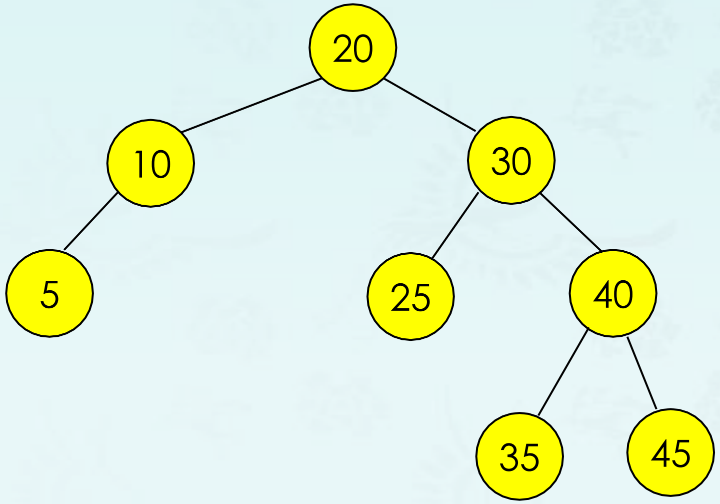
inside case



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

Double Rotation -??case

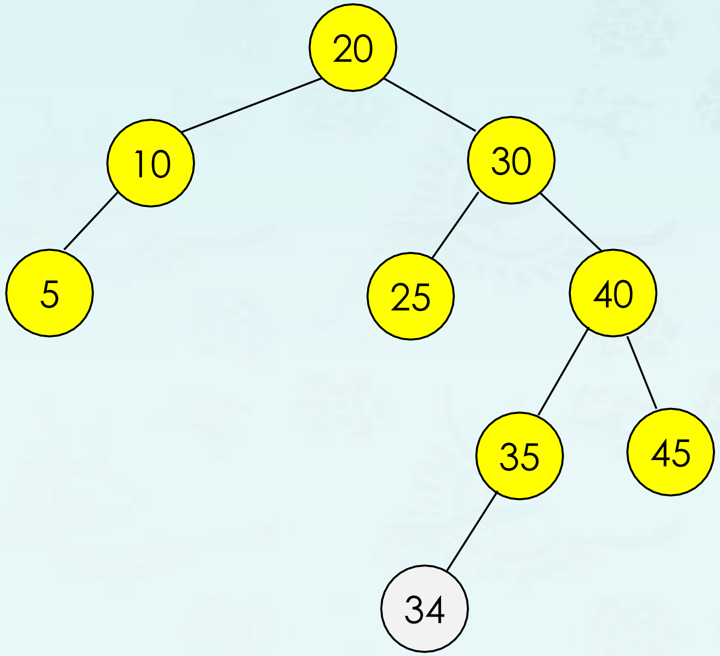
- **Insertion of 34**
- Imbalance at ?
- Balance factor ??
- Rotation ___??___ case



AVL balanced tree

Double Rotation -??case

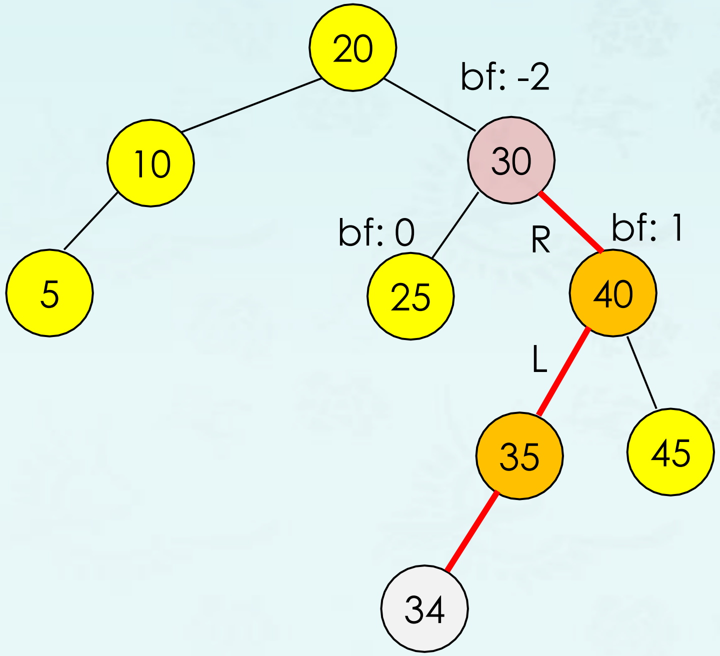
- Insertion of 34
- Imbalance at ?
- Balance factor ??
- Rotation ___??___ case



After insertion, AVL imbalanced tree

Double Rotation -??case

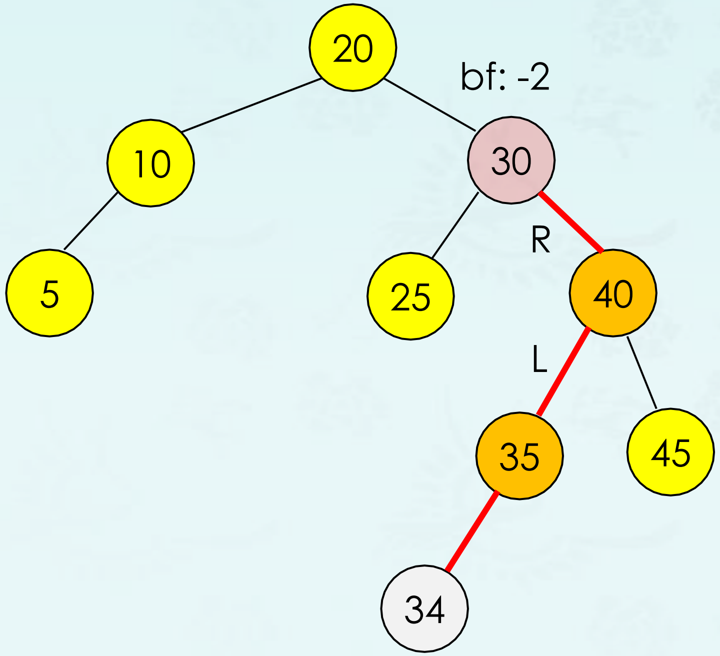
- Insertion of 34
- Imbalance at 30
- Balance factor **-2**
- Rotation ___??___ case



Double Rotation - RL case

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

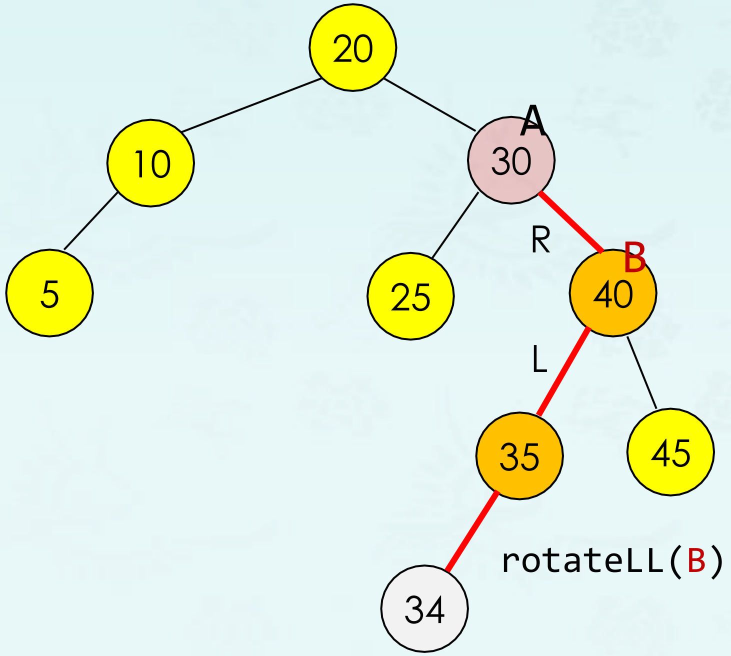
```
tree rotateRL(tree A) {
  tree B  = A->right;
  A->right = rotateLL(B);
  return rotateRR(A);
}
```



Double Rotation - RL case

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

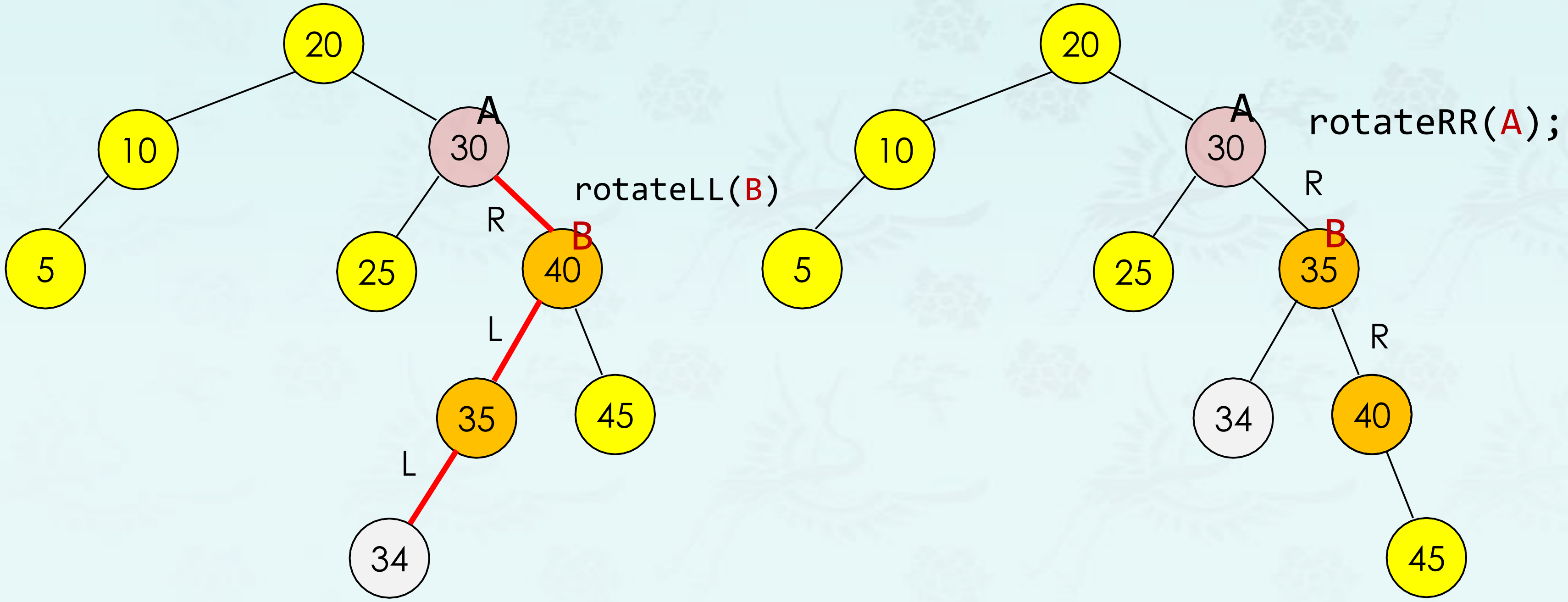
```
tree rotateRL(tree A) {
  tree B  = A->right;
  A->right = rotateLL(B);
  return rotateRR(A);
}
```



Double Rotation - RL case

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

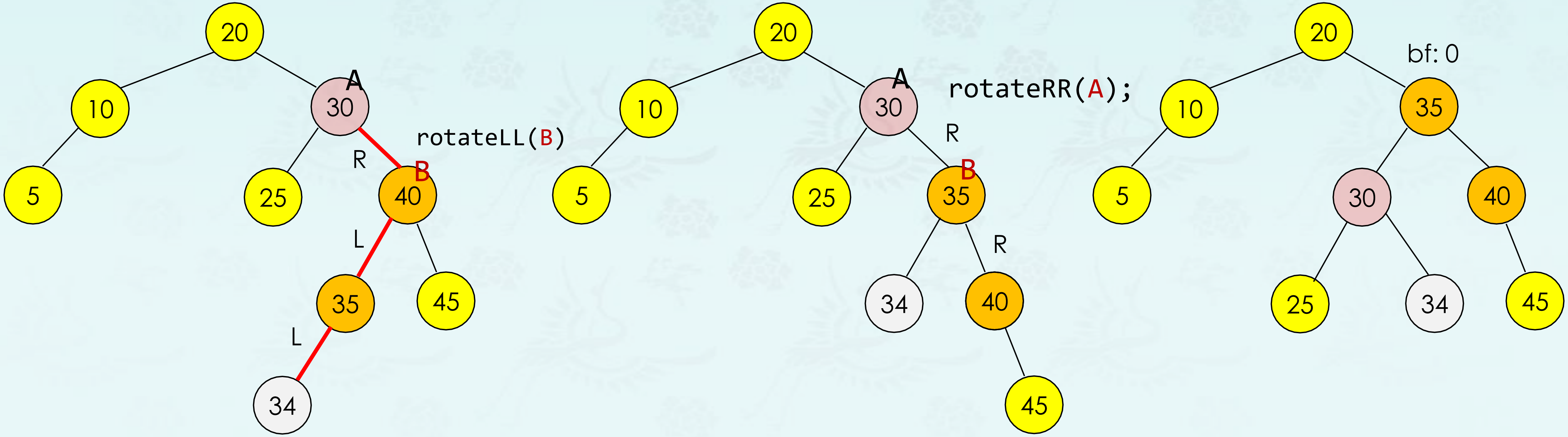
```
tree rotateRL(tree A) {
  tree B  = A->right;
  A->right = rotateLL(B);
  return rotateRR(A);
}
```



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

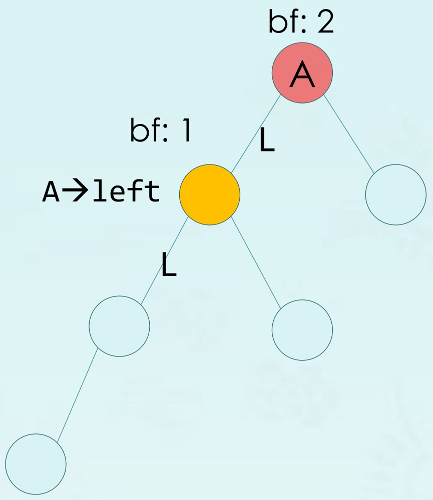
Balance Factor and Height

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

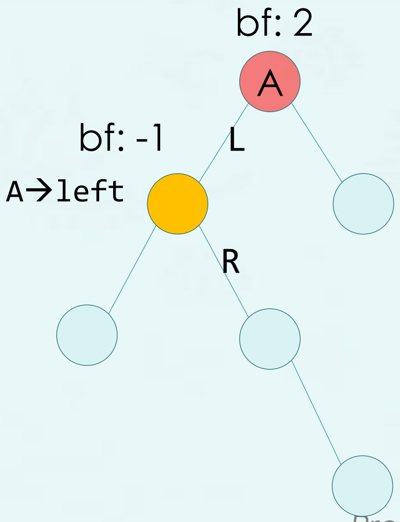
```
int balanceFactor(tree node) {
  if (node == NULL) return 0;
  int left  = height(node->left);
  int right = height(node->right);
  return left – right;
}
```

Rebalance

outside case



inside case



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

  }
```

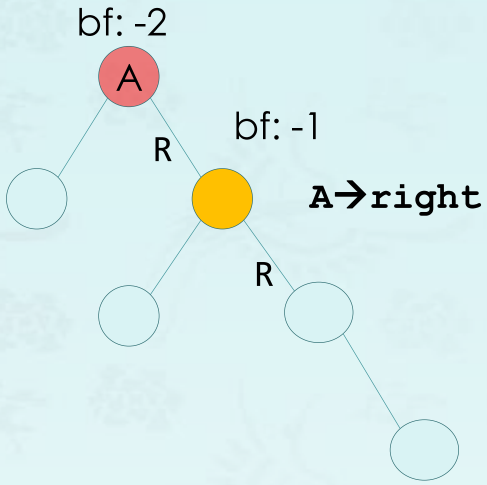
checking single or double rotation

Rebalance

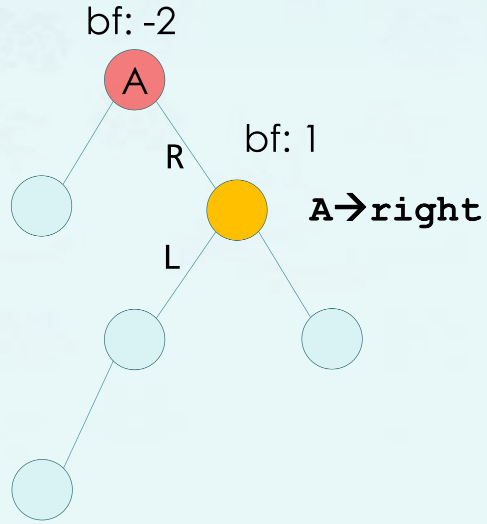
```
tree rebalance(tree A) {
  int bf = balanceFactor(A);
  if (bf == 2) {





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



  }
  return A;
}
```

checking single or double rotation

outside case

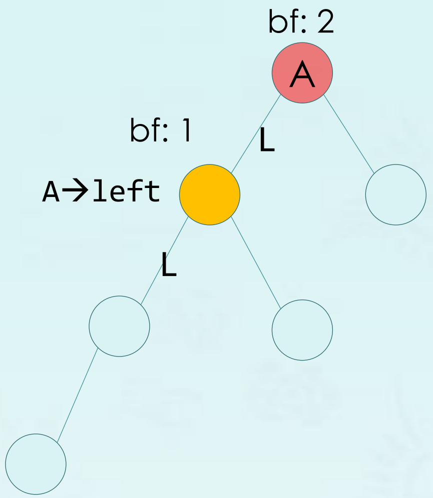


inside case

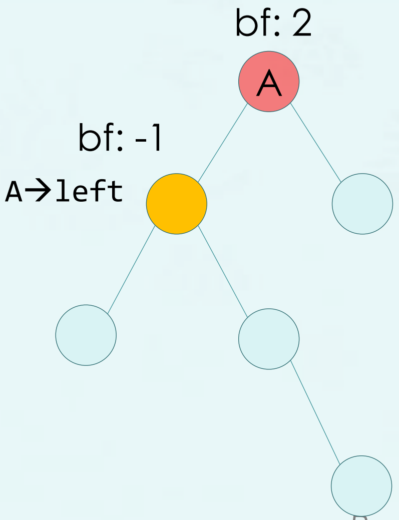


Rebalance

outside case



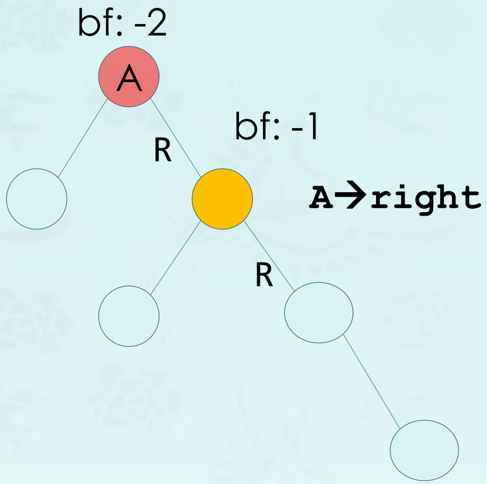
inside case



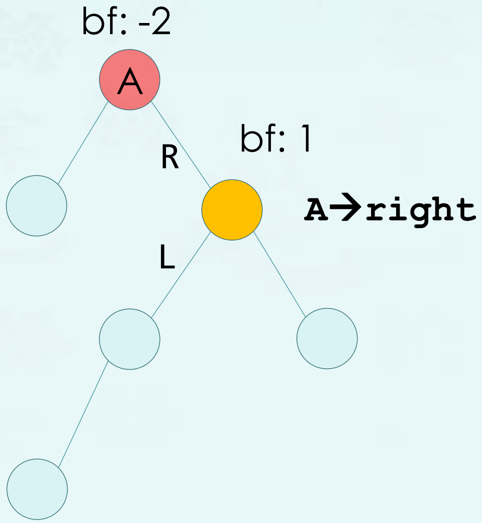
```
tree rebalance(tree A) {
  int bf = balanceFactor(A);
  if (bf == 2) {
    if (balanceFactor(A->left) == 1)



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



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

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

outside case



inside case



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 new TreeNode(key);

  // 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
   - **Coding**

Height of an AVL Tree

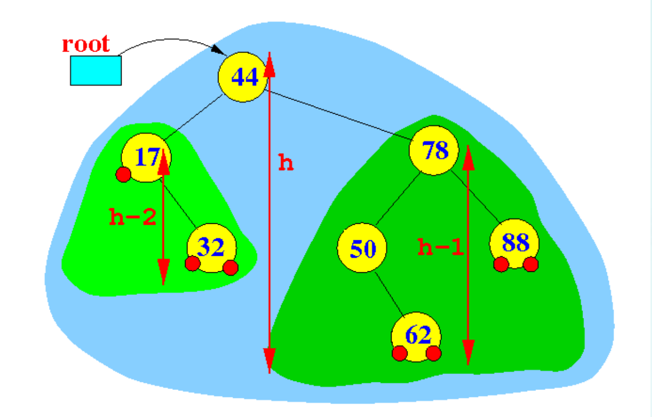
- What is the maximum height of an AVL tree having exactly n nodes?
  - To answer this question, we must ask this question first:
    What is the minimum number of nodes (sparsest possible AVL tree) an AVL tree of height h?

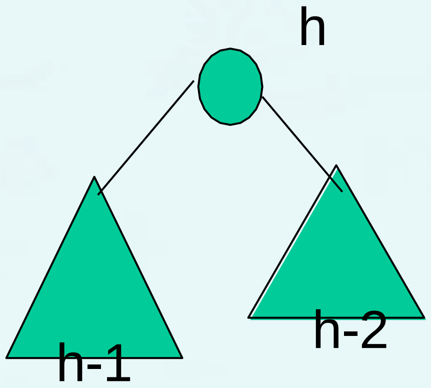
- Consider **the minimum number of nodes** in an AVL tree of height h:
- We can get the recurrence relationship:

$$\begin{aligned} n(0) &= 1 \\ n(1) &= 2 \\ n(2) &= 4 \\ n(h) &= n(h-1) + n(h-2) + 1 \end{aligned}$$

where $h > 1$

- This approximate solution of the recurrence is known as $\mathrm{n(h)} \cong 1.618^h$

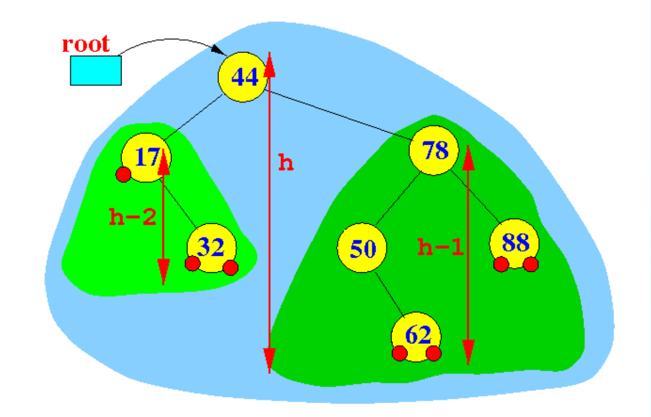


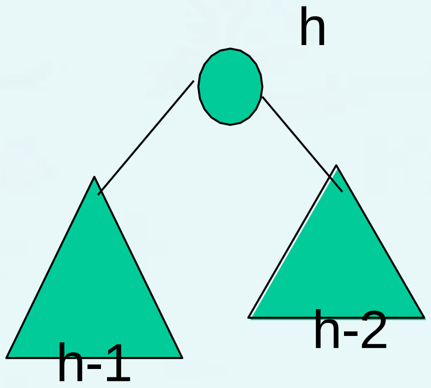


Height of an AVL Tree

- $n(h) = n(h-1) + n(h-2) + 1$
  where $h > 1$
- This approximate solution of the recurrence is known as $\mathrm{n(h)} \cong 1.618^h$

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





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**

- If there are n nodes in AVL tree, minimum height of AVL tree is floor(
- If there are n nodes in AVL tree, maximum height can't exceed
- If height of AVL tree is h, maximum number of nodes can be
- 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
- 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) =

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(
- If there are n nodes in AVL tree, maximum height can't exceed
- If height of AVL tree is h, maximum number of nodes can be
- 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
- 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) =

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
- If height of AVL tree is h, maximum number of nodes can be
- 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
- 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) =

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
- 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
- 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) =

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
- 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) =

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**.

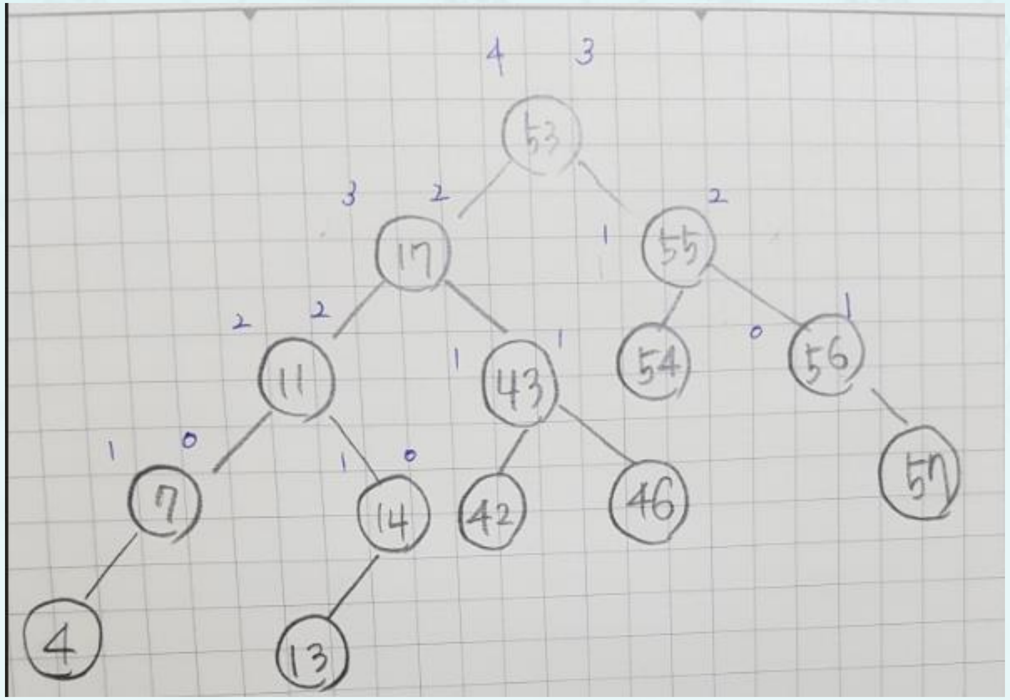
- If there are n nodes in AVL tree, minimum height of AVL tree is floor($\log_2 n$).
- If there are n nodes in AVL tree, maximum height can't exceed $1.44 * \log_2 n$.
- If height of AVL tree is h, maximum number of nodes can be $2^{h+1} - 1$.
- Minimum number of nodes in a tree with height h can be 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) =

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**.

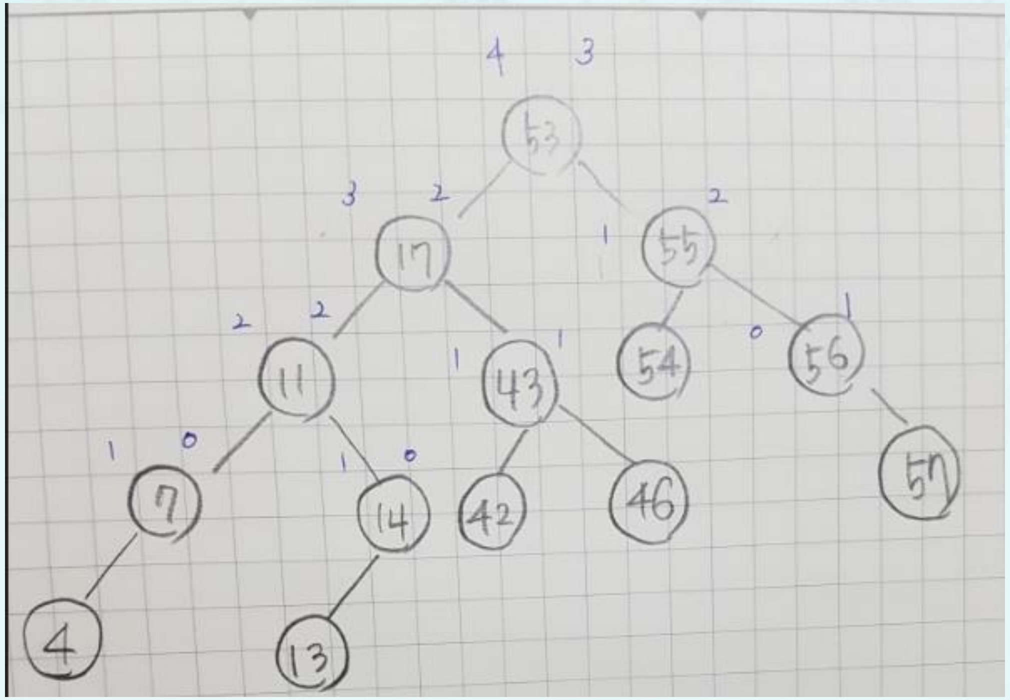
- If there are n nodes in AVL tree, minimum height of AVL tree is floor($\log_2 n$).
- If there are n nodes in AVL tree, maximum height can't exceed $1.44 * \log_2 n$.
- If height of AVL tree is h, maximum number of nodes can be $2^{h+1} - 1$.
- Minimum number of nodes in a tree with height h can be 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)

이것도 AVL tree가 될 수 있을까요?

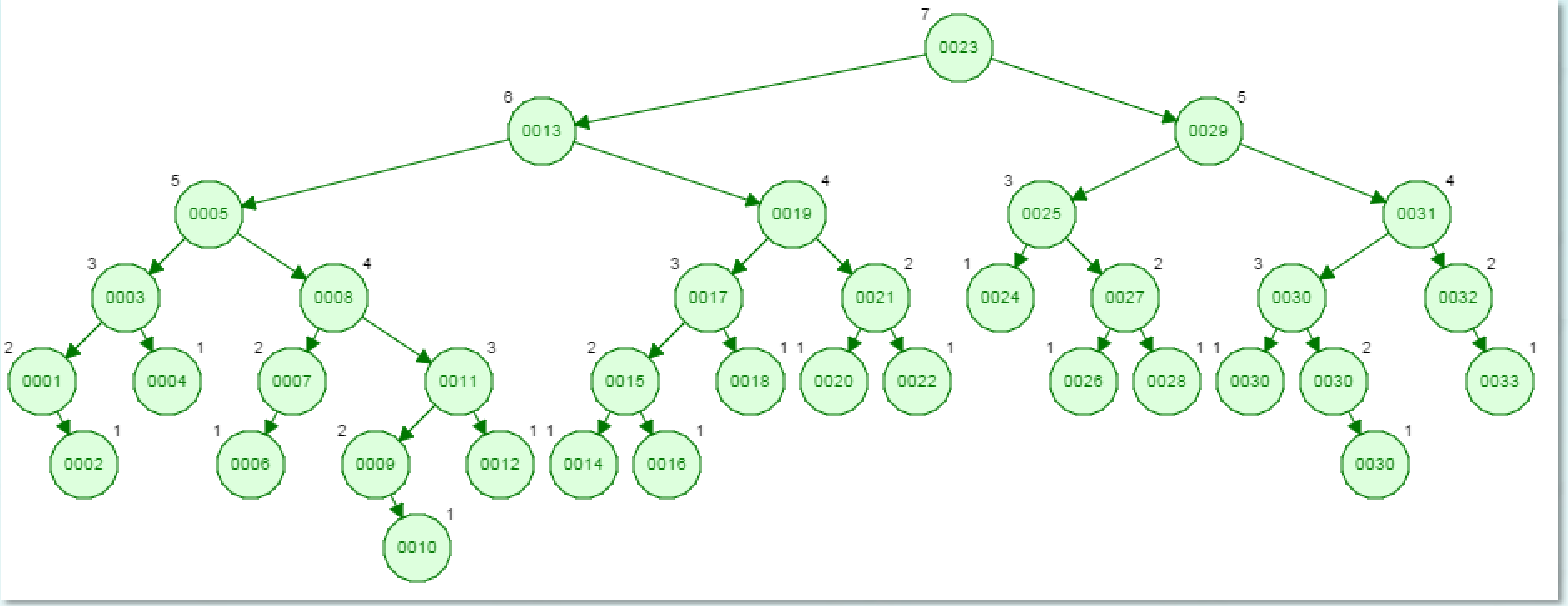


이것도 AVL tree가 될 수 있을까요?

- 저는 AVL tree가 모든 노드의 왼쪽과 오른쪽의 height의 차이가 절대값 1을 넘어서지 않는 것이라고 알고있습니다. 근데 이 트리는 모든 노드에서 왼쪽과 오른쪽의 height의 차이가 절대값 1을 넘지는 않지만 55-54가 연결되어 있는 부분의 높이가 다른 쪽에 비해 2이상 차이나는 것을 보았습니다. 제가 아는 정의상으로는 AVL tree인거 같으면서도 저렇게 height가 2이상 차이가 나니… 결론을 내릴 수가 없어 질문 드립니다.
- 제가 AVL tree의 정의를 잘못 알고 있는건가요?



Example with leaf 24 on level 3 and leaf 10 on level 6:



- AVL maintain the maximum height difference of 1 between two children subtree, not any two leaves.
- The difference in levels of any two leaves can be any value!
  The definition of AVL describes height difference only on two sub-trees from one node.

https://stackoverflow.com/questions/28964971/height-difference-between-leaves-in-an-avl-tree

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 new TreeNode(key);

  // your code here

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

almost same as trim()

AVL rotation if necessary

growN() & TrimN()

```
// removes randomly N numbers of nodes in the tree(AVL or BST).
// It gets N node keys from the tree, trim one by one randomly.
tree trimN(tree root, int N, bool AVLtree) {  // testing purpose
  vector<int> vec;

  // your code here

  delete[] arr;
  return root;
}
```

growN() & TrimN()

```
tree growN(tree root, int N, bool AVLtree) {    // coding a faster version
  int start = empty(root) ? 0 : value(maximum(root)) + 1;
  int* arr = new (nothrow) int[N];
  assert(arr != nullptr);
  randomN(arr, N, start);

#if 0  // use BST grow() first. then, if AVLtree, reconstruct it as AVL.
  for (int i = 0; i < N; i++) root = grow(root, arr[i]);
  if (AVLtree) root = reconstruct(root);
#else // use its own grow() function, respectively. it is too slow
  if (AVLtree)
    for (int i = 0; i < N; i++) root = growAVL(root, arr[i]);
  else
    for (int i = 0; i < N; i++) root = grow(root, arr[i]);
#endif

  delete[] arr;
  return root;
}
```

Reconstruct() – Building AVL tree from BST in O(n)

- **Goal:** Reconstruct a new AVL tree from BST in O(n).
- **Intuition:** Since we can get a sorted key values from the binary search tree, we take advantage of the sorted list to form a well balanced AVL tree faster.
- **Recreation Method**
  - Use an array of keys, using an existing inorder() function that returns a sorted keys in vector.
  - Clear the original tree since it is not used any more.
  - Since it goes through the tree twice only, the time complexity is O(n).

Reconstruct() – Building AVL tree from BST in O(n)

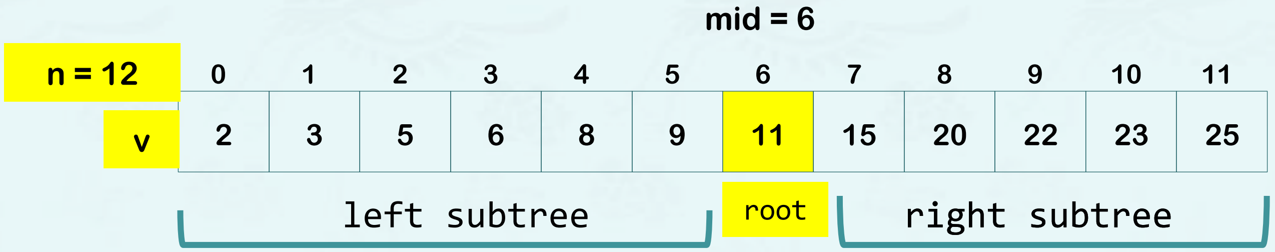
- **Goal:** Reconstruct a new AVL tree from BST in O(n).
- **Intuition:** Since we can get a sorted key values from the binary search tree, we take advantage of the sorted list to form a well balanced AVL tree faster.
- **Recreation Method**
  - Use an array of keys, using an existing inorder() function that returns a sorted keys in vector.
  - Clear the original tree since it is not used any more.
  - Since it goes through the tree twice only, the time complexity is O(n).
- **Recycling Method**
  - Use an array of nodes, simply reconstructs (or relink) the existing nodes.
  - Write a new inorder() that returns the sorted nodes of the tree.
  - Since it goes through the tree twice only, the time complexity is O(n).

Reconstruct() – Building AVL tree from BST in O(n)

- **Goal:** Reconstruct a new AVL tree from BST in O(n).
- **Intuition:** Since we can get a sorted key values from the binary search tree, we take advantage of the sorted list to form a well balanced AVL tree faster.
- **Recreation Method**
  - Use an array of keys, using an existing inorder() function that returns a sorted keys in vector.
  - Clear the original tree since it is not used any more.
  - Since it goes through the tree twice only, the time complexity is O(n).
- **Recycling Method**
  - Use an array of nodes, simply reconstructs (or relink) the existing nodes.
  - Write a new inorder() that returns the sorted nodes of the tree.
  - Since it goes through the tree twice only, the time complexity is O(n).
- **For pedagogical purpose**, let us use the recycling method if the number of nodes are more than 10, otherwise use the recreation method.

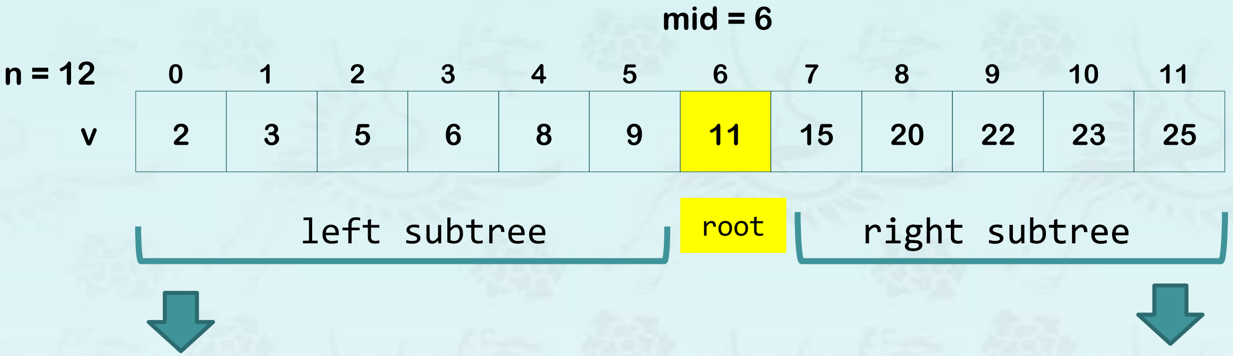
Reconstruct() – Building AVL tree from BST in O(n)

```
// reconstructs a new AVL tree from BST in O(n).
tree reconstruct(tree root) {
  if (root == nullptr) return nullptr;

  if (size(root) > 10) {           // recycling method

        cout << your code here

  }
  else {                           // recreation method

        cout << your code here

  }
  return root;
}
```



n = 12	0	1	2	3	4	5	6	7	8	9	10	11
v	2	3	5	6	8	9	11	15	20	22	23	25

Reconstruct() – Building AVL tree from BST in O(n)



mid = 6

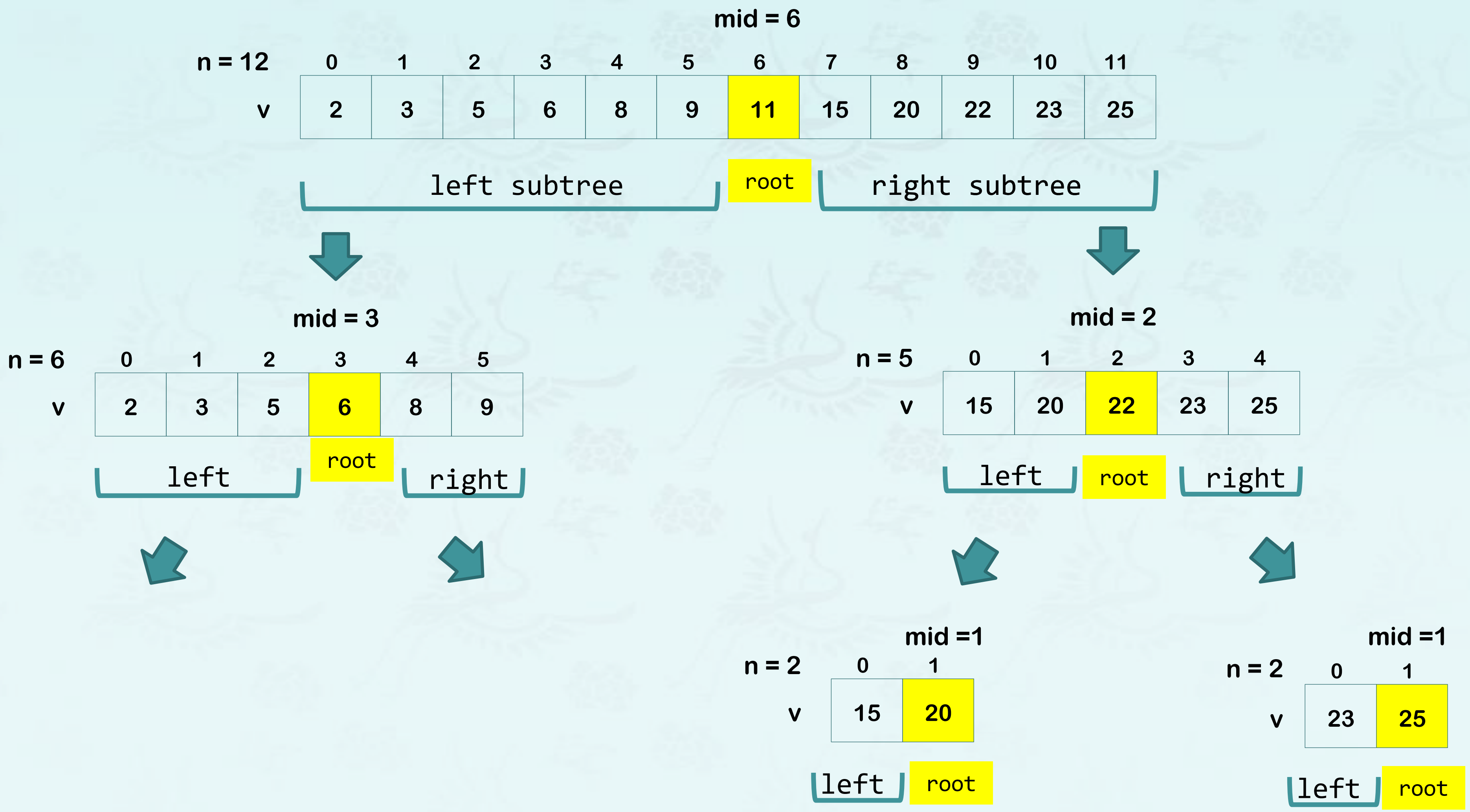
n = 12	0	1	2	3	4	5	6	7	8	9	10	11
v	2	3	5	6	8	9	11	15	20	22	23	25

left subtree

root

right subtree

Reconstruct() – Building AVL tree from BST in O(n)



mid = 6

n = 12

	0	1	2	3	4	5	6	7	8	9	10	11
v	2	3	5	6	8	9	11	15	20	22	23	25

left subtree / root / right subtree

mid = 3

n = 6

	0	1	2	3	4	5
v	2	3	5	6	8	9

left / root / right

mid = 2

n = 5

	0	1	2	3	4
v	15	20	22	23	25

left / root / right

mid =1

n = 2

	0	1
v	15	20

left / root

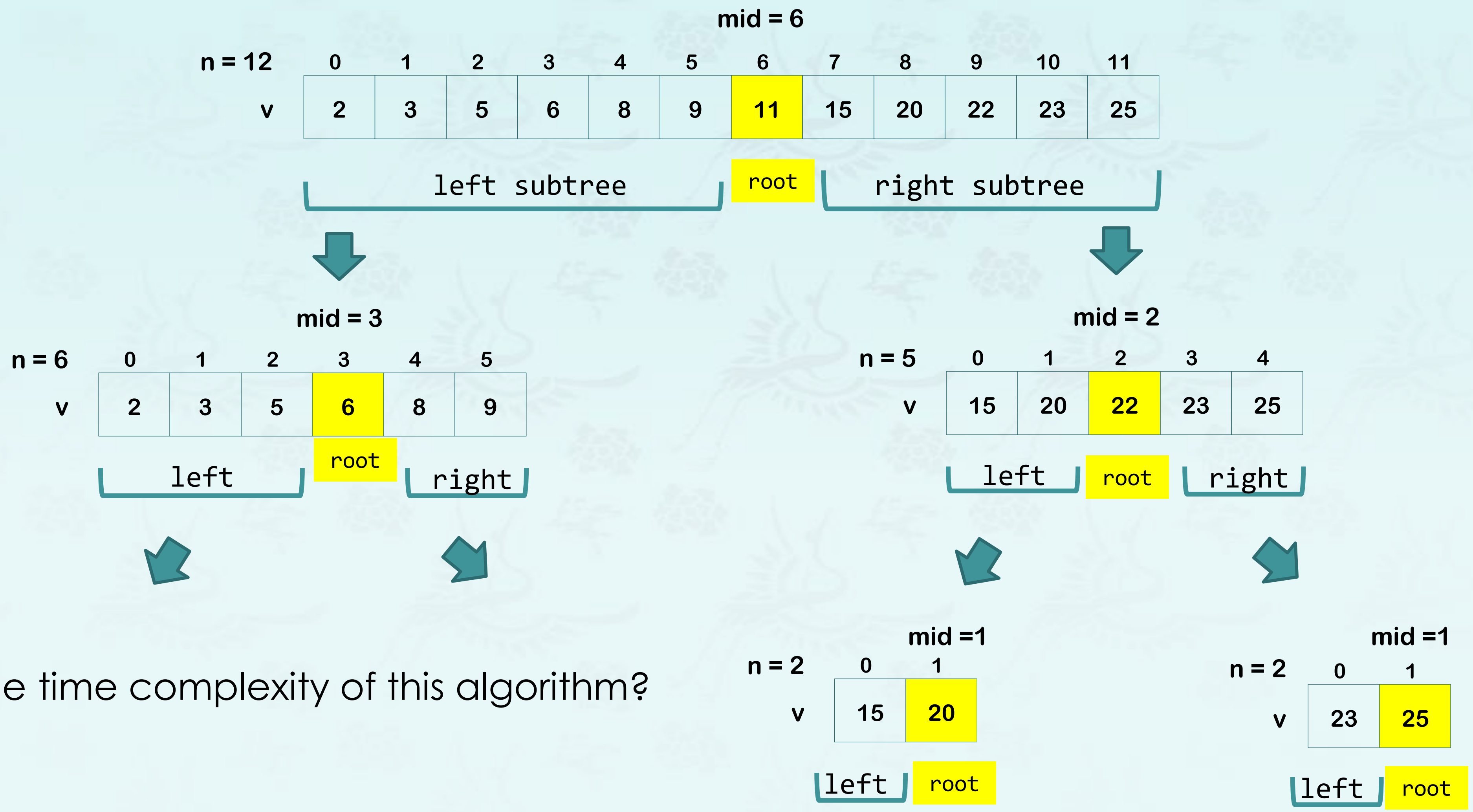
mid =1

n = 2

	0	1
v	23	25

left / root

Reconstruct() – Building AVL tree from BST in O(n)



mid = 6

n = 12	0	1	2	3	4	5	6	7	8	9	10	11
v	2	3	5	6	8	9	11	15	20	22	23	25

mid = 3

n = 6	0	1	2	3	4	5
v	2	3	5	6	8	9

mid = 2

n = 5	0	1	2	3	4
v	15	20	22	23	25

mid =1

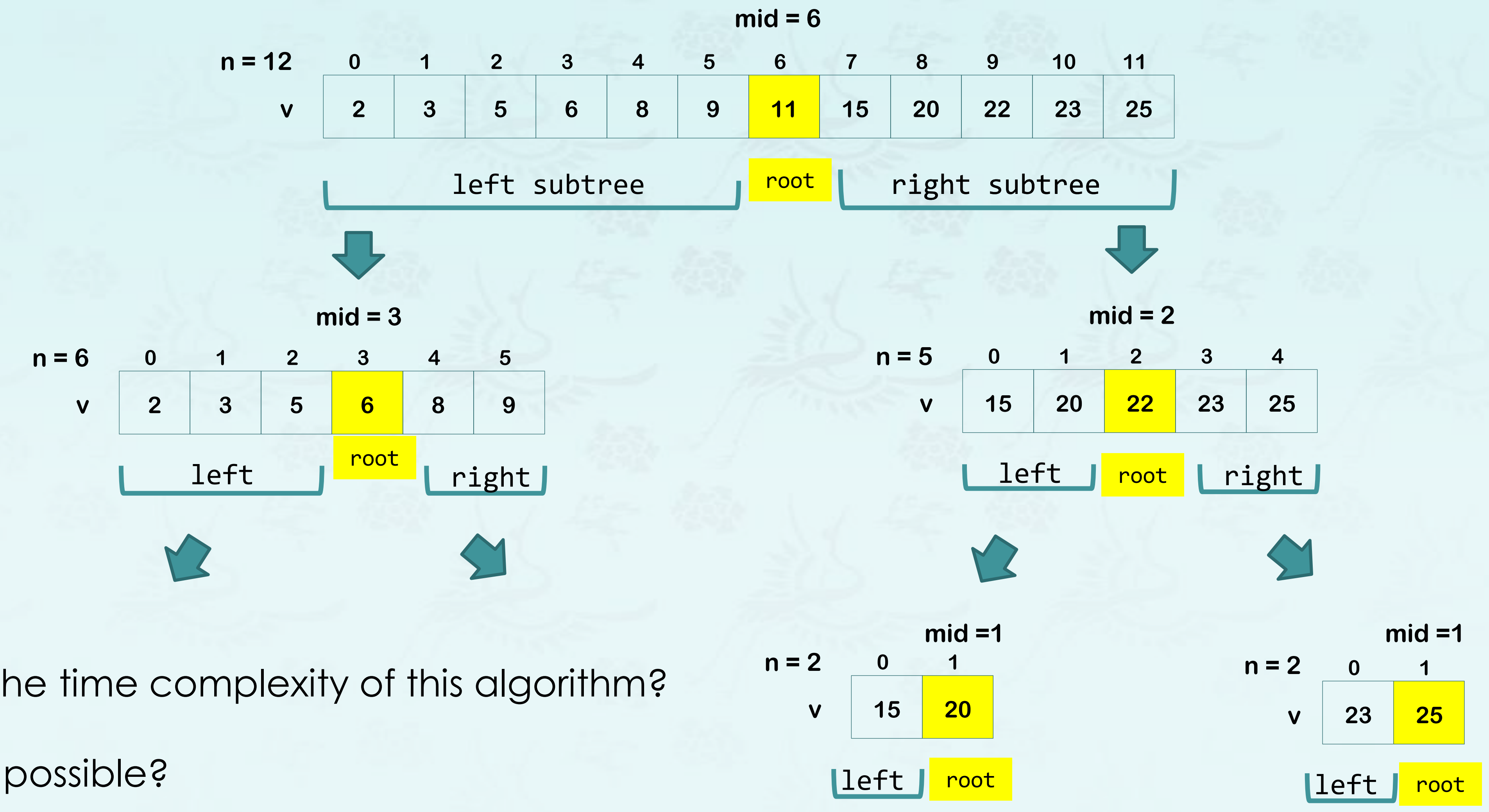
n = 2	0	1
v	15	20

mid =1

n = 2	0	1
v	23	25

What is the time complexity of this algorithm?

Reconstruct() – Building AVL tree from BST in O(n)



mid = 6

n = 12	0	1	2	3	4	5	6	7	8	9	10	11
v	2	3	5	6	8	9	11	15	20	22	23	25

left subtree · root · right subtree

mid = 3

n = 6	0	1	2	3	4	5
v	2	3	5	6	8	9

left · root · right

mid = 2

n = 5	0	1	2	3	4
v	15	20	22	23	25

left · root · right

mid =1

n = 2	0	1
v	15	20

left · root

mid =1

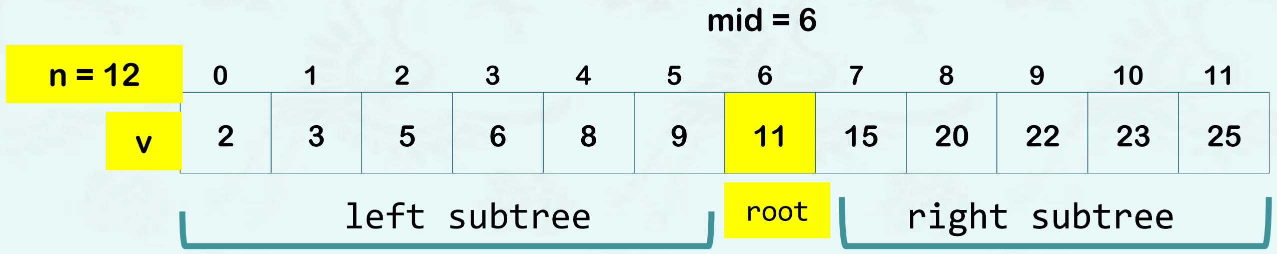
n = 2	0	1
v	23	25

left · root

What is the time complexity of this algorithm?
O(n)
How is it possible?

Building AVL tree from BST in O(n) – recreation method

```
// rebuilds an AVL tree with a list of keys sorted.
// v - an array of keys sorted, n - the array size
tree buildAVL(int* v, int n) {
  if (n <= 0) return nullptr;
  int mid = n / 2;

  tree root = 

  // recursive buildAVL() calls for left & right, return it to root->left & root->right



  return root;
}
```

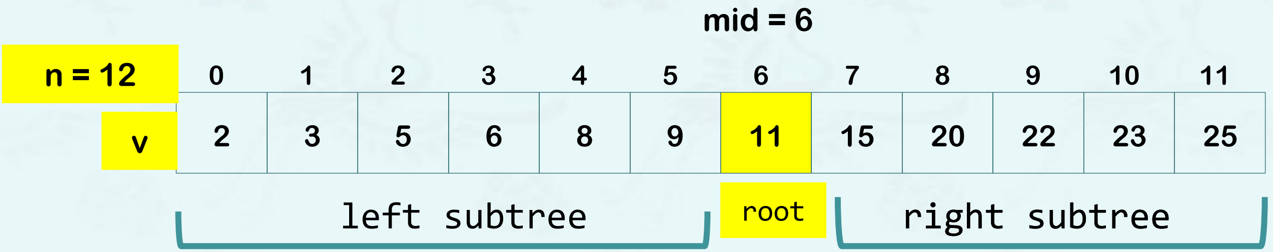


mid = 6

n = 12	0	1	2	3	4	5	6	7	8	9	10	11
v	2	3	5	6	8	9	11	15	20	22	23	25

Building AVL tree from BST in O(n) – recreation method

```
// rebuilds an AVL tree with a list of keys sorted.
// v - an array of keys sorted, n - the array size
tree buildAVL(int* v, int n) {
  if (n <= 0) return nullptr;
  int mid = n / 2;

  tree root = new TreeNode(v[mid]);    // create a root node

  // recursive buildAVL() calls for left & right, return it to root->left & root->right



  return root;
}
```

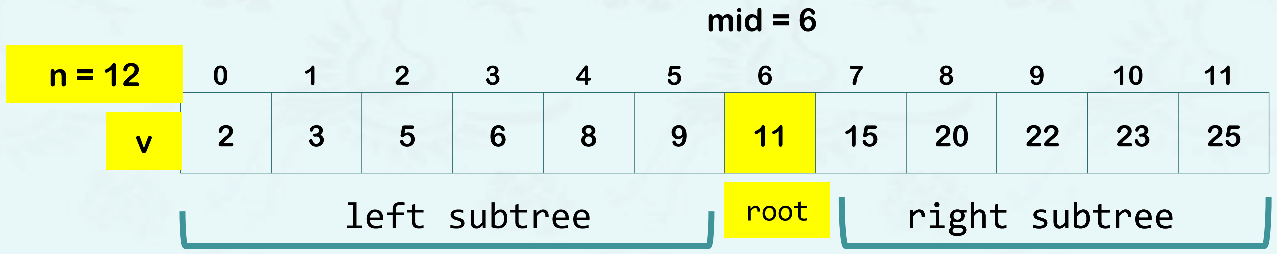


mid = 6

n = 12	0	1	2	3	4	5	6	7	8	9	10	11
v	2	3	5	6	8	9	11	15	20	22	23	25

Building AVL tree from BST in O(n) – recycling method

```
// rebuilds an AVL tree using a list of nodes sorted, no memory allocations
// v - an array of nodes sorted, n - the array size
tree buildAVL(tree* v, int n) {
  if (n <= 0) return nullptr;
  int mid = n / 2;

  tree root = 
                                             // set leaf nodes to null for recycling.
  // recursive buildAVL() calls for left & right, return it to root->left & root->right



  return root;
}
```

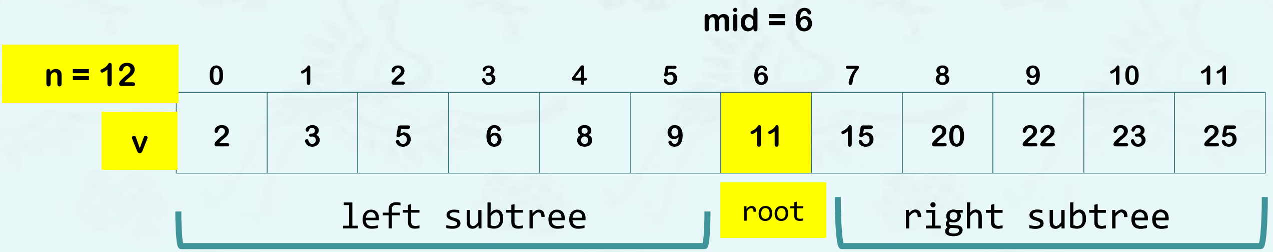


mid = 6

n = 12	0	1	2	3	4	5	6	7	8	9	10	11
v	2	3	5	6	8	9	11	15	20	22	23	25

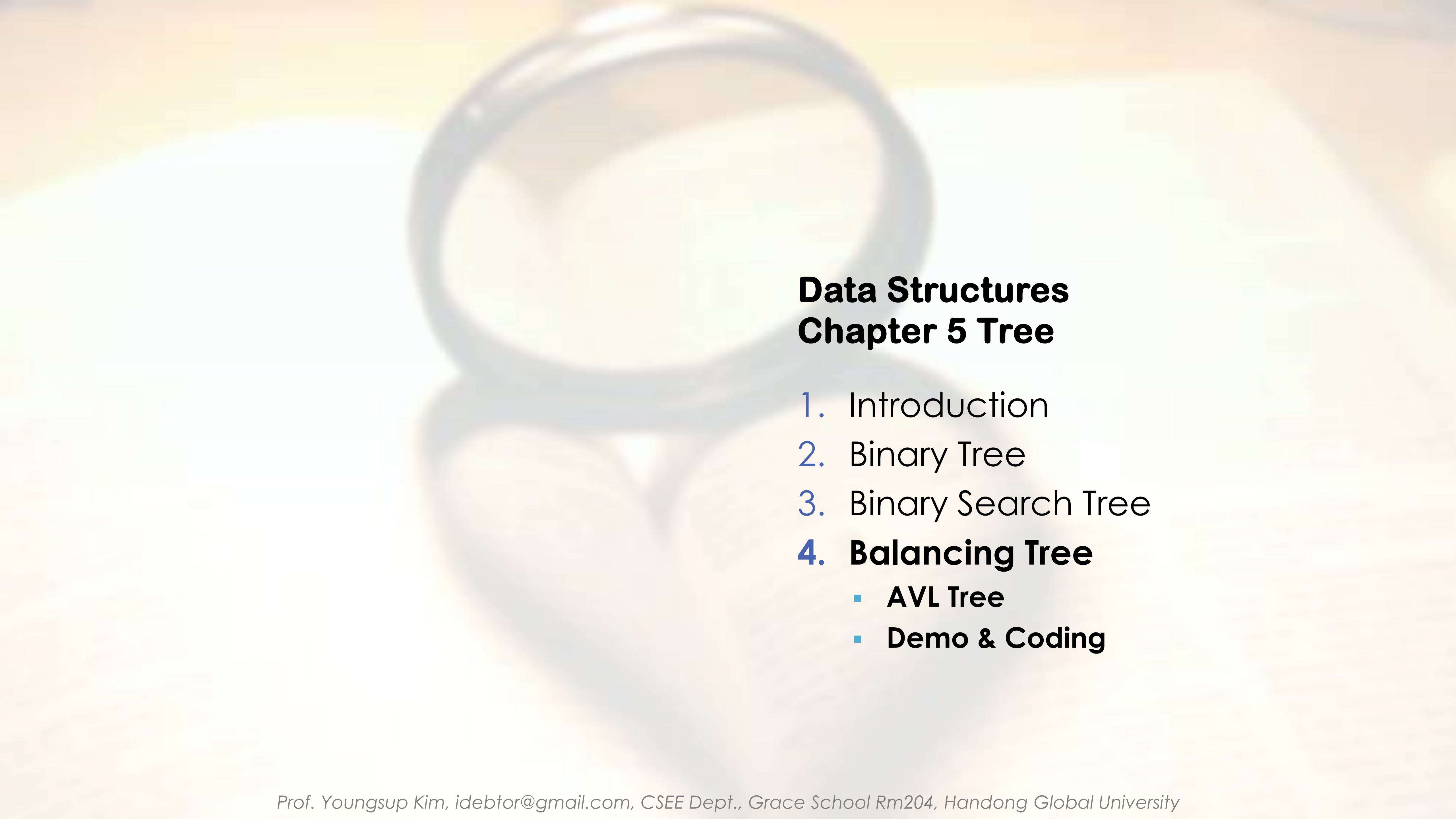
Building AVL tree from BST in O(n) – recycling method

```
// rebuilds an AVL tree using a list of nodes sorted, no memory allocations
// v – an array of nodes sorted, n – the array size
tree buildAVL(tree* v, int n) {
  if (n <= 0) return nullptr;
  int mid = n / 2;

  tree root = v[mid];                        // mid becomes the root; don't call new TreeNode.
                                             // set leaf nodes to null for recycling.
  // recursive buildAVL() calls for left & right, return it to root->left & root->right




  return root;
}
```



mid = 6

n = 12	0	1	2	3	4	5	6	7	8	9	10	11
v	2	3	5	6	8	9	11	15	20	22	23	25



Data Structures
Chapter 5 Tree

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