AVL Tree

引入与定义

数据结构作用

1. 存储数据
2. 支持操作
    - 查找
    - 插入
    - 删除
    - etc.

考虑BST，其查找、插入和删除操作都是$O(h)$的，为了尽可能降低时间复杂度，我们会采取h尽可能小的建树方式，那么complete BT（完全二叉树）就符合这个性质。

但是由于在树的维护过程中，我们插入一个数据之后再将这个数变成完全二叉树的时间代价是$O(N)$的，其维护成本非常高，所以我们就放宽了一些限制条件，定义了**balanced binary tree**:

对于树的任意节点u，其左子树的高度h_L和右子树的高度h_R满足条件：$|h_L - h_R| \leq 1$

其中，$|h_L - h_R|$被称作这个节点的balance factor，而满足这个条件的搜索树被称为BBST，也称**AVL tree**。

证明

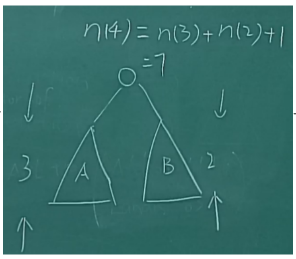
1. 这个条件虽然放宽了，但是仍然可以较快地做查找

LEMMA: A balanced binary tree with n nodes must have a height of O(logN)

proof:

主要需要证明 any BBT of height h has at least c^h (c is a constant) nodes

定义 $n(h)$: nodes in the smallest BBT of height h

以此为例

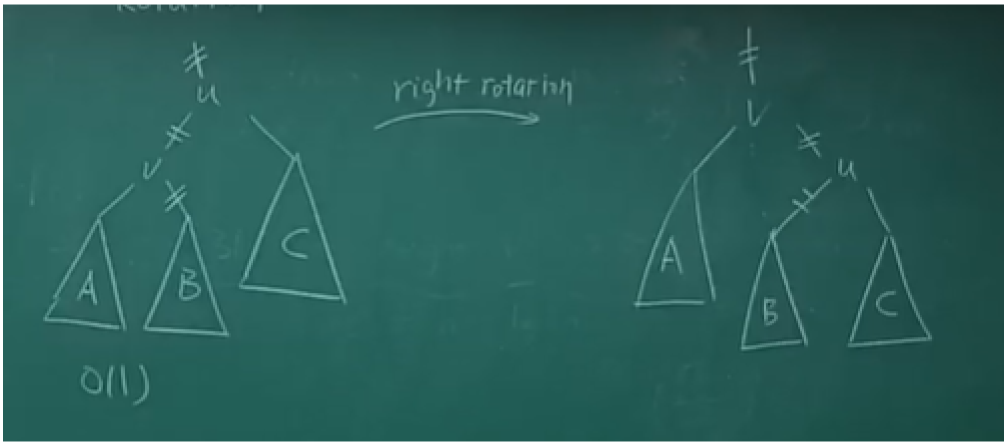
可推得 $n(h) = n(h-1) + n(h-2) + 1$ for $n \geq 2$

由递推式可得 $n \approx (\sqrt{5}+1/2)^h$

故 $n \geq c^h \rightarrow h \leq log_c n$

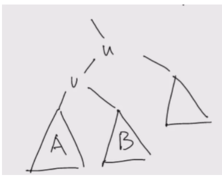
2. 这个性质的维护代价小

先理解一个操作：

Rotation: 

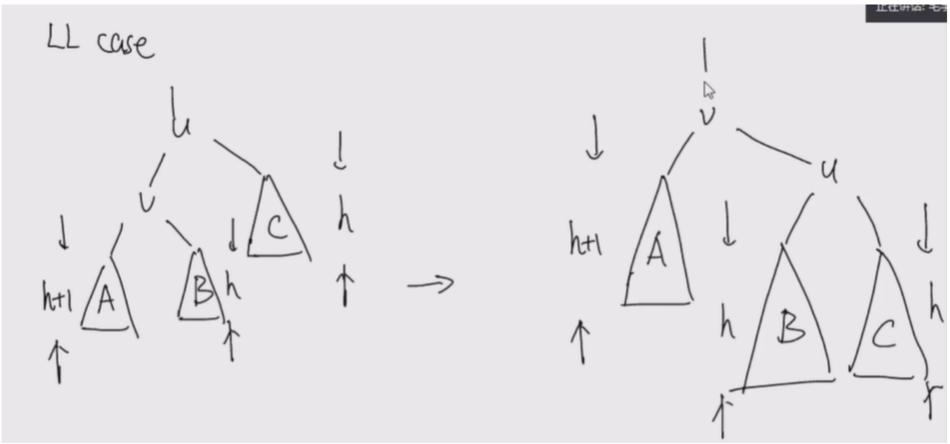
- 只需要改3个指针，是$O(1)$时间复杂度的
- 若旋转前满足BST性质，旋转后仍然满足
- 左右子树的高度发生了变化
- left rotation就是right rotation的逆操作

考虑不同操作的维护：

- 插入
  1. insert as in BST
     - 哪些点的平衡可能会受到影响？ - 被插入节点的ancestor
     - 变得不平衡之后高度差（平衡因子）是多少？ - 只可能是2
  2. restore the balance
     - 选取路径上最低的不平衡节点u
     - 以$h_L - h_R = 2$（**L case**）为例讨论（相反的情况**R case**类比即可），左右子树的高度分别被记为$h+2$和h
     - 考虑左子树的根节点v，其左右子树分别记为A和B：

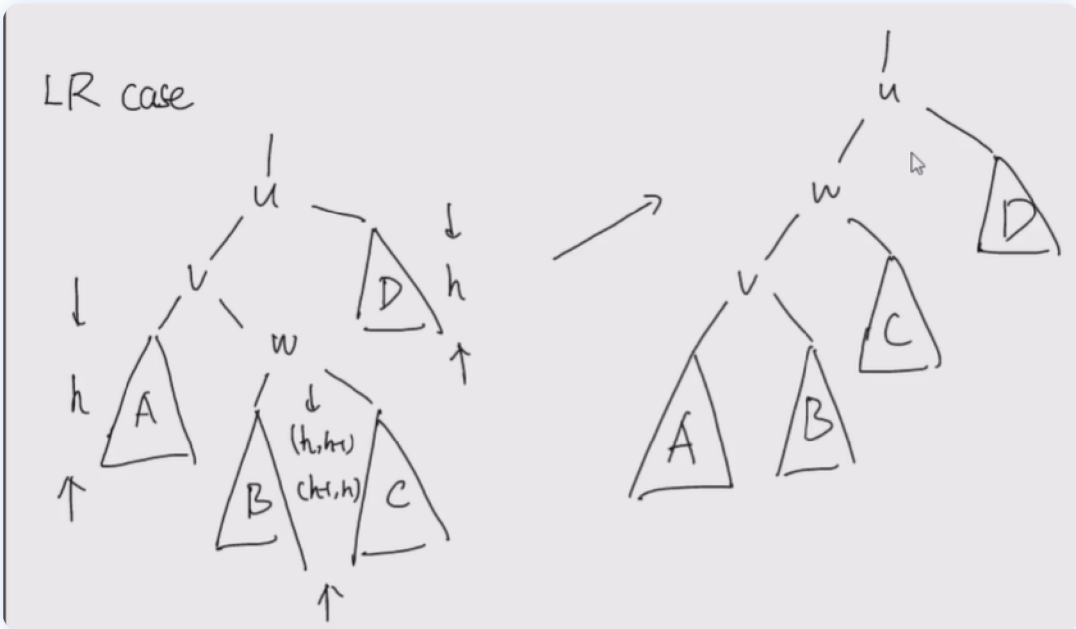
     - 由于在插入操作前为平衡树，故h_A和h_B有且仅有两种可能性：
       - $h_A = h + 1, h_B = h$ - **LL case**

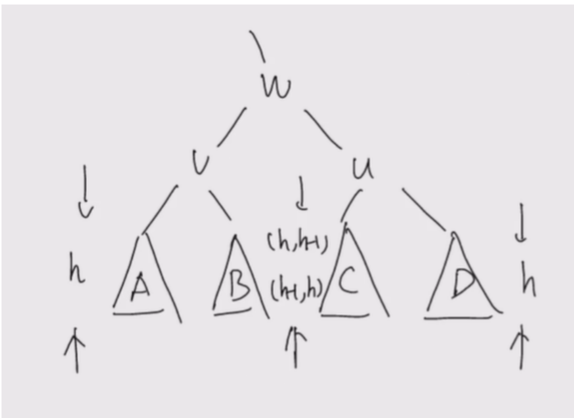
把v拎起来作旋转即可



    - $h_A = h, h_B = h+1$ - LR case

      将B子树再分成以w为根节点的左右子树，把w拎起来做两次旋转：



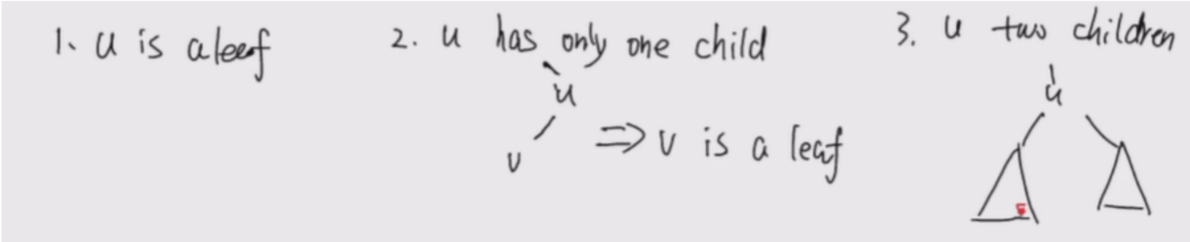


此时三个节点都平衡了

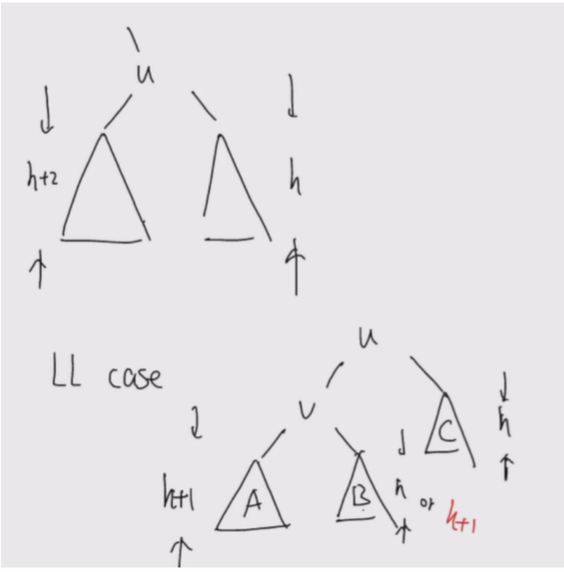
  - 那么总共需要旋转多少次可以修复所有节点?
    - 做插入前树的高度为$h+2$，插入后（不平衡）为$h+3$，把u修复平衡后高度为$h+2$，因此对上面的原先不平衡的ancestors节点，在最低不平衡节点被修复后也都修复了，所以总共1或2次（即修复最下面的不平衡点）就可以
- 总时间复杂度$O(logN)$

- 删除

1. delete as in BST
    - BST deletion in BBST is essentially removing a leaf
    - proof:

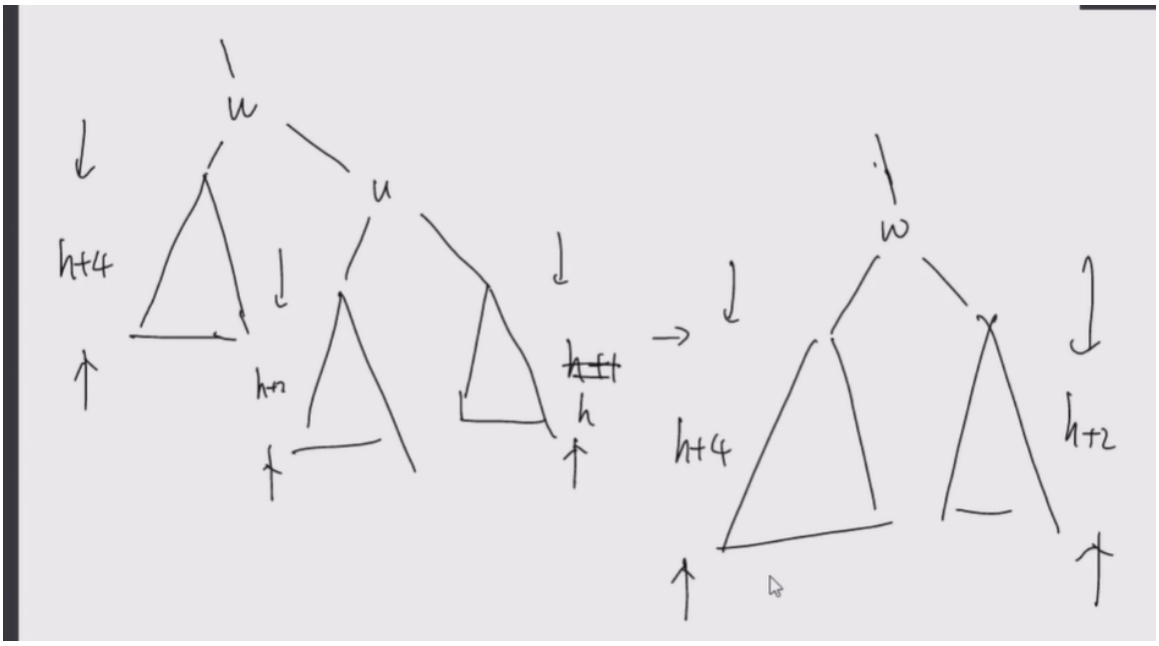


    - 如果删掉一个leaf，可能会使得某一个子树高度下降，导致**至多一个节点不平衡**（刚刚做完删除操作），且高度差为2
2. restore the balance
    - **LL case**



只是B子树高度可能性不同，但是把u变平衡的操作和插入相同
    - **LR case**同理
    - 那么总共需要旋转多少次可以修复所有节点?

- 虽然最开始只有一个节点不平衡，但是修复那个节点的过程可能会导致上面的节点不平衡：



- 考虑到不平衡的向上传递，删除+修复的时间复杂度为 $O(logN) + O(1) * O(logN) = O(logN)$

树的高度和维护代价都是O(logN)数量级的

红黑树

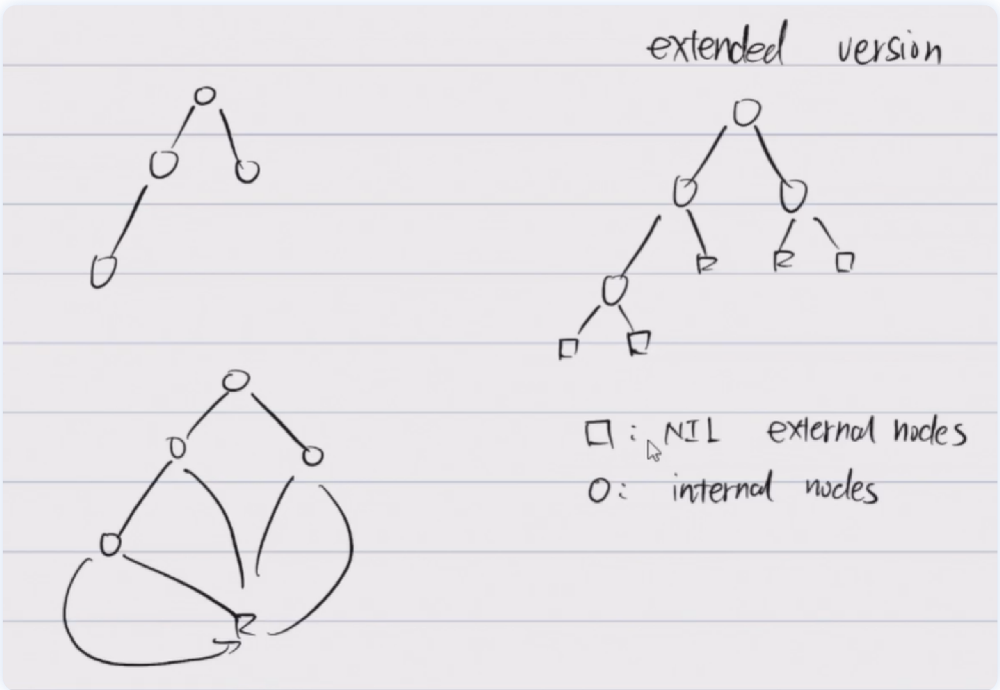
引入

complete binary tree: 每一个leaf的height相差最多为1

→ 红黑树：每一个leaf的height相差最多是两倍

实现方法：染色

树的拓展表现形式



- 将空的节点合并为一个节点
- 方块数量 = 圆点数量 + 1

定义

A red black tree is a BST whose extended satisfies the following properties:

1. node color: red or black
2. root is black
3. leaves (NIL) are black
4. children of red must be black
5. for each node v, all descending paths from v to leaves contain **the same number(excluding v)**(→black height of v: **bh(v)**) of black nodes → bh(T) = bh(root)

结合45可推：树上的黑点比红点多且每条路的黑点数目一致→最长路的height不会超过最短路的两倍，且$h(T) \leq 2 * bh(T)$

证明

1. 树的高度是$O(logN)$数量级的

LEMMA: A RBT(in extended version) with n internal nodes has height of at most $2log_2(n+1)$.

红黑树

Proof:

首先我们定义，一棵树T中的节点u，记Tu为以u为root的子树，$size(Tu)$则为此子树内部节点的个数。

Tu :

size (Tu) = # internal nodes

我们需要证明：$\underbrace{size(Tu) \geq 2^{bh(u)} - 1 \text{ for any } u}$ $\rightarrow size(T) \geq 2^{bh(T)} - 1 \rightarrow bh(T) \leq log_2(n+1) \rightarrow h(T) \leq 2log_2(n+1)$

归纳法： base case: $h(Tu) = 0$

u:  $\underbrace{size(Tu) = 0 \quad bh(Tu) = 0}$

$\Rightarrow size(Tu) \geq 2^{bh(u)} - 1$

Inductive hyposize:

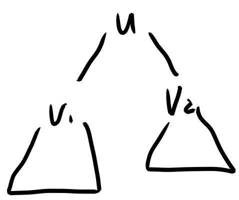
Assume that for all Tu with $h(Tu) \leq k$, $size(Tu) \geq 2^{bh(u)} - 1$

↑ 用

高度 $\leq k$.

↑

Inductive Step. when $h(Tu) = k+1$.



$\Rightarrow size(Tu) = 1 + size(T_{V_1}) + size(T_{V_2})$

$\geq 2^{bh(V_1)} - 1 + 2^{bh(V_2)} - 1 + 1$

$\geq 2 \cdot 2^{bh(u)-1} - 1$

又 ∵ $\begin{matrix} bh(V_1) \\ bh(V_2) \end{matrix} \geq bh(u) - 1$

$= 2^{bh(u)} - 1$

2. 维持性质的代价是 log(N) 级别的

① Insertion

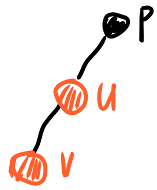
步骤

1. insert as in BST

2. make the new node red

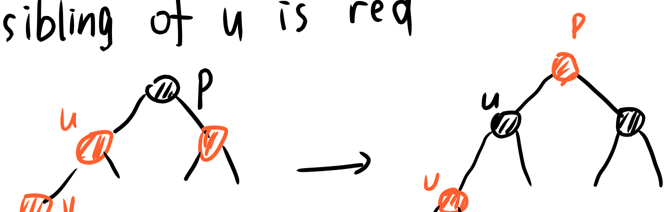
考虑插入节点的 parent —— black 不用任何维护

└ red 维护，分成左右两种情况，时对称，考虑 new node 在左的情况



3. case 1:

sibling of u is red



前5条性质依然 √

考虑 (a) p is root, mark p as black, done

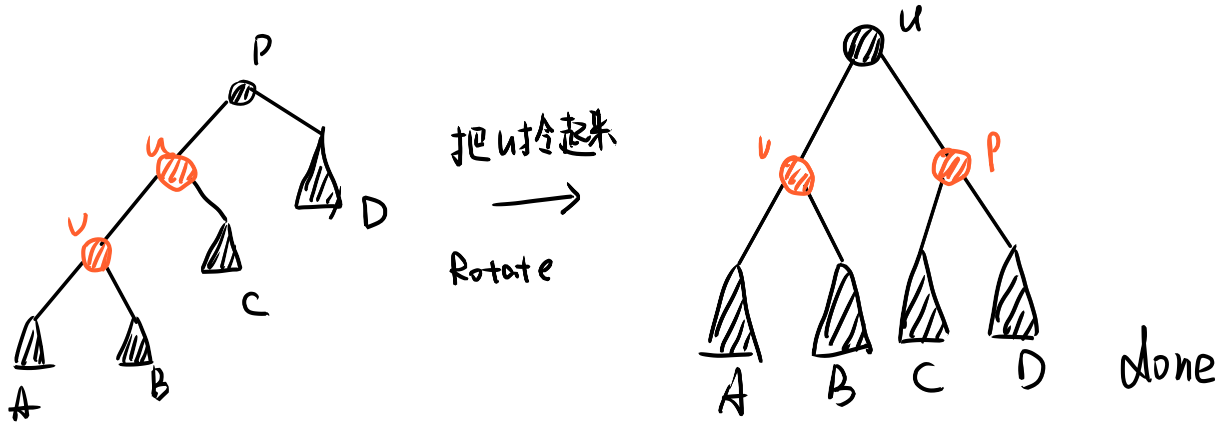
(b) parent of p is black, done

(c) parent of p is red, violation goes upwards 继续迭代

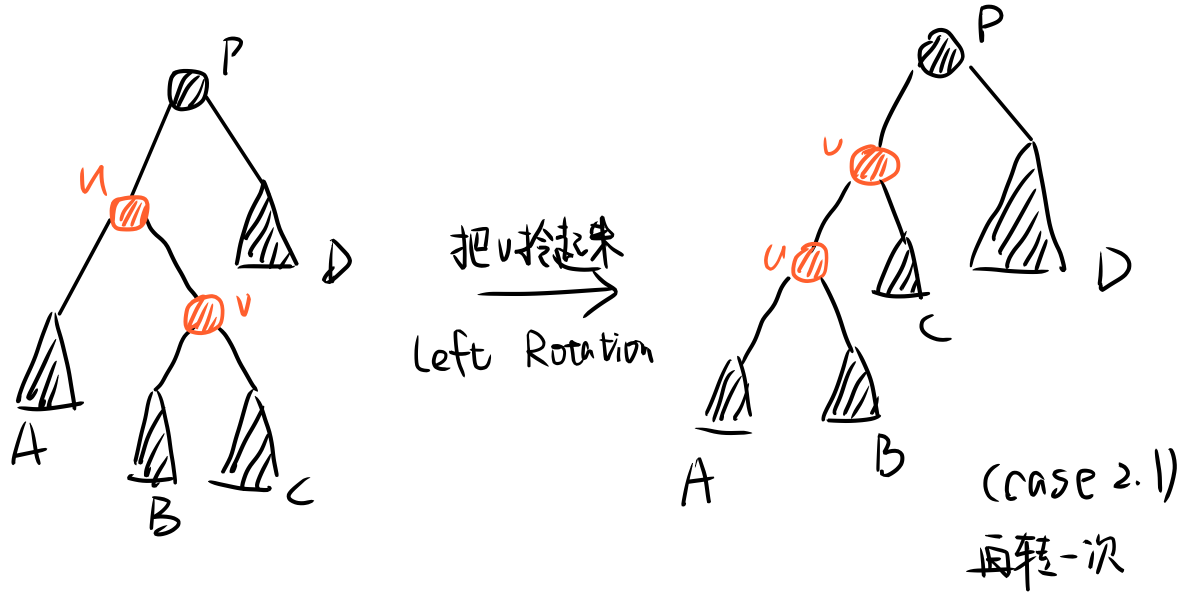
case 2.

sibling of u is black

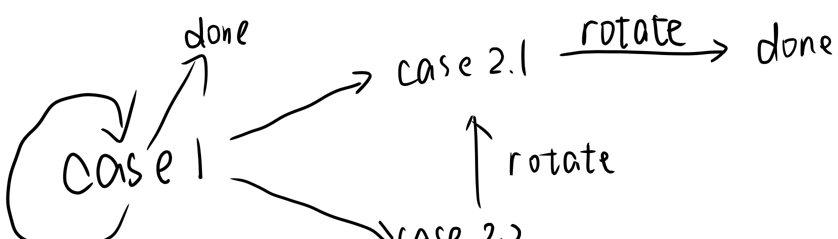
case2.1 v is left child



case 2.2 v is right child of u.



流程图：



case 2.2

时间复杂度： ① 插入 $\log(N)$

② "向上推"流程最多为树高 $\log(N)$

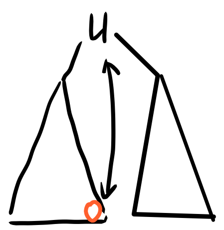
③ rotate 最多两次

$$\Rightarrow \text{总} = O(\log N) + O(\log N)\cdot O(1) + O(1)\cdot 2$$
$$= O(\log N)$$

② deletion

步骤：

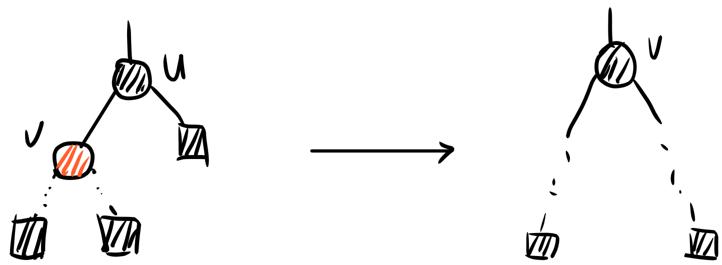
1. delete it as in BST



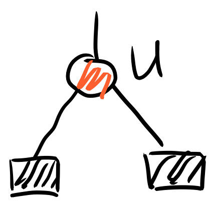
* 只交换 key，不交换颜色

→ deleted node u has one most child (excluding NULL)

2、case 1:

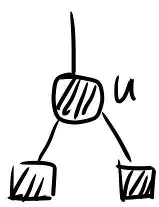


case 2:



直接删除即可

case 3.

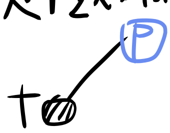


直接删会使得 RBT 的第5条性质不满足

⇓

← double black

处理双层黑问题



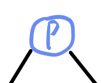
left | right

↑

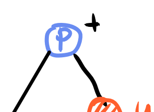
对称，仅考虑此情况

双层黑问题： case 1 the sibling of $\oslash^+$ is black

case 1.1 children of u are all black



把u的黑向上推一层



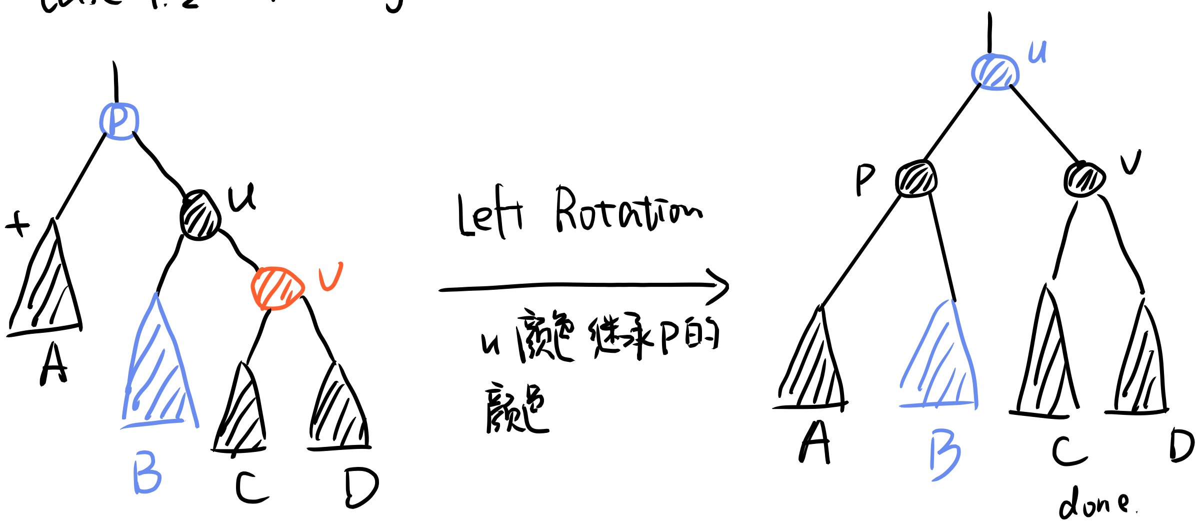
(a) p is the root，直接把加号去掉，done

(b) p is red，mark p as black，done

(c) p is black (not root)，p becomes double black

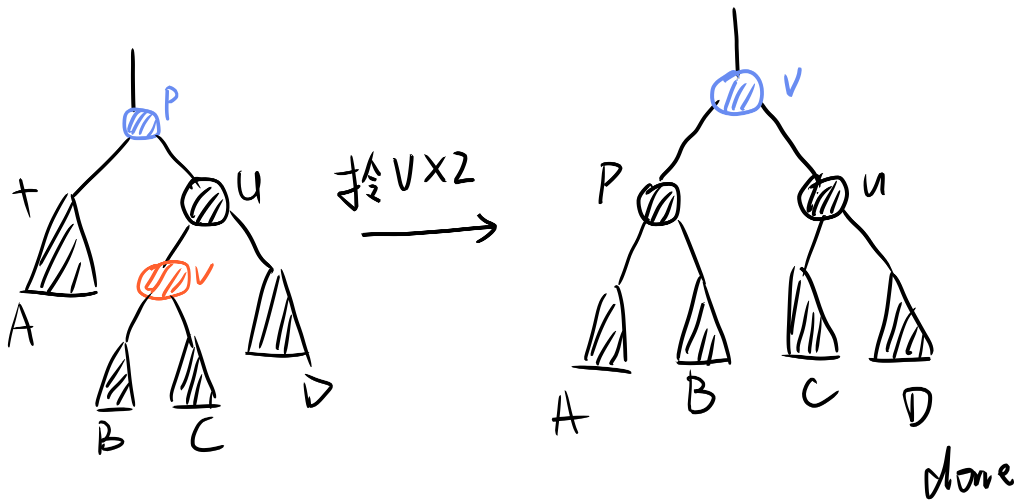
把问题向上推了一层

case 1.2 the right child of u is red

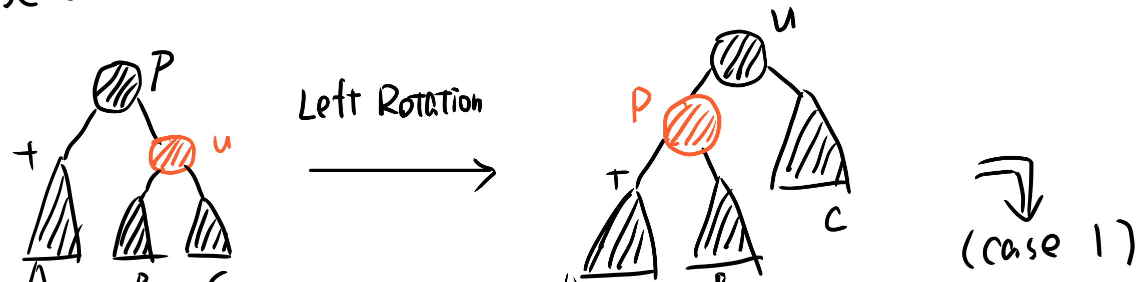


case 1.3 the right child of u is black

the left child of u is red



Case 2. the sibling of ⊘$^{+}$ is red



A B C A B

流程图:

$$
\begin{tikzcd}
 & \text{case 1.2} \arrow[dr, "\text{single rotation}"] & \\
\text{case 2} \arrow[ur] \arrow[dr] \arrow[d] & \text{case 1.1} \arrow[l, "(c)"'] \arrow[u] \arrow[loop above, "(c)"] \arrow[d] \arrow[r, "(a)(b)"'] & \text{done} \\
\text{case 1.1 (b)} \arrow[d] & \text{case 1.3} \arrow[ur, "\text{double rotation}"'] & \\
\text{done} & &
\end{tikzcd}
$$

$$T \begin{cases} \text{向上推 } O(h) \\ \text{旋转最多3次 } O(1)\cdot 3 \end{cases}$$

$$T = O(h) + 3\cdot O(1) = O(\log N)$$

	AVL	RBT
height	$\approx \log_{1.618} N$	$\approx 2\log_2 N = \log_{\sqrt{2}} N$
insertion	2# rotation	2# rotation
deletion	$O(\log N)$ #rotation	3# rotations

均摊分析

previous: worst-case bound for a single operation

amortized: worst-case bound for a sequences operation
↓
from empty structure

eg.

Dynamic Array

- A[i]
- Insertion

Insertion

- $O(n)$ space

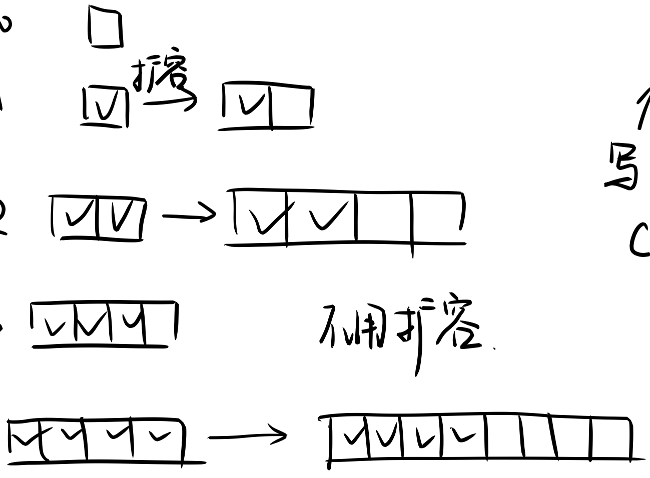
$m=0$

$m=1$

$m=2$

$m=3$

$m=4$



$m=5$

$m=6$

$C+2C+C = C+3C$

写 新建 粘贴 写 扩容

$C+4C+2C=C+6C$

C

Insertion　一次的 worst case: $O(n)$　但发现扩容很贵 但次数少

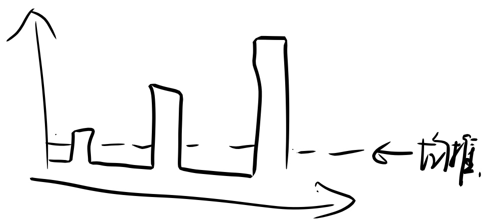
定义 $T(m)$ = cost of the worst sequence of insertion.

$$= \underset{\text{写}}{cm} + \underbrace{2^0\cdot 3c + 2^1\cdot 3c + 2^2\cdot 3c + \cdots + 2^i\cdot 3c}_{\text{扩容}} \quad i=\lfloor \log_2 m \rfloor$$

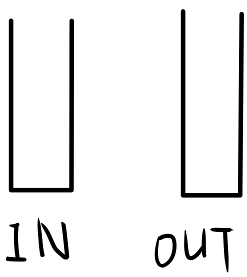
$$= c\cdot m + 3c\cdot(2^{\lfloor \log_2 m\rfloor+1}-1) \le cm + 6cm = 7cm$$

平均每次操作 $\frac{T(m)}{m} = 7c \leftarrow O(1)$

法1　amortized cost



eg2. 考虑 Two-Stack Queue 数据结构



enque(x):

IN.PUSH(x)

deque(x):

```
If OUT not empty
    OUT.pop()
else if IN not empty
    while IN not empty
        x = IN.pop()
        OUT.push(x)
OUT.pop().
```

enque: $O(1)$

deque: # elements moved

"在一次很贵的 deque 前一定有很多次便宜的 deque."

考虑之前的做法 - 什么是 worst sequence? 不好找

法2. Accounting Method

amortized cost = actual cost + credit 存取款变化 [存+ 取−]

$\sum$ amortized cost $= \sum$ actual cost $+ \sum$ credit

$\sum$ credit ≥ 0 "不欠钱"

$\Downarrow$

$\sum$ amortized cost $\geq \sum$ actual cost

⇒ 考虑 eg2.

	actual cost	credit	amortized
enque	c	2c	3c
deque	2c · #elements moved + c	−2c · #elements moved	c → $O(1)$ amortized cost

要保证不欠钱 ⇒ 证明: 每一个 enque 之后的 deque 最多 2c.

用函数记录总存款的量:

$\Phi(i)$: total # credits in the bank after the ith operation

⇒ 改写上面的 accounting method:

amortized cost of O_i = actual cost of O_i + $\Phi(i) - \Phi(i-1)$ 就是前面的 credit.

$\Phi(i) \geq \Phi(0)$ (不欠钱)

↑

势能函数 potential function 法3

$\Phi(Q) = (\#\text{ element in Stack } I_n)\cdot 2c$ (定义eg2的势能函数)

for any sequence of operation $O_1, O_2, \ldots, O_m$

$Q_0 \quad Q_1 \quad Q_2 \quad \ldots \quad Q_m$

Q_0: empty

$\Phi(Q_0)=0$, $\Phi(Q_i)\geq 0$

if O_i = enque, $\hat{d_i} = d_i + 2c = c + 2c = 3c$

if O_i = deque, $\hat{d_i} = c + 2c\cdot\#\text{elements moved} - \#\text{elements moved}\cdot 2c = c$

给出均摊费用定义: Given k types of operation $1,\ldots,k$ with actual cost $T_1(D), \ldots, T_k(D)$, we say they have amortized cost $A_1(D), \ldots, A_k(D)$, if

通常是与当前数据结构D相关的函数, 例如 insertion of a BST $T(D)$= height of D)

for any $m>0$, for any sequence of m operations $O_1, \ldots, O_m$ ($D_1, D_2, \ldots, D_m$),

$$\sum_{i=1}^{m} A_{type(O_i)}(D_{i-1}) \geq \sum_{i=1}^{m} T_{type(O_i)}(D_{i-1})$$

---

potential function: $\Phi: \mathcal{D} \to \mathbb{Z}$

定义

- $\Phi(D) \geq \Phi(\text{empty})$ for any $D \in \mathcal{D}$
- for any $t \in [1,2,\ldots,k]$, $A_t(D) = T_t(D) + \Phi(D') - \Phi(D)$

D': after operation t

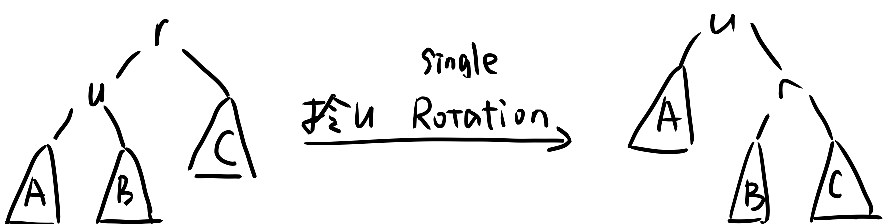
Splay Tree

- :($\Theta(n)$ in worst case
- :) $O(\log n)$ amortized cost
- :) easy to implement
  - no extra space
  - adaptive → m次连续find(4) ⇒ $O(m+\log n)$

(基于BST的)

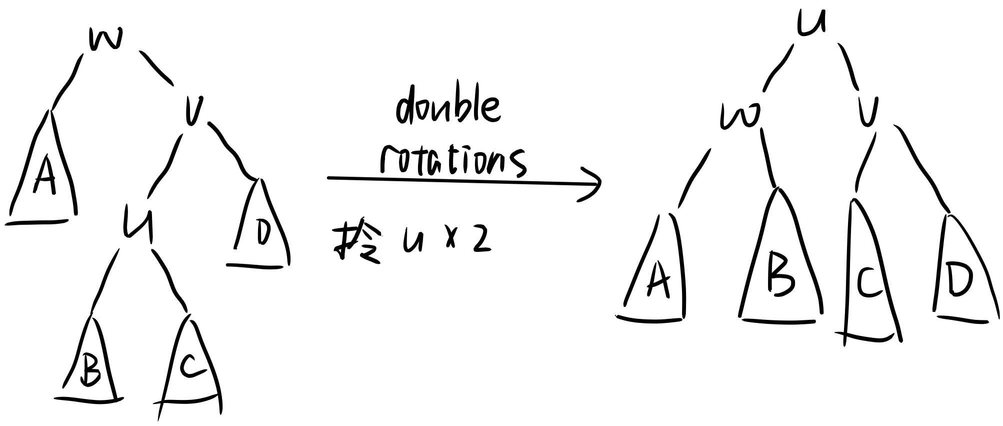
splay(u): repeat the follows until u is the root.

case 1. u is a child of the root (对称情况略)

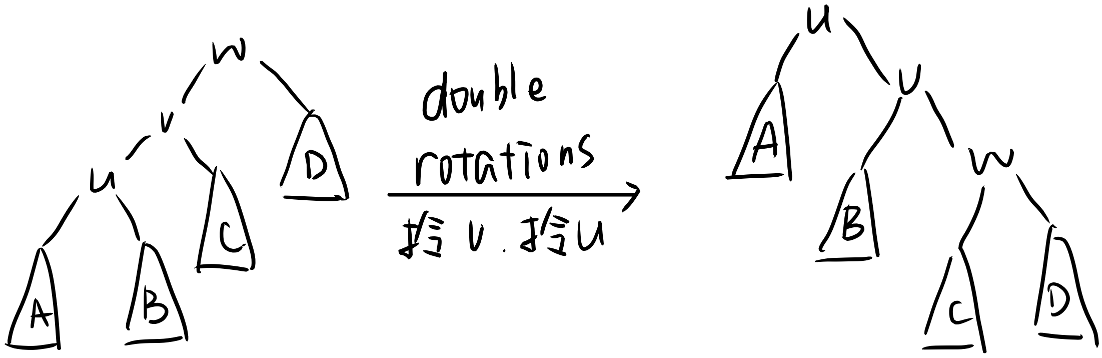


case2. u has a grandparent

case 2.1 zig-zag (对称情况略)



case 2.2 zig-zig (对称情况略)



findkey:

1. find as in BST
2. splay the node you found

Insert:

1. insert as in BST
2. splay the new node

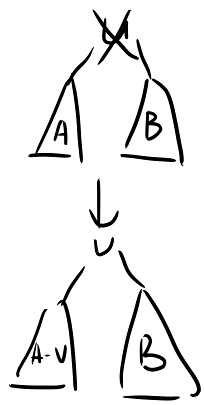
delete(u):

1. splay(u)
2. if u has only one child → delete u directly

else u has two children → ① delete u

② splay the largest element v in A

③ attach B to v



证明以上三种操作均摊费用为 $O(\log N)$

Observation

actual cost of each opration is $\underset{\text{constant}}{c} \cdot \#\text{rotations}$

amortized cost $= c \cdot \lg n \leftarrow$ goal

$\Delta\Phi = c \cdot \lg n - c \cdot \#\text{rotations}$ 由目标推出势函数变化值

↓定义势函数

Given a BST T, for each $u \in T$, $size(u) = \#\text{nodes in } T_u$ 以u为root的子树的结点数量

rank $r(u) = \lg(size(u)) \rightarrow$ 定义势函数:

$\Phi(T) = c \cdot \sum_{u \in T} r(u)$, $\Phi(\text{empty}) = 0$, $\Phi(T) \geq 0$

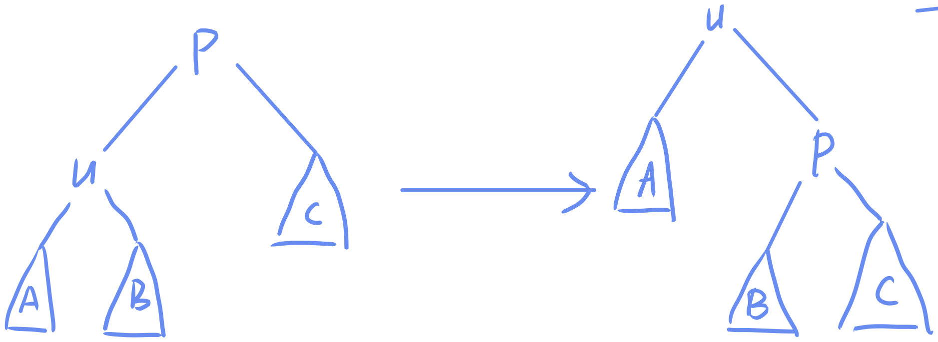
LEMMA: let T be a splay tree, let $u \in T$

let T' be the tree obtain from T by performing splay(u)

$\Phi(T') - \Phi(T) \leq 3c \cdot [r'(u) - r(u)] - 2c \cdot (\#\text{rotations} - 1)$

$r'(u)$: rank of u in T' $\#\text{rotations}$: during splay(u)

proof: case 1. u is a child of root

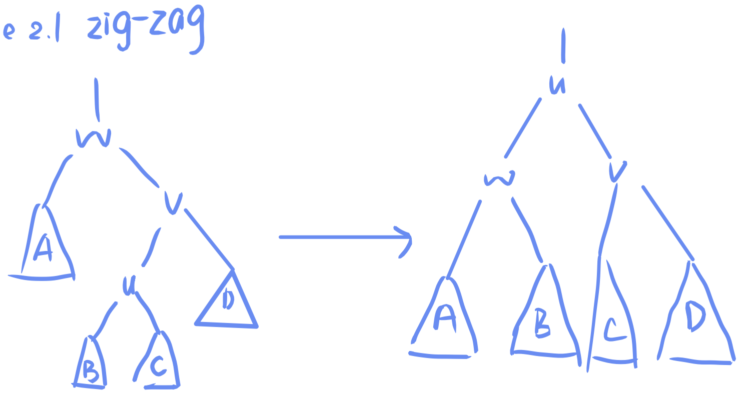


$\frac{\Delta\Phi}{c} = r'(u) - r(u) + r'(p) - r(p)$

$\leq r'(u) - r(u)$

$\leq 3[r'(u) - r(u)] - 2(\#\text{rotation} - 1)$

case 2.1 zig-zag



相消,抵消

$\frac{\Delta\Phi}{c} = r'(u) - r(u) + r'(v) - r(v) + r'(w) - r(w)$

$= r'(v) + r'(w) - r(u) - r(v)$

$\leq r'(v) + r'(w) - 2r(v)$

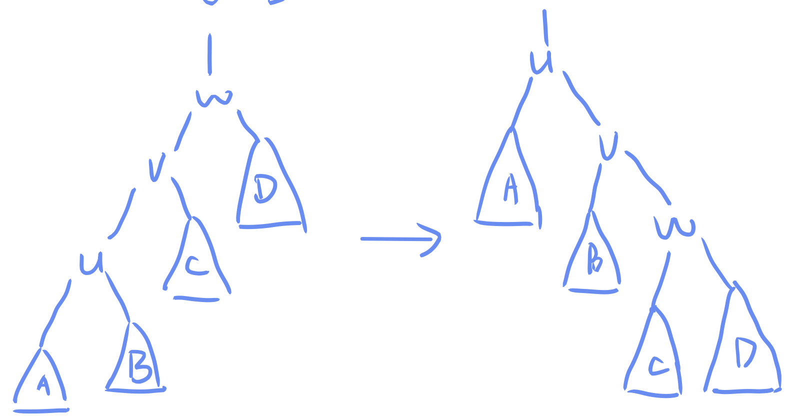
$size(v') + size(w) + 1 = size(u')$

$\Rightarrow r'(v) + r'(w) \leq 2r'(u) - 2$

$\leq 2r'(u) - 2r(u) - 2$

$\leq 3(r'(u) - r(u)) - 2(\#\text{rotations} - 1)$

case 2.2 zig-zig



$\frac{\Delta\Phi}{c} = r'(u) - r(u) + r'(v) - r(v) + r'(w) - r(w)$

$\leq r'(w) - r(u) + \underbrace{[r'(v) - r(v)]}$ 放缩成 $r'(u) - r$

$\leq r'(w) + r'(u) - 2r(u)$

$\leq r'(w) + r(u) + r'(u) - 3r(u).$

$size'(w) = |C| + |D| + 1$
$size(u) = |A| + |B| + 1$
$size'(w) + size(u) + 1 = size'(u)$
$\Downarrow$
$r'(w) + r(u) \leq 2r'(u) - 2$

$\Rightarrow \leq 3[r'(u) - r(u)] - 2$

$\leq 3[r'(u) - r(u)] - 2(\# rotation - 1).$

接下来使用LEMMA证明三种操作的均摊费用:

① findkey:

1. find as in BST
2. splay the node you found

actual cost = $c \cdot \#$ rotations

$\Delta\Phi = 3c \cdot [r'(u) - r(u)] - 2c \cdot (\# rotations - 1)$

$\leq 3c \cdot \lg n - 2c\ \# rotations + 2c$

amortized cost $\leq 3c \cdot \lg n - 2c \cdot \# rotations + 2c + c \cdot \# rotation$

$\leq 3c \cdot \lg n + 2c \leq 5c \cdot \lg n$

② Insert:

1. insert as in BST
2. splay the new node

actual cost = $c \cdot \#$ rotations

$\Delta\Phi = \#rotations + 3c \cdot [r'(u) - r(u)] - 2c(\# rotations - 1)$

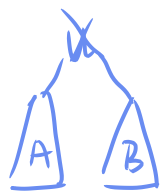
第一步的$\Delta\Phi$ 其实是 h
这里 h = #rotations

$\leq 3c \cdot \lg n - (2c - 1)\# rotations + 2c$

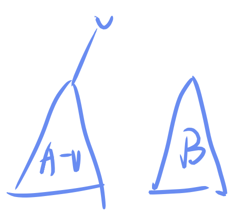
$\leq 3c \cdot \lg n + 2c \leq 5c \cdot \lg n$

③ delete(u):

1. splay(u) $\Delta 1$
2. if u has only one child → delete u directly → 和insert一样
   else u has two children → ① delete u $\Delta 2$
   ② splay the largest element v in A $\Delta 3$
   ③ attach B to v $\Delta 4$



$\Delta_2 = -\lg(|A| + |B| + 1)$
(rank(u))



$\Delta 4 = \lg(|A| + |B|) - \lg(|A| + 1)$

$\Delta 2 + \Delta 4 \leq 0$ 可放缩.

仅考虑 $\Delta 1+\Delta 3$：

$\Delta\Phi \le \Delta 1+\Delta 3$

amortized cost $\le$ actual cost $+ \Delta\Phi$

$\le$ #rotations in step1 $+ \Delta 1 +$ #rotations in step2 ② $+ \Delta 3$

$\le 3c \cdot \lg n + 2c + 3c \cdot \lg n + 2c \le 10c \cdot \lg n$