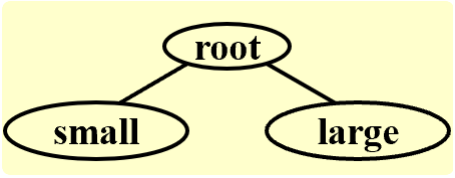
#ADS01 AVL Trees,Splay Trees and Amortized Analysis

1.1 AVL Trees

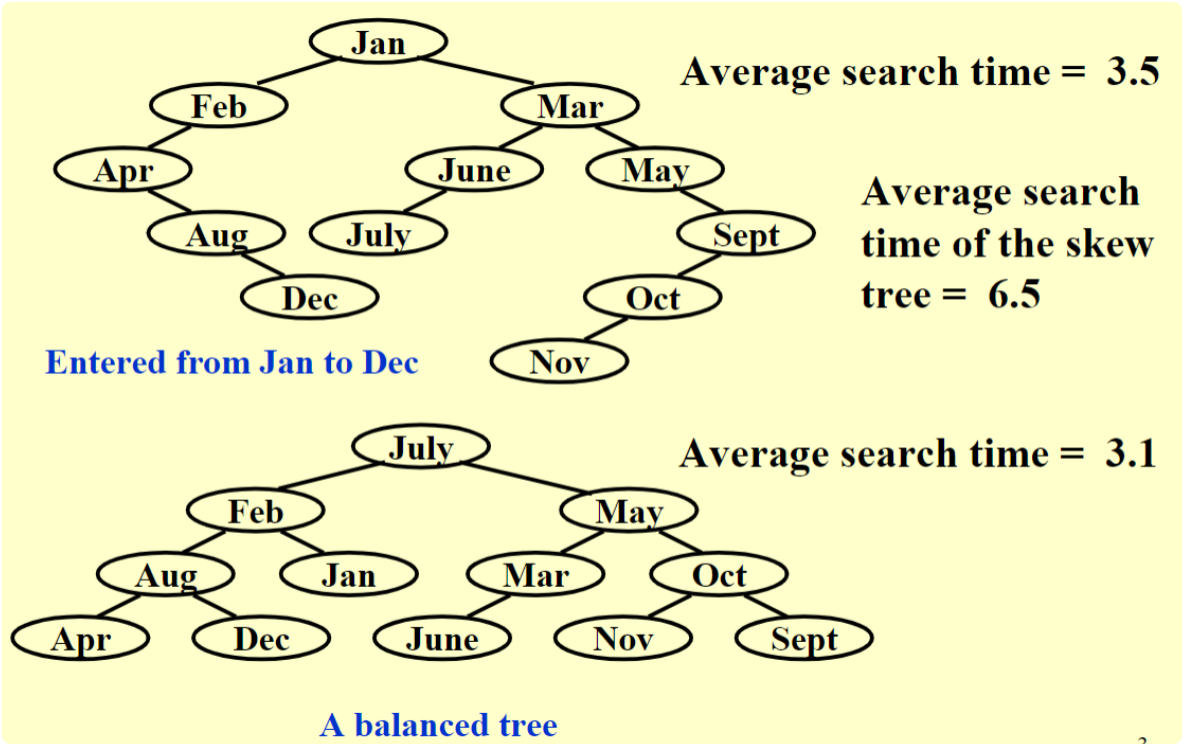
1.1.1 Definition

Target: Speed up searching (with insertion and deletion). O(ln n)

Tool: Binary Search Trees



Problem: Although T = O(height),but the height can be as bad as N



AVL Trees

Definition: An empty binary tree is height balanced. If **T** is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is **height balanced** iff

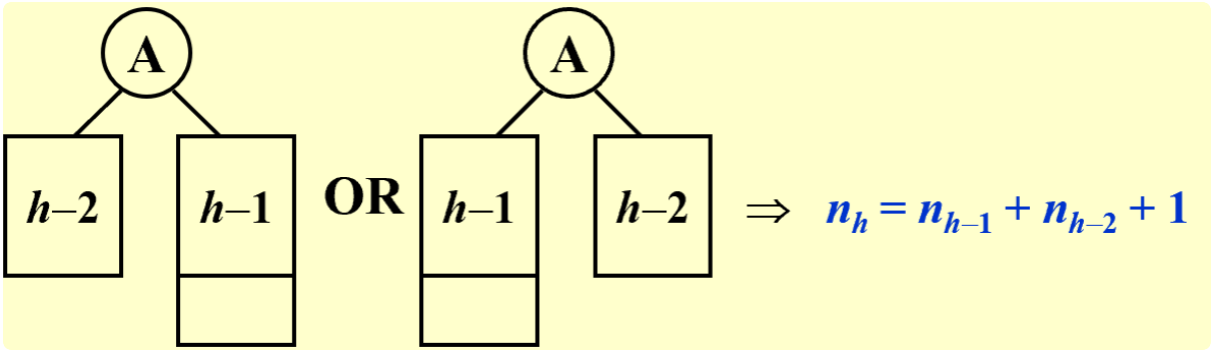
- T_L and T_R are height balanced, and
- $|h_L - h_R| <= 1$, where H_L and H_R are the heights of T_L and T_R , respectively.

The height of an empty tree is defined as `-1` .

Definition: The **balance factor**（平衡因子） **BF(node) = $h_L - h_R$** . In an AVL tree, BF(node) = –1, 0, or 1.

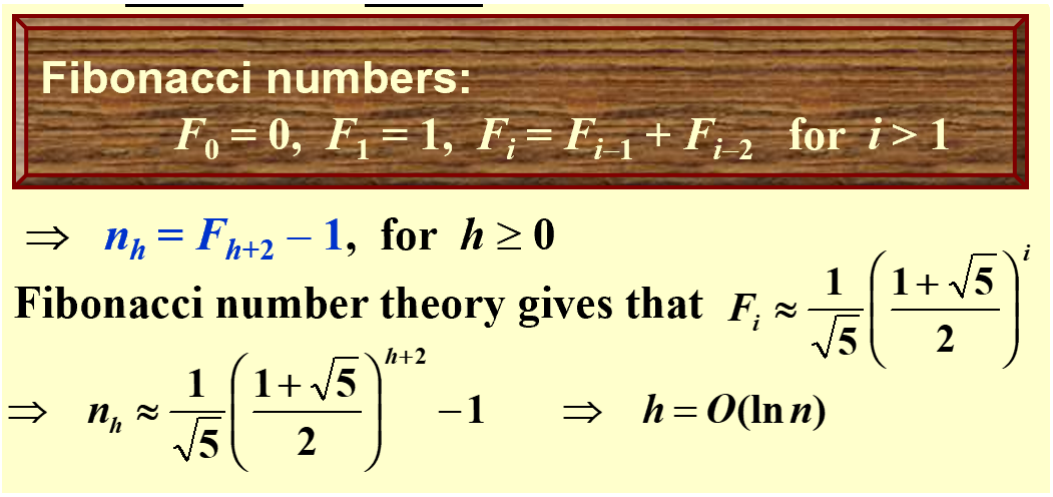
Obviously we have T_p = (height), but height = ? 显然搜索时间与高度相关，高度?

Let n_h be the **minimum number** of nodes in a height balanced tree of height h. Then the tree must look like:



Hence, we have $n_h = n_{h-1} + n_{h-2} + 1$

Fibonacci number theory gives that



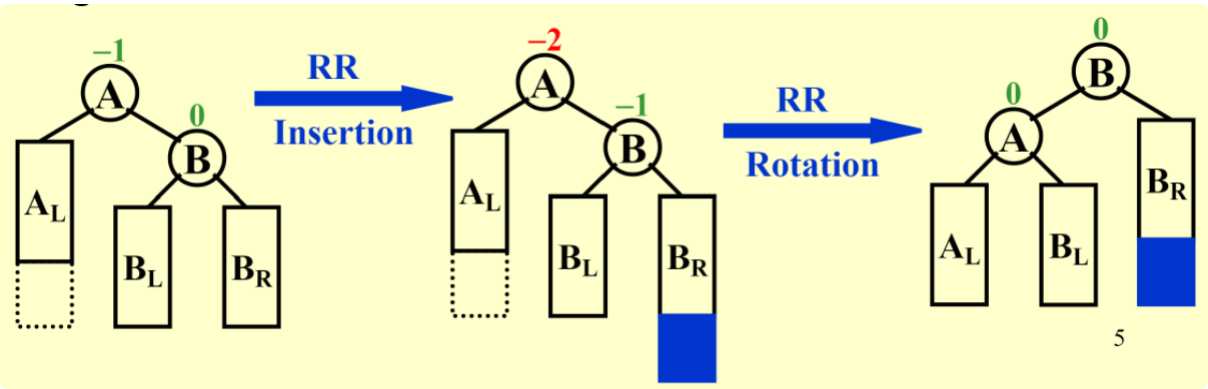
n为当高度为h时，树有的最少节点。可得到 `h = O(ln N)`

1.1.2 Methods to Build AVL Trees

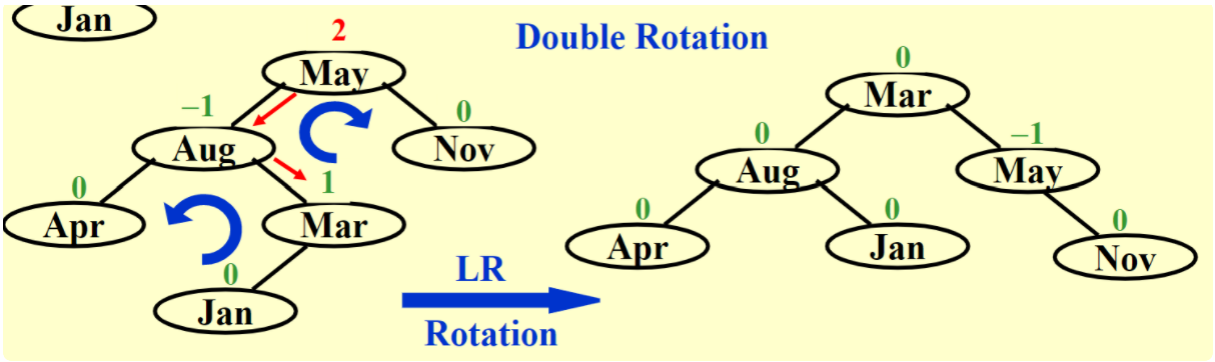
- trouble maker
- trouble finder

RR rotation

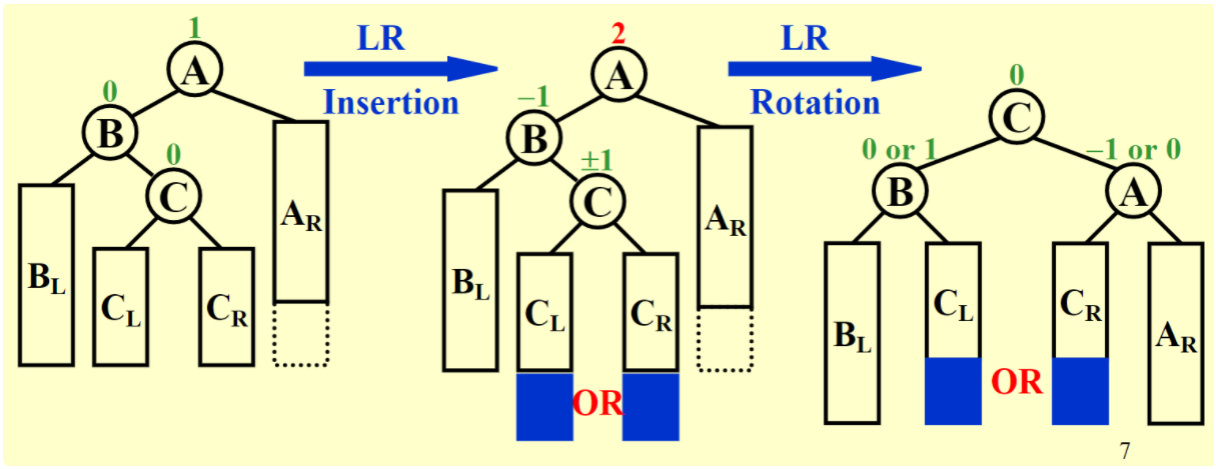
The `trouble maker` is in the right subtree's right subtree of the `trouble finder` . Hence it is called an RR rotation.



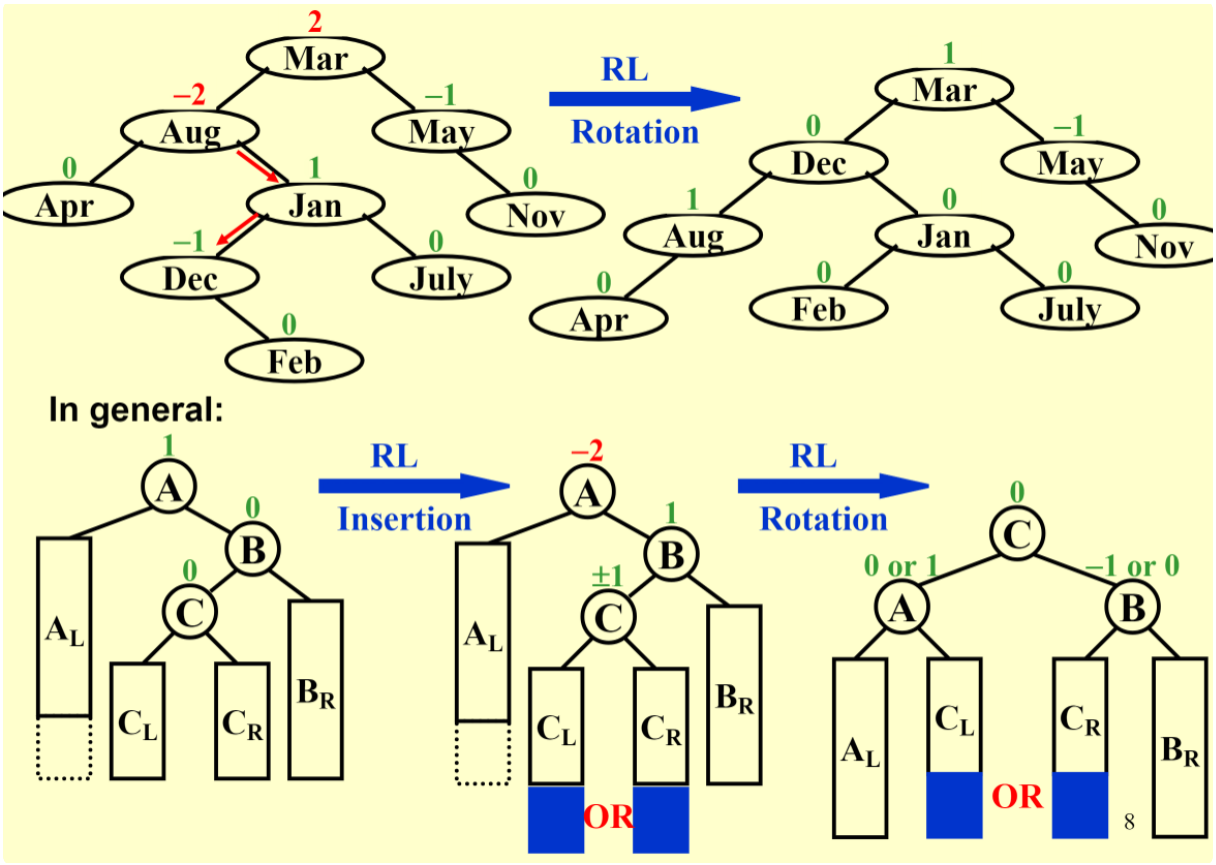
LR rotation

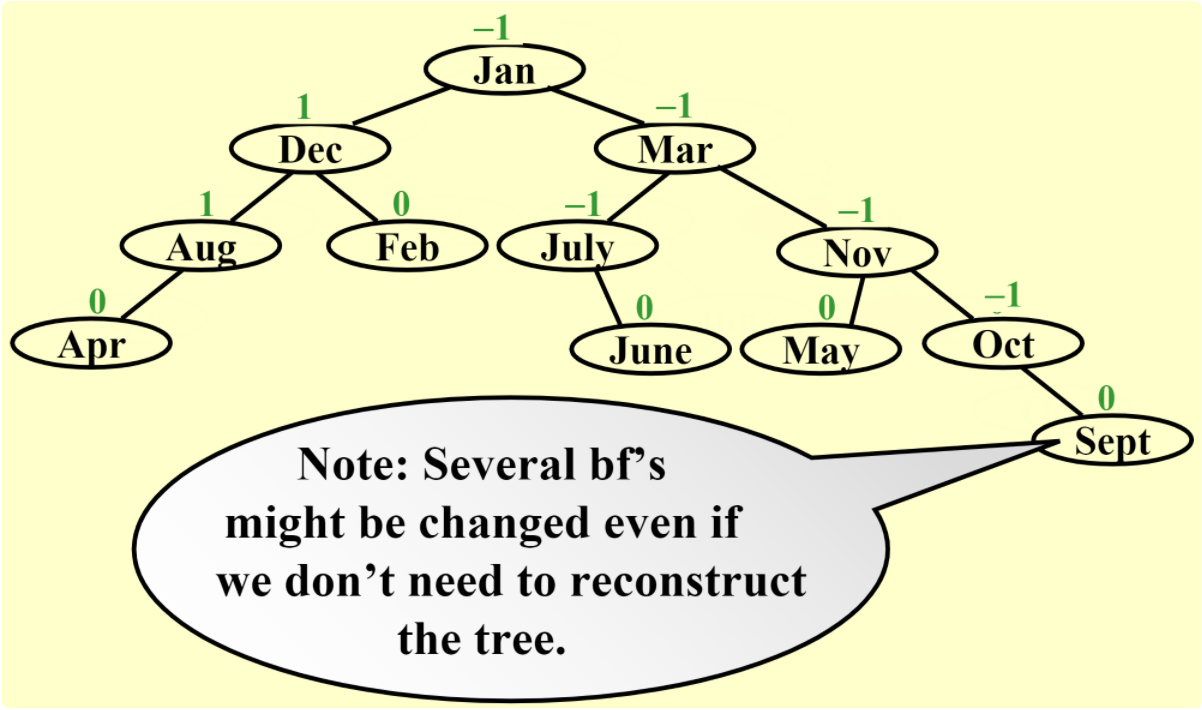


In general:


LR
Insertion
LR
Rotation
OR
OR
7

RL rotation


Mar
Aug
May
Apr
Jan
Nov
Dec
July
Feb
RL
Rotation
In general:
RL
Insertion
RL
Rotation
OR
OR
8



即使没有重构树，某个节点的平衡因子也可能发生变化，所以要实时更新。

1.1.3 Programming

Declaration

Declaration C 复制代码

```c
struct AvlNode
{
    ElementType Element;
    AvlTree Left;
    AvlTree Right;
    int Height; //记录高度而非bf.
}
typedef struct AvlNode *Postion;
typedef struct AvlNode *AvlTree;
```

这个函数必须处理**NULL指针即空树**的情形

Height

C | 复制代码

```c
static int Height(Position P)
{
    if(P == NULL)
        return -1;
    else
        return P->Height;
}
```

Insert

Insert C 复制代码

```c
AvlTree
Insert(ElementType X,AvlTree T)
{
    //如果树为空，则创建一个新树
    if(T == NULL)
    {
        T = malloc(sizeof(struct AvlNode));
        if(T == NULL)
            FatalError("Out of space!!!");
        else
        {
            T->Element = X;
            T->Height = 0;
            T->Left = T->Right = NULL;
        }
    }
    else
    if(X < T->Element) // 插入的数据小于根，则需要插在左子树上
    {
        T->Left = Insert(X,T->Left); //插入左子树
        if(Height(T->Left) - Height(T->Right) == 2) // 如果插入的数据是Trouble Maker,则要进行翻转
        {
            if(X < T->Left->Element)  //LL
                T = SingleRotateWithLeft(T);
            else  //LR
                T = DoubleRotateWithLeft(T);
        }
    }
    else if(X > T->Element)
    {
        T->Right = Insert(X,T->Right);
        if(Height(T->Right) - Height(T->Left) == 2)
        {
            if(X > T->Right->Element)
                T = SingleRotateWithRight(T);
            else
                T = DoubleRotateWithRight(T);
        }
    }

    T->Height = Max(Height(T->Left),Height(T->Right)) + 1;
    return T;
}
```

SingleRotate

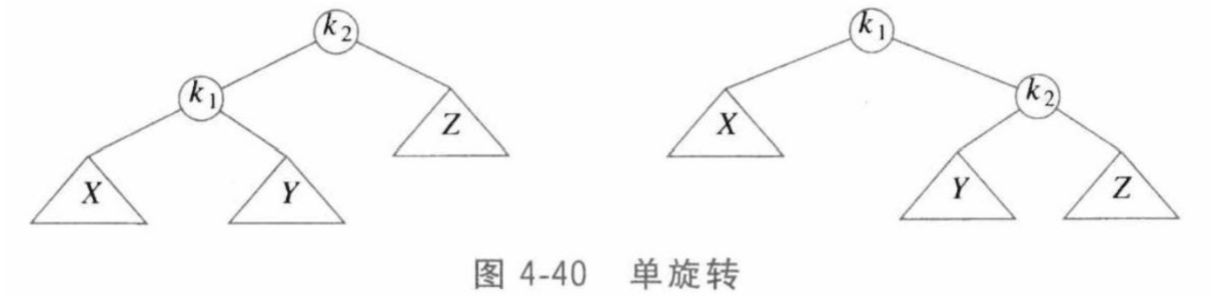


图 4-40　单旋转

Single Rotate　C　复制代码

```c
static Position
SingleRotateWithLeft (Position k2)
{
    Position K1;

    K1 = K2->Left;
    K2->Left = K1->Right;
    K1->Right = K2;

    //更新height
    K2->Height = Max(Height(k2->Left),Height(Height(K2->Right))) + 1;
    K1->Height = Max(Height(k1->Left),K2->Height) + 1;

    return K1;
}
```

DoubleRotate

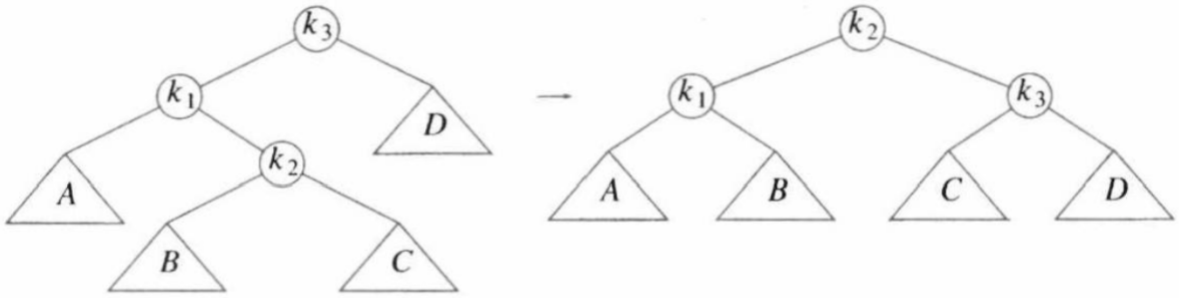


图 4-42　双旋转

Double rotate　C　复制代码

```c
static Position
DoubleRotateWithLeft (Position k3)
{
    K3->Left = SingleRotateWithRigh(K3->Left);

    return SingleRotateWithLeft(K3);
}
```

对于AVL树，懒惰删除比较好。

懒惰删除：为了删除某元素，我们只标记这个元素已被删除。

1.1.4 Exercise

练习-PTA-DataStructure：https://pintia.cn/problem-sets/16/problems/668

1.

2-3 分数 2 作者 DS课程组 单位 浙江大学

If the depth of an AVL tree is 6 (the depth of an empty tree is defined to be -1), then the minimum possible number of nodes in this tree is:

- A. 13
- B. 17
- C. 20
- D. 33

答案正确：2 分 创建提问

利用递推公式 $n_h = n_{h-1} + n_{h-2} + 1$

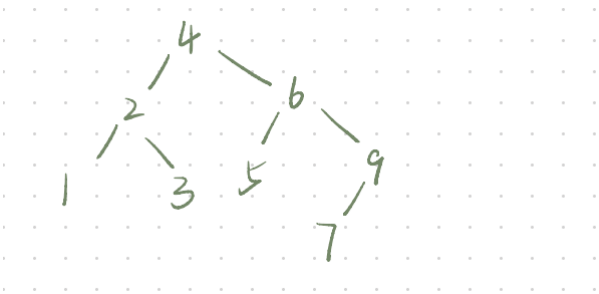
2.

2-1 分数 1 作者 陈越 单位 浙江大学

Insert 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL tree. Which one of the following statements is FALSE?

- A. 4 is the root
- B. 3 and 7 are siblings
- C. 2 and 6 are siblings
- D. 9 is the parent of 7

答案正确：1 分 创建提问



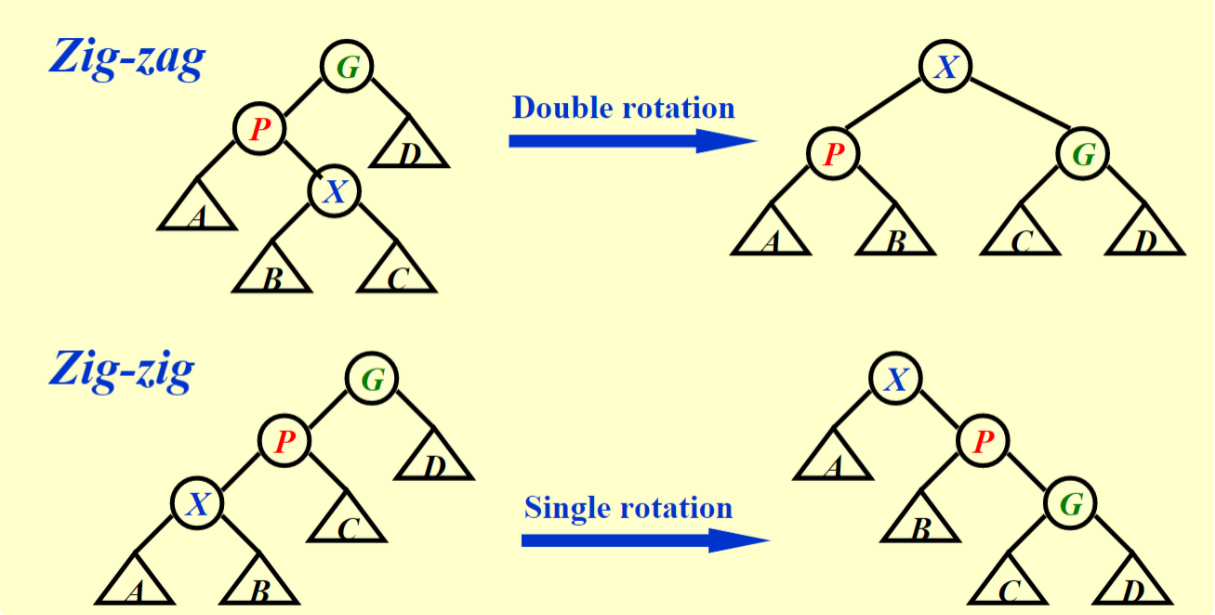
1.2 Splay Trees

1.2.1 Definition

Target: Any M consecutive tree operations starting from an empty tree take at most **O(MlogN)** time.
Idea: After a node is accessed, it is pushed to the root by a series of AVL tree rotation.

1.2.2 Methods

For any **nonroot** node **X** , denote its parent by **P** and grandparent by **G**.
Case 1: P is the root –> Rotate X and P
Case 2: P is not the root
Zig–zag
Zig–zig



Splaying not only moves the accessed node to the root, but also roughly halves the depth of most nodes on the path.

1.2.3 Deletions

1. **Find X**(X will be at the root)
2. **Remove X**(There will be two subtrees TL and TR)
3. **FindMax(TL)** (The largest element will be the root of TL and has no right

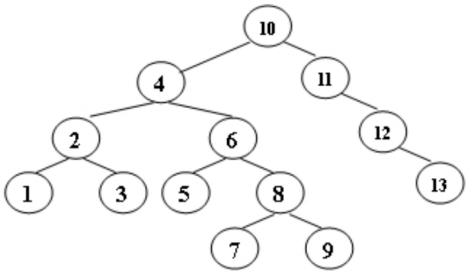
child)

4. Make T_R the right of the root of T_L.

1.2.4 Exercise

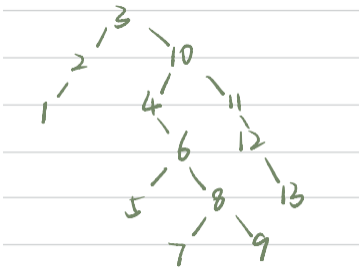
2-2 分数 2 作者 陈越 单位 浙江大学

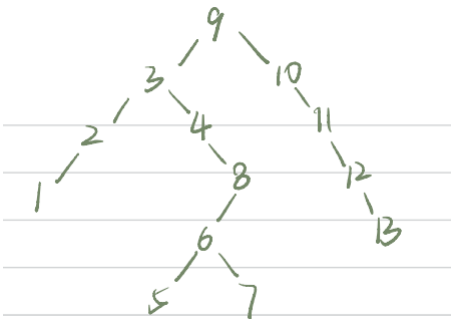
For the result of accessing the keys 3, 9, 1, 5 in order in the splay tree in the following figure, which one of the following statements is FALSE?

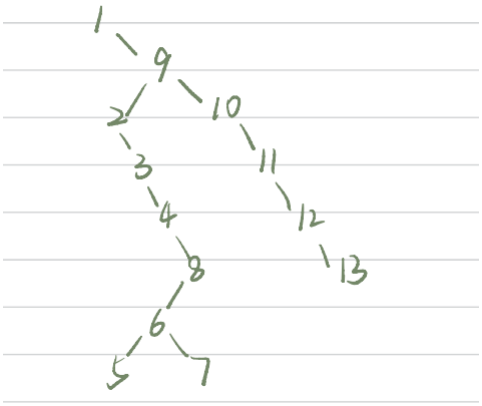


- A. 5 is the root
- B. 1 and 9 are siblings
- C. 6 and 10 are siblings
- D. 3 is the parent of 4

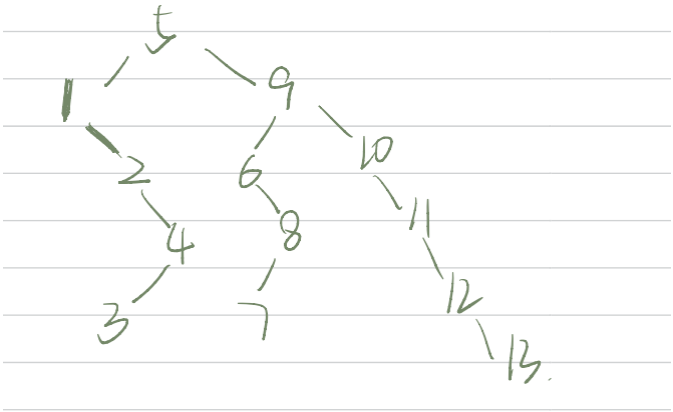
答案正确：2 分 创建提问







Final：



1.3 Amortized Analysis

1.3.1 Definition

Target: Any M consecutive operations take at most `O(MlogN)` time.

—— Amortized time bound

worst-case time >= amortized time >= average-case bound

Probability is not involved in amortized analysis

摊还分析中，我们求数据结构的一个操作序列中所执行的所有操作的平均时间，来评价操作的代价。这样，我们就可以说明一个操作的平均代价是很低的，即使序列

中某个单一操作的代价很高。摊还分析不同于平均情况分析，并不涉及概率，可以保证最坏情况下每一个操作的平均性能。

Types：

- Aggregate analysis 聚合分析
- Accounting method 核算法
- Potential method 势能法

1.3.2 Aggregate analysis

聚合分析
Idea: Show that for all n, a sequence of n operations takes `worst-case` time `T(N)` in total. In the worst case, the average cost, or amortized cost, per operation is therefore T(n)/n.

确定一个n个操作的序列的总代价的上界T(N), 因而每个操作的平均时间为T(N) / n, 我们将平均代价作为每一个操作的代价，因此所有的代价都是相同的。

Example: Stack with MultiPop(int k,Stack S)

```c
while(!IsEmpty(S) && k>0)
{
    Pop(S);
    k--;
}
```

> T = min(sizeof(S),k)

We can pop each object from the stack at most once for each time we have pushed it onto the stack.
amortized time = O(n)/n = O(1)

1.3.3 Accounting method

核算法
Idea: When an operation's amortized cost Ci exceeds its actual cost ci, we assign the difference to specific objects in the data structure as **credit**. Credit can help pay for later operations whose amortized cost is less than their actual cost.

对不同的操作赋予不同的费用，可能多于或少于实际代价。我们将赋予一个操作的费用成为它的摊还代价。当摊还代价超出实际代价时，我们将差额存入数据结构中特定对象，存入的差额称为**信用**， 对于后续操作中摊还代价小于实际代价的情况，信用可以用来支付差额。不同的操作有不同的代价。

数据结构所关联的信用必须一直为非负值。

sizeof(S) >= 0 –> Credits >=0
–> amortized time = O(n)/n = O(1)

push 2

pop 0

pop（s） 0

对于任意个序列，总摊还代价时实际代价的上界，由于总的摊还代价为n，所以实际的代价也是。

1.3.4 Potential method

势能法
Idea: Take a closer look at the credit ––

> c′i=ci+Θ(Di)−Θ(Di−1)
> Where Θ is the potential function.

势能法并不将预付代价表示为数据结构中特定对象的信用，而是表示为势能，将势能释放即可用来支付未来操作的代价。我们将势能作为整个数据结构而不是特定对

象相关联。

势函数将每个数据结构Di 映射到一个实数，此值即为关联到数据结构的势。第i个操作的摊还代价为：

> $c'i=ci+\Theta(Di)-\Theta(Di-1)$

即，每个摊还操作的摊还代价等于其实际代价加上此操作引起的势能变化。

Hence,

In general, a good potential function should always assume its minimum at the start of the sequence.开始是最小值,一般为0.

总摊还时间给出了实际时间的一个上限。

对于势函数的选取，我们选择将栈映射到其内部元素个数的函数，即Θ(Di)表示第i次操作栈时，栈中元素的个数。对于初始的空栈，有 `θ(D0)=0` ；并且由于栈中元素不可能为负，因此 `θ(Di)≥0=θ(D0)` 。

2-4 分数 2 作者 陈越 单位 浙江大学

When doing amortized analysis, which one of the following statements is FALSE?

- (●) A. Aggregate analysis shows that for all n, a sequence of n operations takes worst-case time $T(n)$ in total. Then the amortized cost per operation is therefore $T(n)/n$
- ( ) B. For potential method, a good potential function should always assume its maximum at the start of the sequence
- ( ) C. For accounting method, when an operation's amortized cost exceeds its actual cost, we save the difference as credit to pay for later operations whose amortized cost is less than their actual cost
- ( ) D. The difference between aggregate analysis and accounting method is that the later one assumes that the amortized costs of the operations may differ from each other

答案错误：0 分 创建提问

B

2-5 有题解 分数 2 作者 张国川 单位 浙江大学

Consider the following buffer management problem. Initially the buffer size (the number of blocks) is one. Each block can accommodate exactly one item. As soon as a new item arrives, check if there is an available block. If yes, put the item into the block, induced a cost of one. Otherwise, the buffer size is doubled, and then the item is able to put into. Moreover, the old items have to be moved into the new buffer so it costs $k+1$ to make this insertion, where k is the number of old items. Clearly, if there are N items, the worst-case cost for one insertion can be $\Omega(N)$. To show that the average cost is $O(1)$, let us turn to the amortized analysis. To simplify the problem, assume that the buffer is full after all the N items are placed. Which of the following potential functions works?

- A. The number of items currently in the buffer
- B. The opposite number of items currently in the buffer
- C. The number of available blocks currently in the buffer
- D. The opposite number of available blocks in the buffer

答案正确：2 分 创建提问

设 $size_i$ 为第 i 次插入前 buffer 的大小。

$\hat{c}_i = c_i + \phi_i - \phi_{i-1}$.

如果插入前 buffer 没满，$c_i = 1$ ，否则 $c_i = size_i + 1$.

A:

如果插入前 buffer 没满, $\hat{c}_i = c_i + \phi_i - \phi_{i-1} = 1 + (size_i + 1) - size_i = 2$.

如果插入前 buffer 满，$\hat{c}_i = c_i + \phi_i - \phi_{i-1} = size_i + 1 + (size_i + 1) - size_i = size_i + 2$.

B:

同理, 两种情况 $\phi_i - \phi_{i-1}$ 要么是 1 要么是 -1， 而 c_i 却和当前 buffer 大小有关，$\hat{c}_i$ 肯定不是常数。

C:

如果插入前 buffer 没满, $\phi_i - \phi_{i-1} = -1, \hat{c}_i = c_i + \phi_i - \phi_{i-1} = 1 + (-1) = 0$.

如果插入前 buffer 满，$\phi_i - \phi_{i-1} = (size_i - 1) - 0 = size_i - 1$,

$\hat{c}_i = size_i + 1 + \phi_i - \phi_{i-1} = 2size_i$.

D:

如果插入前 buffer 没满, $\phi_i - \phi_{i-1} = 1, \hat{c}_i = c_i + \phi_i - \phi_{i-1} = 1 + 1 = 2$.

如果插入前 buffer 满，$\phi_i - \phi_{i-1} = 1 - size_i$,

$\hat{c}_i = size_i + 1 + \phi_i - \phi_{i-1} = 2$.

要让摊还代价为常数。