COMP 251

Algorithms Structures (Winter 2022)

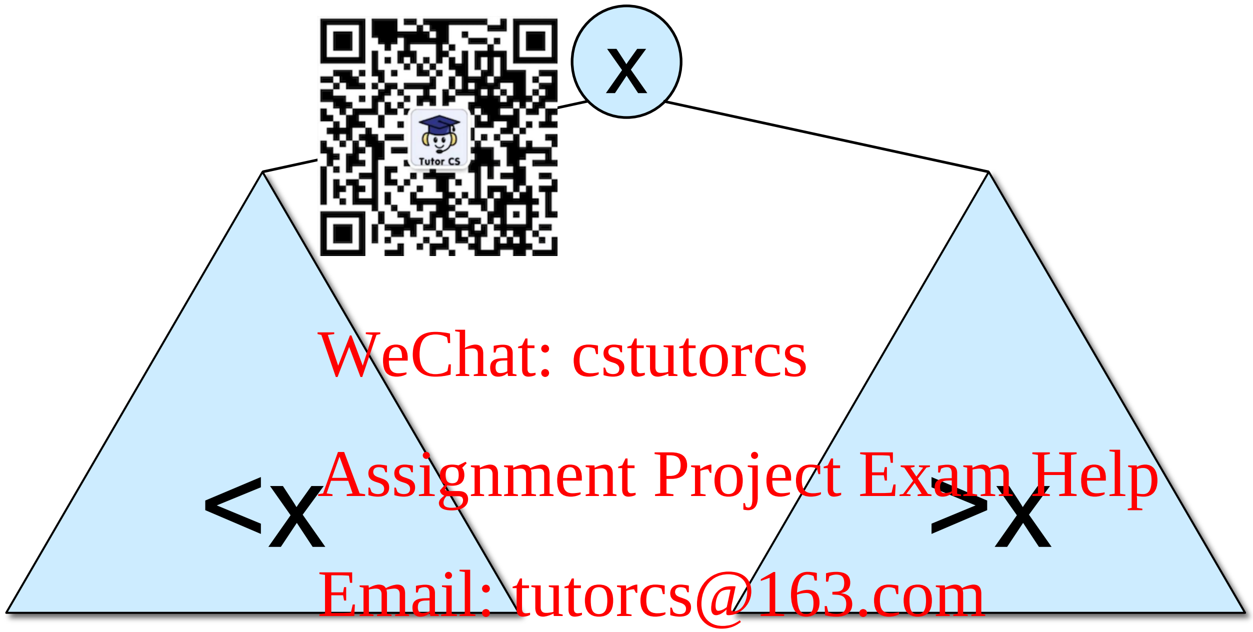
AVL

Announcements

Outline

- Introduction.
- Operations.
- Application.

Introduction – Binary Search Trees



- T is a rooted binary tree
- Key of a node x > keys in its left subtree.
- Key of a node x < keys in its right subtree.

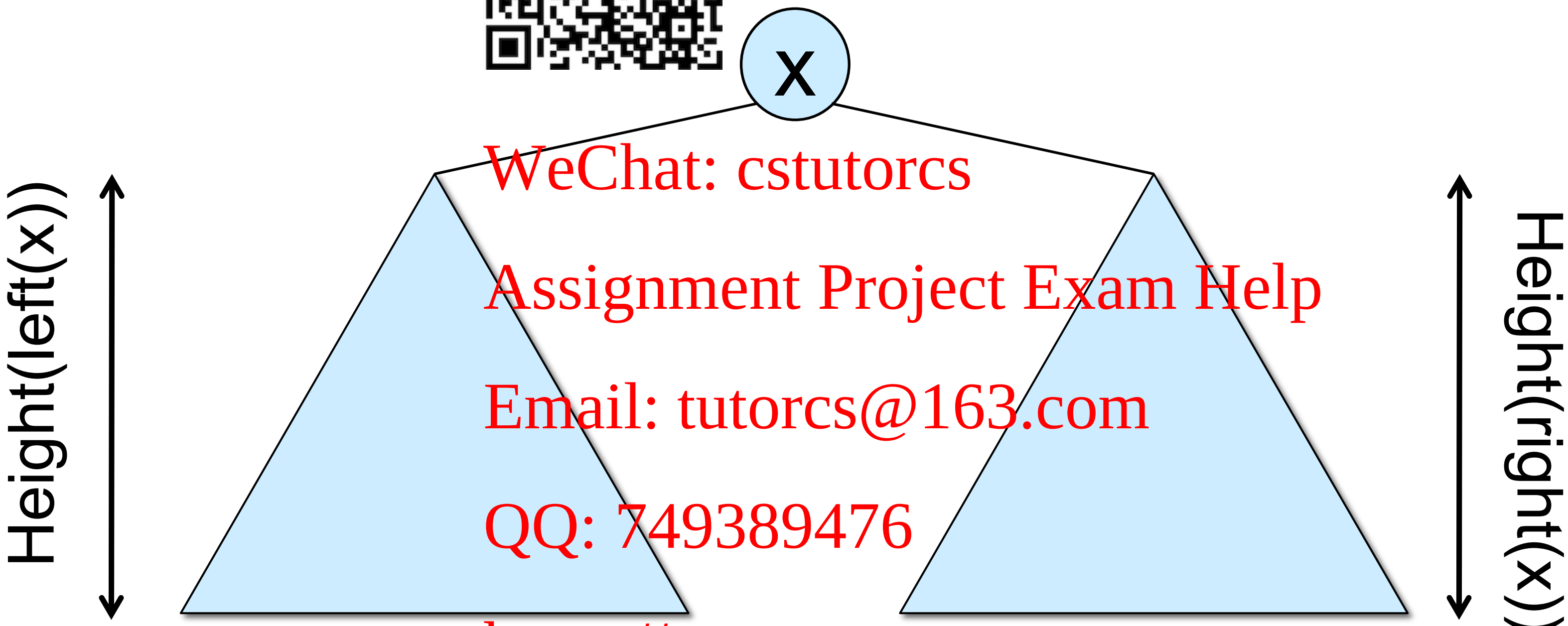
BST – Operations

- Search(T,k): O(h)
- Insert(T,k): O(h)
- Delete(T,k): O(h)

Where h is the height of the BST.

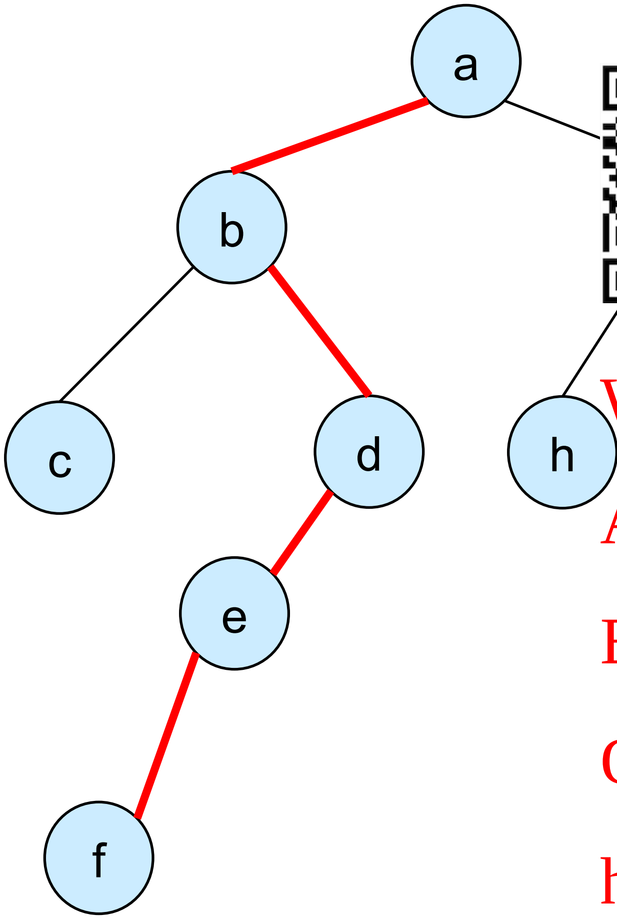
BST – Height of a tree

Height(n): le[illegible]dges) of longest downward path from no[illegible]a leaf.



Height(x) = 1 + max(height(left(x)), height(right(x)))

BST – Height of a tree - Example



h(a) = ?

= 1+max(h(b) , h(g))

= 1+max(1+max(h(c),h(d)),1+h(h))

= 1+max(1+max(0,h(d)),1+0)

= 1+max(1+max(0,1+h(e)),1)

= 1+max(1+max(0,1+(1+h(f)))),1)

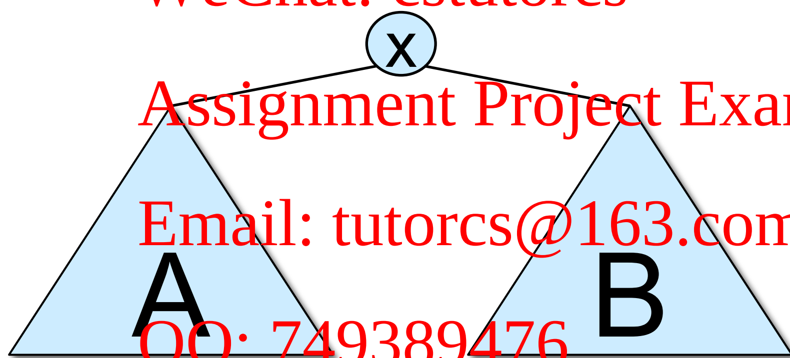
= 1+max(1+max(0,1+(1+0))),1)

= 1+max(3,1)

= 4

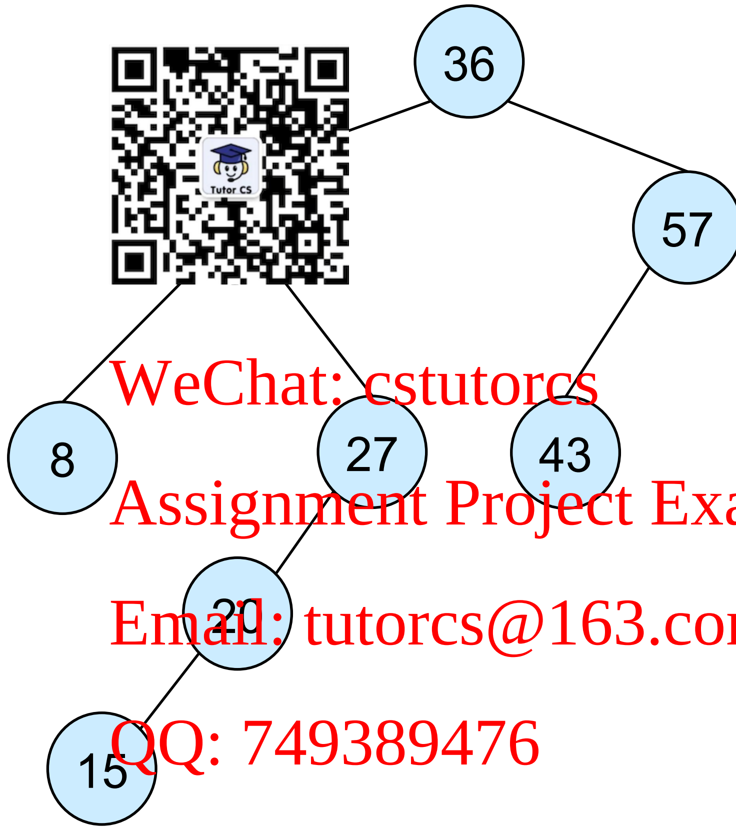
BST – In-order traversal


```
inorderTraversal(treeNode x)
    if x != NIL
        inorderTraversal(x.leftChild);
        print x.value;
        inorderTraversal(x.rightChild);
```

- Print the nodes in the left subtree (A), then node x, and then the nodes in the right subtree (B)
  - In a BST, it prints first keys < x, then x, and then keys > x.

BST – In-order traversal



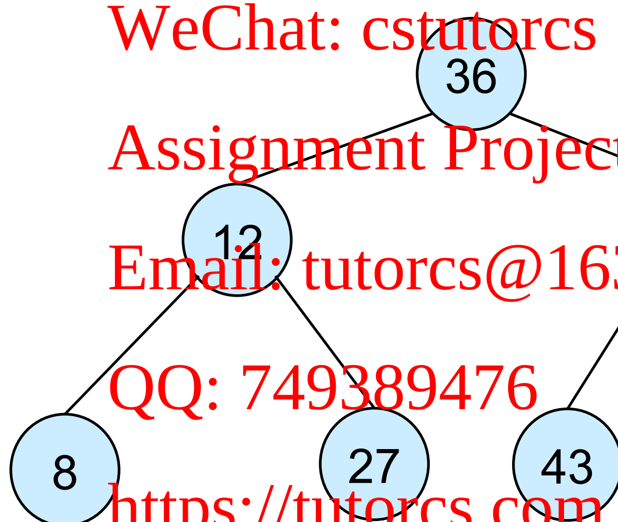
8, 12, 15, 20, 27, 36, 43, 57

All keys come out sorted!

BST – In-order traversal - Sort

1. Build a BST from the list of keys (unsorted)
2. Use in-order traversal on the BST to print the keys.

36	12	8	57	43	27



→ 8, 12, 27, 36, 43, 57

Running time of BST sort: insertion of n keys + tree traversal.

BST – Sort – Running time

- In-order traversal is O(n)
- Running time of insert is O(h)

Best case: The BST is always balanced for every insertion.

$$\Omega(n\log(n))$$

Worst case: The BST is always un-balanced. All insertions on same side.

$$\sum_{i=1}^{n} i = \frac{n\cdot(n-1)}{2} = O(n^2)$$

BST – Good vs Bad BSTs

Balanced

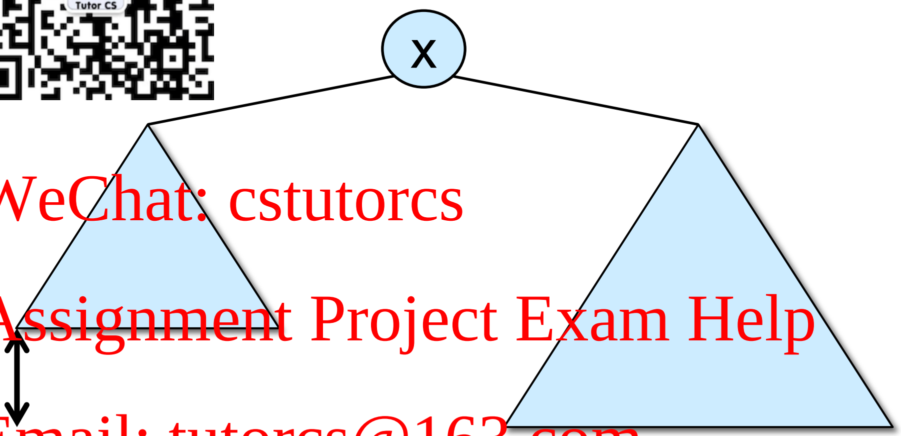
$h=O(\log n)$

Unbalanced

$h=O(n)$

AVL - Trees

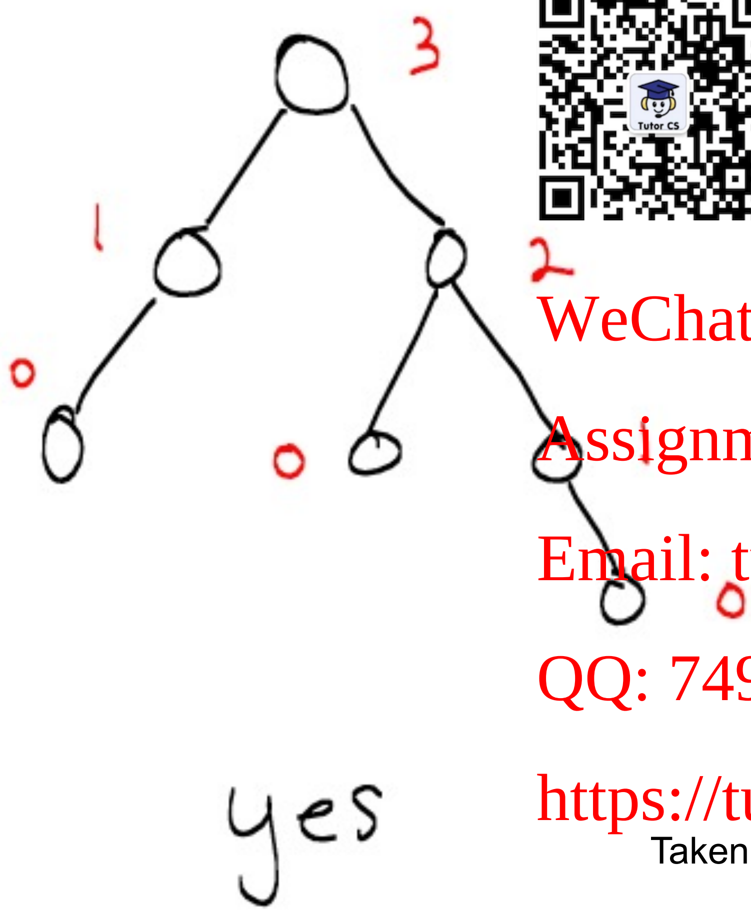
Definition: BST such that the heights of the two child subtrees of any node differ by one.

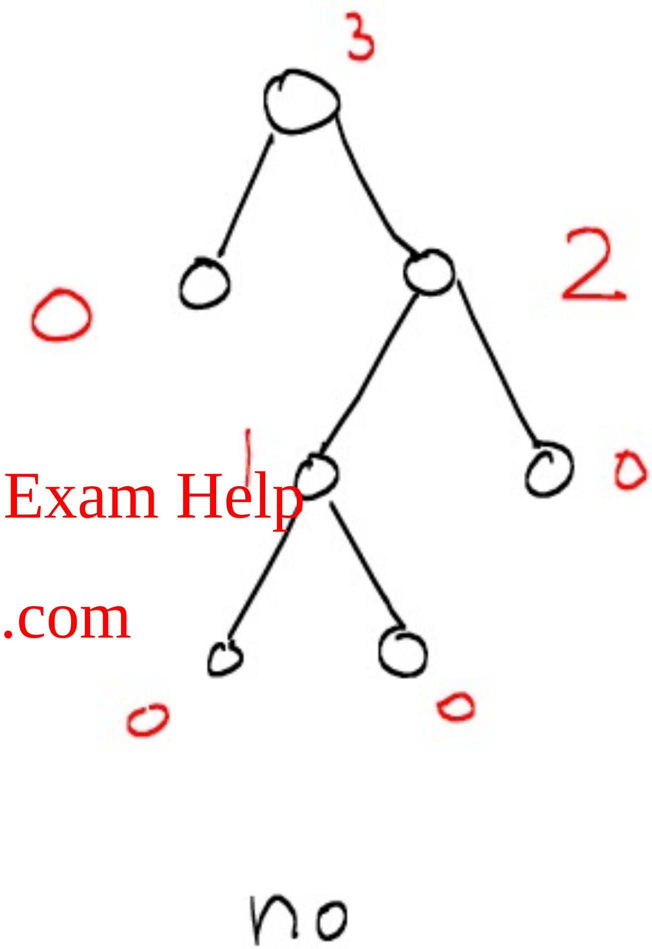


$|h_{left}-h_{right}| \le 1$

- Invented by G. Adelson-Velsky and E.M. Landis in 1962.
- AVL trees are self-balanced binary search trees.
- Insert, Delete & Search take O(log n) in average and worst cases.
- To satisfy the definition, the height of an empty subtree is -1

AVL – Trees -Example

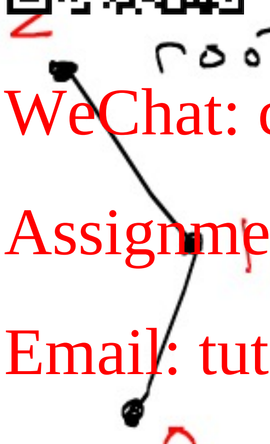




Taken from Langer2014

AVL – Trees -Example

Weird [illegible]ommon example



root's left child is null

Define the height of an empty tree to be −1.

Taken from Langer2014

AVL – Trees height – Worst case

- AVL trees with a minimum number of nodes are the worst case examples.
  - every node's subtrees ight by one.
  - we cannot make thes worse / any more unbalanced.
    - If we add or remove a either get a non-AVL or balance one of the subtree.

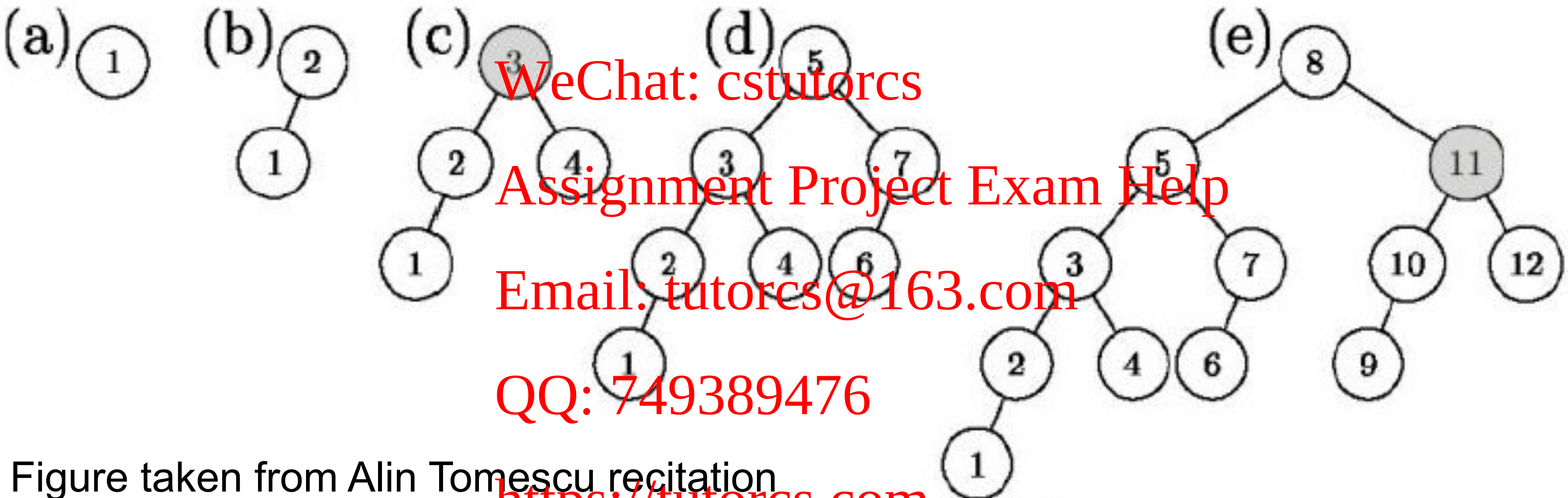


Figure taken from Alin Tomescu recitation

"If we can bound the height of these worst-case examples of AVL trees, then we've pretty much bounded the height of all AVL trees"

AVL – Trees height

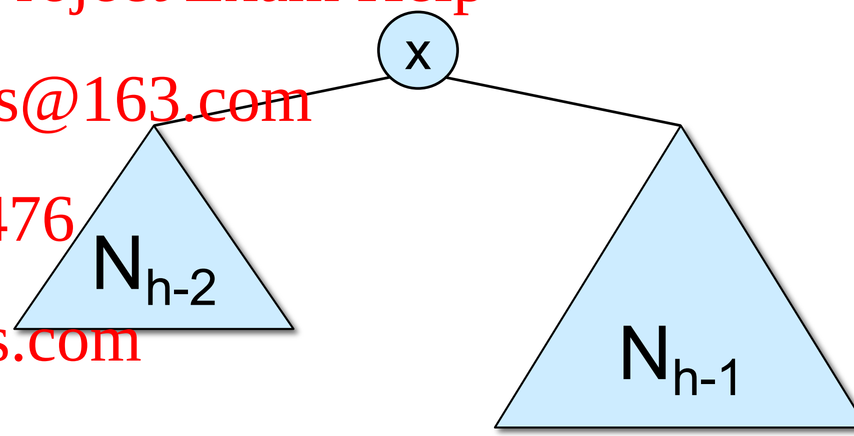
N_h = minimum #nodes in an AVL tree of height h.

$$N_h = 1 + N_{h-1} + N_{h-2}$$
$$N_h > 1 + N_{h-2} + N_{h-2}$$
$$N_h > 2 * N_{h-2}$$
$$N_h > 2 * 2 * N_{h-4} > 2 * 2 * 2 * N_{h-6} > \ldots > 2^{h/2}$$
$$N_h > 2^{h/2}$$

$\Longrightarrow \log(N_h) > \log(2^{h/2})$

$\Longrightarrow 2^*\log(N_h) > h$

$\Longrightarrow h = O(\log(n))$



Larger height when tree is unbalanced.

Outline

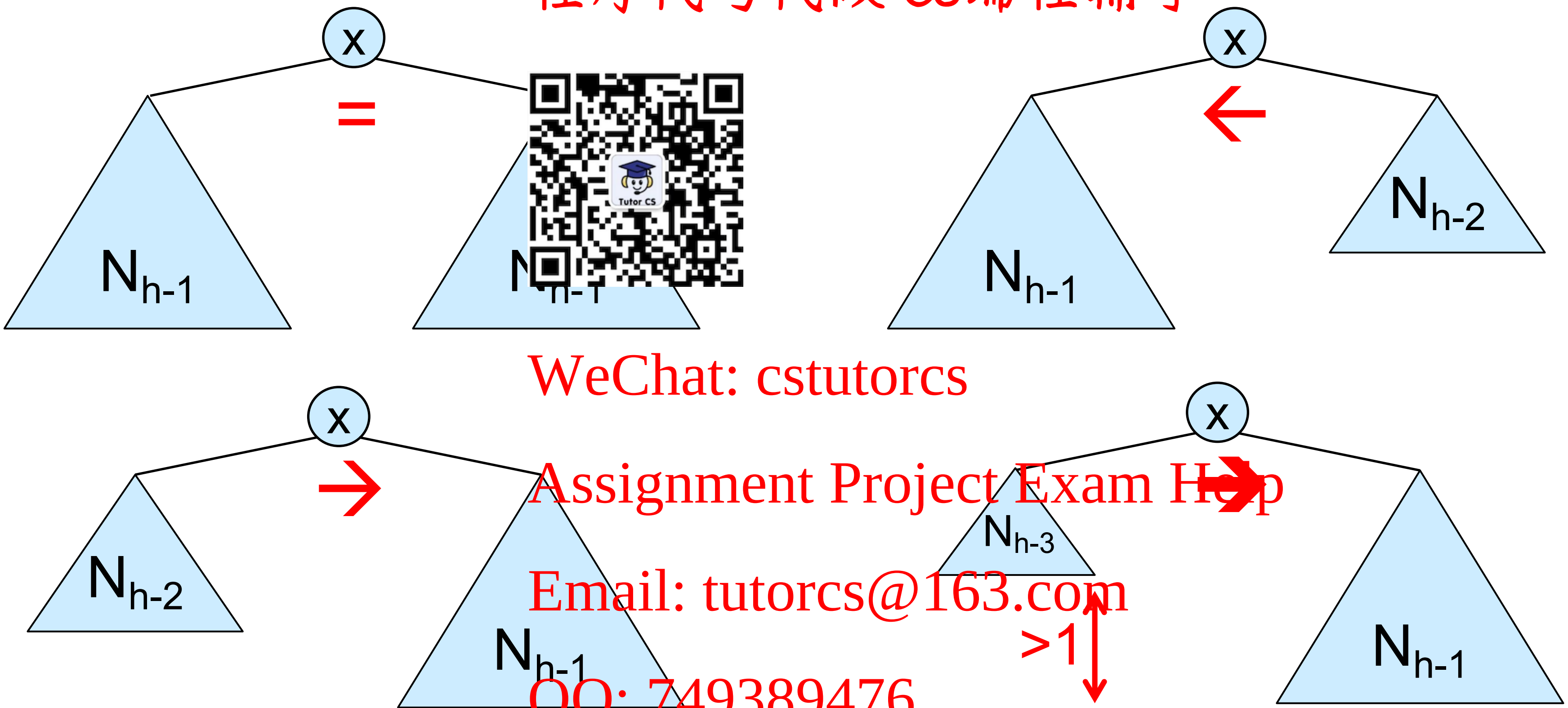
- Introduction.
- Operations.
- Application.

Definition: Balance Factor

- The **balance factor** of a binary **tree** is the difference in heights of its two subtrees (hL - hR). It m one of the values -1, 0, +1.

Figure taken from randerson112358.medium.com.

Definition: Balance Factor



←: Left tree is higher (left-heavy)

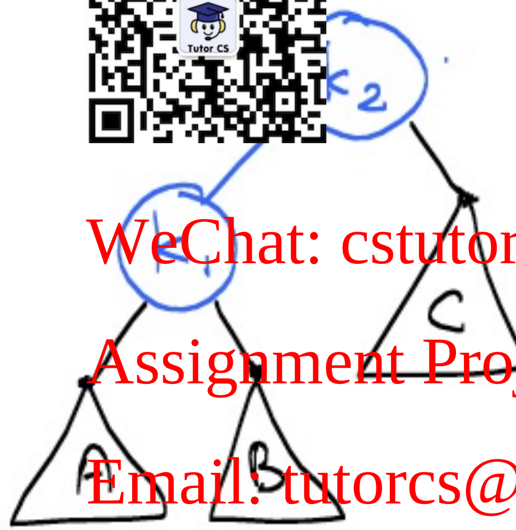
= : Balanced

→ : Right tree is higher (right-heavy)

Definition: Rotations

- Suppose we have.

Taken from Langer2014



- k_1 k_2 are keys
- A, B, C are sub trees

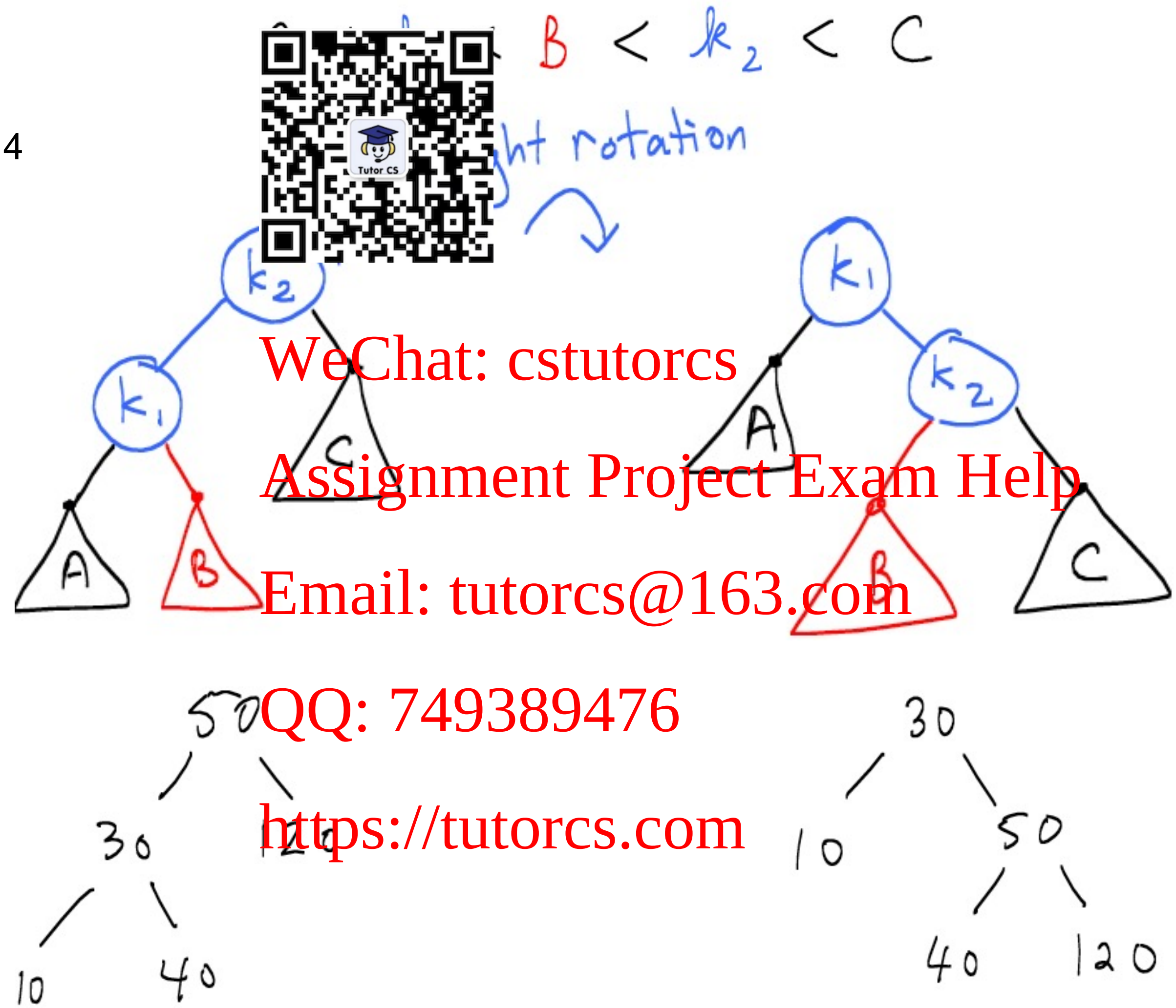
(of unspecified shape)

$$A < k_1 < B < k_2 < C$$

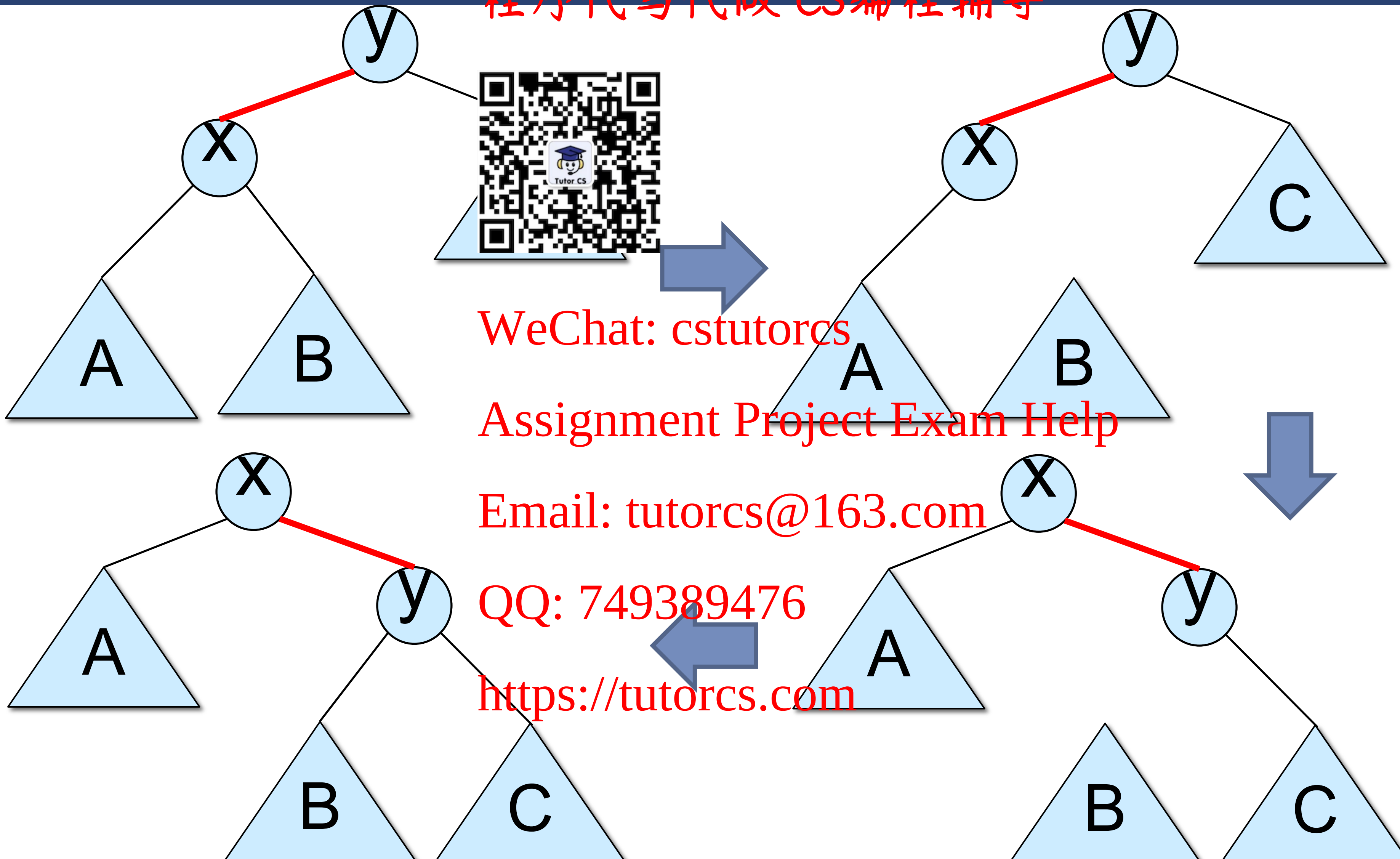
- All keys in A are less than key k_1. k_1 is less than all keys in B, which are less than k_2. k_2 is less than all keys in C

Definition: Right Rotation

Taken from Langer2014

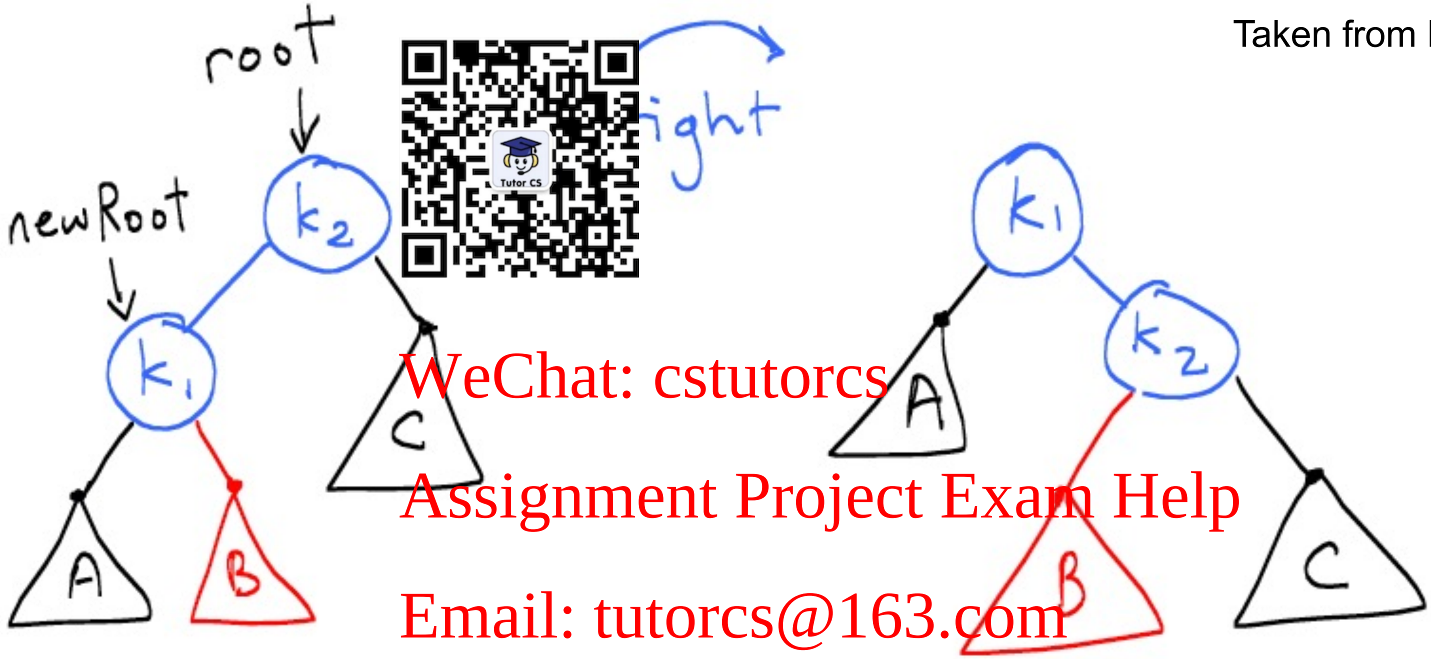


Definition: Right Rotation



Definition: Right Rotation

Taken from Langer2014

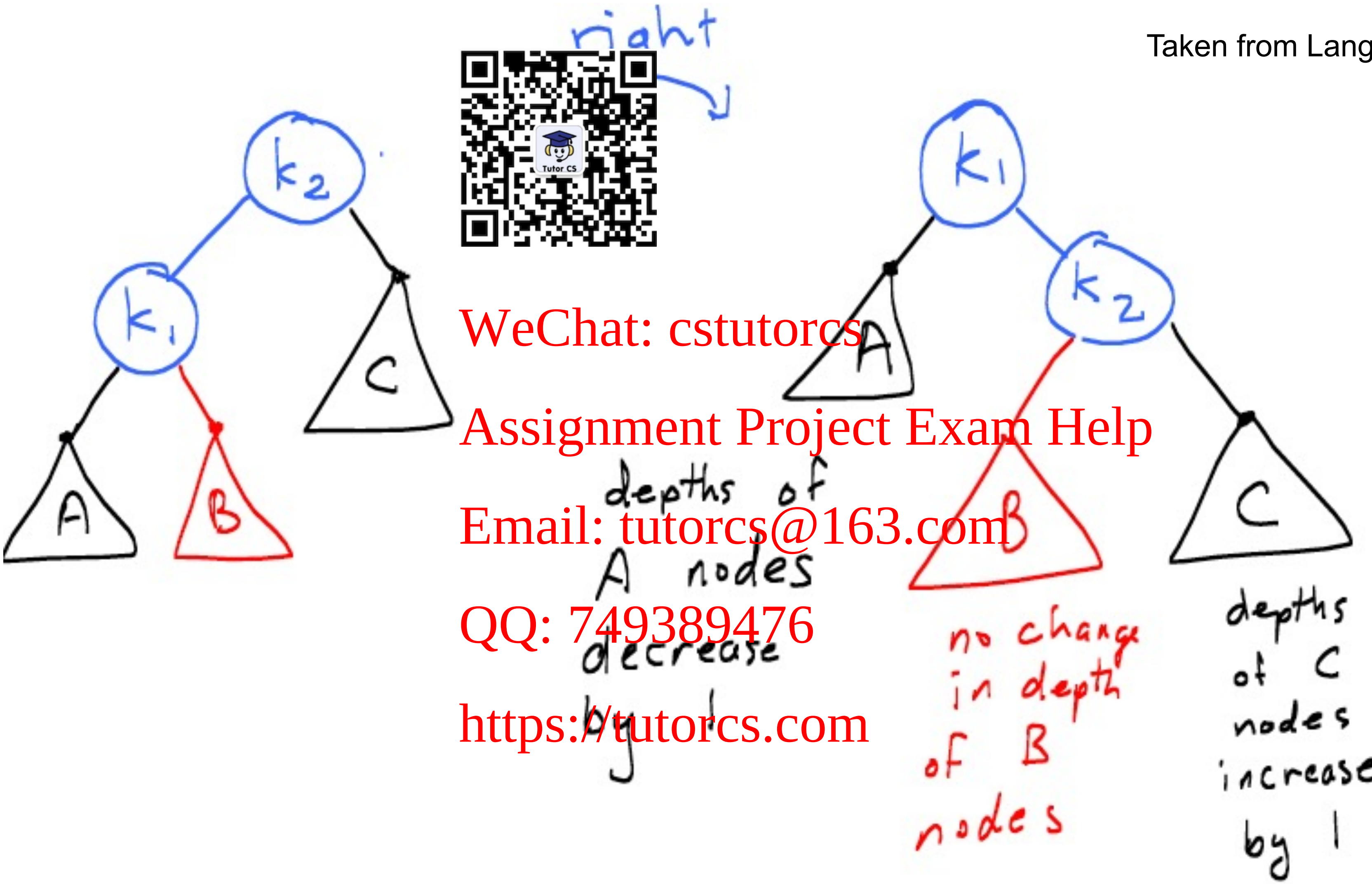



```
rotateRight(root){
    newRoot = root.left
    root.left = newRoot.right
    newRoot.right = root
    return newRoot
}
```

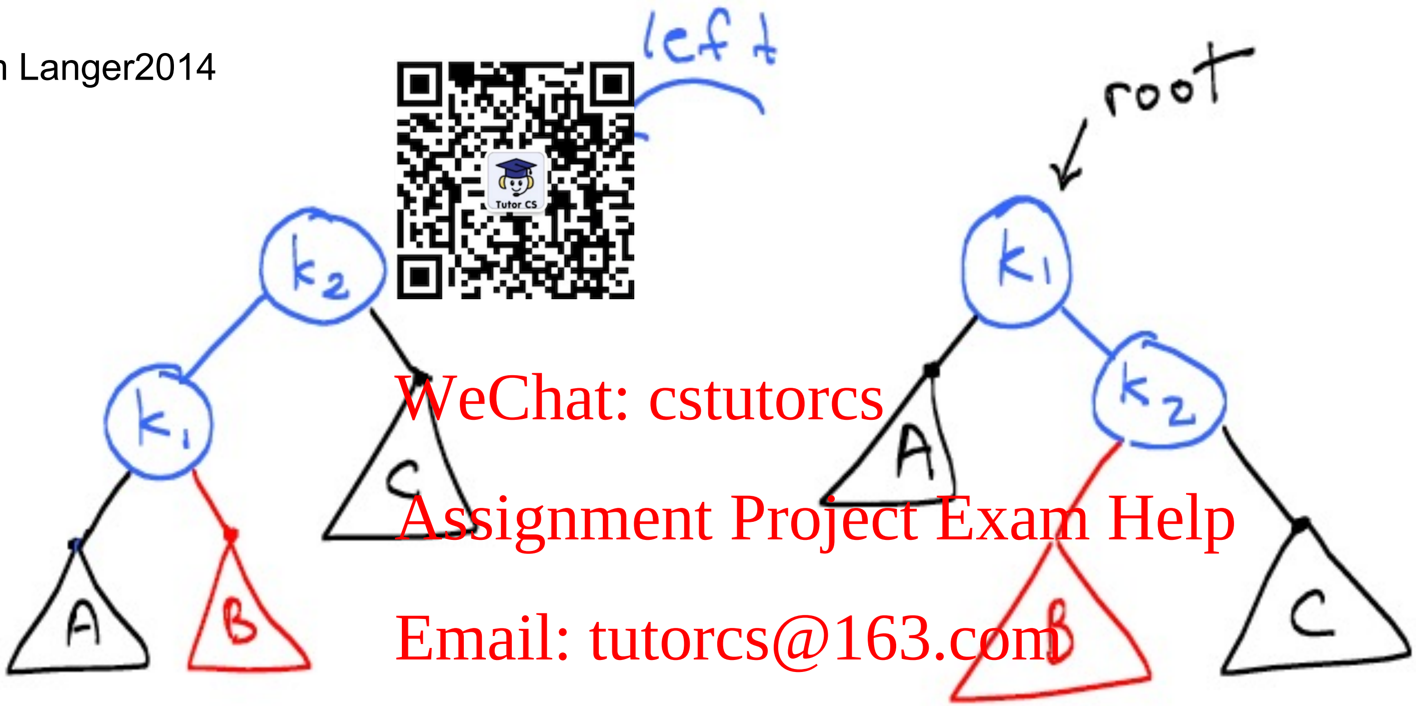
argument and returned value are nodes, not keys

Definition: Right Rotation

Taken from Langer2014



Taken from Langer2014

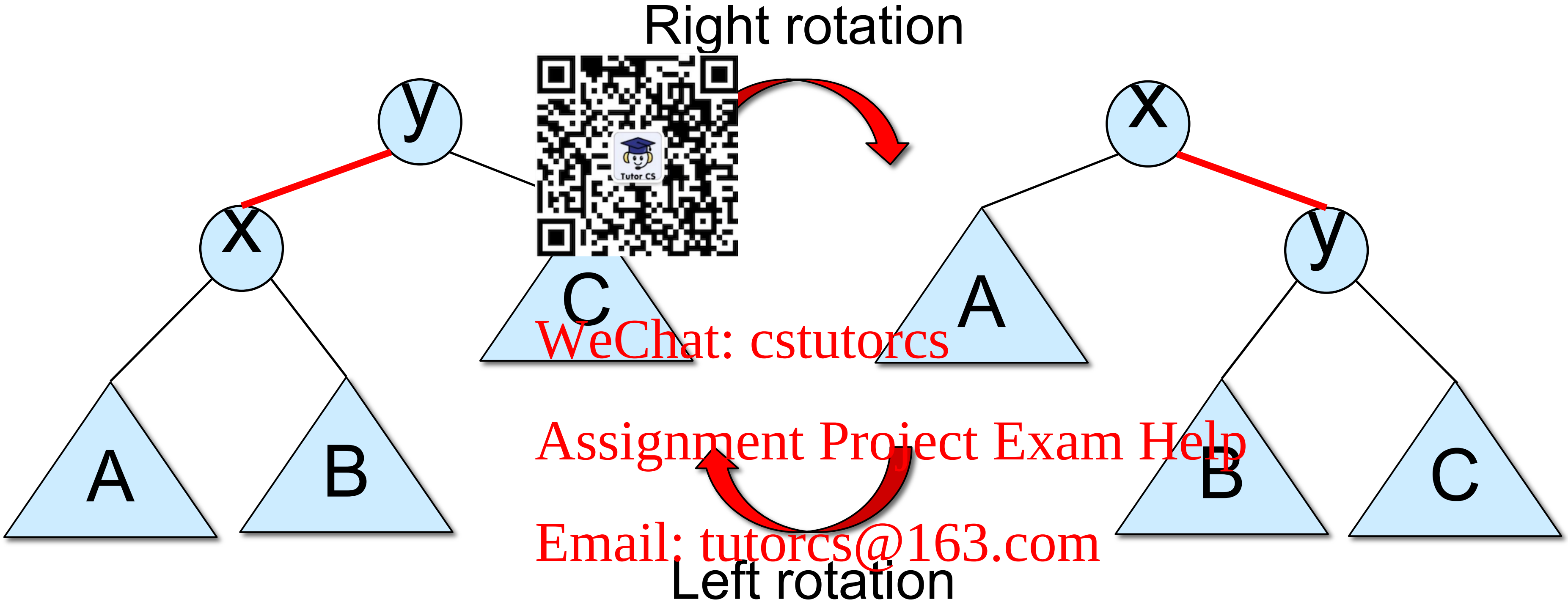


Exercise =>

```
rotateLeft( root ){
    :
    :
}
```

no change in depth of B nodes

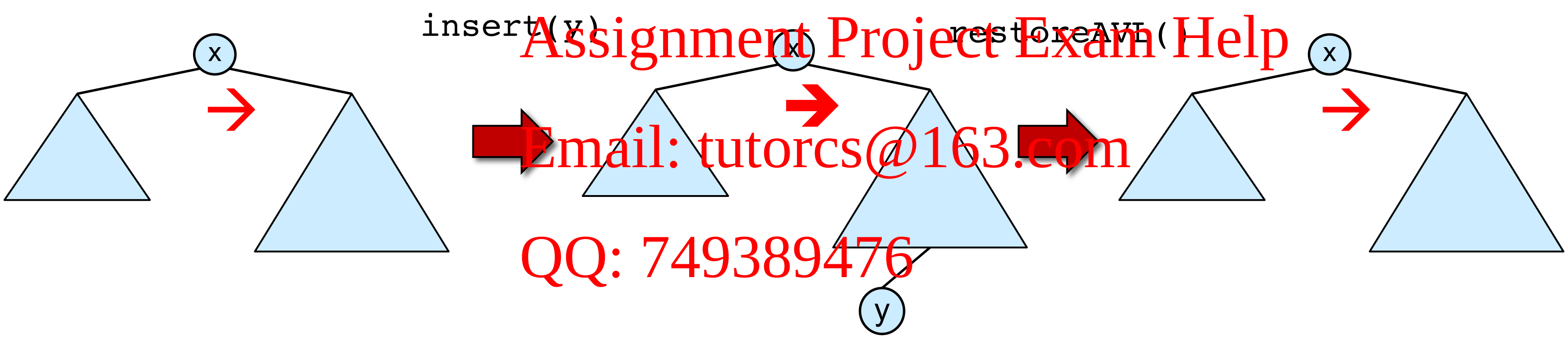
Definition: Rotations



Rotations change the tree structure & **preserve the BST property**.

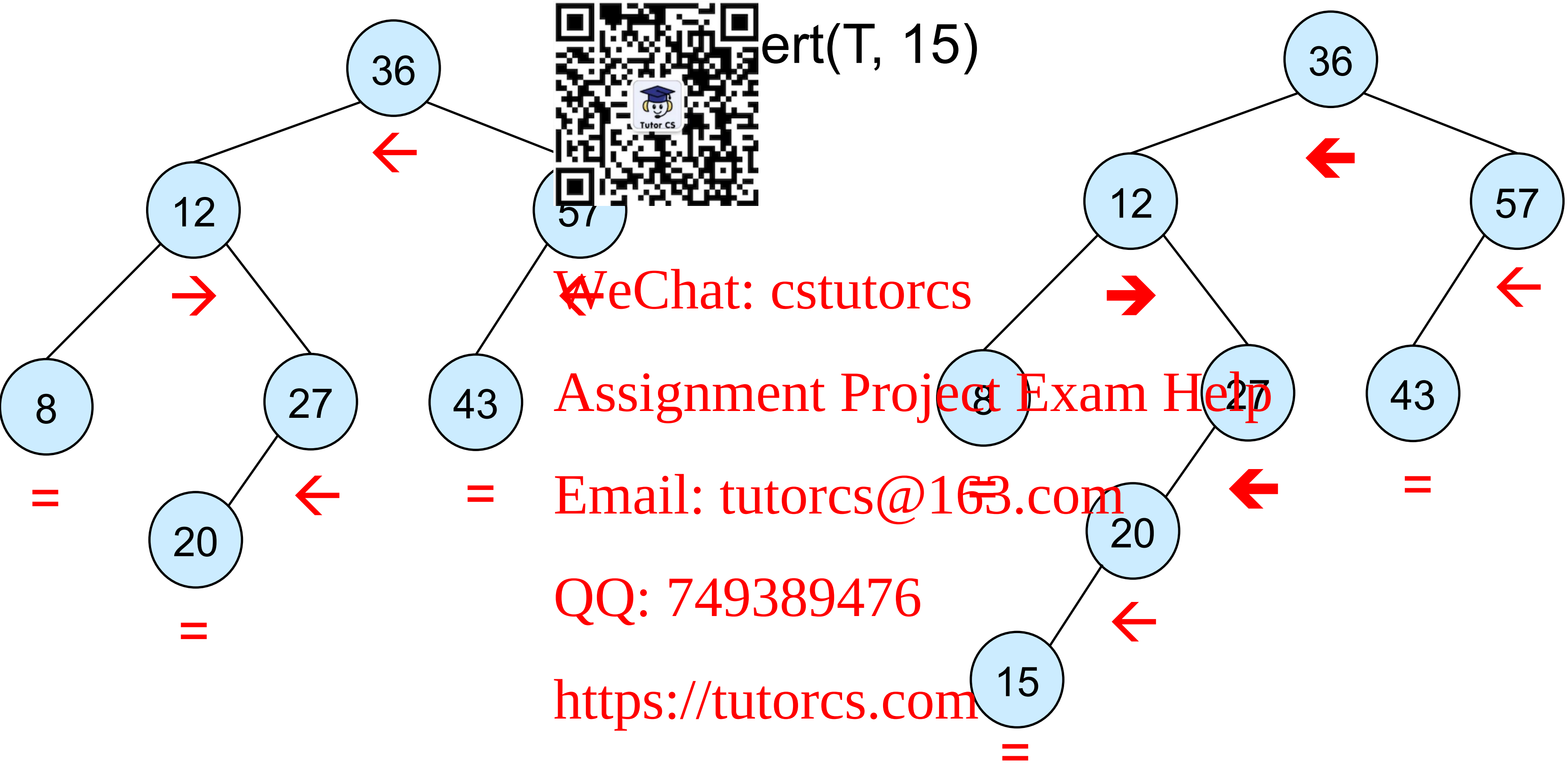
Operations: AVL insertion

1. Insert standard BST
2. Resto L tree properties



Operations: AVL insertion - Example

ert(T, 15)



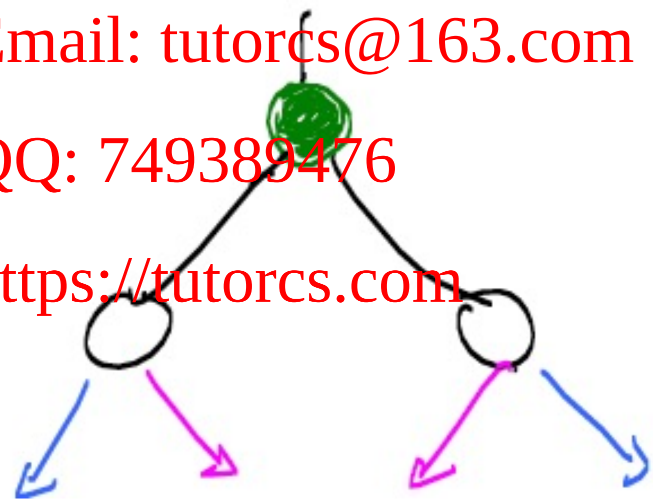
How to restore AVL property?

Operations: AVL insertion

There are four ways (two pairs of ways) that the imbalance could have occured, ... he insertion was:

- to the left subtree ... child (outside)
- to the right subtree of the right child (outside)

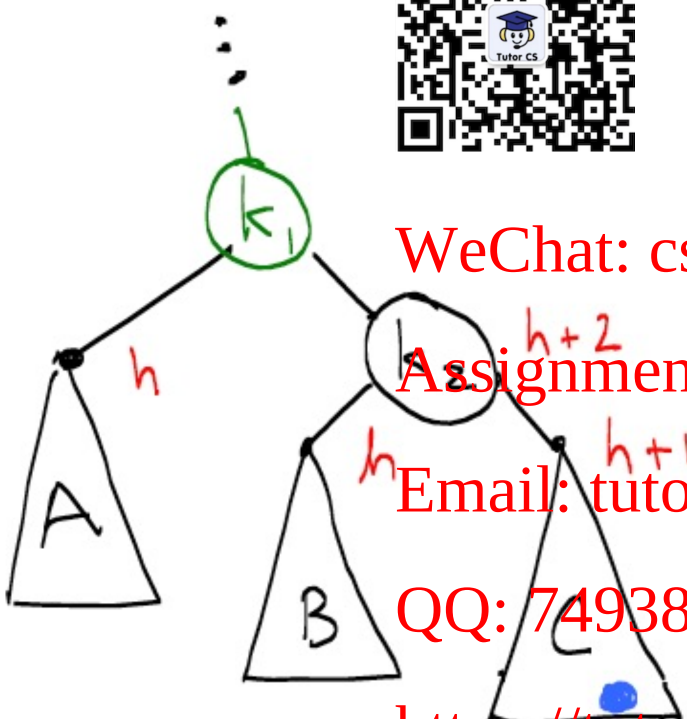
- to the right subtree of the left child (inside).
- to the left subtree of the right child (inside)



Taken from Langer2014

Operations: AVL insertion

"outside"

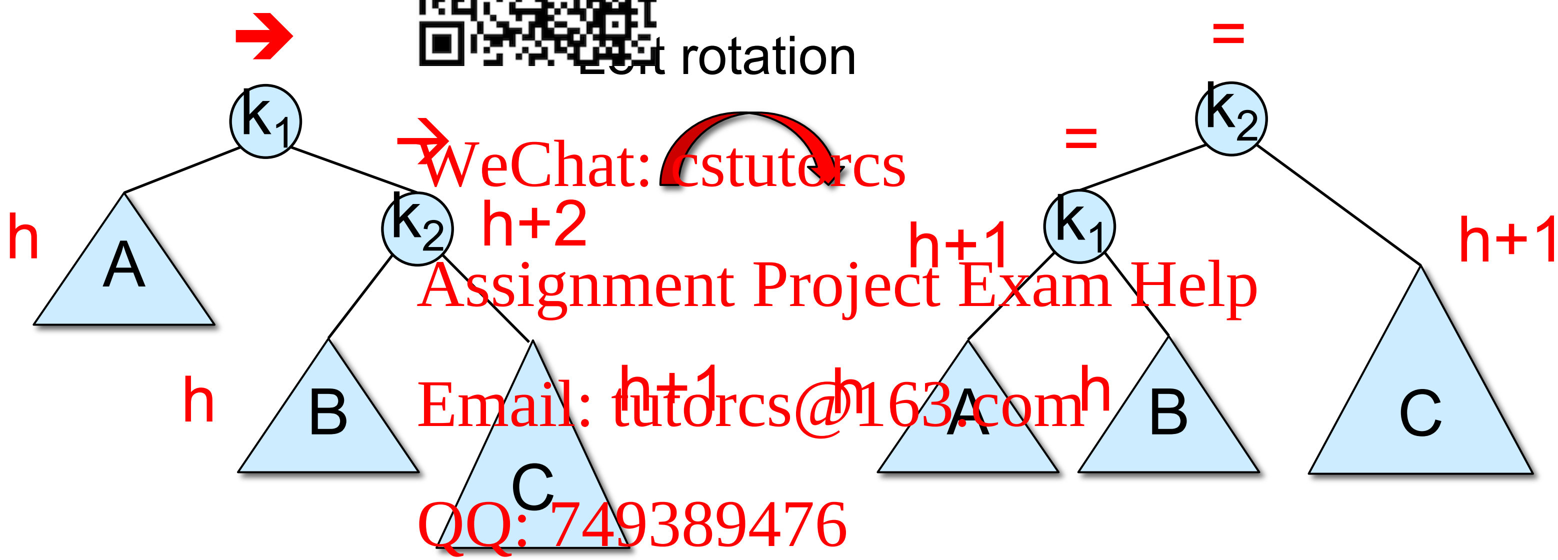


e.g. Insertion was in tree C and extended C's height from h to h+1, creating imbalance at subtree rooted at k_1 (but not k_2).

Question: How to rebalance k_1?

Operations: AVL insertion

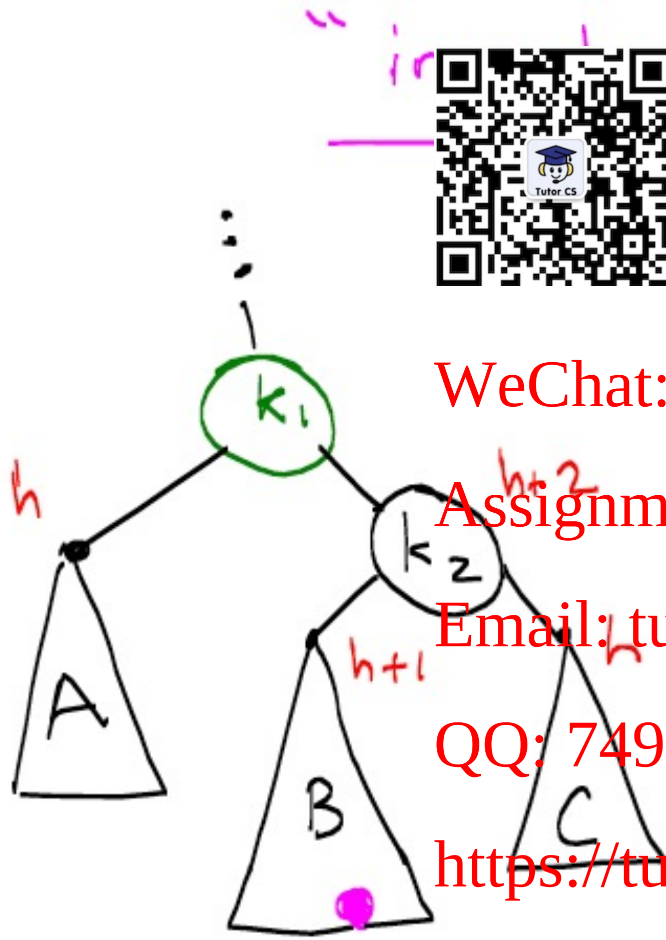
Answer: rotate Left(k_1)



before

After (balanced)

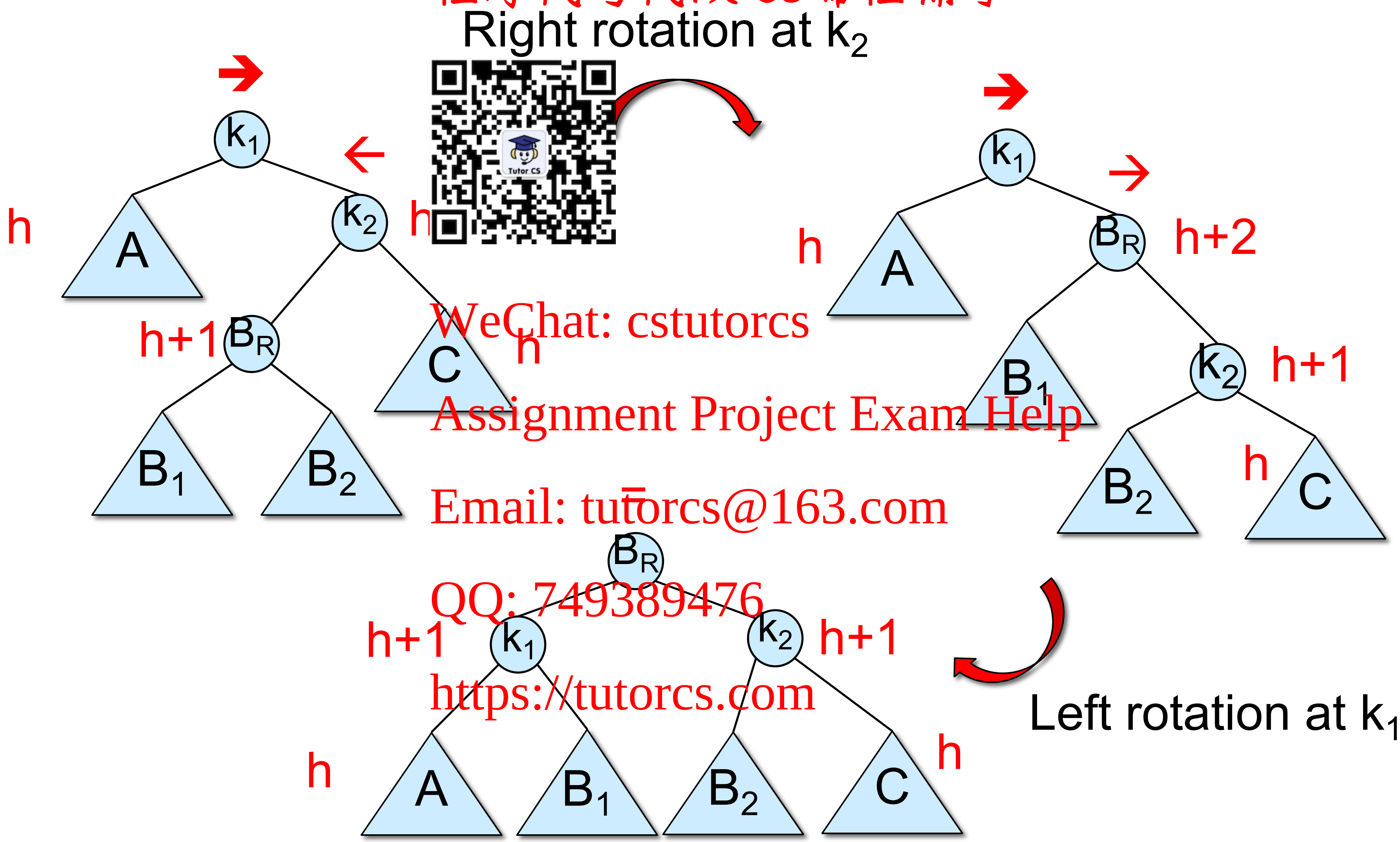
Operations: AVL insertion



e.g. Insertion was into tree B, extending its height from h to h+1, and creating imbalance at subtree rooted at k_1

How to rebalance k_1?

Operations: AVL insertion

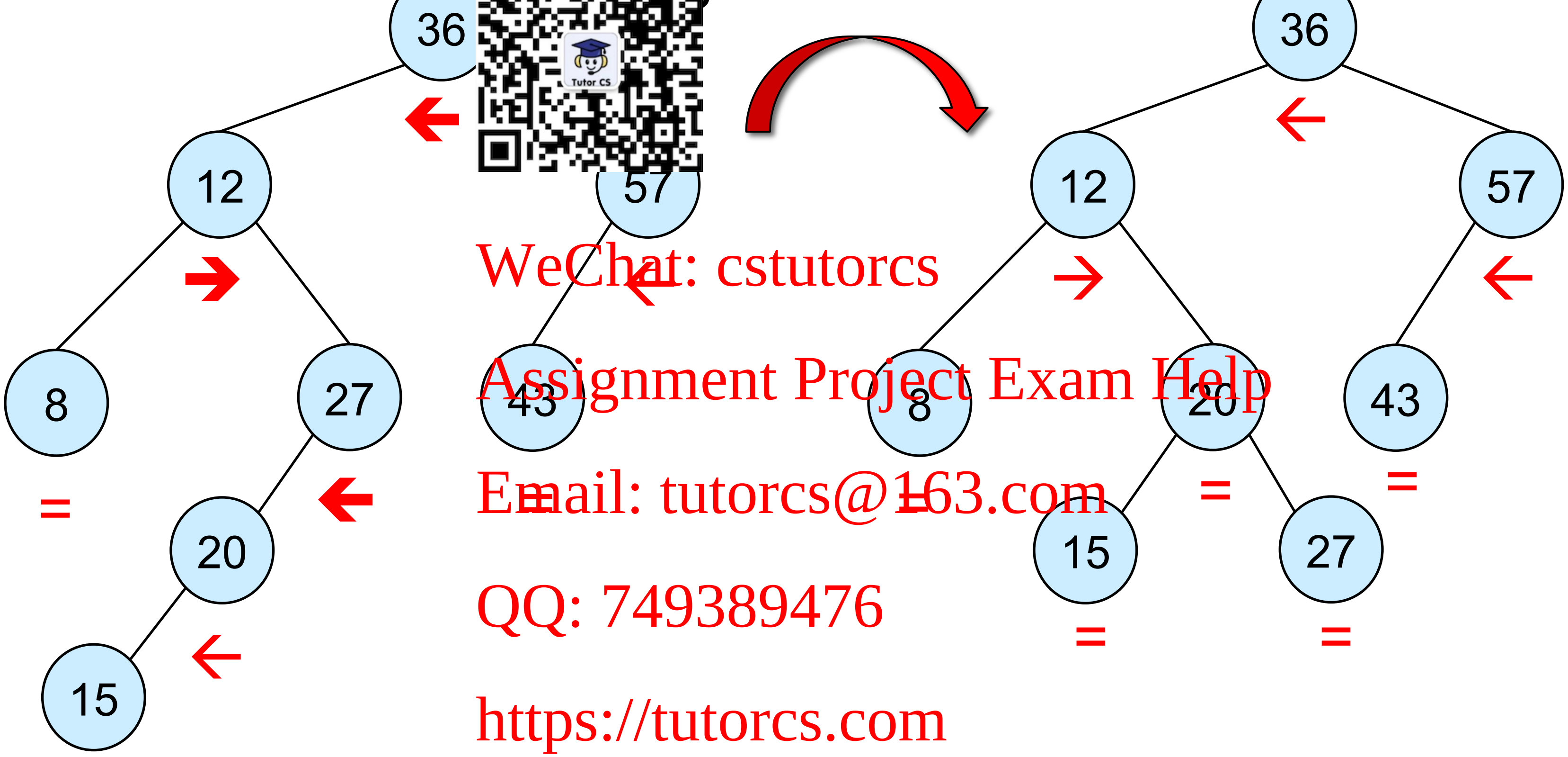


Operations: AVL insertion

1. Suppose x is lowest node violating AVL
2. If x is right-heavy:
   - If x's right child is right-heavy or balanced: Left rotation (case outside)
   - Else: Right followed by left rotation (case inside)
3. If x is left-heavy:
   - If x's left child is left-heavy or balanced: Right rotation (sym. of case outside)
   - Else: Left followed by right rotation (sym. of case inside)

Operations: AVL insertion - Example

ght rotation at 27

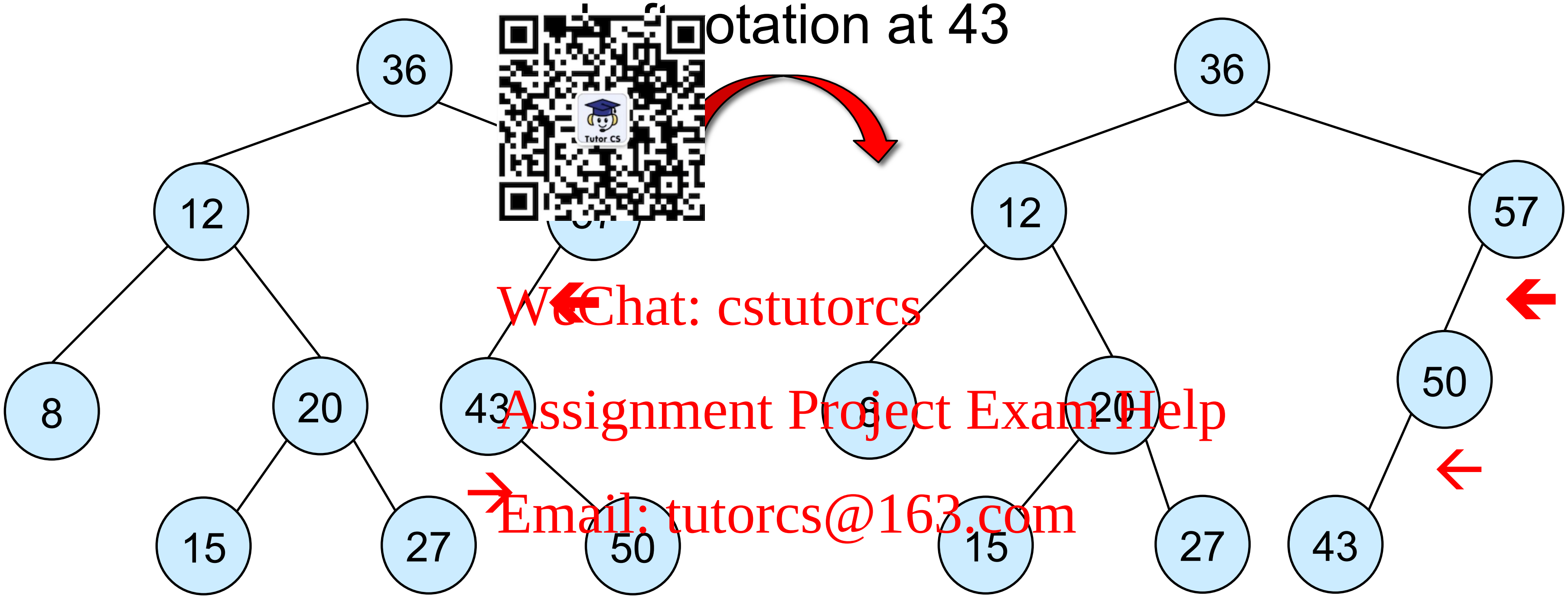


Insert(T, 15)

How to restore AVL property?

Operations: AVL insertion - Example

...otation at 43

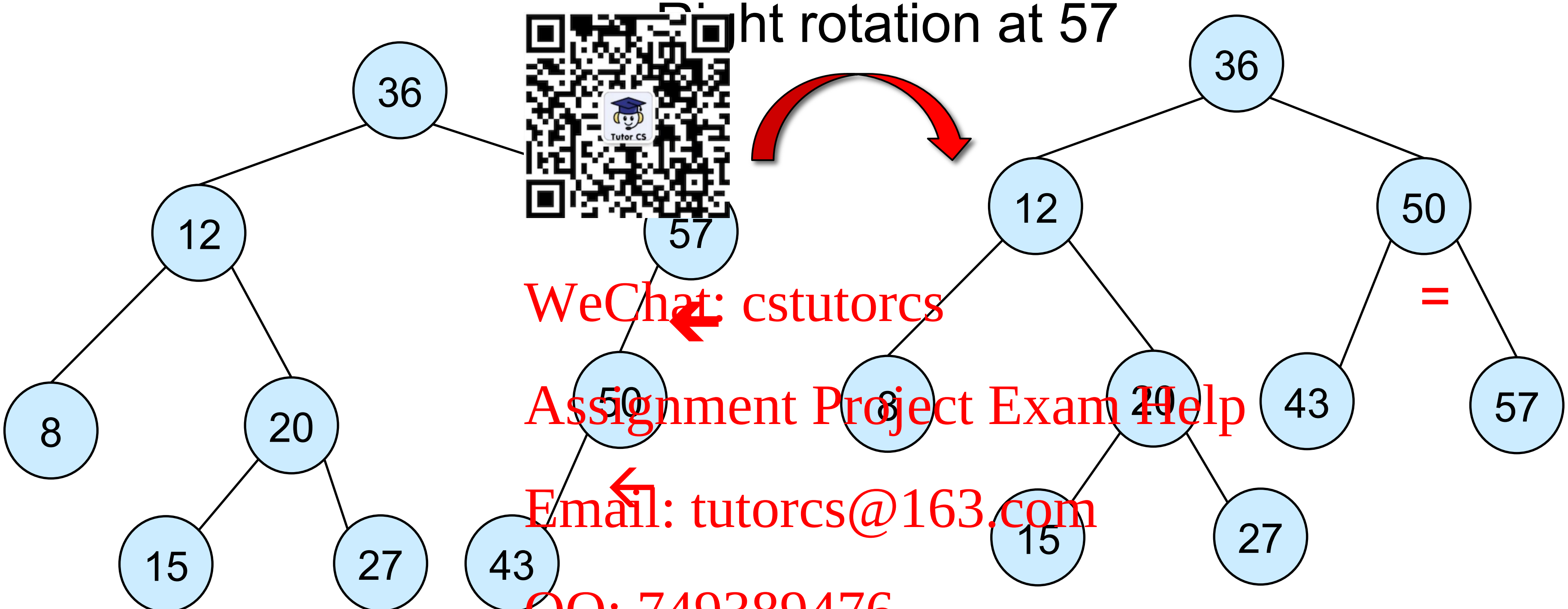


We remove the zig-zag pattern

Insert(T, 50)
RotateLeft(T,43)

Operations: AVL insertion - Example

Right rotation at 57



AVL property restored!

RotateRight(T,57)

AVL insertion: running time

- Insertion in O(h)
- At most 2 rotation [illegible]ns which take O(1)
- Running time is O(h) + O(1) = O(h) = O(log n) in AVL trees.

AVL sort: running time

Same as BST sort [illegible] AVL trees and AVL insertion instead.

- Worst case running time can be brought to O(n log n) if the tree is always balanced.
- Use AVL trees (trees are balanced).
- Insertion in AVL trees are O(h) = O(log n) for balanced trees.


time for questions