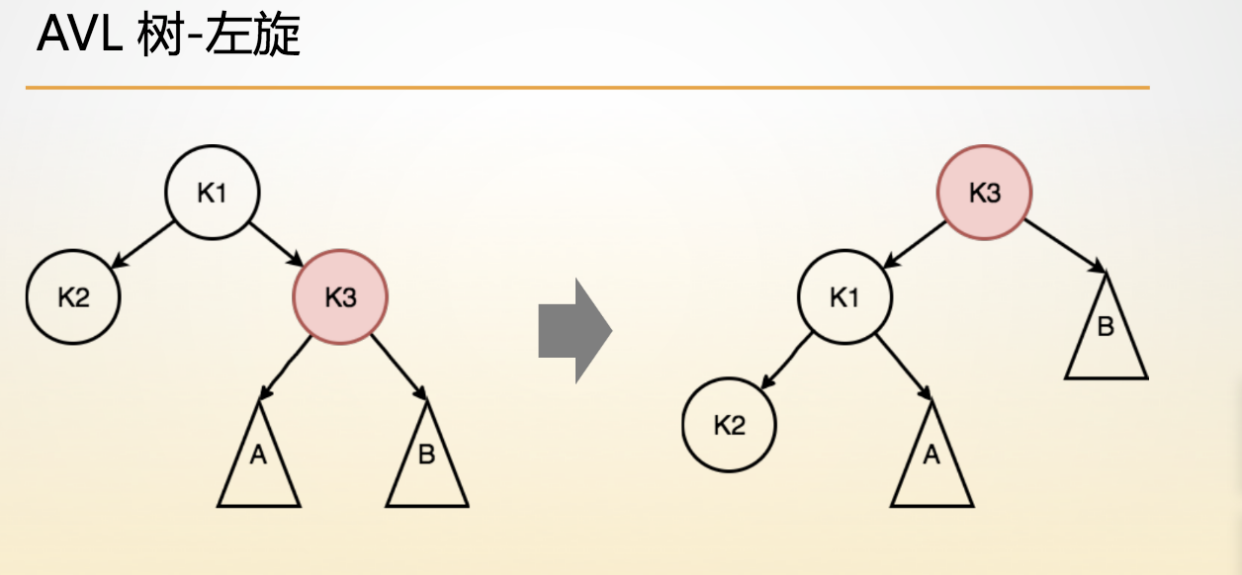
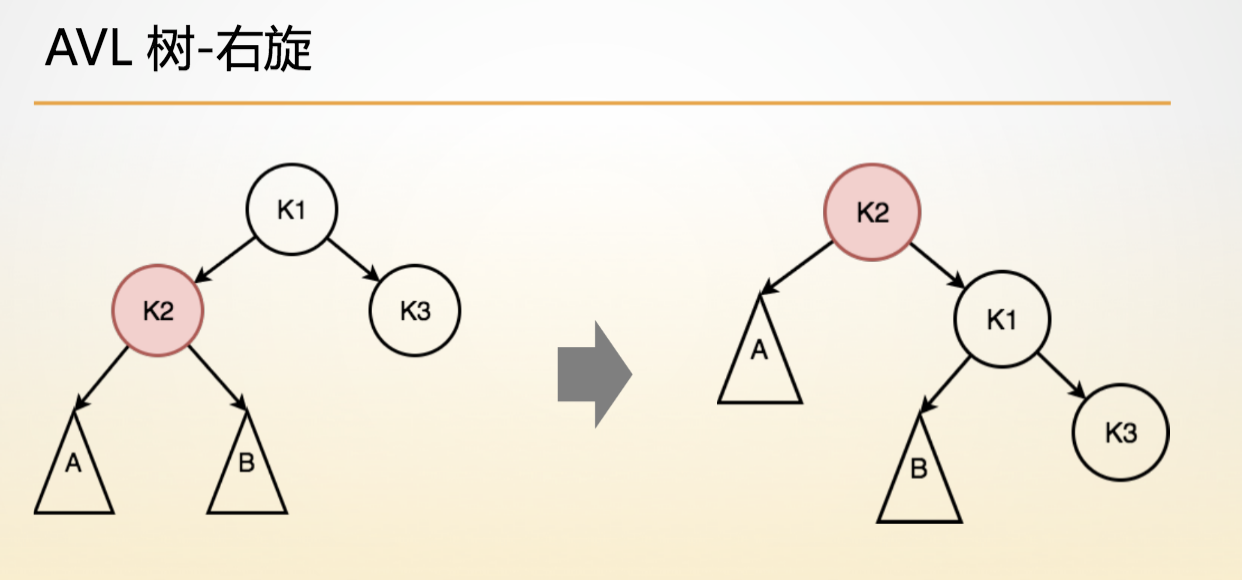
AVL树

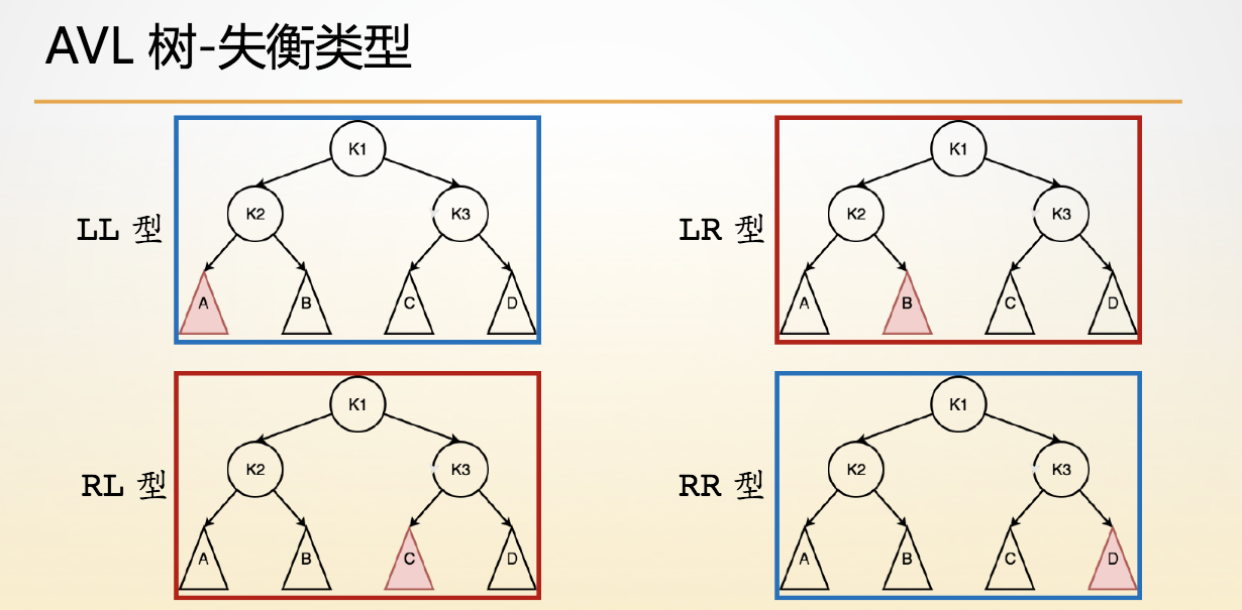
发明者： AV 、L
性质： |H（left）- H（right）| <= 1
特点： 不会退化成链表
高度为H
BST H <= 节点数量 <= 2^H - 1
AVT F(H - 1) + F(H - 2) <= 节点数量<= 2^H - 1
左旋： 把当前节点的左子树接到父节点的右子树 当前节点作为新的父节点



右旋： 把当前节点的右子树接到父节点的左子树 当前节点作为新的父节点

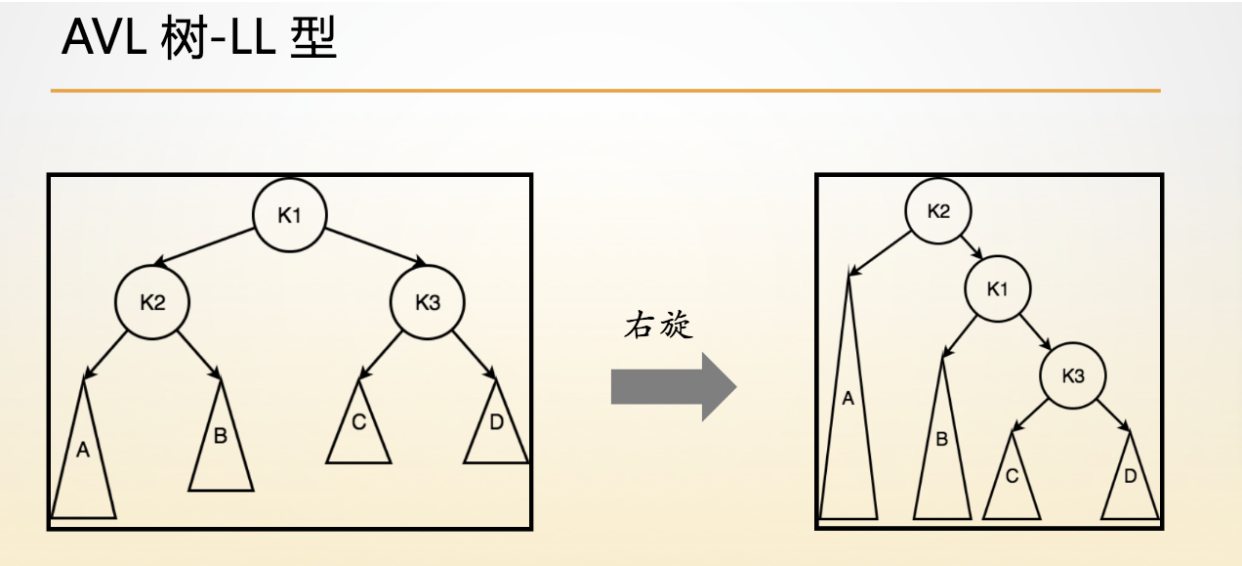


失衡类型： LL LR RR RL
LL : K1是第一个失衡的节点 K1左子树深度比右子树高2，并且K1左子树的左子树高于右子树
LR : K1是第一个失衡的节点 K1左子树深度比右子树高2，并且K1左子树的左子树低于右子树
RR RL 同上



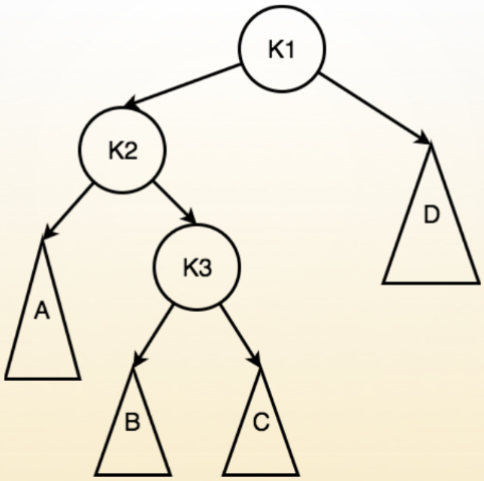
调整策略：

ll型： 直接右旋
rr型： 直接左旋

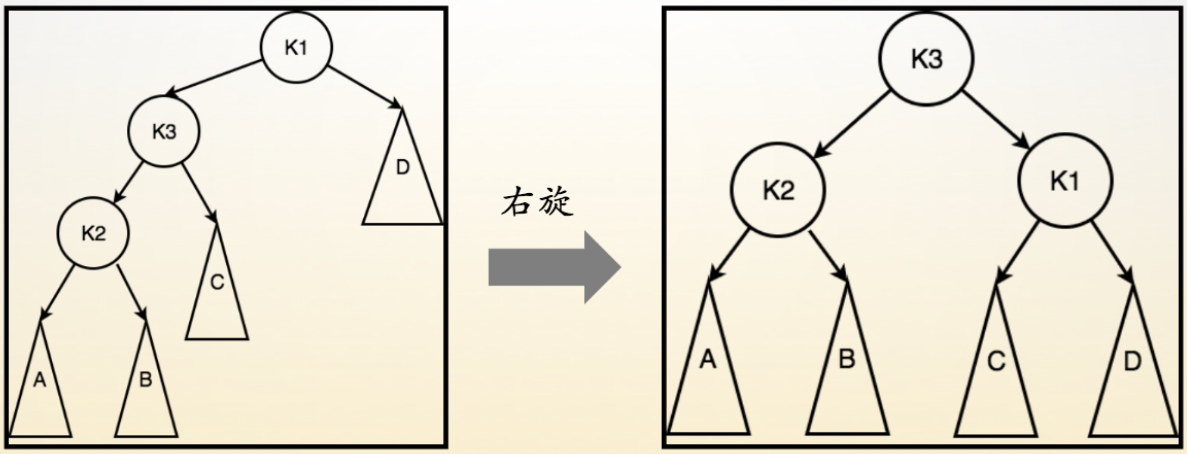


lr型： 先小左旋 在右旋
rl型： 先小右旋 在左旋

AVL 树-LR 型



AVL 树-LR 型-先左旋



代码演示

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// #define MAX(a, b) ({   \ // __typeof() 自动推倒参数的类型
//  __typeof(a) _a = (a); \
//  __typeof(b) _b = (b); \
//  _a > _b ? _a : _b;    \
// })

#define MAX(a, b) (a > b ? a : b)

typedef struct AVLTNode {
    int data, h;
    struct AVLTNode *lchild, *rchild;
} Node;
```

```
Node *NIL;

__attribute__((constructor)) // 程序启动直接运行
void init_nil() {  // 方便计算MAX值 不用做特判
    NIL = (Node *)malloc(sizeof(Node));
    NIL->data = 0;
    NIL->lchild = NIL->rchild = NIL;
    NIL->h = 0;
}

Node *init(int data) {
    Node *p = (Node *)malloc(sizeof(Node));
    p->lchild = p->rchild = NIL;
    p->h = 1;
    p->data = data;
    return p;
}

void clear(Node *root) {
    if (root == NIL) return ;
    clear(root->lchild);
    clear(root->rchild);
    free(root);
    return;
}

Node *left_rotate(Node *root) {
    Node *temp = root->rchild;   // 把root的右子树作为根节点
    root->rchild = temp->lchild; // 让root的右子树的左子树给root的右子树
    temp->lchild = root;         // root作为根节点的左子树
    root->h = MAX(root->lchild->h, root->rchild->h) + 1; // 更新树高
    temp->h = MAX(temp->lchild->h, temp->rchild->h) + 1;
    return temp;
}

Node *right_rotate(Node *root) {
    Node *temp = root->lchild;
    root->lchild = temp->rchild;
    temp->rchild = root;
    root->h = MAX(root->lchild->h, root->rchild->h) + 1; // 更新树高
    temp->h = MAX(temp->lchild->h, temp->rchild->h) + 1;
    return temp;
}

Node *maintain(Node *root) {
    if (abs(root->lchild->h - root->rchild->h) < 2) return root;
    if (root->lchild->h > root->rchild->h) {
        if (root->lchild->lchild->h < root->lchild->rchild->h) { // LR
            root->lchild = left_rotate(root->lchild); // 小左旋
        }
        root = right_rotate(root); // 大右旋
```

```
    } else {
        if (root->rchild->lchild->h > root->rchild->rchild->h) { // RL
            root->rchild = right_rotate(root->rchild); // 小右旋
        }
        root = left_rotate(root); // 大左旋
    }
    return root;
}

Node *insert(Node *root, int data) {
    if (root == NIL) return init(data);
    if (root->data == data) return root;
    if (root->data < data) root->rchild = insert(root->rchild, data);
    else root->lchild = insert(root->lchild, data);
    root->h = MAX(root->lchild->h, root->rchild->h) + 1;
    // 当rchild为空 访问NIL节点 不影响计算
    root = maintain(root); // 调整
    return root;
}

void output(Node *root) {
    if (root == NIL) return ;
    output(root->lchild);
    printf("%d ", root->data);
    output(root->rchild);
    return ;
}

// void output(Node *root) {
//  if (root == NIL) return;
//  printf("(%d %d %d)\n", root->data, root->lchild->data, root->rchild->d
ata);
//  output(root->lchild);
//  output(root->rchild);
//  return ;
// }

int main() {
    srand(time(0));
    Node *root = NIL;
    #define OP_NUM 20

    for (int i = 0; i < OP_NUM; i++) {
        int data = rand() % 30;
        root = insert(root, data);
        printf("insert %d to tree\n", data);
        output(root), printf("\n");
    }
    clear(root);

    #undef OP_NUM
    return 0;
```

```
    }
```