第九章 查找 （2）

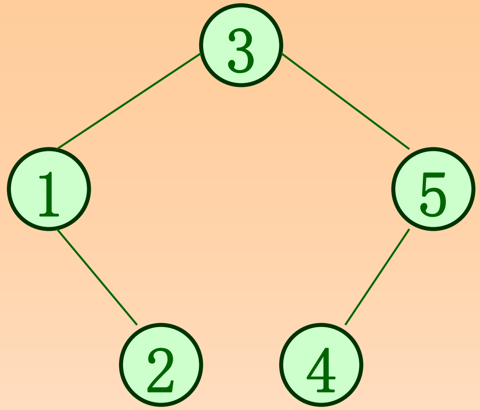
二叉查找树

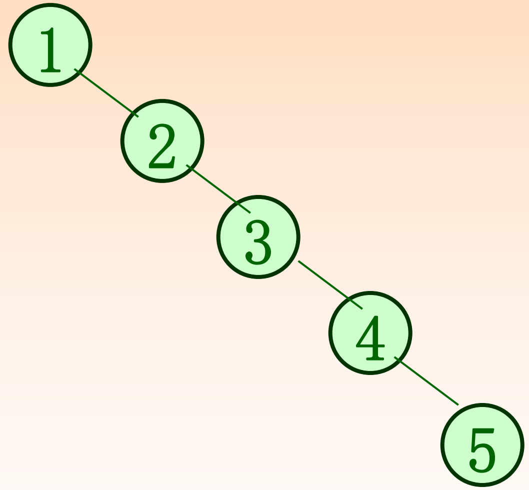
平衡二叉树

Treap(***)

回顾:二叉查找树

依次插入 3，1，2，5，4





依次插入 1，2，3，4，5

在最坏情况下：

树深h = O(n)。

能否让最坏情况下h=O(log n)？

平衡二叉树的定义

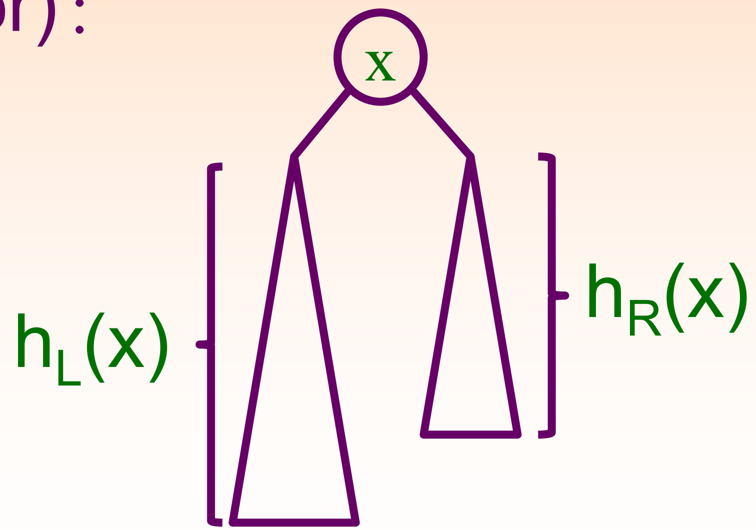
Adelson-Velsky和Landis发明了一种self-balancing BST

称作平衡二叉树（或AVL树)。

定义：平衡因子(balance factor)：

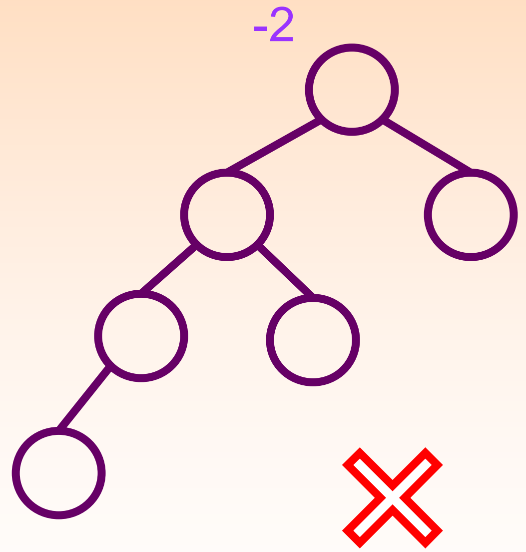
- ★ $BF(x) = h_R(x) - h_L(x)$
  - ❖ $h_R(x)$:节点x的右子树的高度
  - ❖ $h_L(x)$:节点x的左子树的高度
- ★ $BF(x) \in \{-1,0,1\}$

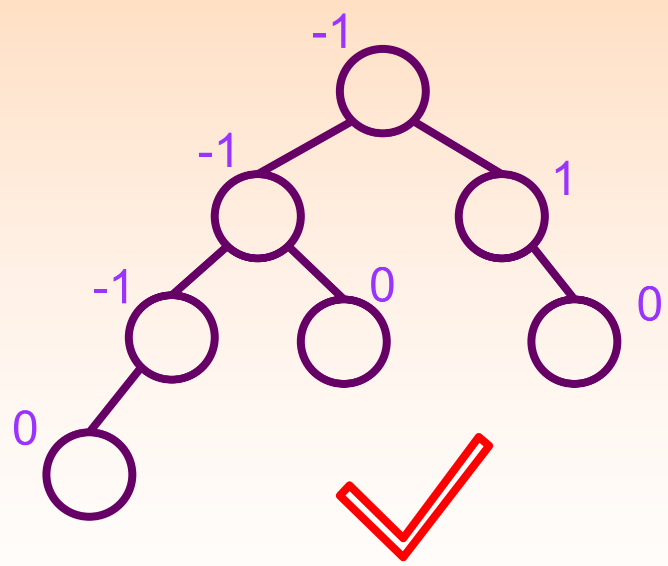
  ➔ x是平衡的(balanced).



平衡二叉树的定义

如果排序二叉树T的所有顶点都是平衡的（也就是说说有的平衡因子都是在{-1,0,1}中）那么T称作平衡二叉树。





平衡二叉树的一个重要性质

定理：如果树T是n个节点的平衡二叉树。

那么它的深度$h=O(\log n)$。

证明： 设N_h表示 深度为h的平衡二叉树最少包含多少个节点。$N_1=1, N_2=2. N_h=N_{h-1}+N_{h-2}+1$

即 $N_h+1= (N_{h-1}+1)+ (N_{h-2}+1)$

记 $A_h=N_h+1$。则$A_1=2,A_2=3,A_h=A_{h-1}+A_{h-2}$。

所以$A_h= F_{h+2}\geqslant \Phi^{h+2}/5^{1/2}-1.$ $\Phi=(1+5^{1/2})/2$

➔$n+2\geqslant N_h+2 \geqslant A_h+1 \geqslant \Phi^{h+2}/5^{1/2}$

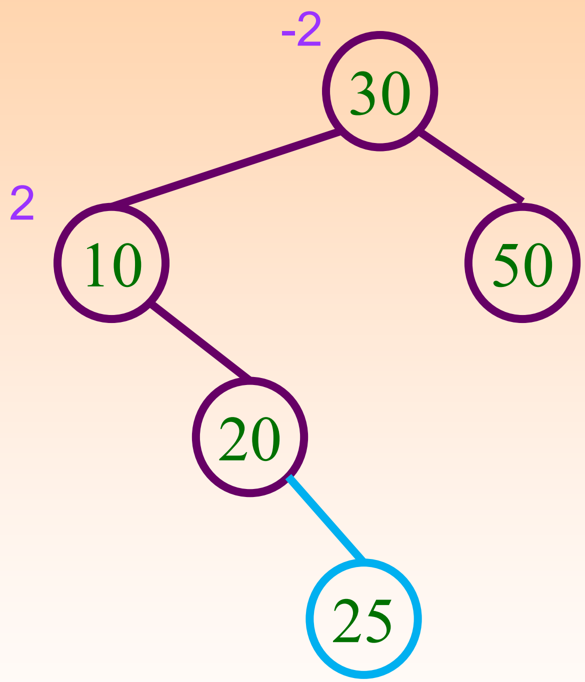
➔$\Phi^{h+2}\leqslant 5^{1/2}(n+2)$ ➔ $h = O(\log(n))$.

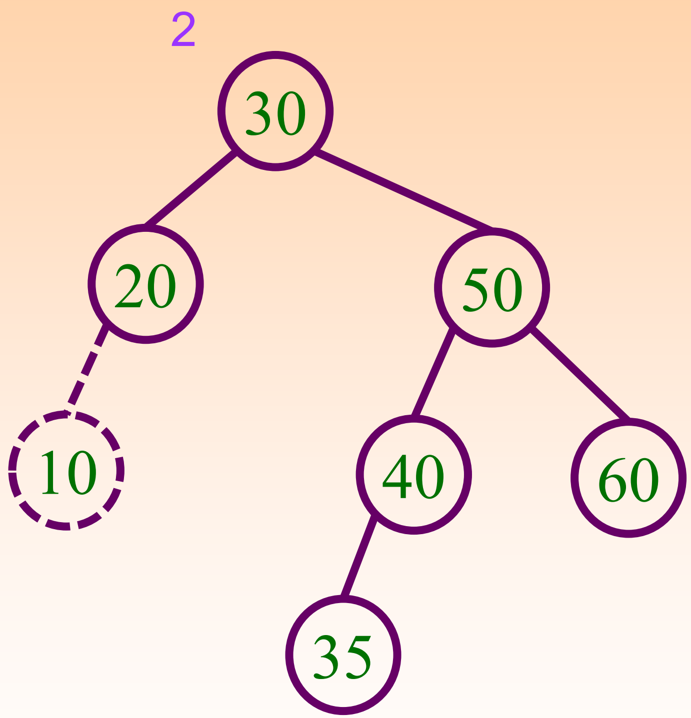
平衡二叉树的基本操作

- § 查询 Lookup(int key)
  - ★与排序二叉树一致。复杂度为O(log n)
- § 插入Insert(int key)
  - ★插入以后，平衡因子会发生变化。
  - ★需要重新调整(re-balancing)
- § 删除delete(int key)
  - ★删除以后，平衡因子会发生变化。
  - ★需要重新调整(re-balancing)

平衡二叉树的基本操作

例:插入和删除后平衡性可能被破坏





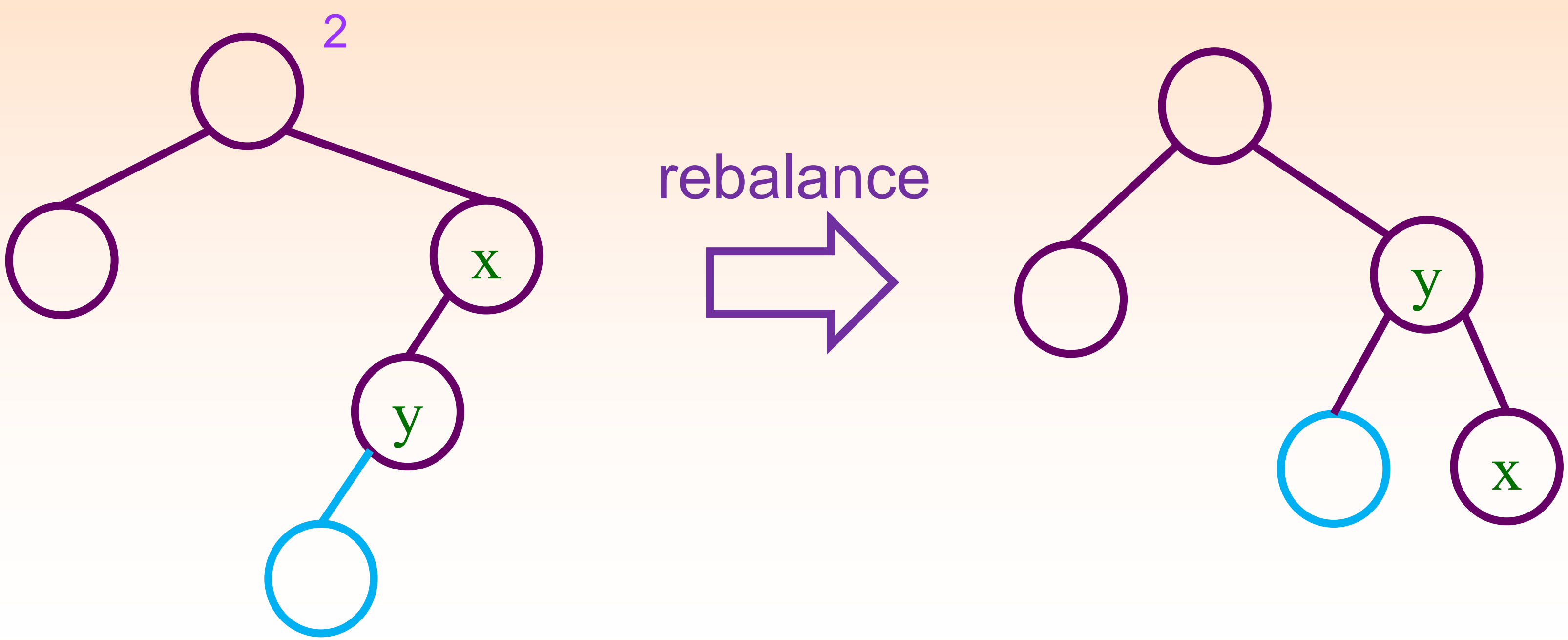
按排序二叉树的方式插入后可能存在节点不再平衡 ☹

按排序二叉树的方式删除后可能存在节点不再平衡 ☹

插入后如何rebalance？

假设插入后存在不平衡的节点

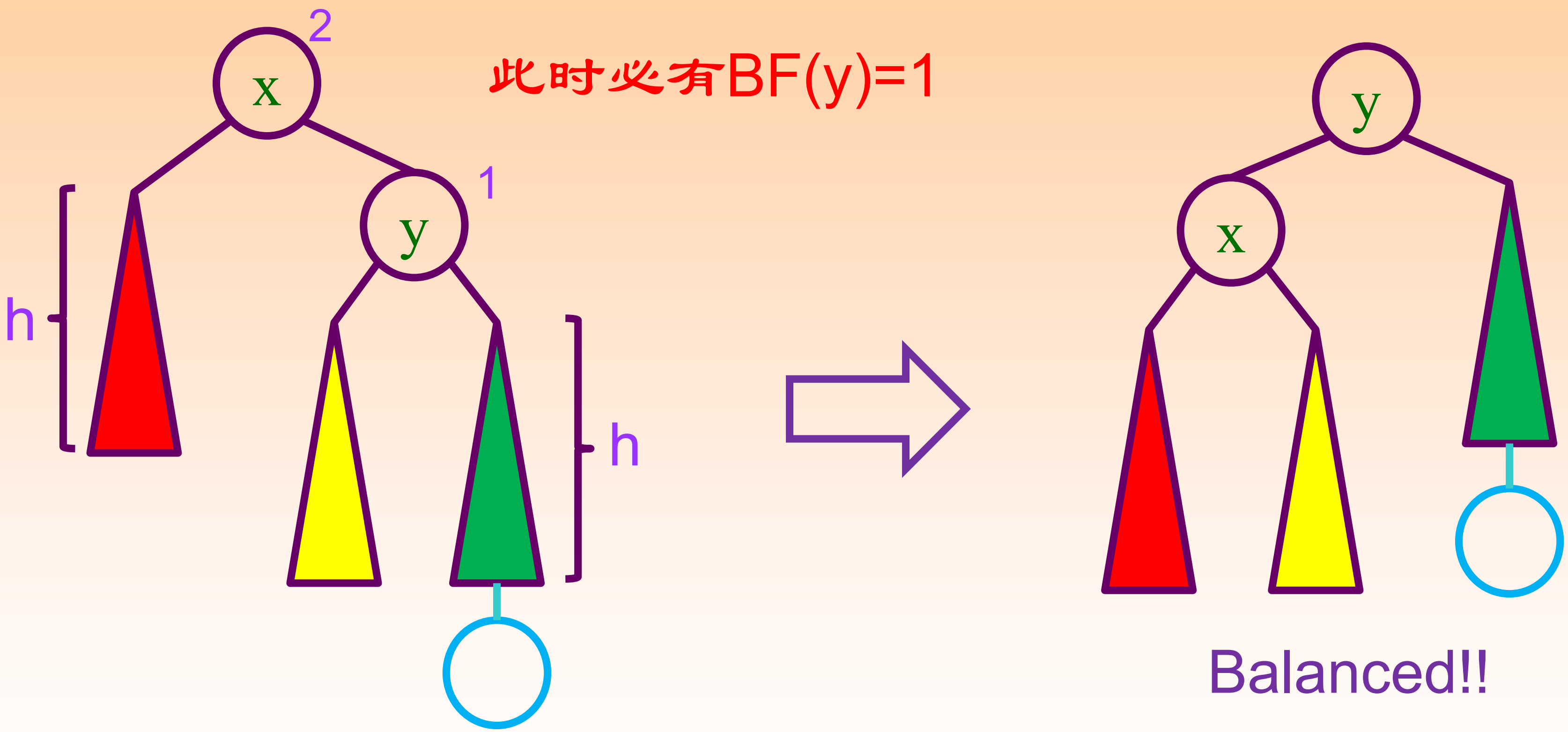
- ★不平衡的节点一定是新插入节点的祖先
- ★不妨设x是最靠下的一个不平衡节点。
- ★BF(x)= ±2. 不妨假设BF(X)=2(否则是对称的)。



插入后的再平衡

★Case 1. 新插入的节点在y的右子树. (y是x的右孩子)

此时必有BF(y)=1



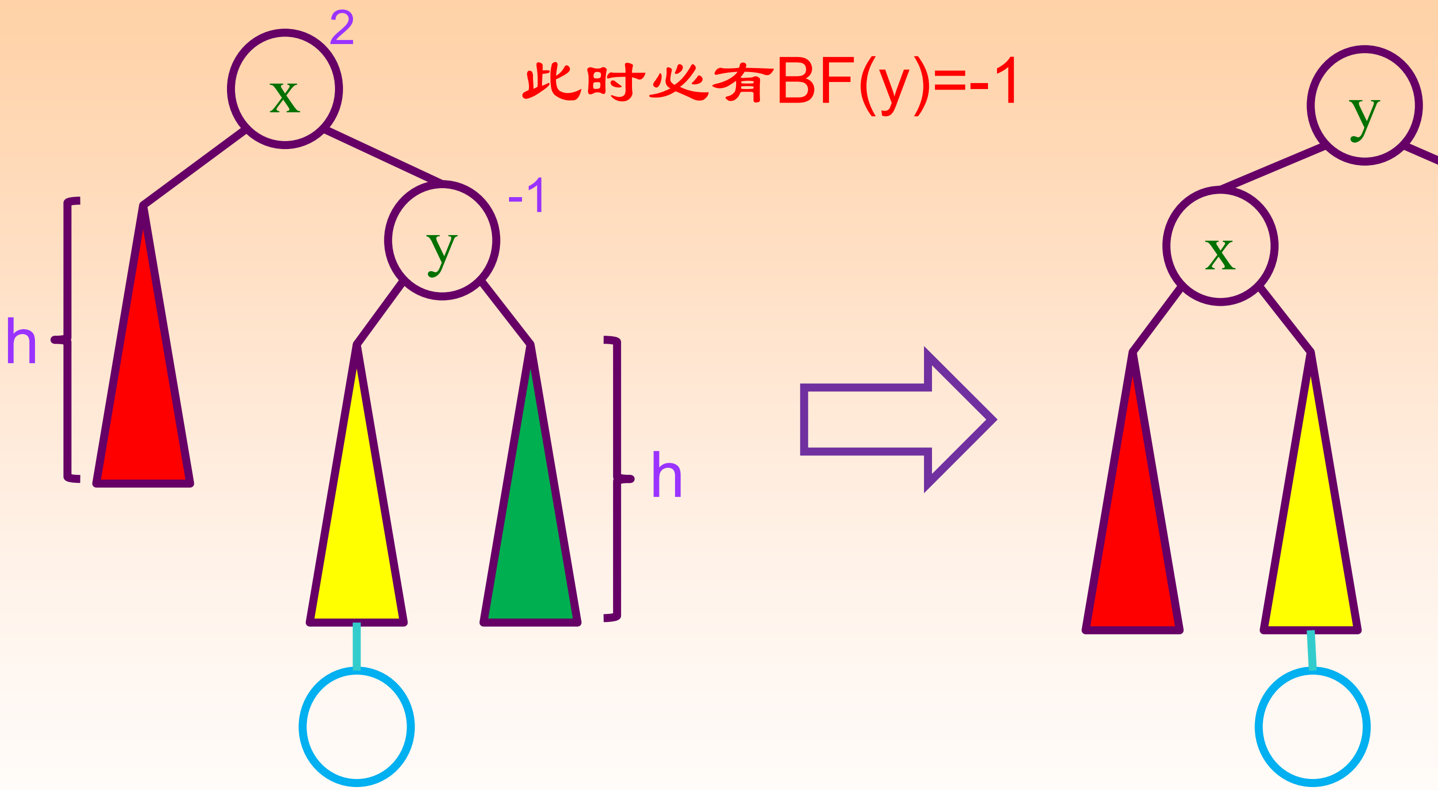
rebalance后该子树的height与插入前一致，均为h+2。

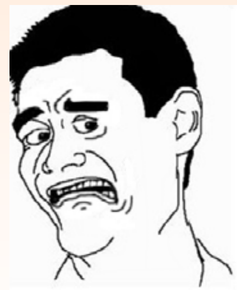
因此更靠上方的节点的BF值与插入前一致，无需平衡。

插入后的再平衡

★Case 2. 新插入的节点在y的左子树.（y是x的右孩子）

此时必有BF(y)=-1



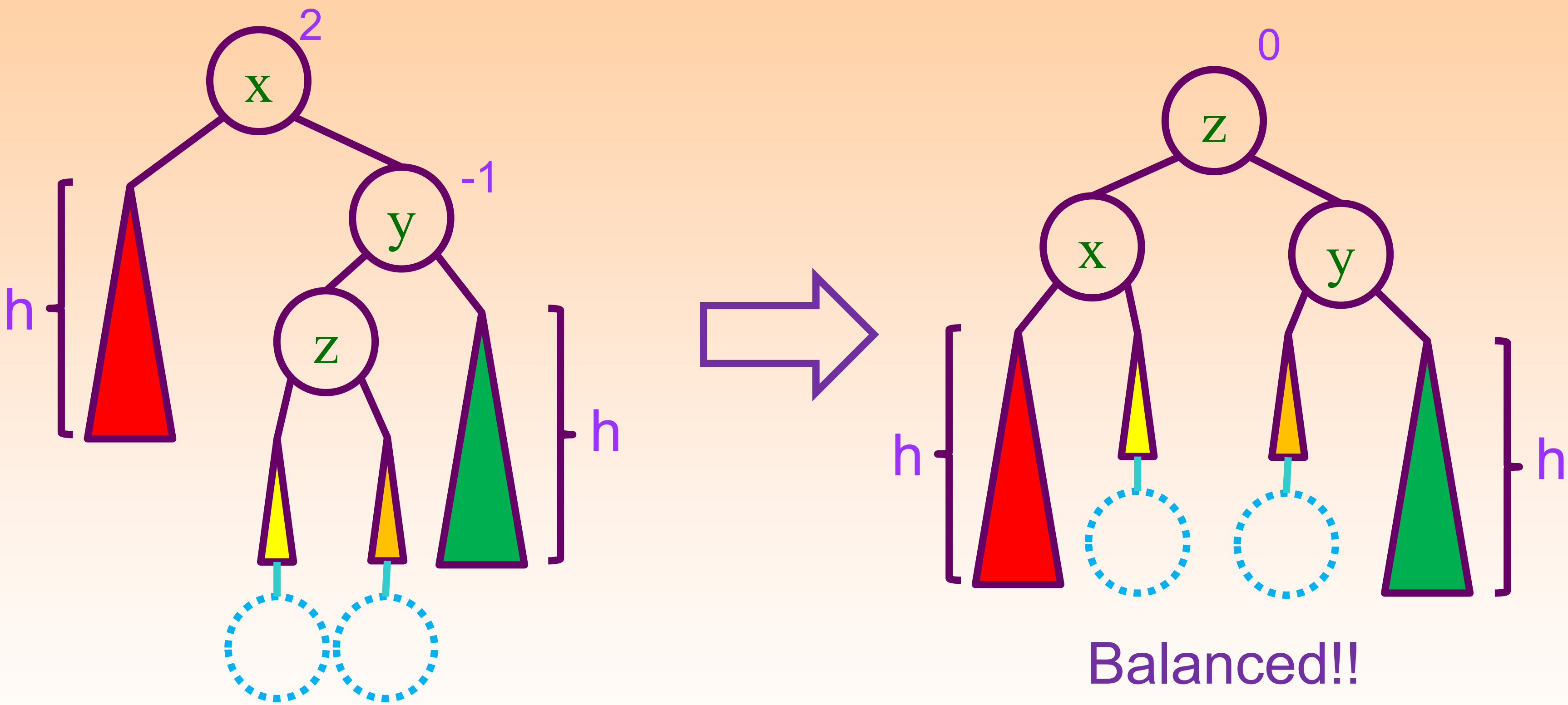


对这一种case，不能按照刚才那种方式进行rebalance。

按这种简单方式rebalance，BF(y)=-2。应该怎么做？

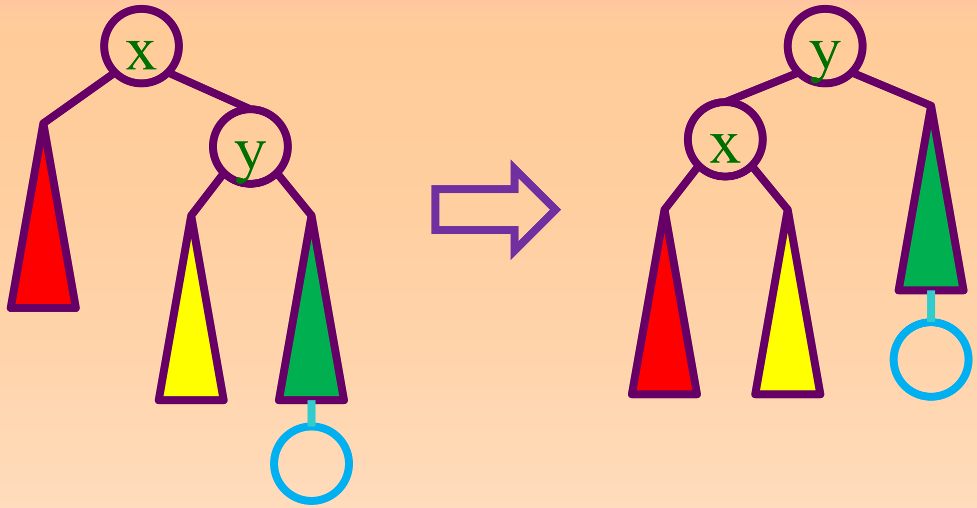
插入后的再平衡

★Case 2. 新插入的节点在y的左子树．(y是x的右孩子)

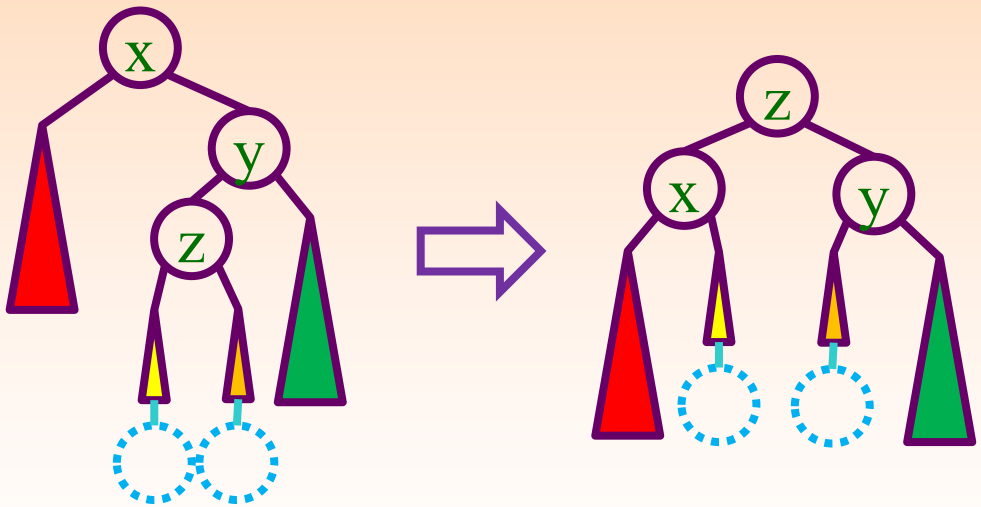


rebalance后该子树的height与插入前一致，均为h+2。
因此更靠上方的节点的BF值与插入前一致，无需平衡。

Case 1

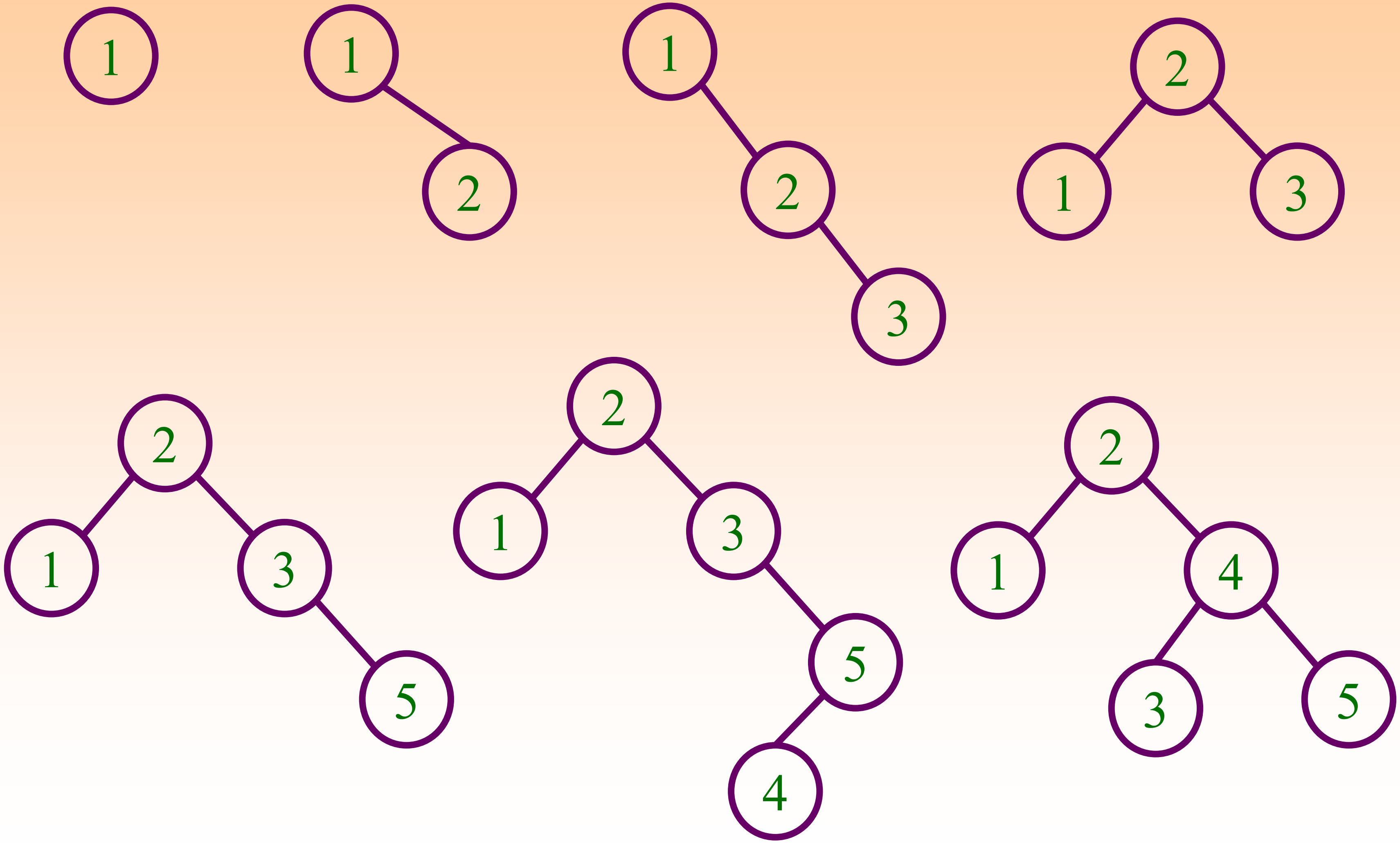


Case 2



Nothing outside this picture needs to be updated.
“Everything happens in Vegas stays in Vegas.”

例子：依次插入1,2,3,5,4

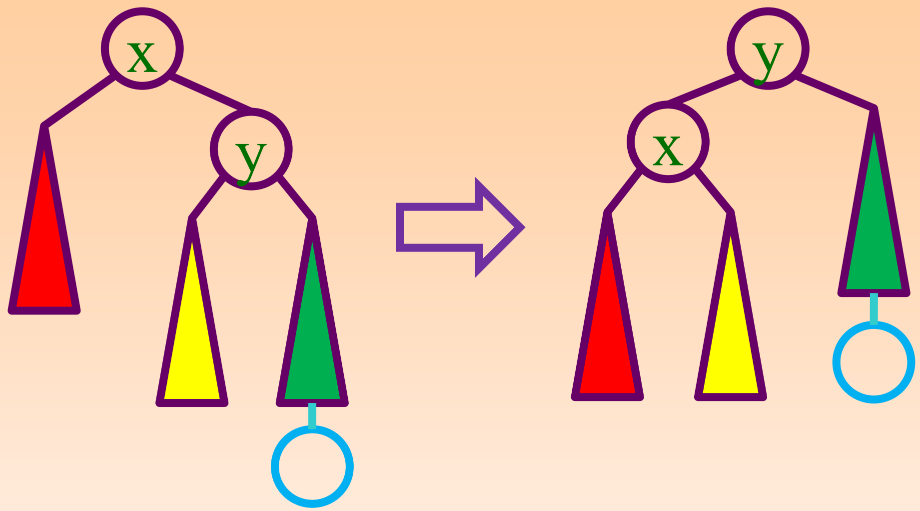


Rebalance过程伪代码实现

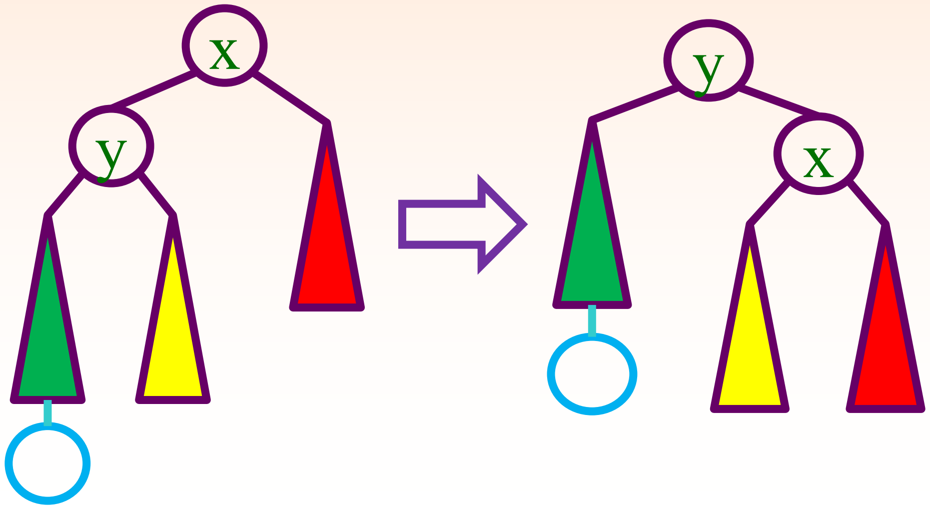
```
void rebalance(node &x){
    计算x的height和BF；  //根据左右子树的height;
    if (x.BF <= 1 && x.BF >= -1) return;  // 无需再平衡
    if (x.BF == 2)
        if (x.rchild->BF == 1)
                {……}            // rebalance_R_1
        else    {……};           // rebalance_R_2;
    else
        if (x.lchild->BF == -1)
                {……}            // rebalance_L_1
        else    {……};           // rebalance_L_2;
}
```

Rebalance过程伪代码实现

```
//以下为rebalance_R_1
node * y = x.rchild;
x.rchild = y->lchild;
y->lchild = x;
x.BF = y->BF = 0; x = y;
```

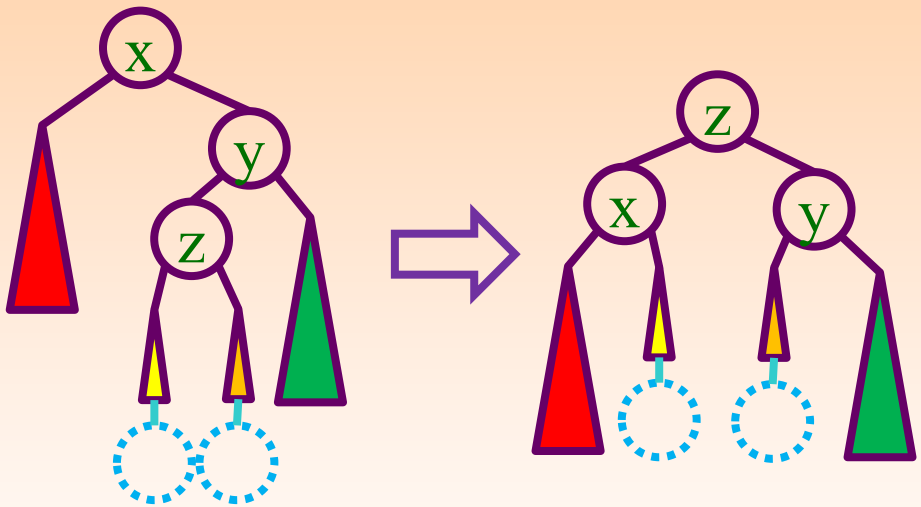



```
//以下为rebalance_L_1
node * y = x.lchild;
x.lchild = y->rchild;
y->rchild = x;
x.BF = y->BF = 0; x = y;
```



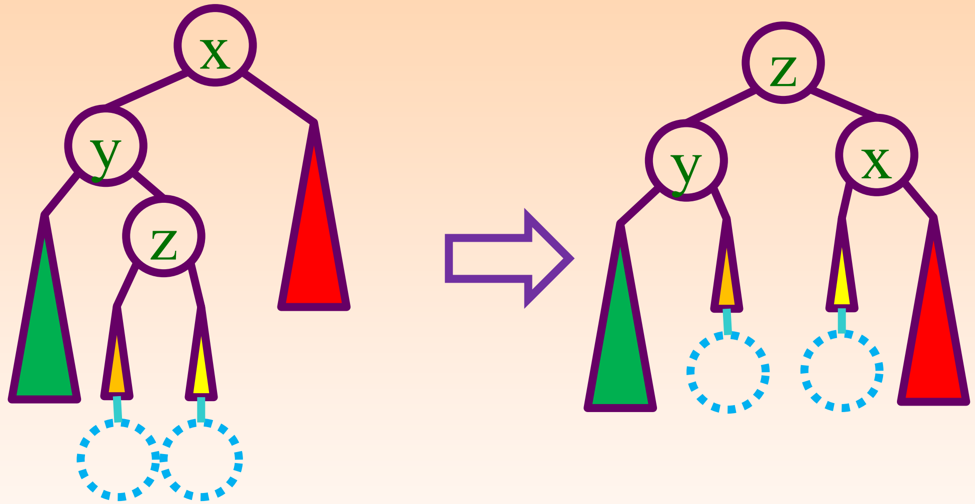
Rebalance过程伪代码实现

```
//以下为rebalance_R_2
node * y = x.rchild;
node * z = y->lchild;

x.rchild = z->lchild;
y->lchild = z->rchild;

z->lchild = x;
z->rchild = y;

重新计算x和y的BF ;
z->BF=0;
x = z;
```



Rebalance过程伪代码实现

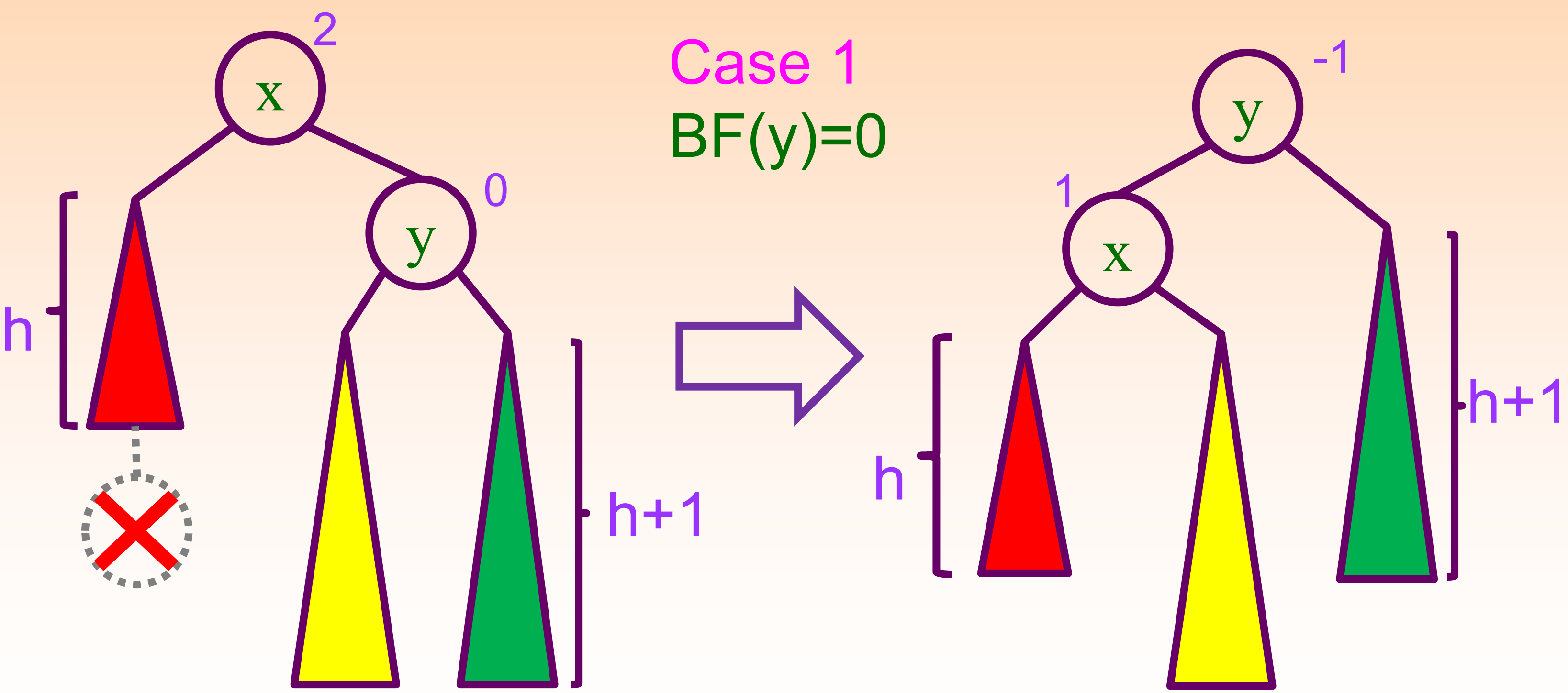
```
//以下为rebalance_L_2
node * y = x.lchild;
node * z = y->rchild;

x.lchild = z->rchild;
y->rchild = z->lchild;

z->rchild = x;
z->lchild = y;

重新计算x和y的BF;
z->BF=0;
x = z;
```



删除叶子节点后的再平衡

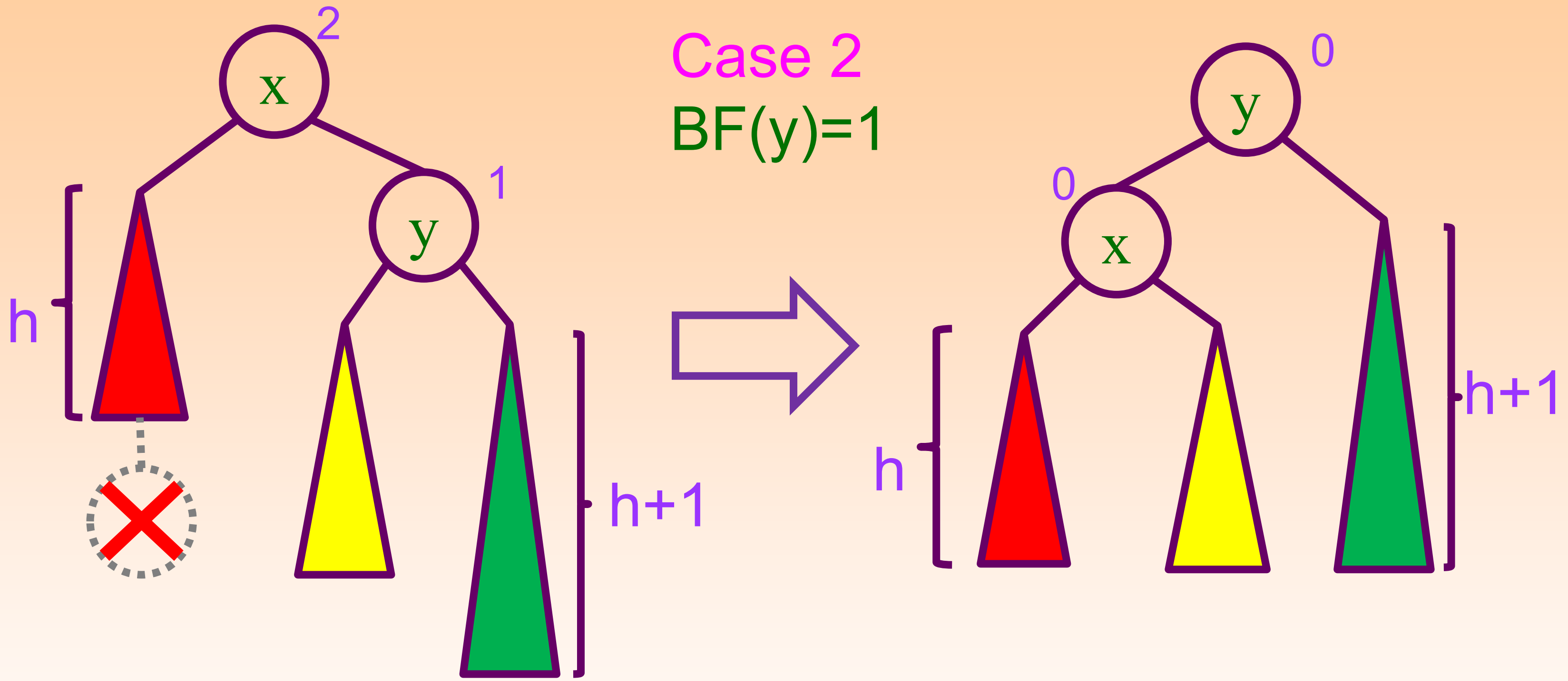
设最靠下方的不平衡的节点为x且BF(x)=2。

($h_L(x)$) 减少了1）。仍设y是x的右子树。分三种情况。



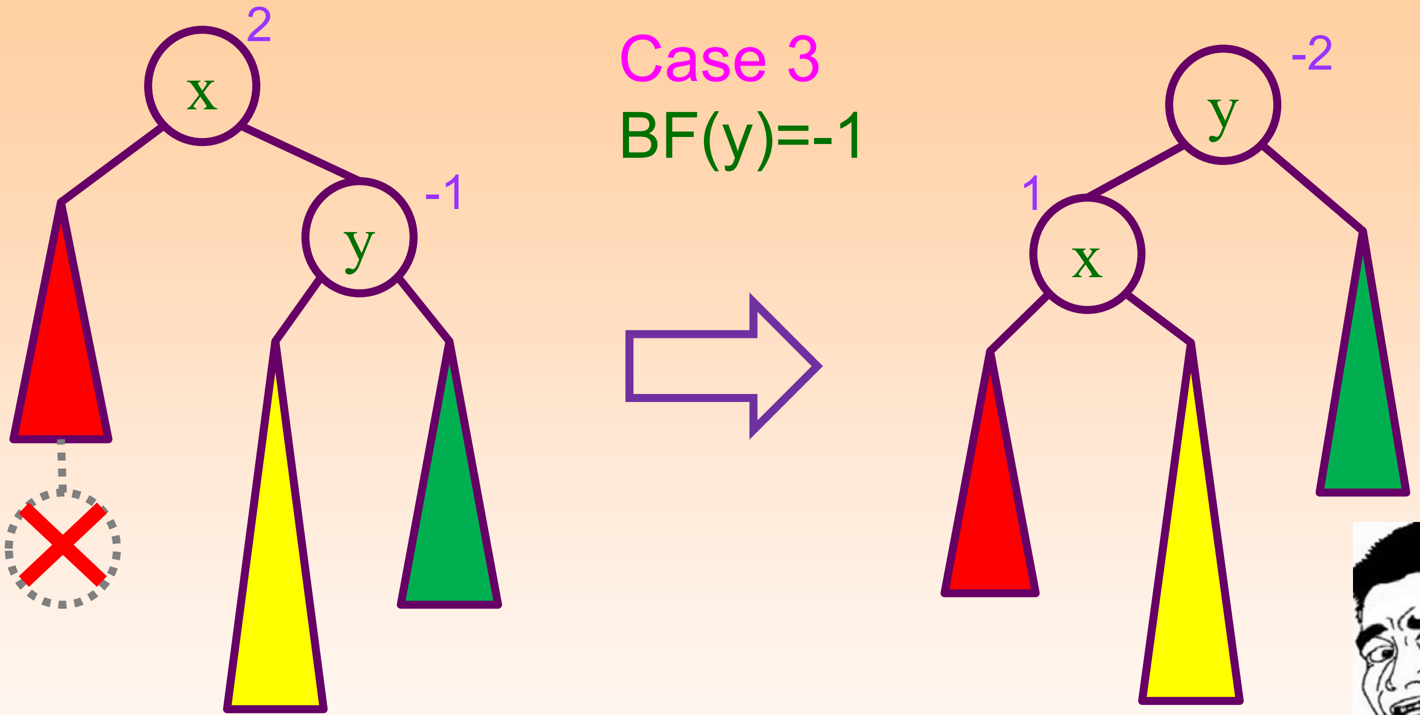
Nothing outside this picture needs to be updated!

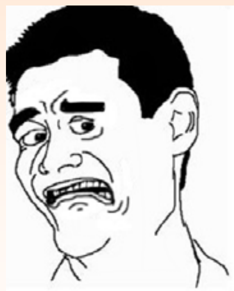
删除叶子节点后的再平衡



注意这个子树的高度减少了1！ 所以仍然可能有其他非平衡节点。接下来，应该去找到下一个最深的非平衡节点，对其进行同样操作。

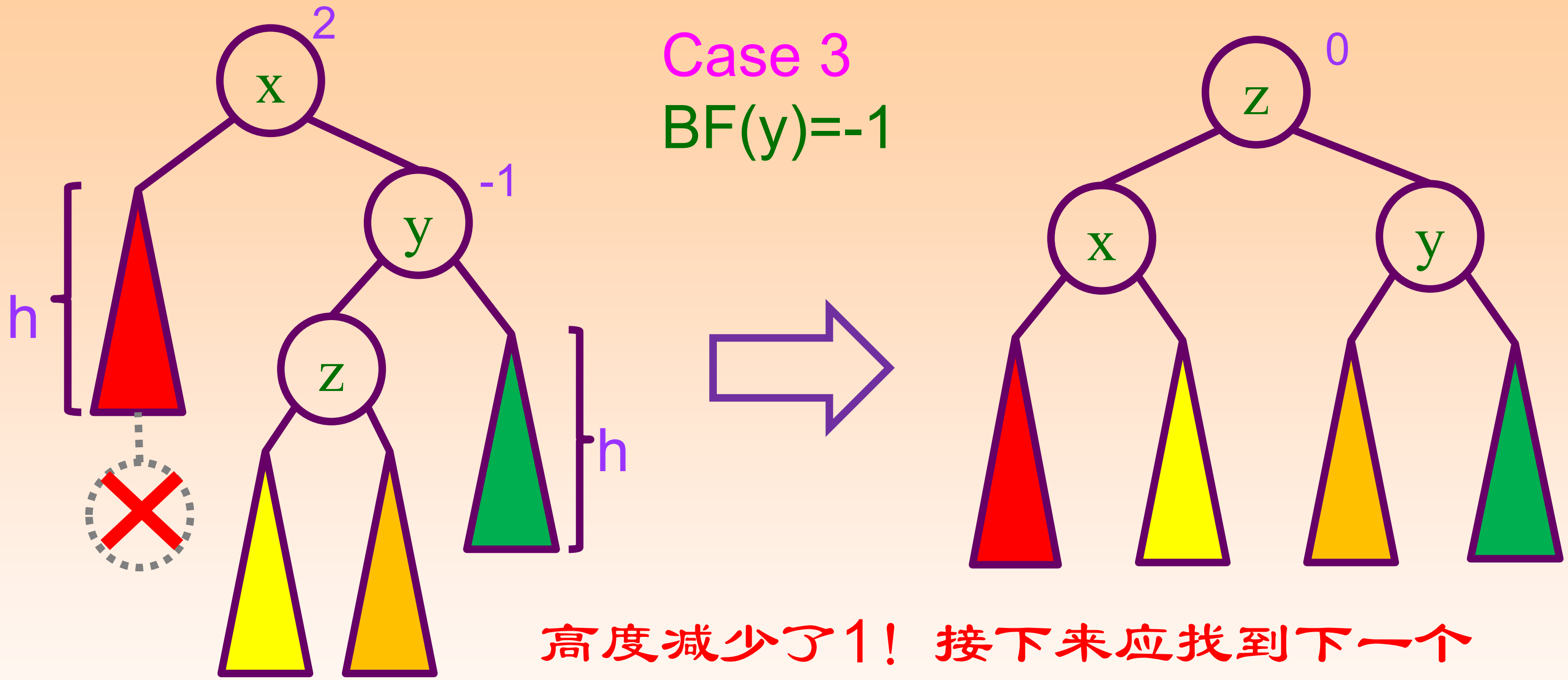
删除叶子节点后的再平衡





BF(y)=-1时，我们不能用简单的调整方法。

删除叶子节点后的再平衡



高度减少了1！接下来应找到下一个最深的非平衡节点对其进行同样操作

注意：Z的左右子树有一个高度为h，另一个为h或h-1。在图片中，我们假设两个子树高度都为h。(不重要)

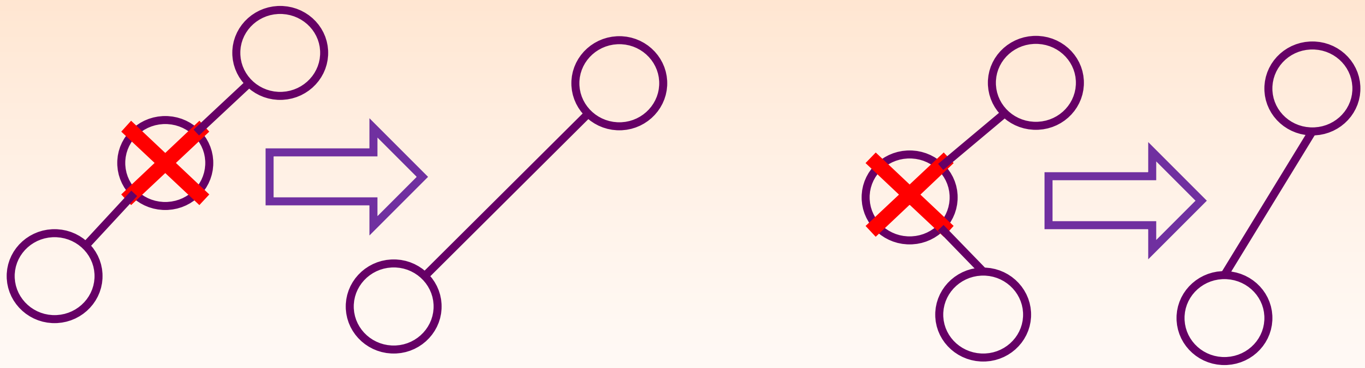
删除非叶子节点的再平衡

回顾删除的另外两种case:

（2）被删除的结点只有左子树或者只有右子树；

（3）被删除的结点既有左子树，也有右子树。

情况（2）处理方式：



被删除节点的祖先有可能不再平衡；要将他们再平衡！

（再平衡方式与删除叶子节点时的方法完全一致）

删除非叶子节点的再平衡

情况（3）被删除的结点既有左子树，也有右子树。

回顾：假设要删除的节点a有左右子树。找到a的前驱a'。将a'的节点信息copy到a中然后删掉a'。实际删除节点为a'。

因此可以按情况(1)或情况(2)的方式进行处理。

删除的类C语言实现

不要求掌握！

平衡二叉树的应用

平衡二叉树的优点

§ 支持多种操作在O(log n)时间内完成：

- ★Insert / delete
- ★successor。找到比某个值s大最小的key值。
- ★min / max
- ★查找第k大的元素 （需额外信息；见后文）

§ 与二叉堆对比

- ★二叉堆更简单、常数更小、但功能更少
- ★Insert / delete / min (or insert / delete / max)
- ★都可用来排序。O(n log n)

应用1：快速最长递增子序列

给过$O(n^2)$算法。用AVL树可优化到$O(n\log n)$。

问题描述：输入序列$(x_1,\ldots,x_n)$。

- ★要找最大的k以及i1<i2<…<ik满足$x_{i1}<x_{i2}<\ldots<x_{ik}$。
- ★即，要找到序列的最长的递增子序列。

动态规划解法

- ★状态描述：
  - ❖F_j表示$(x_1,\ldots,x_j)$的以x_j结束的最长递增子序列的长度。
- ★状态转移方程：
  - ❖ $F_j = \max\{1, \max_{i<j, x_i<x_j} F_i + 1\}$
- ★依据转移公式，容易在$O(j)$时间内算出F_j。从而可在$O(n^2)$时间内算出$F_1,\ldots,F_n$。最终答案$=\max(F_1,\ldots,F_n)$。

应用1：快速最长递增子序列

§ 可否更快速的计算 $A = \max_{i<j, x_i<x_j} F_i$?

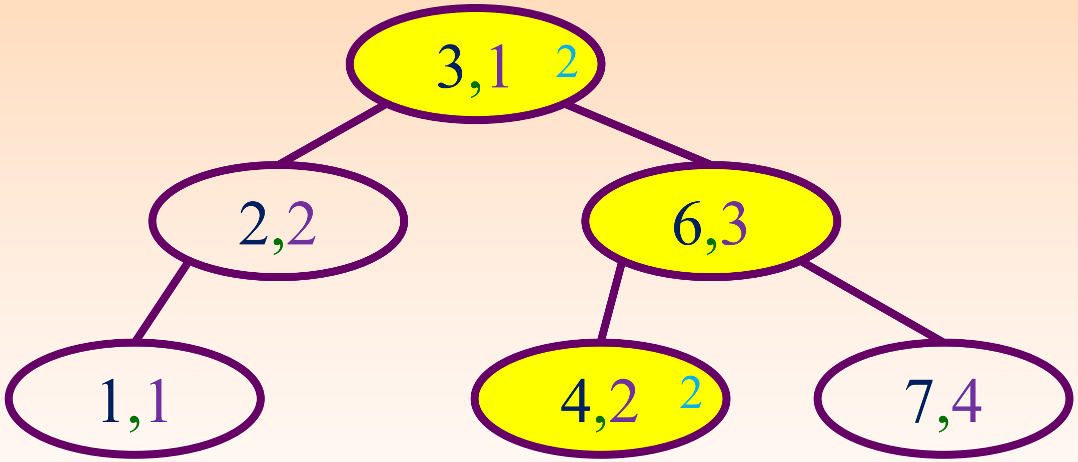
§ 将(x_i,F_i)作为一个数据项插入到集合S中。

- ★计算公式A时，S中有$(x_1,F_1),\ldots,(X_{j-1},F_{j-1})$。
  - ❖当计算出F_j后，将(X_j,F_j)加入到S中。
- ★将x_i当作(x_i,F_i)的关键值；S用一棵AVL树维护。
- ★给定一个阈值x_j，需要计算的是：S中关键值比该阈值小的那些数据项中，F的最大值。
- ★为快速计算这个值，在每个节点添加一个额外信息，储存以该节点为根的子树中F的最大值。

应用1：快速最长递增子序列

例　$x=(3,1,4,2,6,7,5)$

$F=(1,1,2,2,3,4,?)$



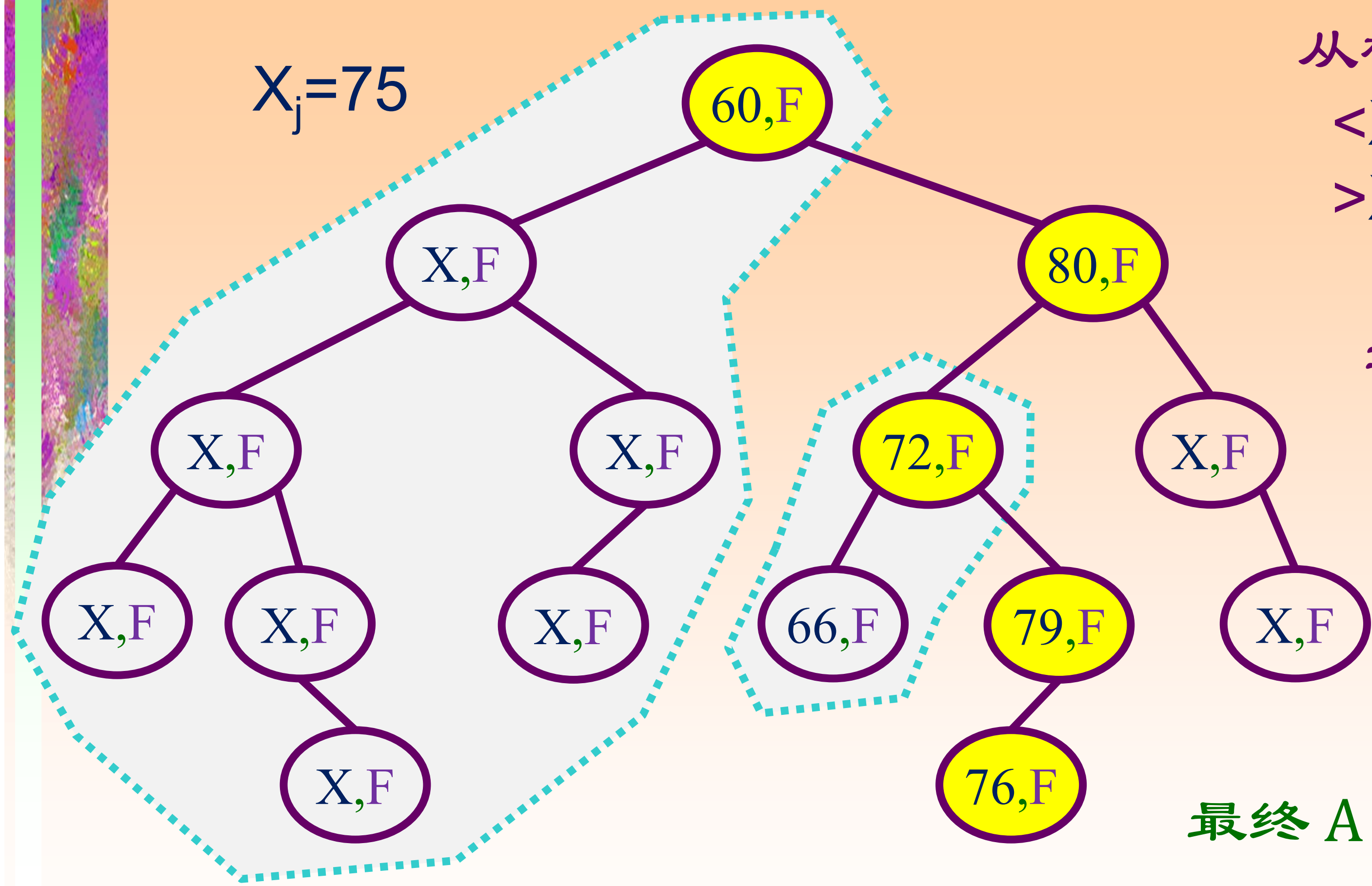
1. 令A=0。

2. 从根出发往下走。
<5往右,>5往左。

3. 遇到<5节点u时：
令$A=\max(A, LF_u)$。
其中LF_u表示该节点及其左子树中所有节点的最大F值。

只需O(log n)算出A。之前O(n)是枚举了所有节点。

应用1：快速最长递增子序列



从根出发往下走。

<X_j往右

>X_j往左

当<X_j时：

令

$A=\max(A,LF_u)$

最终 $A = \max_{i<j, x_i<x_j} F_i$

课后思考：找更简单的O(nlogn)算法。Hint:无需AVL树.

应用2：动态计算凸包

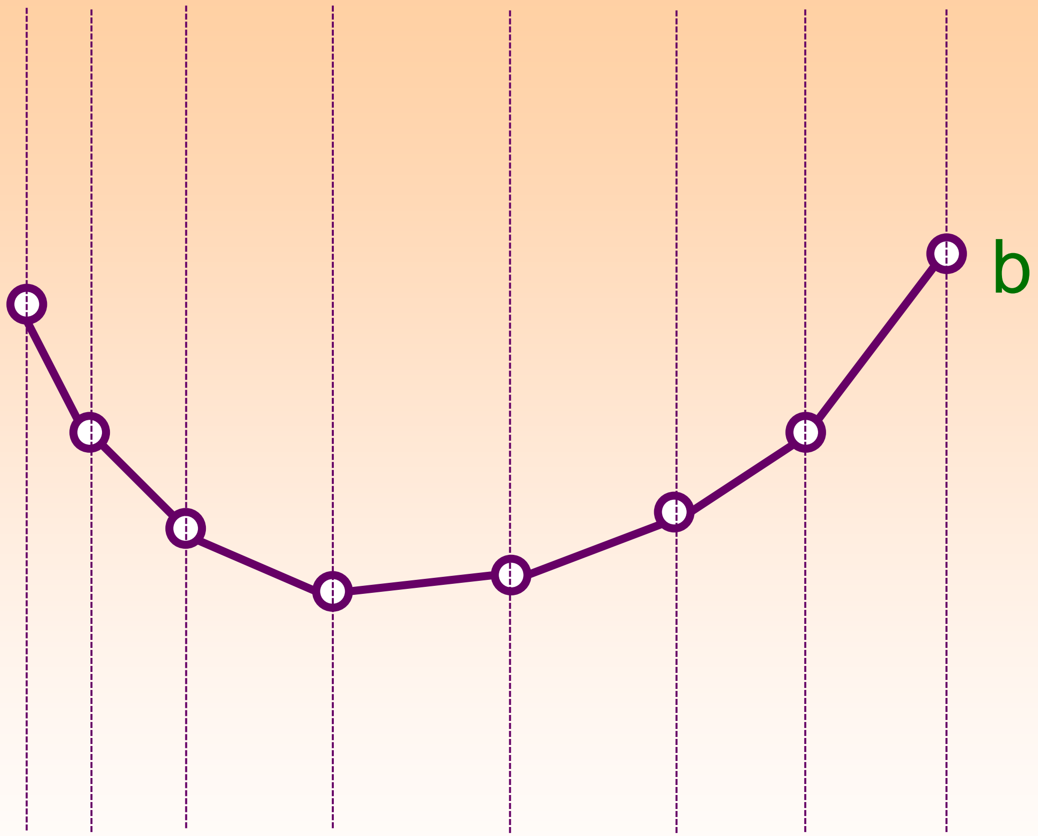
【问题描述】假设你需要支持两种操作：

- ★集合P中加入一个点，坐标(x,y)。
- ★输出P的凸包。

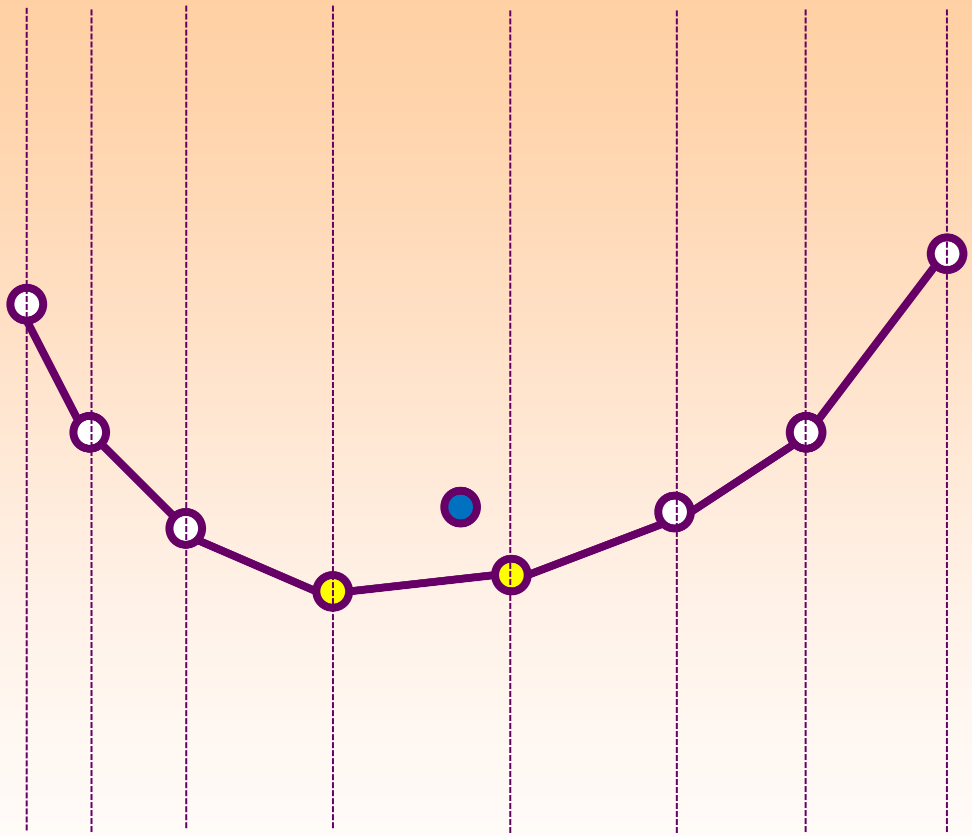
§ 方法

- ★用AVL树维护凸包的下边界b。上边界对称处理
  - ❖b由一系列顶点组成，从左到右，x坐标为关键值。
- ★当加入一个新的点(x,y)时，
  - ❖判断其是不是在b上方
    - 若在b上方，则下边界b不需要更新。
    - 否则，(x,y)要插入b中，并且b的某些顶点要删除。

应用2：动态计算凸包



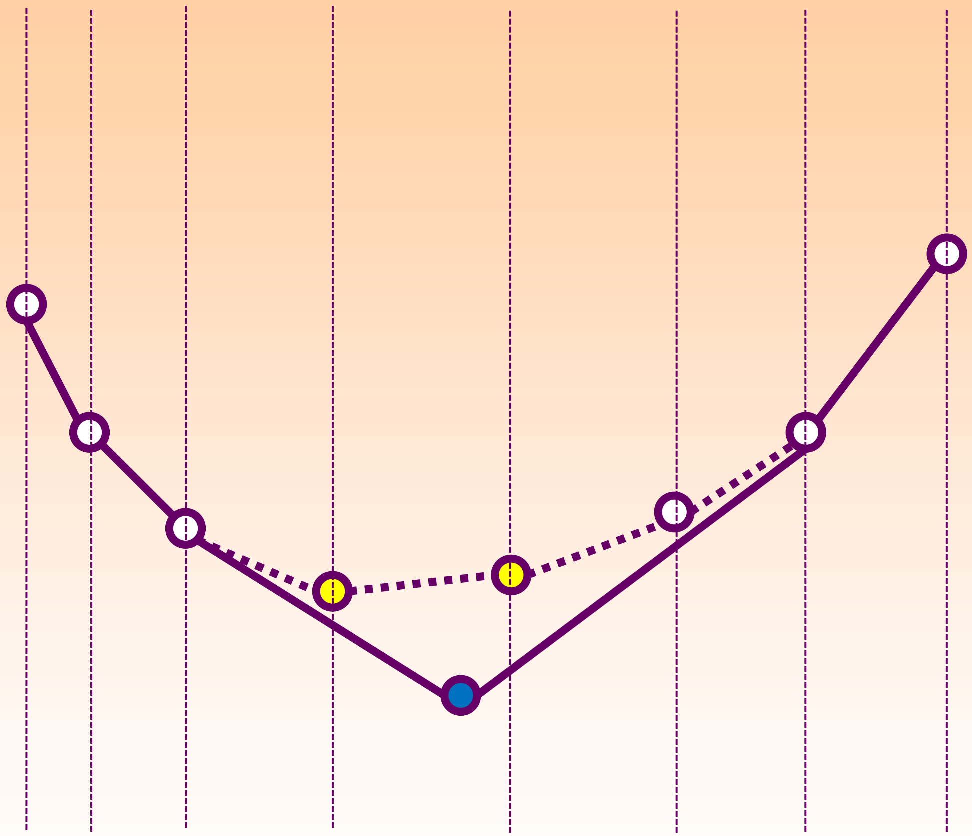
应用2：动态计算凸包



加入点(x,y)时
先判断它与b的关系
——找到b中两个邻点
使得(x,y)的横坐标
在它们的横坐标之间

判断(x,y)与b的关系仅
需要O(log n)时间。

应用2：动态计算凸包



如果(x,y)在b下方
要将(x,y)插入
同时还要删除b中
原有的某些点。

如果在b左方/右方
也可以类似处理。

插入/删除 一个点的复杂度为O(log n)。 (总共是O(nlogn))
注意：加入(x,y)时可能许多点被删除；每步未必是O(log n)的。

应用2：动态计算凸包

基于AVL树的增量凸包算法。

v.s.

Graham Scan以及分治求凸包算法。

优点：

新算法是一个在线算法。

不需要拿到所有的点就可开始计算。

每新增一个点，都能较快速度得到现有点集合的凸包。

缺点：

编程难度稍微大一些。隐藏在O后面的常数更大。

应用3：寻找某个rank的key

【问题描述】

★动态维护一个集合S（支持插入/删除）

★要求支持如下查询（query）：

❖Query有一参数为k

❖要求输出S中第k小的元素。

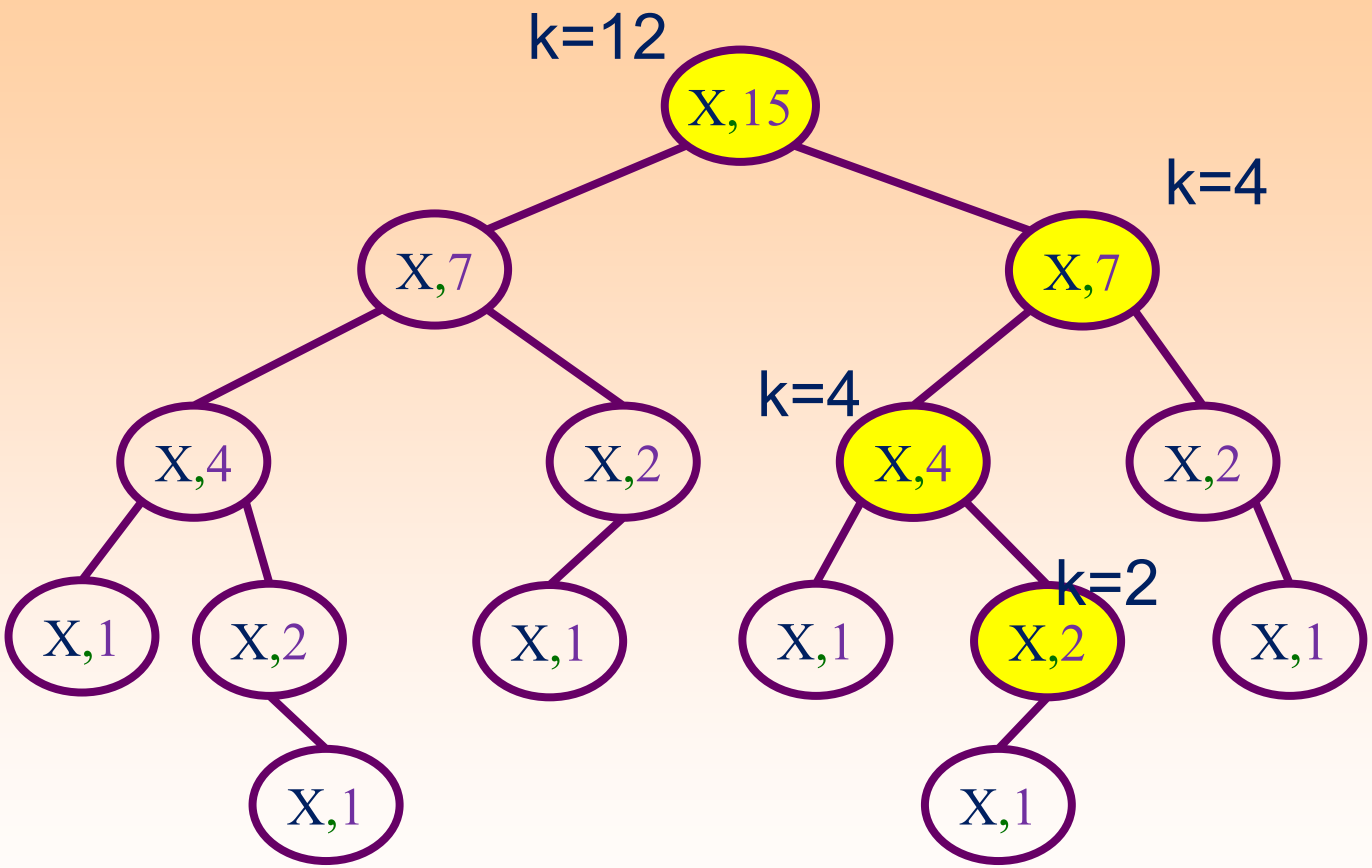
解决办法：

★用一棵AVL树保存S中的元素

★每个节点u增加一个信息域t_u：表示以u为根的子树中节点的个数。

★利用t_u，很容易找到第k小元素（见下页）

应用3：寻找某个rank的key



应用4：查询某个key的rank

【问题描述】

- ★动态维护一个集合S（支持插入/删除）
- ★要求支持如下查询（query）：
  - ❖Query有一参数为k
  - ❖要求输出S中key值为k的元素的rank

    （即，k在S中是第几小的元素？）

解决方法：

- ★类似前一个应用的解决方法。

  每个节点u增加一个信息域t_u：表示以u为根的子树中节点的个数。

应用5. Voronoi图的Fortunes算法

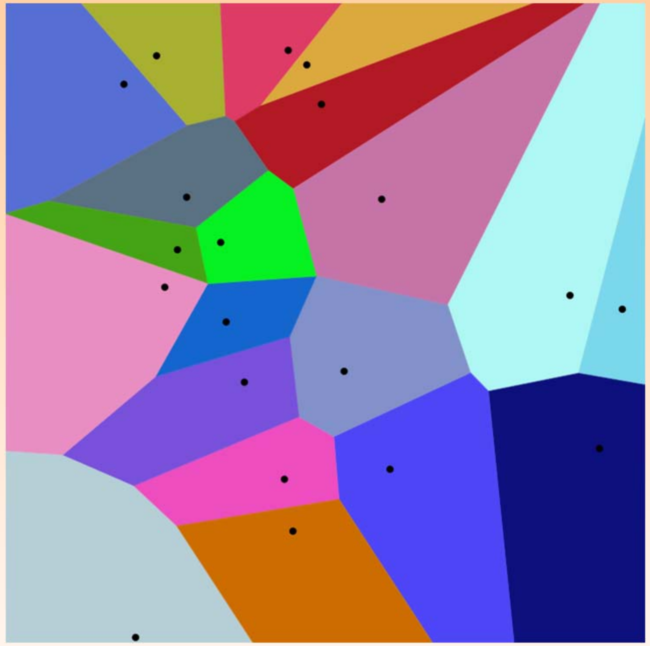
【问题背景】Voronoi图

★给定平面上的n个点$p_1,\ldots,p_n$
每一个叫做一个site（站点）

★VoronoiDiagram($p_1,\ldots,p_n$)
is a partition of a plane into regions close to each site.

❖每个点p_i对应一个区域r_i——
该区域的点的最近的site为p_i。

❖可以把site看成商店/邮局等等。



Voronoi图是非常重要的几何结构，应用特别广。

https://en.wikipedia.org/wiki/Voronoi_diagram#Illustration

应用5. Voronoi图的Fortunes算法

- § Voronoi Diagram包含n个区域、O(n)条边。
- § 计算Vorono Diagram有许多O(nlogn)的算法。
  - ★ 如Fortunes算法。
    - ❖ 维护一个sweepling line和一个beach line。
    - ❖ 要用到binary search tree

阅读：

https://en.wikipedia.org/wiki/Fortune%27s_algorithm

<Computational Geometry:Algorithms and applications> 7.2章Computing the Voronoi Diagram (***)

Connection with最优判定树 **

§ 最优判定树

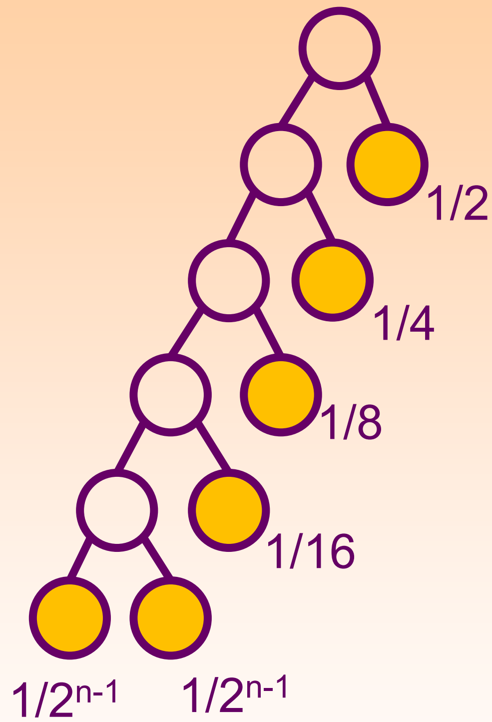
(optimal binary search tree)

★ 已知每个key的概率$p_1,\ldots,p_n$。
要构造一棵BST平均查找时间最低。

★ O(entropy($p_1,\ldots,p_n$))期望时间找到key

❖比O(log n)可能还要块! (entropy<log n)

❖但是单次可能比O(log n)差得多; 如右图



§ 平衡二叉树

(self-balancing binary search tree)

★ 未知概率。保证每次都是O(log n)。

其他类似平衡二叉树的结构

§ 线段树　（功能类似，更简单，适用性差）

§ 红黑树　（功能更强，更复杂，常数更低）

§ 伸展树　（后续课程将做介绍！）

Treap (简单，最坏复杂度差)**

§ 另一种二叉查找树（不一定平衡）

§ 它是一棵random binary search tree.

★ 为每一个节点设置一个随机的distinct index

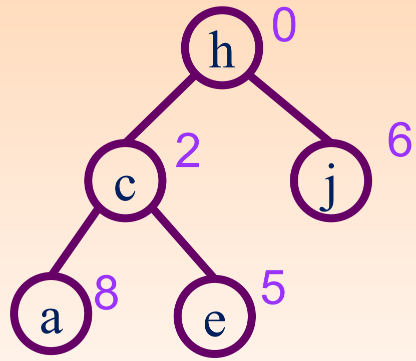
★ 这棵树中的节点满足:

❖ index满足堆性质

❖ Key满足排序二叉树性质

★ 插入和删除维护上述性质

❖ 最坏情况很差。平均情况是O(log n)。

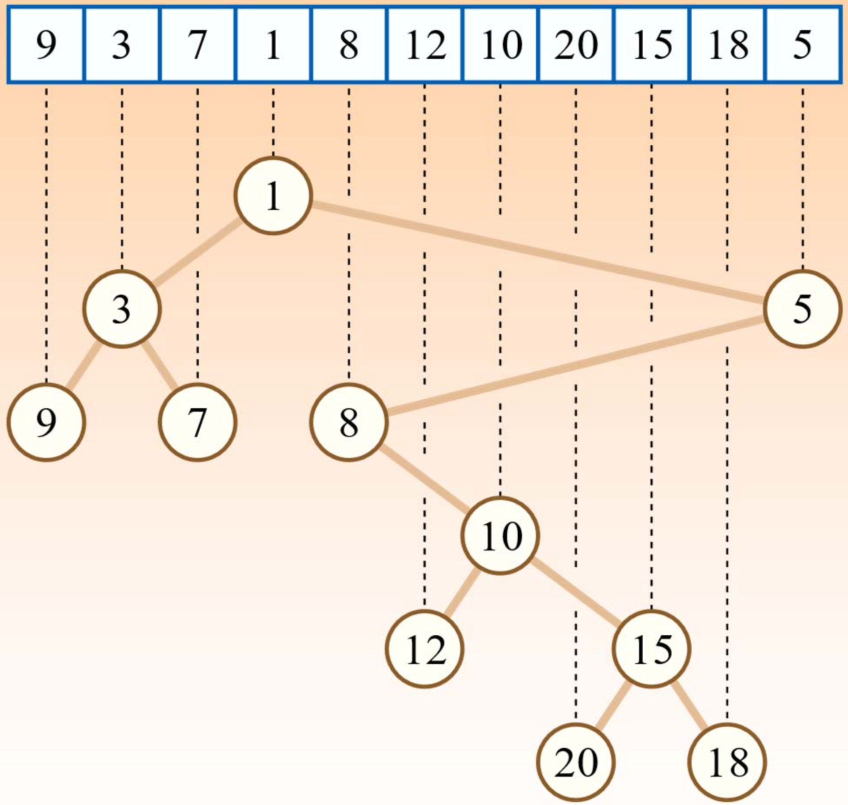


§ 阅读：

https://en.wikipedia.org/wiki/Treap

https://en.wikipedia.org/wiki/Cartesian_tree

Treap (简单， 最坏复杂度差)**



Reference

http://www.cs.toronto.edu/~toni/Courses/263-2015/lectures/lec04-balanced-augmentation.pdf