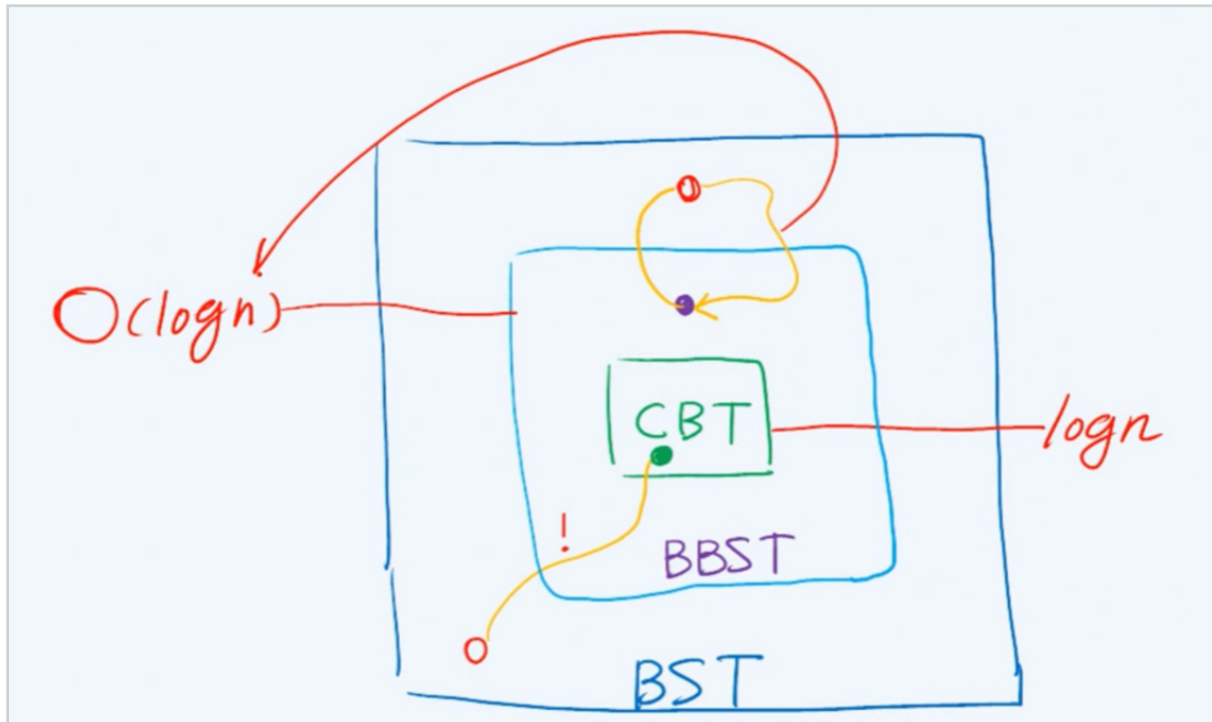


08D1&2 AVL树

#数据结构邓神

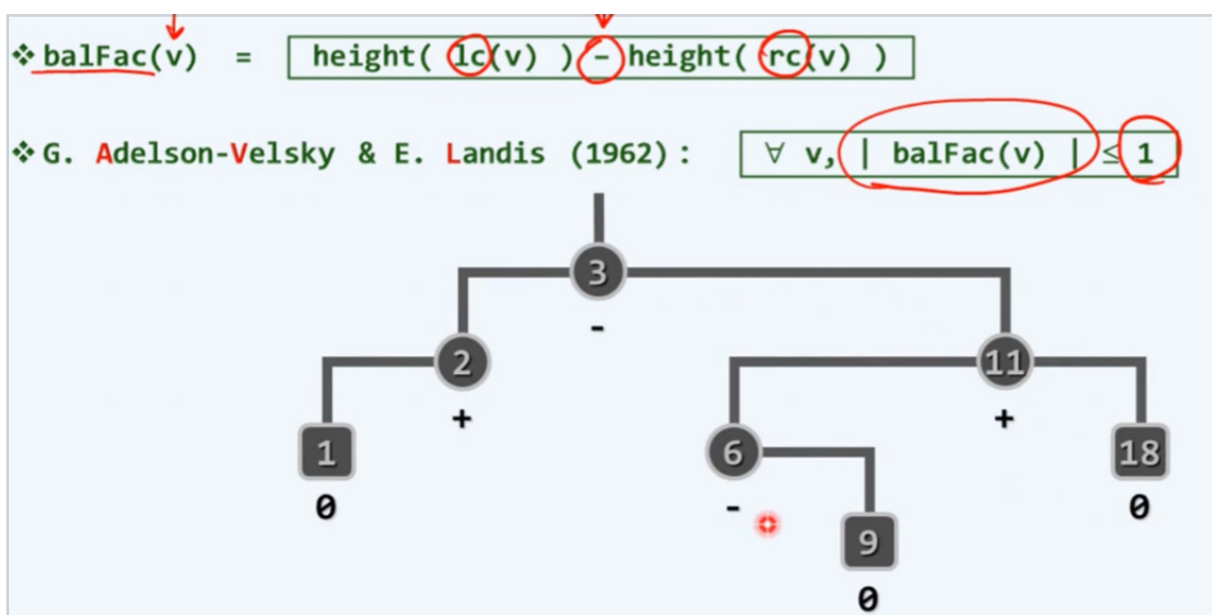
AVL = BBST



BBST:

1. 如何界定是否为BBST
2. 如何重平衡 Rebalance?

平衡因子



其中所有节点的平衡因子都不超过1和小于-1

适度平衡



$$\text{height}(\text{AVL}) = O(\log n)$$



$$n = \Omega(2^{\text{height}(\text{AVL})})$$

AVL = 适度平衡

❖ 高度为 h 的 AVL 树，至少包含 $S(h) = \text{fib}(h+3) - 1$ 个节点

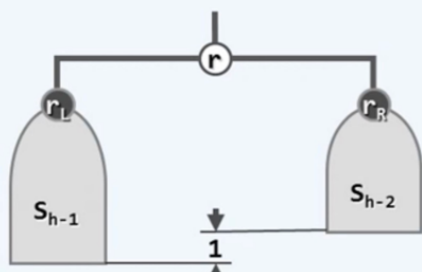
❖ $S(h) = 1 + S(h-1) + S(h-2)$ \Leftarrow

❖ $S(h) + 1 = [S(h-1) + 1] + [S(h-2) + 1]$

$$T(h) = T(h-1) + T(h-2) = \text{fib}(h+?)$$

$$n = \Omega(\Phi^h)$$

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



h	n	$T(h)$
0	1	$2 = \text{fib}(3)$
1	2	$3 = \text{fib}(4)$

接口

AVL : 接口

```
❖ #define Balanced(x) \ //理想平衡 ✓
    ( stature( (x).lChild ) == stature( (x).rChild ) )

#define BalFac(x) \ //平衡因子 ✓
    ( stature( (x).lChild ) - stature( (x).rChild ) )

#define AvlBalanced(x) \ //AVL平衡条件 ✓
    ( ( -2 < BalFac(x) ) && ( BalFac(x) < 2 ) )

❖ template <typename T> class AVL : public BST<T> { //由BST派生
public: // BST::search()等接口, 可直接沿用 ✓

    BinNodePosi(T) insert( const T & ); //插入重写

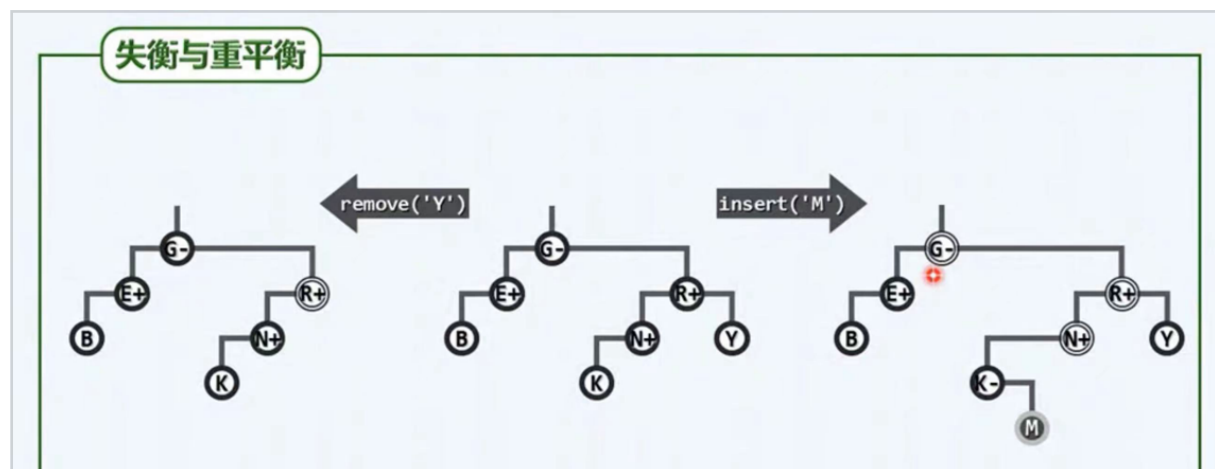
    bool remove( const T & ); //删除重写

};
```

Data Structures & Algorithms (Fall)

```
template <typename T> class AVL:public BST<T> {
public:
    BinNodePosi<T> insert(const T &); // 插入重新实现
    bool remove(const T &); // 删除重新实现
}
```

失衡 + 复衡



中间就是一个BBST

1. 插入一个节点后可能会导致很多祖先失衡, 但是除了祖先以外, 另外的节点是不可能失衡。
2. 删除节点之后的瞬间, 至多只有一个节点会失衡。

是否可以说：AVL树删除节点要比插入操作更为简单呢？

实际情况恰恰相反，如果我们将插入操作和删除操作比喻为孩子，插入操作可能会闯下下一连串的祸，但是往往只要改正其中的一个错误，其他的错误都会烟消云散

删除操作是一个不吸取教训的孩子，虽然他每次都只会闯下一个祸，每当你修复他时，他会闯下另外一个祸....