AVL Trees

为了加速查找，包括搜索与删除，我们使用了二分搜索树，其性能一般为$O(height)$。但是，二分搜索树的性能在最坏情况（即链的情况）下可能会退化到$O(n)$。为了解决这个问题，我们引入了AVL树。

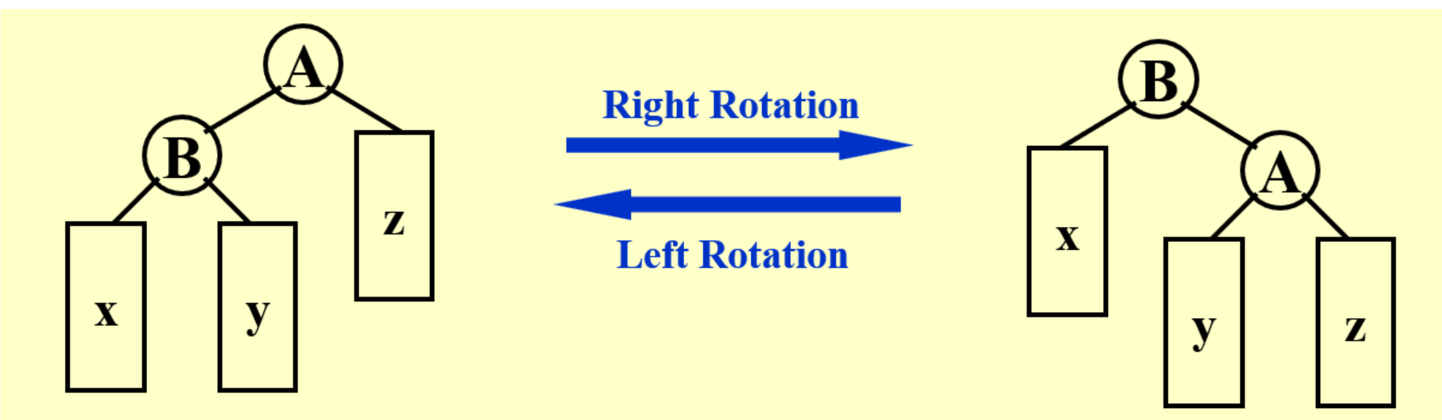
1.定义

- 空二叉树是高度平衡的（Height balanced）定义空二叉树的高度为-1
- 设非空二叉树的左右子树为T_L, T_R。那么非空二叉树高度平衡等价于
  - T_L和T_R都是高度平衡的
  - 并且$|h_L - h_R| \le 1$，其中h_L和h_R分别是左右子树的高度
- 平衡因子（balanced factor）$BF(node) = h_L - h_R$。在AVL树中，每个节点的平衡因子只能是-1,0,1

2.旋转（Tree rotation）

在插入和删除的过程中，树的高度平衡有可能被破坏，我们需要设法来修复这种破坏。有两种思路来修复，一种是改变插入的顺序，但这样在一般情况下是不可取的。另一种就是我们改变插入后的结构，改变已有的结构来维持平衡，这种操作就是旋转。

旋转是一种**改变树的结构但是不改变元素顺序**的操作。



2.1 单次旋转

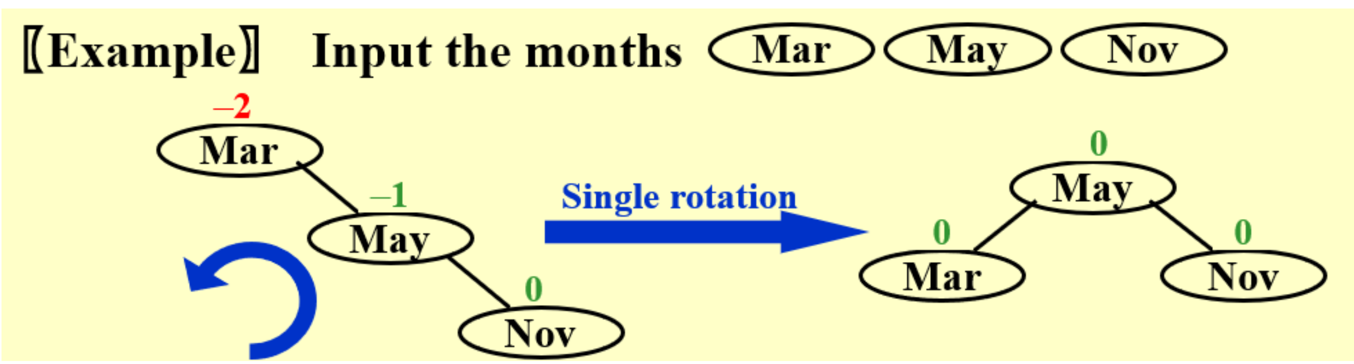
单次旋转是对**出问题节点的儿子**进行旋转。

- 右旋：将B向右旋转，A变为B的右儿子，由于y原先属于A的左子树，此时将y接到A的左子树也是满足二分搜索条件的。也可以看作是把B拎起来，A就落下去了，那么多的部分y就接到A上。
  - 旋转后，左子树高度减1，右子树高度加1；而且保证顺序性质不变（即二分搜索树的条件）

- 旋转中受到影响的节点只有A，B，A的父节点，A的左子树的根节点，因此操作的复杂度为$O(1)$
- 左旋：将A向左旋转，B变为A的左儿子，A的左子树y成为B的右子树。
  - 旋转后，右子树高度减1，左子树高度加1；而且保证顺序性质不变
  - 操作复杂度为$O(1)$
- 右旋的实现：

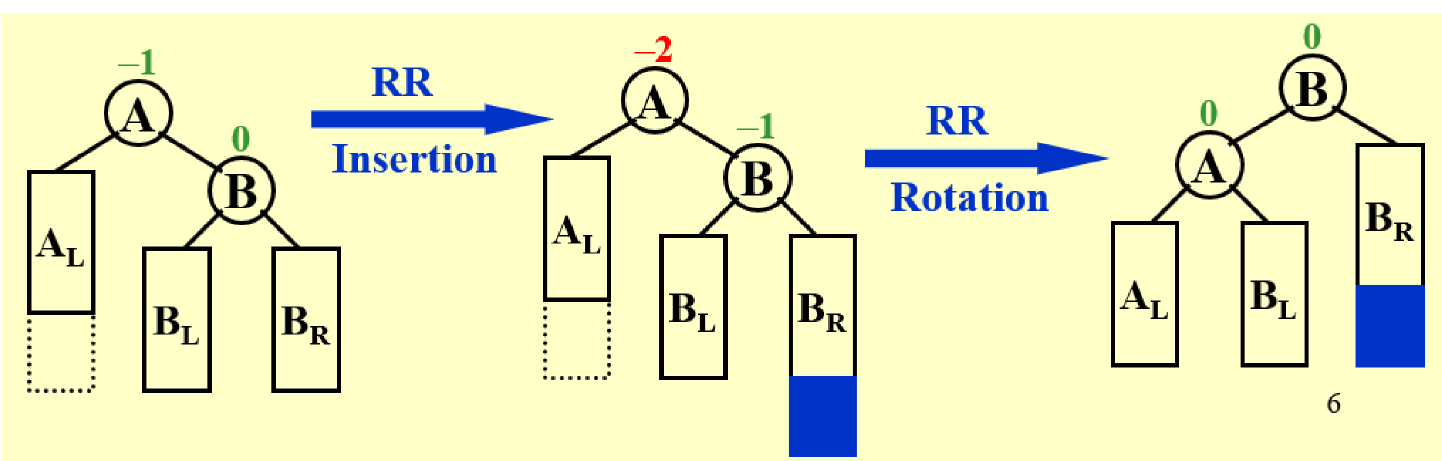
```
A.Left = B.Right;
B.Right = A;
A.Father.son = B;
```

Example



这种情况下，出现问题的节点（BF为2）是March，而产生这个问题的节点是November。这个时候我们要做的旋转操作是对May进行左旋。由于Nov是May的右子树的右子树，所以这种操作叫做 **RR rotation**.

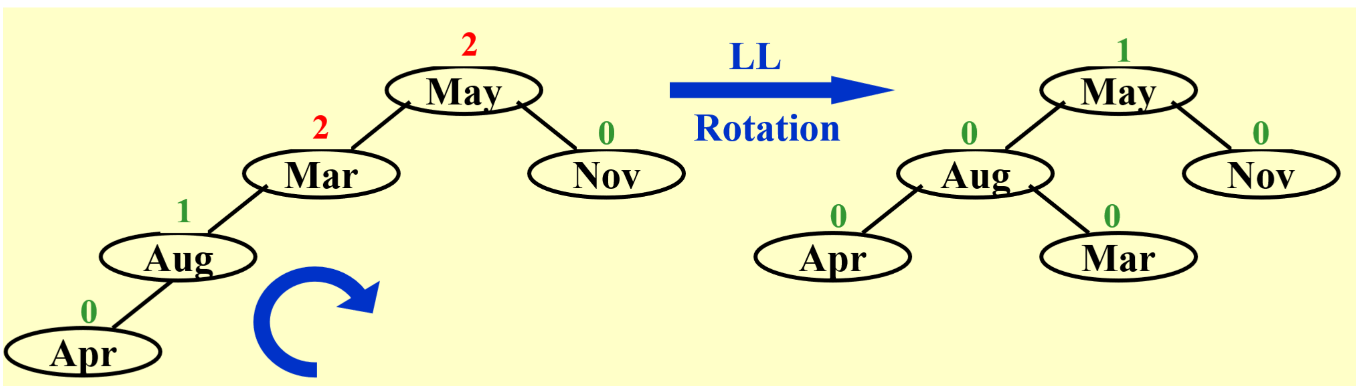
一般来说：



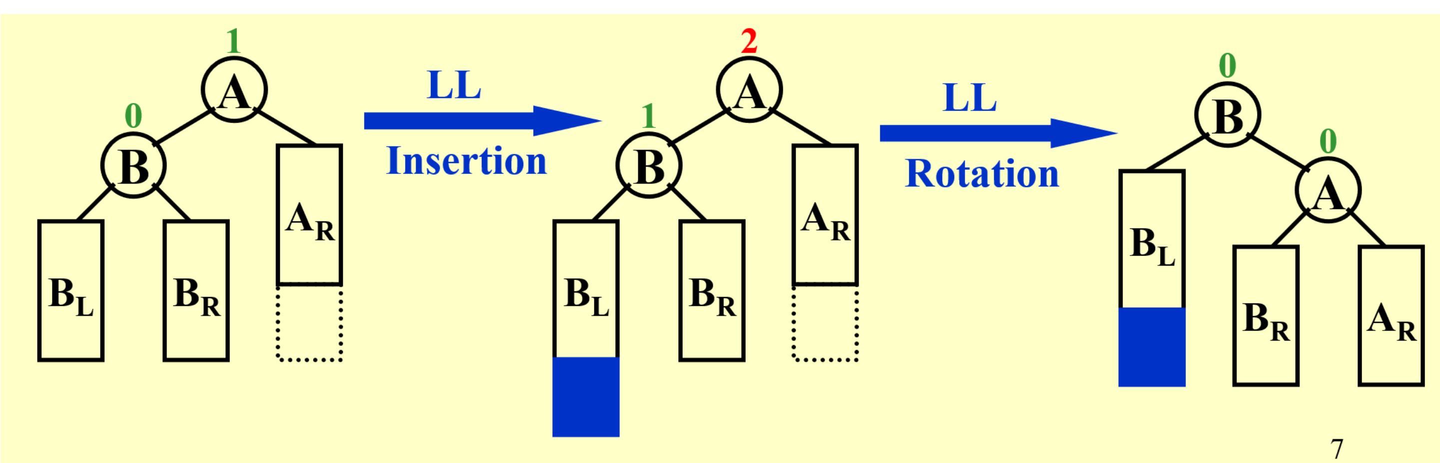
状态1是高度平衡的，经过插入一个节点即蓝色部分（插在右子树的右子树上）后，A的BF变为-2，发生了不平衡，为了维护平衡状态，我们需要对B进行RR rotation。

> **Note**
>
> A不一定要是树的根节点。

与 RR rotation 相对应的是 LL rotation，即插入的节点在左子树的左子树上。需要进行右旋的是左儿子。

> **Example**
>
> 
>
>
> 这里插入的节点是Apr，发生问题的节点是Mar，由于是自插入节点开始的维护，我们县发现了Mar出现的问题，于是我们就处理Mar的左儿子Aug，进行右旋。这种操作叫做 **LL rotation**。这里涉及到维护的顺序，应当是自下而上进行维护。

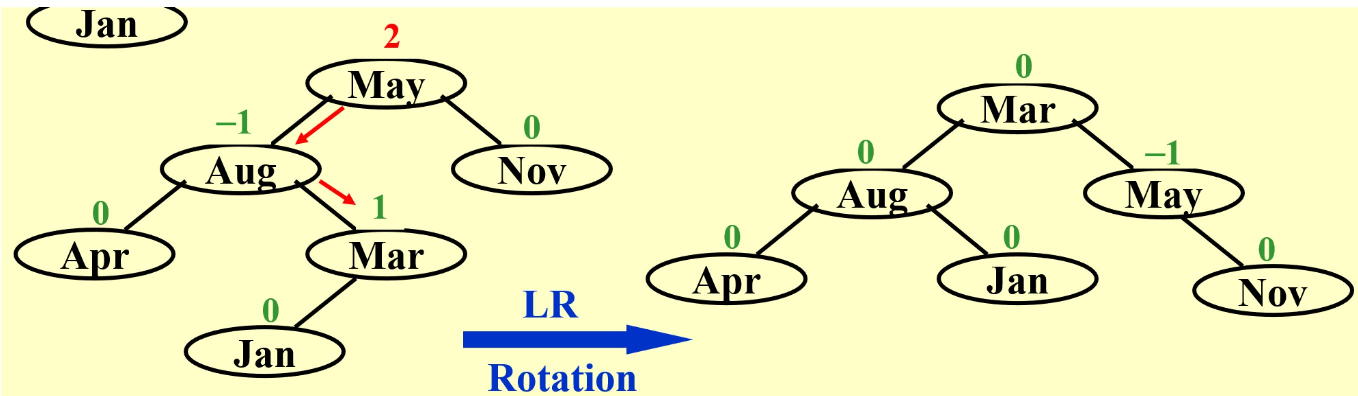
一般来说：



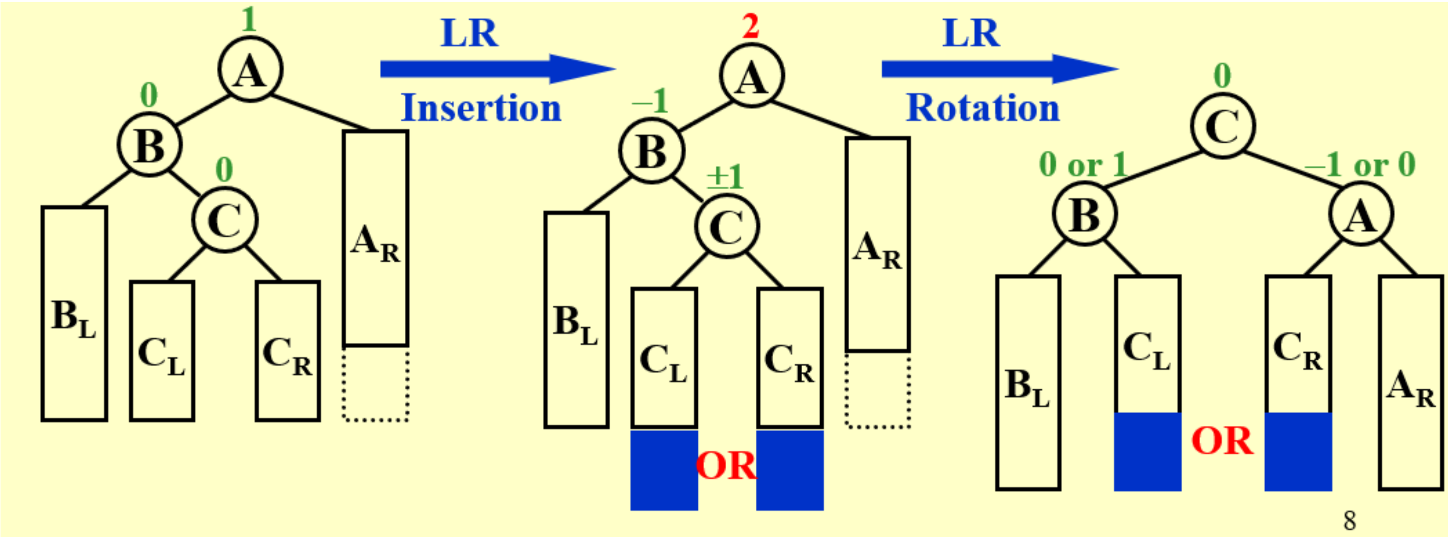
> **Note**
>
> 小结：单次旋转是针对出问题的节点与插入节点在一条线上的情况，即没有出现折叠的情况。旋转的是出问题节点的儿子，只需要一次旋转。

2.2 二次旋转 Double Rotation

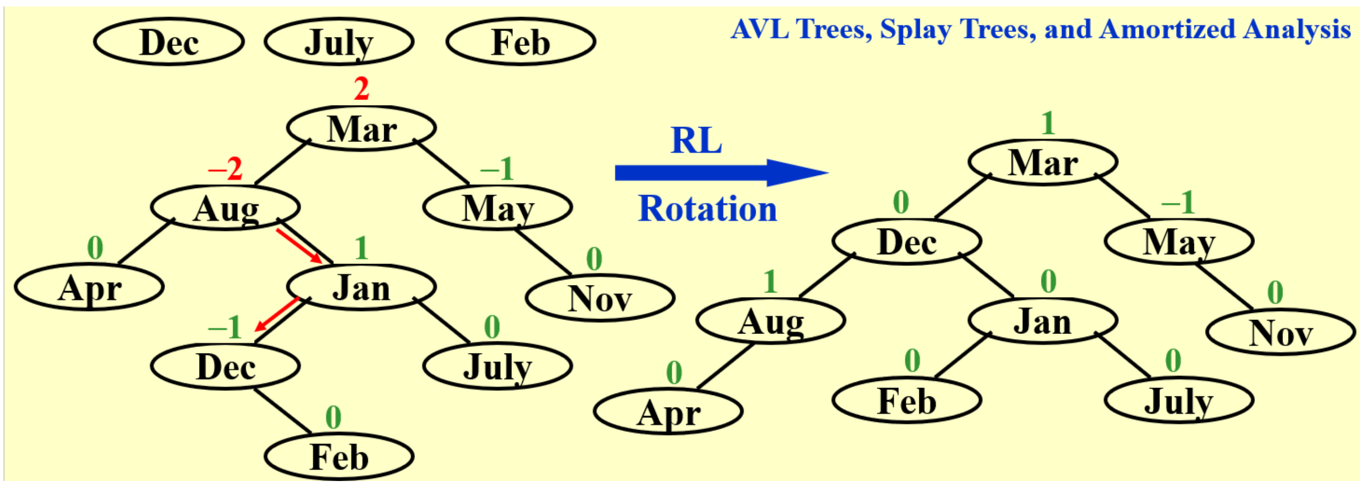
针对插入的节点在左子树的右子树上（LR），或者在右子树的左子树上（RL）的情况，我们需要进行两次旋转。此时旋转的是**出问题节点的孙子节点**。

> **Example**
>
> 
>
>
> 这个例子中，插入的节点是Jan，导致了May出现问题，Jan是在May的左子树的右子树中，是LR的情况，我们需要做两次旋转：找到May的左儿子的右儿子——Mar。两次旋转都是将它提起来。先对它左旋软后再对它右旋。

一般来说：

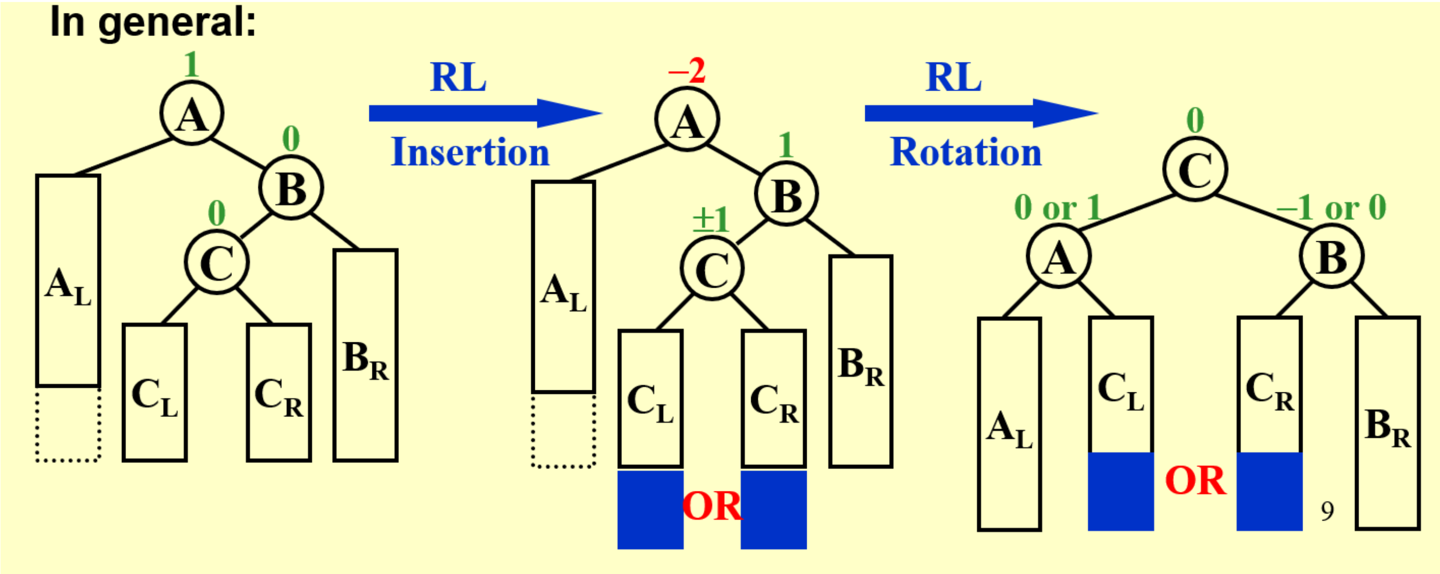


Example



按照从下往上修复的顺序，我们发现第一个出问题的节点是Aug，而且导致问题的节点再它的右子树的左子树中，这是RL的情况，我们需要旋转它的孙子节点。将孙子节点提起来两次就可以了。

一般来说：



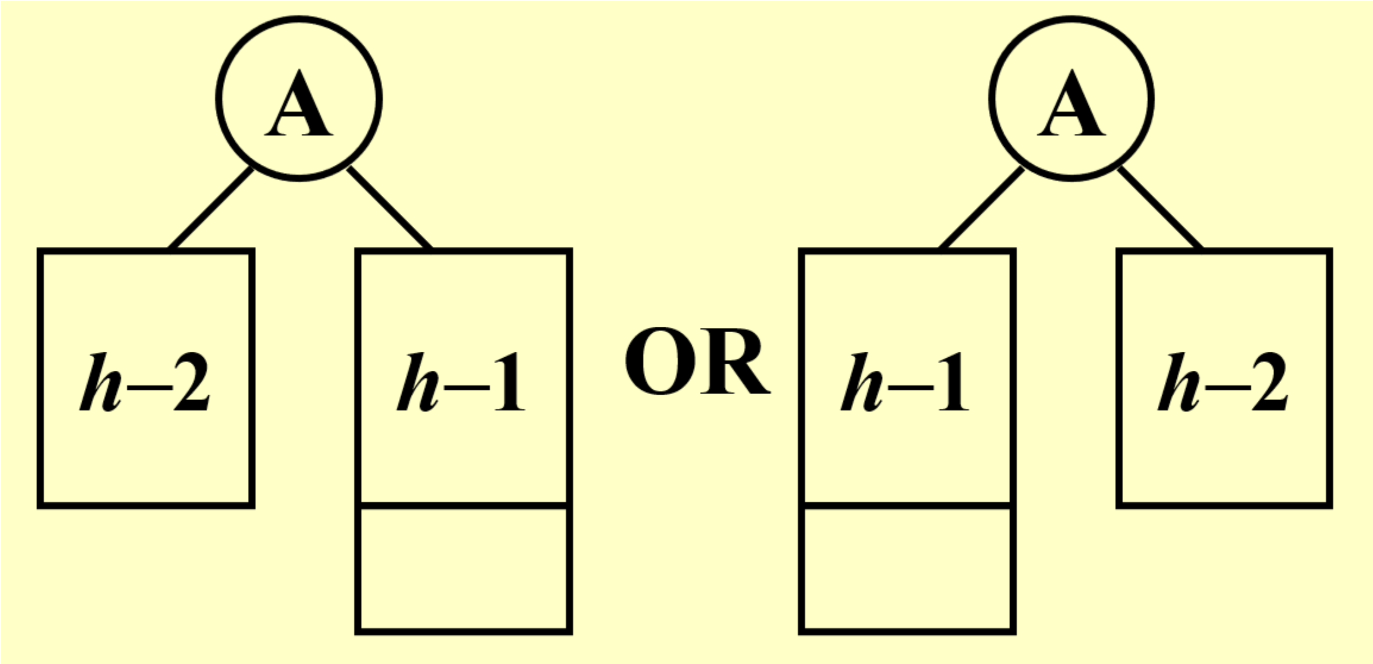
> **Note**
>
> 小结：双旋转是针对出现折叠的情况，旋转的是出问题节点的孙子节点，需要两次旋转。

> **Note**
>
> 即使树的结构不发生变化，但是由于插入和删除的操作，节点的bf（平衡因子）可能会发生变化，因此我们需要对平衡因子进行维护。

3. Avl树的复杂度

很显然，我们有$T_p = O(h)$，那么树的高度该如何计算呢。我们考虑计算h的渐进上界。作出以下定义：

> n_h表示一棵高度为h的高度平衡树的最小节点数。
> 据这个定义，我们的树一定是以下形状的：

> 
>
>
> 如果两个子树的高度一致，我们可以将其中一个子树的叶子节点去掉一个，仍然满足高度平衡和最小高度的条件。

由以上的定义，我们可以得到以下的递推关系：

$$n_h = 1 + n_{h-1} + n_{h-2}$$

注意到这个递推式与斐波那契数列有着类似的结构，我们有以下结论，对任意$h \geq -1$:

$$n_h = F_{h+3} - 1 \text{ 其中 } F_0 = 0, F_1 = 1$$

由斐波那契数列的通项公式：

$$F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

我们有近似值：

$$n_h \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} - 1$$

由此：

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

4. 总结

Avl树可以保证每一次插入和删除的操作都是$O(\log n)$的时间复杂度。但是，相对而言Avl树的实现代码比较复杂。

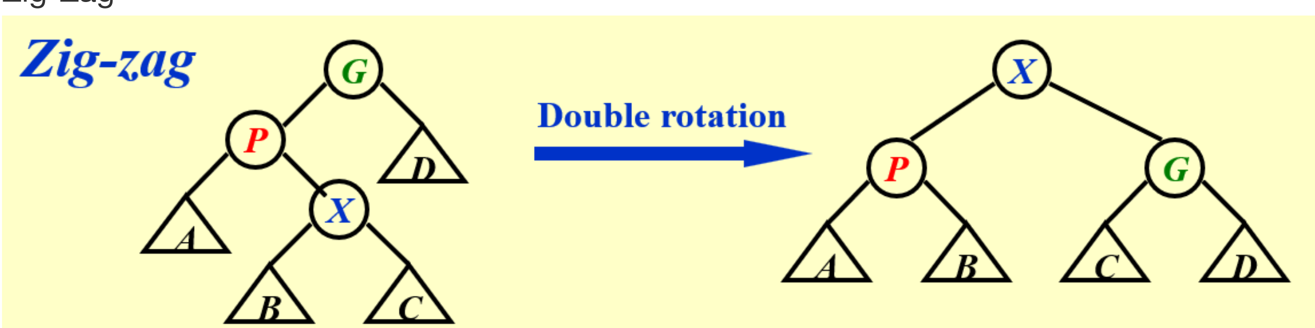
Splay Trees

- 目标：从空树开始的任意M次树操作的时间复杂度最多是$O(M\log N)$

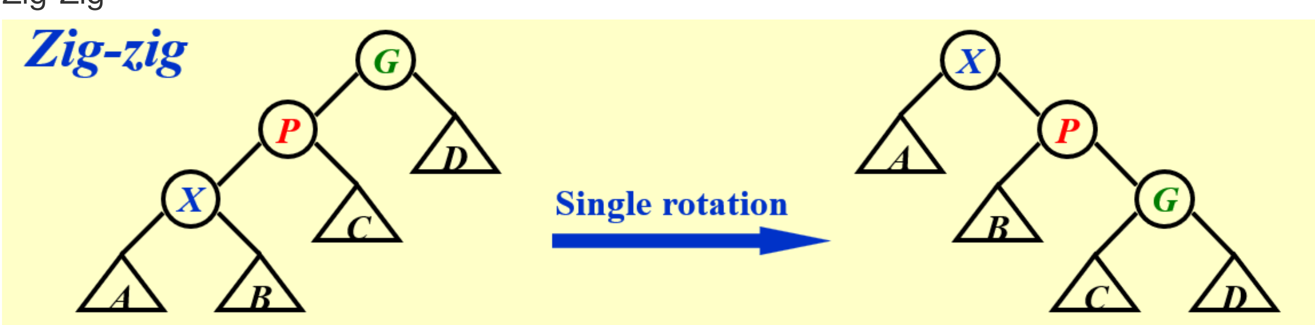
> 思路：我们要让M次操作总的时间复杂度得到约束，每一次的时间复杂度可能会达到worst-case bound，我们考虑让这个worst-case bound变成没有worst-case bound的情况。比如第一次查询一个链的叶子节点，那么这次查询是$O(n)$的时间复杂度，但是我们可以通过旋转操作将这个节点提到根节点，这样下一次查询这个节点的时间复杂度就是$O(1)$了，整体的复杂度就降下来了。

操作如下：对于被查询的非根节点 X，其父节点为 P，祖父节点为 G，我们可以通过以下操作将X提到根节点：

- Case1:P是根节点，对X进行一次旋转
- Case2:P不是根节点
  - Zig-Zag

    旋转两次X即可，相当于Avl树的Double Rotation。
  - Zig-Zig

这与Avl树的操作不同，先对P进行一次旋转，再对X进行一次旋转。

- Deletions:
  - Step1:Find X;
  - Step2:Remove X;
  - Step3:FindMax(T_L)
  - Make T_R the right child of the roor of T_L

Amortized Analysis

定义：Amortized Analysis（摊还分析）：对任何M次操作最多消耗$O(MlogN)$的时间，即均摊界(Amortized time bound)。

$$\text{worst-case bound } \geq \text{ amortized bound } \geq \text{ average-case bonud}$$

worst-case bound 是要求每次都是最坏的情况；

amortized bound 是对M次操作均摊后的情况

我们有以下方法分析：

Aggregate analysis（聚合分析）

对所有的n，n次操作一共消耗 worst-case 时间 $T(n)$。所以此时的均摊复杂度为$T_{amortized} = T(n)/n$

Example

Stack with MultiPop(int k,Stack S)

```
Algorithm {
  while(!IsEmpty(S) && k > 0){
    Pop(S);
    k--;
  }
}
    T = min (sizeof(S),k)
```

考虑一个在空栈中有n次push,pop 和 MultiPop操作的集合，那么

$$sizeof(S) \leq n$$

$$T_{amortized} = O(n)/n = O(1).$$

Note

聚合分析就是将所有的操作的时间复杂度加起来，然后除以操作的次数，就得到了均摊界。

Accounting method（记账法）

当一个操作的均摊开销是$\hat{c}_i$超过了实际开销c_i时，我们将$\hat{c}_i - c_i$的差额存入一个账户中。当一个操作的均摊开销是$\hat{c}_i$小于了实际开销c_i时，我们将账户中的钱拿出来用。

可以通过设计一个均摊代价，验证其合理性（credit一直>=0），根据均摊代价计算均摊界

Note

对于所有有n个操作的序列，我们必须有

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

此时的均摊界为

$$T_{amortized} = \frac{\sum_{i=1}^{n} \hat{c}_i}{n}$$

Example

Stack with MultiPop(int k, Stack S)
c_i for Push: 1; Pop: 1; MultiPop: min(sizeof(S),k)
$\hat{c}_i$ for Push: 2; Pop: 0; MultiPop: 0
在每次push的时候，我们支付的代价为2，提前将未来pop这个元素的代价支付了。那么在将来的pop和multipop中，我们就不需要支付代价了。

Starting from an empty stack —— Credits for
Push: 1; Pop: -1; MultiPop: -1 for each +1

$$\text{sizeof(S)} \geq 0 \text{ 意味着 Credits} \geq 0$$

也就是说

$$O(n) = \sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

所以均摊界为

$$T_{amortized} = O(n)/n = O(1)$$

Potential method（势能法）

Idea:Take a closer look at the credit:

$$\hat{c}_i - c_i = Credit_i = \Phi(D_i) - \Phi(D_{i-1})$$

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= (\sum_{i=1}^{n} c_i) + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

其中$\Phi(D_n) - \Phi(D_0) \geq 0$

In general, a good potential function should always assume its minimum at the start of the sequence.
一般来说，一个好的势函数应该总是在序列的开始处假设它是最小的。

Note

均摊代价不好设计：通过势能函数计算credit，根据credit和实际代价计算均摊代价

Example

Stack with MultiPop(int k, Stack S)
D_i = the stack that results after the i-th operation

势能函数的选取可以是任意的，只要满足其中的条件。
$\Phi(D_i)$ =the number of objects in the stack Di

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push:$\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) + 1) - sizeof(S) = 1$

$$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$

Pop:$\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) - 1) - sizeof(S) = -1$

$$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (-1) = 0$$

MultiPop:$\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S) - k) - sizeof(S) = -k$

$$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k + (-k) = 0$$

因此

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} O(1) = O(n)$$
$$\Rightarrow T_{amortized} = O(n)/n = O(1)$$

[LEMMA] 如果$a + b \leq c$，a、b都是正数，那么：

$$\log a + \log b \leq 2\log c - 2$$

[PROOF]

$$\begin{aligned} (ab)^{1/2} \ \leq \ (a+b)/2 \ \leq \ c/2 \\ \Rightarrow ab \ \leq \ c^2/4 \\ \Rightarrow \log a + \log b \leq 2\log c - 2 \end{aligned}$$

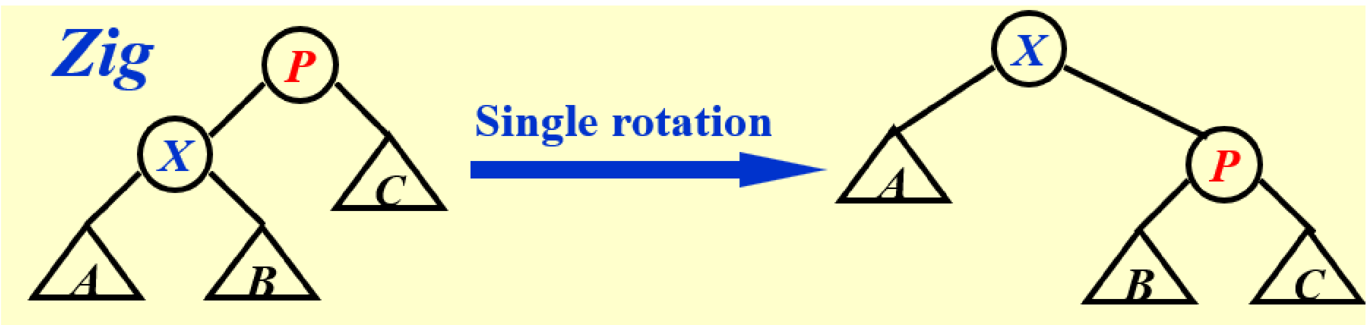
Example

Splay Trees: $T_{amortized} = O(\log N)$
D_i: the root of the resulting tree 本质上还是是整个树的形状
$\Phi(D_i)$: 必须满足在n步中至多增加$O(\log N)$，AND will also cancel out the number of rotations.在分析Splay Trees的时候，我们需要考虑到旋转的次数，这是比较困难的地方，我们希

望理想的势能函数能够消除旋转的次数。
关于势能函数的猜想：

1. 如果定义每个节点为根的子树的个数，那在一次旋转之后，变化的节点的值波动会比较大。
2. 如果定义每个节点的高度，那在每次旋转后很多节点的高度都发生了变化，很难精准地确定。
3. 一个直观的想法是缩小波动的值，那么取log是一个比较理想的操作。

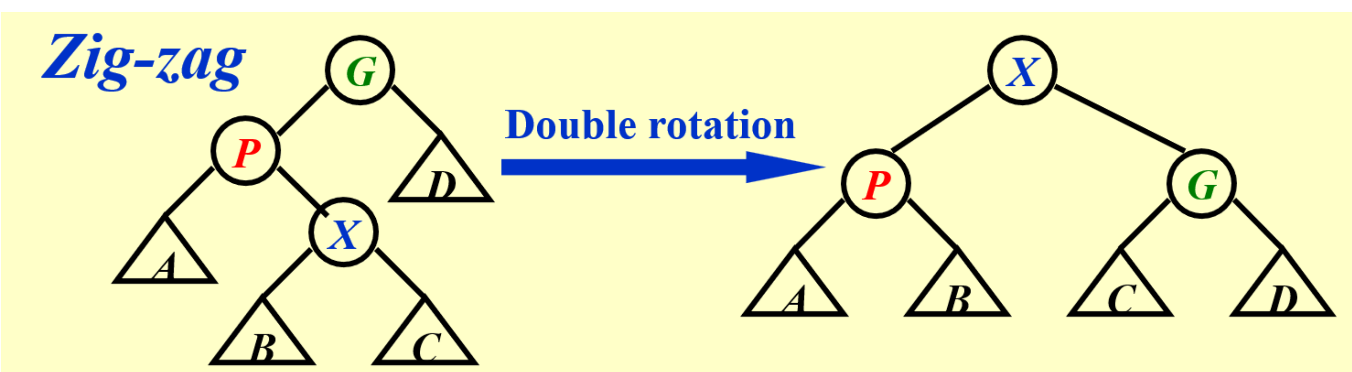
$\Phi(T)=\sum_{i\in T}\log S(i)$，其中$S(i)$是节点i的子树的大小（包括$i$）。这个定义也叫做*Rank of the sub tree* 即秩，约等于树的高度。
以下是基于势能函数$\Phi(T)=\sum_{i\in T}Rank(i)$的均摊分析。
Case1:



Zig 操作转一次X，实际代价为1，所以

$$
\begin{aligned}
\hat{c_i} &= 1 + R_2(X) - R_1(X) + R_2(P) - R_1(P) \\
&\leq 1 + R_2(X) - R_1(X)
\end{aligned}
$$

操作之后只跟X相关了。
Case2:

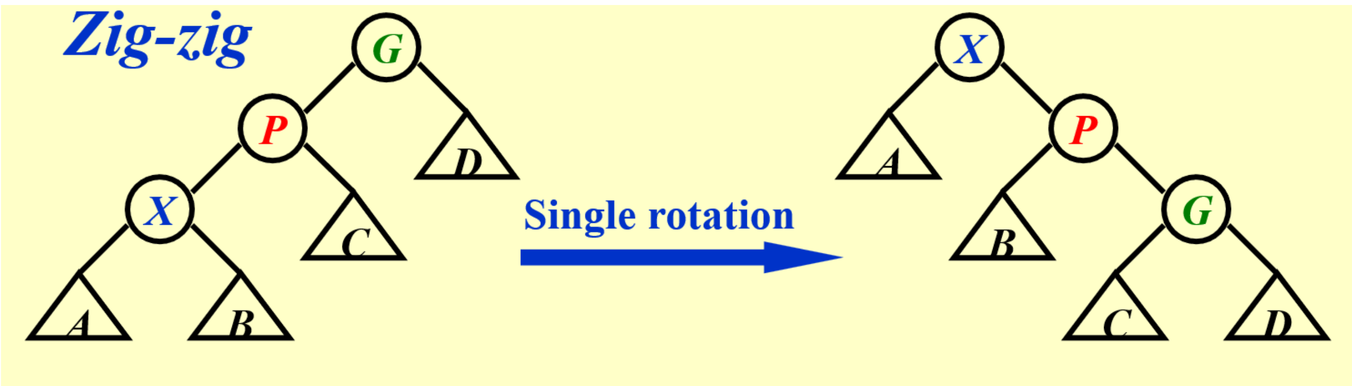


Zig-Zig 操作转两次X，实际代价为2，所以

$$
\begin{aligned}
\hat{c}_i &= 2 + R_2(X) - R_1(X) \\
&\quad + R_2(P) - R_1(P) \\
&\quad + R_2(G) - R_1(G) \\
&\le 2(R_2(X) - R_1(X))
\end{aligned}
$$

注意到$R_2(X), R_1(G)$实际上都是代表整个树的秩，所以这俩应该一样。根据引理，我们有$S_2(P) + S_2(G) \le S_2(X)$，因此$R_2(X) + R_2(G) \le R_2(X) - 2$，这样就把常数消掉了。最后再把$R_1(P) \ge R_1(X)$放缩即得。

Case3:



Zig-Zag 操作转两次X，实际代价为2，所以

$$
\begin{aligned}
\hat{c}_i &= 2 + R_2(X) - R_1(X) \\
&\quad + R_2(P) - R_1(P) \\
&\quad + R_2(G) - R_1(G) \\
&\le 3(R_2(X) - R_1(X))
\end{aligned}
$$

同上有$R_2(X) = R_1(G)$，此时再考虑把$R_2(P), R_2(G)$合起来发现没有那么好办。如果直接把他们都放大到$R_2(X)$的大小，那么就会有$2 + 2R_2(X) - 2R_1(X)$的上界，这里有常数就会导致计算困难，因为我们不清楚操作的次数。我们需要想办法把常数消去。可以看到$R_1(X)$代表的是$S(A) + S(B) + 1$，$R_2(G)$代表的是$S(C) + S(D) + 1$，而整棵树的大小是$S_2(X) = S(A) + S(B) + S(C) + S(D) + 3$，这就符合了引理的前提，因此可以利用引理：$R_2(G) \le 2R_2(X) - 2 - R_1(X)$，而$R_2(P) - R_1(P) \le R_2(X) - R_1(X)$，消掉了常数，得到了上面的结论。

现在我们要把三部分合起来分析，Zig操作最多会被执行一次，因此它的常数可以被保留。为了算整体的均摊代价，将zig-zag的上界放大至和zig-zig一样，而相邻两次的操作中前一次的$R_2(X)$与后一次的$R_1(X)$相同，因此都可以消掉，最后就得到了以下的定理：

> The amortized time to splay a tree with root T at node X is at most $3(R(T) – R(X)) + 1 = O(\log N)$.