I. AVL Trees, Splay Trees, and Amortized Analysis

AVL Tree

AVL Tree Attribution

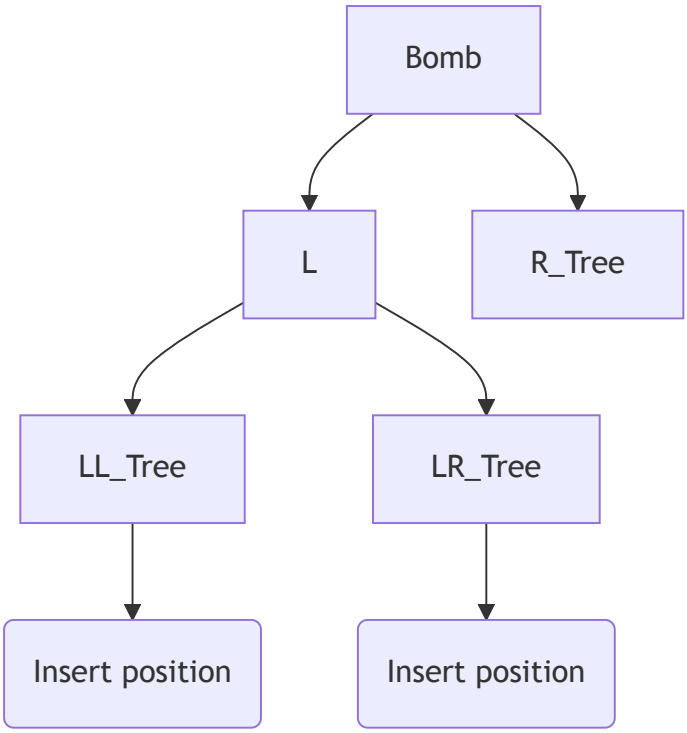
- Definition of height balanced
  - 空树一定是balanced
  - $|h(T_l) - h(T_r)| \leq 1$，其中 $h(T_l)$ 表示的是左子树的高度
- Definition of BF(balance factor)
  - 记节点 T_p 及其子树 T_l, T_r, 则 $BF(T_p) = h(T_l) - h(T_r)$
  - 所以balanced tree可以记为
    对于任意一节点 $BF(T_p) \in \{0, 1, -1\}$
- Question 1: n nodes 的 AVL Trees 的 h maximum 是多少?
  Answer:
  记n_h为高度为 h 的 AVL Trees的最小节点数，$n_h = F_{h+2} - 1,\ \text{for } 0 \leq n$
  $\Rightarrow h = O(lnn_h) \Rightarrow h = O(lnn)$ ■

> Tips:
>
> - $\mathrm{O}(n)$: The Worst Case
> - $\Omega(n)$: The best Case
> - $\Theta(n)$: Usual Case("=")
>
> More Details

AVL Tree Build

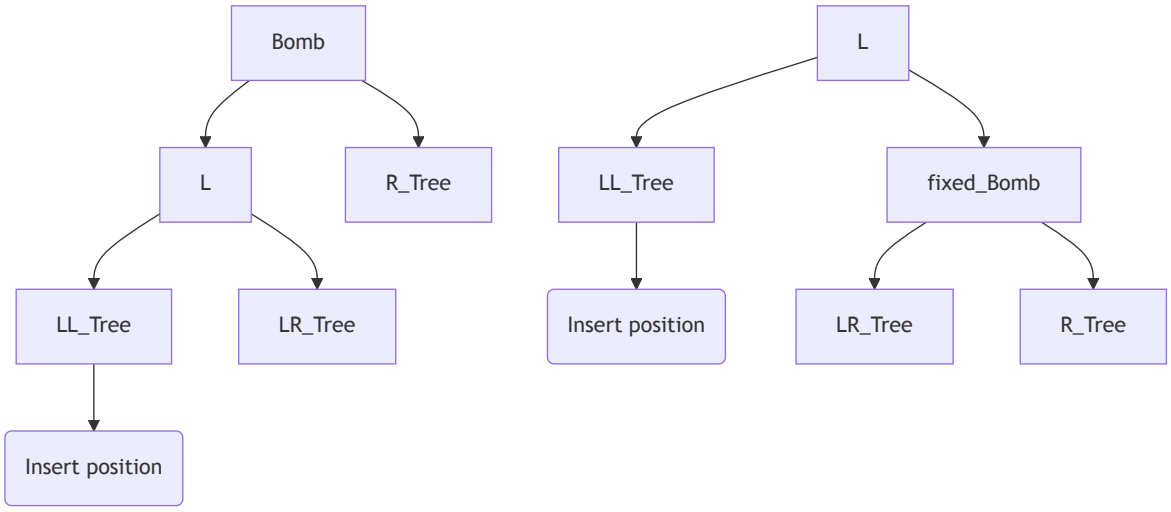
我们尝试以递归的方式来构建一个AVL Tree, 即在已知一个数为AVL树的情况下，随意插入一个Node，怎样保证新生成的树是仍然是一个AVL Tree

Lemma:显然任意一个插入节点不会导致其父节点失衡，所以任意一个AVL树(对称性)可以被抽象为下图.其中_Tree可以是空树，Bomb是离插入节点最近的、BF最先出问题的那个节点，Insert position则是可能插入的位置。



因此只有两种情况：

case1: LL-Insert(下图左)



此时有(可用反证法证明)

$$\mathrm{BF}(\mathrm{bomb}) == 2, h(LL_{Tree}) = h(LR_{Tree}) = h(R_{Tree})$$

得到这个等式之后其实很容易想明白怎样移动树(上图右，结合Rotation的说法则更加明显了)

下证明其移动合法:

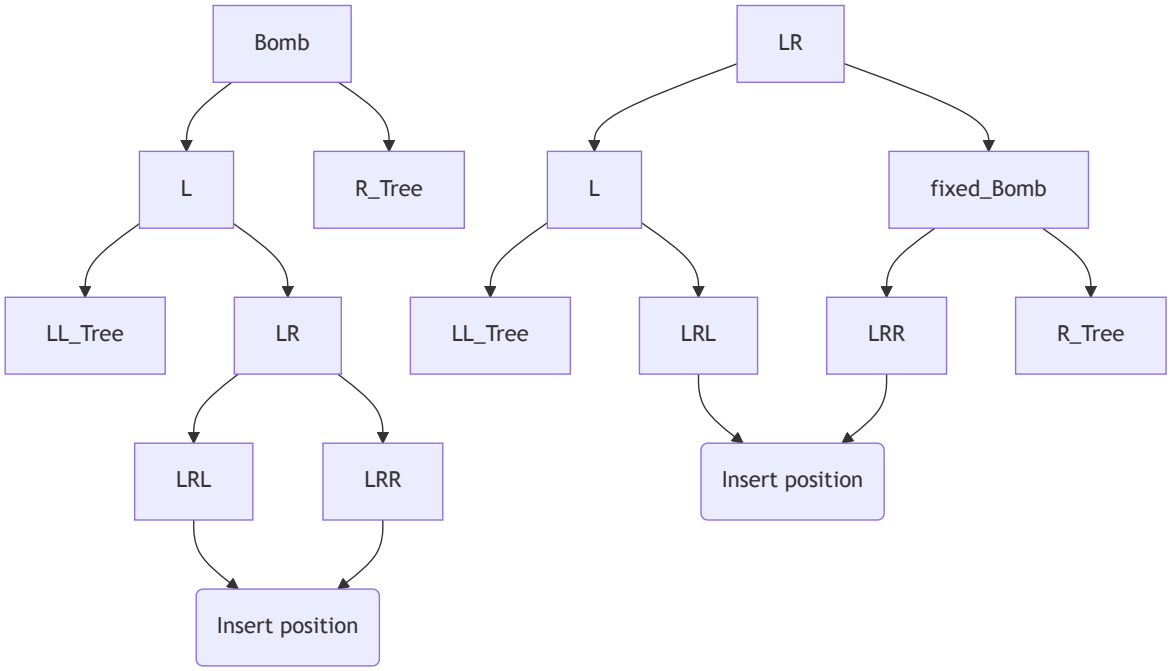
Proof:

Define notion H as the initial tree height, h(L) as the final tree height, we have:

$$h(L) = H$$

并且移动后的树仍然满足BST的基本特性 ■

case2: LR-Insert(下图左)



此时为了保证树的有序性，比较复杂(因此需要引入Rotation来方便理解？)

LRL和LRR仍然是可空树~~才不是懒得打_Tree子~~

我们依然可以有如下结论:

$$\mathrm{BF(bomb)} == 2, h(LL_{Tree}) = h(LRL) + 1 = h(R_{Tree})$$
$$h(LRL) = h(LRR)$$

此时L已经不适合做 "root" 节点了，我们选取LR作为新的 "root" 节点, 此时左子树的分配十分显然,但是右边的分配会略有些奇怪,~~但是其实是唯一合理的分配方式~~分配结果如上右

证明同上，略. 至此所有情况都证毕 ■

最后给出AVL Tree插入节点的可行代码

```
Tree insert(Tree T, int data){
    if(T == NULL){
        T = getNode(data);
        return T;
    }
    if(data > T->data ){
        T->Right = insert(T->Right,data);
        if(tHeight(T->Right) - tHeight(T->Left)>1){
            if(data < T -> Right -> data)
                T = rlRotate(T);
            else
                T = rrRotate(T);
        }
    }else{
        T->Left = insert(T->Left,data);
        if(tHeight(T->Left) - tHeight(T->Right)>1){
            if(data < T -> Left -> data)
                T = llRotate(T);
            else
                T = lrRotate(T);
        }
    }
    return T;
}
```

Splay Trees

Introduction to Splay

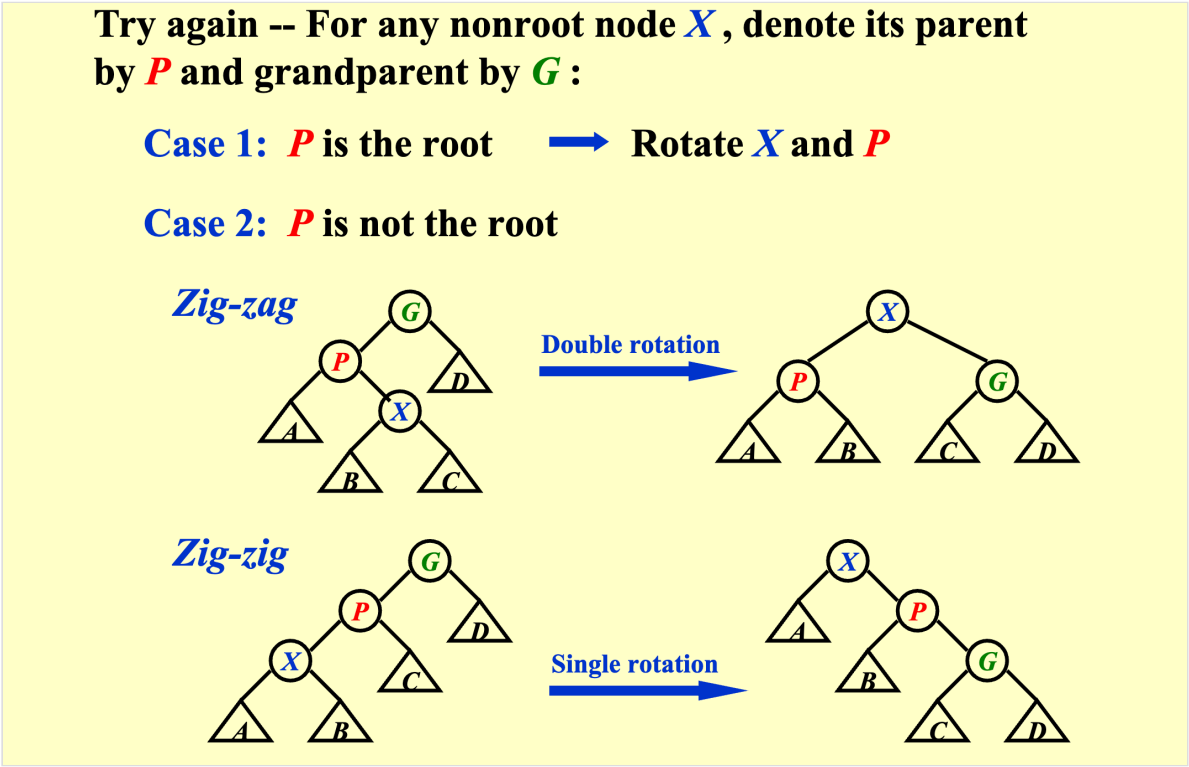
Spaly Tree 的核心是: 当我们只要试图访问 Splay Tree 中的节点，那么我们一定会在访问之后将其以一种方式移动到 `root` ，并且移动过程中 "无意识地" 使得这个树变得比较 "伸展"。据此引出了两个问题:

1. 怎么上移到 `root` , 即 operation to nodes
2. 怎么证明这个移动方式会使得这个树最后比较 "伸展", 即 proof of $O(f(n))$

Operations

关于介绍 Splay Tree 操作的具体步骤可以参考课件 Or 修佬的笔记

在这里只放一张经典老图(关于find)，以及声明：所有对 Splay Trees 的操作其实都需要 **find**, 并且所有操作的开销与 **find** 的开销是一个量级的。



其实可以更进一步，关于Splay Trees 的操作其实与rotate的次数是一个量级的, 这是因为

一次 rotate 一定只会让 X 上升一个高度，其操作量为 $O(height_{node})$, 这与find, delete, ... 一样

O(f(n)) of Splay operations

对任意一个节点，我们只关注 rotate 的操作量，那不妨记 $case1$的操作数为 C_1, $Zig-zag$ 的操作数为 C_2, $Zig-zig$ 的操作数为 C_3, 那么有:

$$\text{Number of operations} = C_2 + C_3 + 1$$

显然我们无法直接分析，但是我们能分析出这个操作的平均数吗？或者有什么方法可以得到平均数的上界呢？ 这里采用 $Amortized\ Analysis$

> 这里应该无法通过分析操作数的期望来解决，因为对于 find 完之后树仍然会很随机

Amoritized Analysis

摊还分析的想法十分朴素，即根据原有的一系列操作 $o_1, o_2, o_3, ...$, 以 "取长补短" 的方式构造出 $\hat{o}_1, \hat{o}_2, \hat{o}_3, ...$, 使得

$$\frac{\sum o_i}{n} \leq \frac{\sum \hat{o}_i}{n} \leq \max_{1 \leq j \leq n} \hat{o}_j \qquad (1)$$

于是原来操作数的平均值必然小于构造出的 "平摊代价" 的均值(当然也比平摊代价中的最大值小)，我们可以得到一个关于 $average-case\ bound$的上界，我们定义为 $Amortized\ bound$

基于这个想法，我们完善该方法的蓝图

$$\Delta_i := \hat{o}_i - o_i, \quad \sum \Delta_i \geq 0 \qquad (2)$$

我们希望(1)式能成立，那么转换成数学语言描述就得到(2), 也即 Δ_i 是基于假设定义的而对于用摊还分析处理的问题，我们一般找到确定的 o_i (即通过分解操作的方式，**把总操作的不可计算性与求和绑定**), 此时我们希望能构造出 Δ_i, 使得 $\hat{o}_i$ 有界 Or 同样有求和的性质，联系到 $\sum \Delta_i \geq 0$, 我们同样想通过差分的方式来构造 Δ_i，据此有:

$$\Phi(Status_i) - \Phi(Status_{i-1}) := \Delta_i \qquad (3)$$

我们将 $\Phi(Status_i)$ 称作势能函数，用来描述做完操作 i 之的一个状态量

$$\sum \Delta_i \geq 0 \Rightarrow \Phi(Status_{final}) - \Phi(Status_{begin}) \geq 0 \qquad (4)$$

$$\Phi(Status_i) - \Phi(Status_{i-1}) = \hat{o}_i - o_i \qquad (5)$$

(4)式是自然推导得出的，需要我们在定义势能函数时注意，~~其实不太重要，某些时候可以绕过这个条件~~，而(5)式很有意思——我们的 "平摊代价" 与实际操作的差，等于状态量因为这个操作的改变量。换种方式说，我们取长补短的过程，其实就是补偿了一个状态量的变化量, 这又巧妙的符合了一开始的设想。

以上两个式子主要是说明了势能函数定义是十分自然的，但是与我们求解 $Amortized\ Bound$ 即，怎么定义势能函数并无关联。如上所说，o_i 大致确定，我们希望 $\hat{o}_i$ 能求和，那么必然要求:

- $\Phi(Status_i) - \Phi(Status_{i-1}) + o_i \leq c$
- $\Phi(Status_i) - \Phi(Status_{i-1}) + o_i$ 是差分 Or 可放缩为差分
- 上述两个条件给我们定义势函数同样有启发性:
  是一个对整体状态量的刻画，同时会灵敏地对不同的操作做出差异化的反应，高代价操作会减少势能，而低代价操作增加势能

Amortized Bound Of Splay Tree

假设开始这个树只有一个节点，我们分析一个安全的包含 m 次操作的任意序列(find, insert, delete...). 定义势能函数: $\Phi(T_j) = \sum_{Node\ i \in T_j} \log V_i$, 其中 V_i 是以 i 为 root 的树的根节点数, 为了方便记 $\log V_i := R_j(i)$

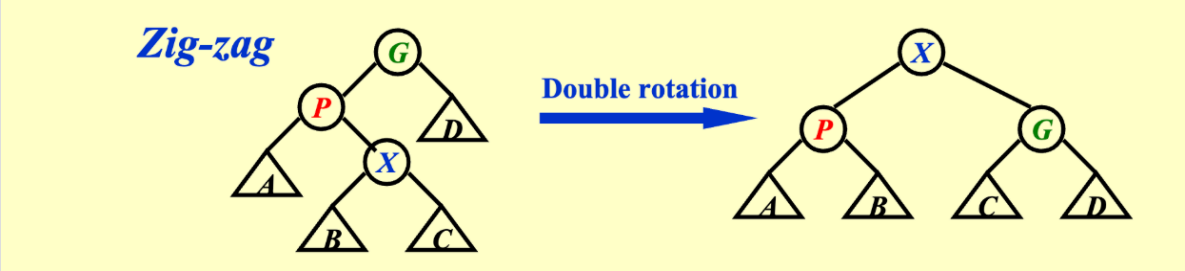
记这 m 次操作中的开销为 A_m, 由上对 Splay Tree 的分析知，$A_m = \lambda \cdot C_m$ 其中 C_m 为这 m 次操作中的 rotate 次数，进而有

$$C_m = \sum_{i=1}^{m} \sum_{\text{第}i\text{次操作中的}c_1\text{次数}} c_1 + \sum_{i=1}^{m} \sum_{\text{第}i\text{次操作中的}c_2\text{次数}} c_2 + \sum_{i=1}^{m} \sum_{\text{第}i\text{次操作中的}c_3\text{次数}} c_3$$

其中 c_1, c_2, c_3 分别指 $case1,\ case2,\ case3$(即$Zig-zag,\ \ Zig-zig$)的操作，且 $c_1 = 1,\ \ c_2 = 2, c_3 = 3$, 根据以上假设计算 $\hat{c}_i$:

$$\hat{c}_2 = R_2(X) + R_2(G) + R_2(P) - (R_1(X) + R_1(G) + R_1(P)) + 2 := RHS$$

以 $\hat{c}_2$ 为例(为方便起见，R 下标采用 1,2), 我们希望将 RHS 变成一个只跟 X 相关的式子, 这是因为每次操作中 G, P 都具有随机性，且不具有差分性质。同时尝试消掉常数，这是因为如果式子内含有常数，且最后能求和，那么必然要求知道具体的操作数。



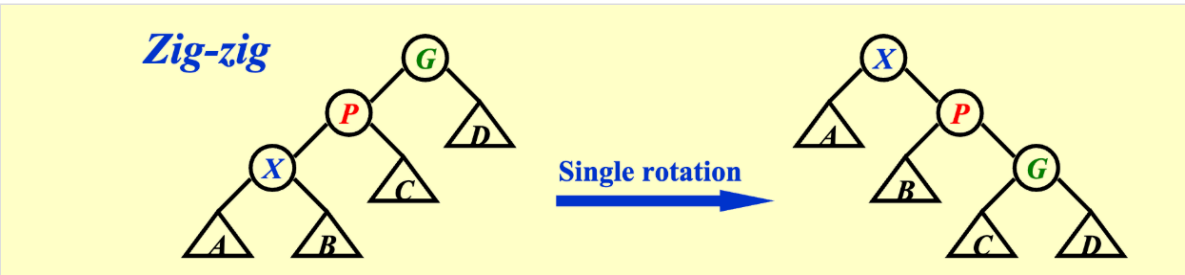
观察上图中的 P_{Right}，G_{Right}, 我们不加说明的给出下面的引理, ~~不知怎么想到的~~

$$2 + \log_2 a + \log_2 b \leq 2\log_2(a+b)$$

据此有

$$RHS = (R_2(X) - R_1(G)) + (R_2(G) + R_2(P) + 2) - R_1(X) - R_1(P)$$
$$From \text{ Lemma} \Rightarrow RHS \leq 0 + 2 \cdot R_2(X) - 2 \cdot R_1(X) \qquad (1)$$

同理可以得到相应的 $\hat{c}_3$ 并完成相应放缩



$$\hat{c}_3 = R_2(X) + R_2(G) + R_2(P) - (R_1(X) + R_1(G) + R_1(P)) + 2 := RHS$$
$$RHS = (R_2(G) + R_1(X) + 2) + (R_2(X) - R_1(G)) + R_2(P) - R_1(P) - 2R_1(X)$$
$$From \text{ Lemma} \Rightarrow RHS \leq 2R_2(X) + 0 + R_2(X) - R_1(X) - 2R_1(X) \qquad (2)$$

而 $\hat{c}_1$ 很容易放缩，并且要求也不严格(可以有常数，因为一共肯定有 m 次)

$$\hat{c}_1 \leq 1 + R_2(X) - R_1(X) \qquad (3)$$

由 (1), (2), (3) 记 X_j 为第 i 次操作的对象节点，我们可以得到

$$C_m \leq m + k \cdot \sum_{i=1}^{m} (R_{last}(X_i) - R_{begin}(X_i)) \leq m + k \cdot \sum_{i=1}^{m} R_{last}(X_i)$$

这里 k 可以是 3，而显然 $R_{last}(X_i) = \log n_i$, 其中 n_i 为操作 i 时的整个树的节点总数，进一步的，这 m 次操作中树必然有最大的节点数 N 且 $N \leq M$, 于是

$$C_m \leq m + 3m \log N = O(m \log N) \Rightarrow A_m = O(m \log N) \quad \blacksquare$$