湖南大學
HUNAN UNIVERSITY

第五章 树

湖南大学信息科学与工程学院

概述

- 机场跑道预定系统:
  - 定义
  - 如何利用列表解决一系列飞机起飞时间节点的安排问题。
  - 二叉搜索树
  - 操作
- 阅读材料：《算法导论》第10章，第12.1-12.3节。

http://izismile.com/tags/Gibraltar/

跑道预定系统

- 问题定义：

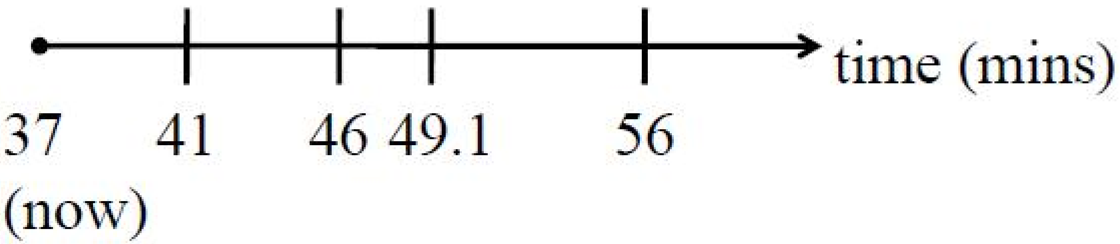
  单一跑道（繁忙）

  预定起飞时间

  - 维护一个集合的起飞时间节点
  - 给定一个新的起飞时间节点申请t
  - 把t加入到集合中：新的时间节点t与其他所有时间节点的间隔小于3分钟
  - 当飞机起飞，则把它的时间节点t从集合中删除

跑道预定系统

例子：



- R = (41, 46, 49.1, 56)
- requests for time:　　新时间节点请求
  - 44 => reject (46 in R)　　拒绝
  - 53 => ok　　允许
  - 20 => not allowed (already past)　　不允许，超过边界

对于有效实现的思路？

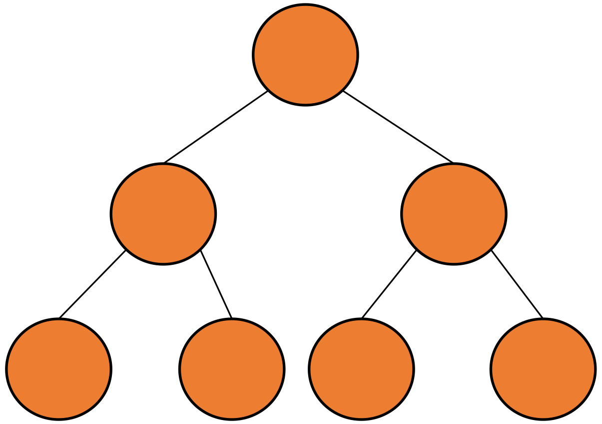
一些选项

- 保持R作为一个未排序的列表
  - ➢ 缺点：需要线性时间来搜索冲突
  - ➢ 优点：可以在O(1)时间内插入t

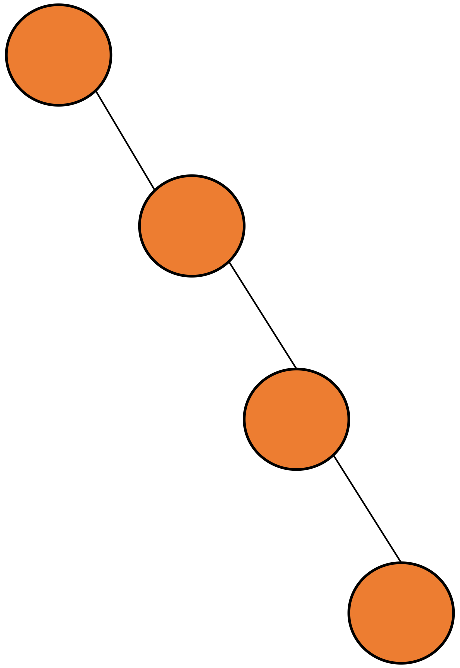
- 保持R为一个排序数组（每次插入后重新排序）
  - ➢ 缺点：需要很多时间来插入元素
  - ➢ 优点：3分钟的检查可以在O(log n)时间内完成——使用二分搜索，找到最小的i使得R[i]>=t（下一个较大的元素）
- 比较t与R[i]和R[i-1]

数组插入效率低，需要更快的插入算法及数据结构

二叉搜索树



一颗好的二叉搜索树，
高度为 log n

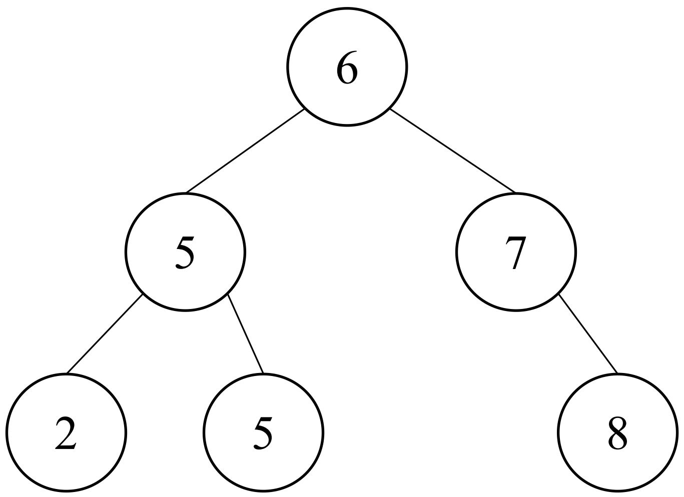


不好的二叉搜索树高度可
能为 n

二叉搜索树

二叉搜索树的每个结点都包含一个关键字key，同时包含属性left、right和p，它们分别指向结点的左孩子、右孩子和双亲。如果某个孩子结点的父结点不存在，则相应属性值为NIL。根结点是树中唯一父指针为NIL的结点。

二叉搜索树



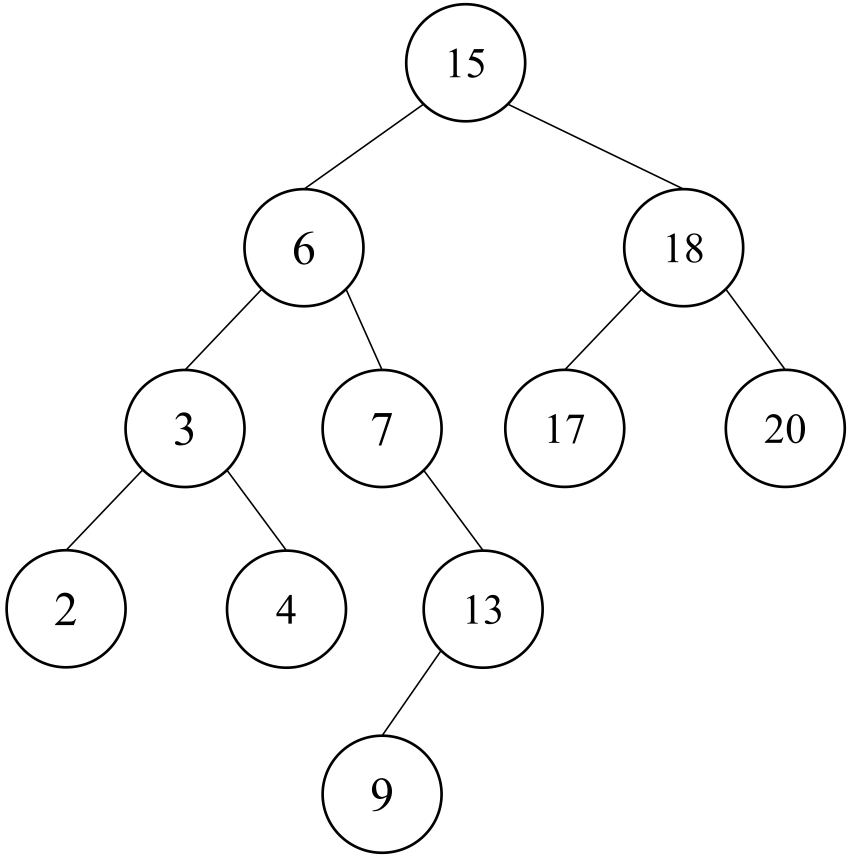
对于任何结点x，其左子树中的关键字最大不超过x.key，其右子树中的关键字最小不低于x.key。上图展示了一颗包含6个结点、高度为2的二叉搜索树。

查询二叉搜索树

在一颗二叉搜索树中查找一个具有给定关键字的结点，输入一个指向树根的指针和一个关键字k，如果这个结点存在，TREE-SEARCH返回一个指向关键字为k的结点的指针，否则返回NIL。

```
TREE-SEARCH(x,k)
If x ==NIL or k == x.key
    return x
If k < x.key
    return TREE-SEARCH(x.left,k)
else return TREE-SEARCH(x.right,k)
```

查询二叉搜索树



查询关键字为13的结点，从树根开始，13<15，故沿着left指针查找左子树；13>6，故沿着right指针查找右子树；13>7，故沿着right指针查找右子树；最终找到13。

查询二叉搜索树

```
ITERATIVETREE-SEARCH(x,k)
  while x ≠ NIL ,and k ≠ x.key
  If k< x.key
        x = x.left
   else  x = x.right
   return x
```

我们可以采用while循环来展开递归，用一种迭代方式重写这个过程，对于大多数计算机，迭代版本的效率要高得多。

最大关键字元素

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

二叉搜索树性质保证了TREE-MAXIMUM是正确的。如果结点没有右子树，那么由于x左子树中的每个关键字都至少小于或等于x.key，则以x为根的子树中的最大关键字是x.key。如果结点有右子树，那么由于其左子树中没有关键字大于x.key，且在右子树中的每个关键字不小于x.key，则以x为根的子树中的最大关键字一定在以x.right为根的子树中。

最小关键字元素

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

二叉搜索树性质保证了TREE-MINIMUM是正确的。如果结点没有左子树，那么由于x右子树中的每个关键字都至少大于或等于x.key，则以x为根的子树中的最小关键字是x.key。如果结点有左子树，那么由于其右子树中没有关键字小于x.key，且在左子树中的每个关键字不大于x.key，则以x为根的子树中的最小关键字一定在以x.left为根的子树中。

查询二叉搜索树

这两个过程在一颗高度为h的树上均能在O(h)的时间内执行完，因为与TREE-SEARCH一样，它们所遇到的结点均形成了一条从树根向下的简单路径。

后继和前驱

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
  return TREE-MINIMUM(x,right)
y = x.p
while y ≠NIL and x == y.right
x = y
y = y.p
return y
```

给定一颗二叉搜索树中的一个结点，有时候需要按中序遍历的次序查找它的后继。如果所有的关键字互不相同，则一个结点x的后继是大于x.key的最小关键字的结点。一颗二叉搜索树的结构允许我们通过没有任何关键字的比较来确定一个结点的后继。如果后继存在，下面的过程将返回一颗二叉搜索树中的结点x的后继；如果x是这棵树中的最大关键字，则返回NIL。

构造一颗二叉搜索树

插入10

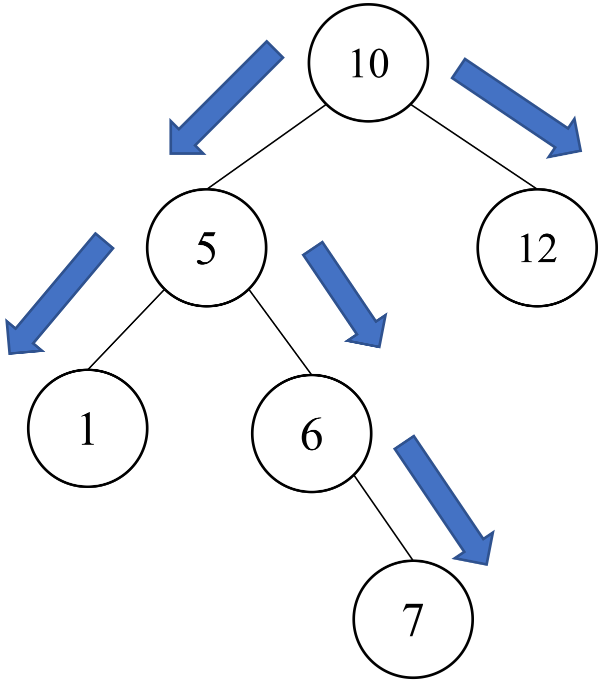
插入12

插入5

插入1

插入6

插入7

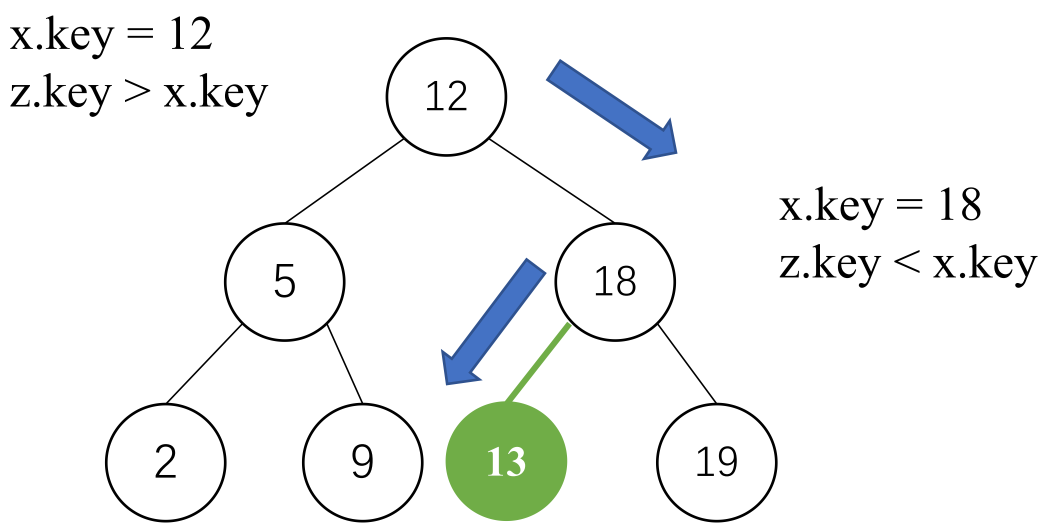


二叉搜索树的插入

```
TREE-SEARCH(T,z)
y == NIL
x == T.root
While x ≠ NIL
    y = x
    If z.key < x.key
        x = x.left
    else  x = x.right
z.p =y
If y == NIL
    T.root = z        //tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

要将一个新值v插入到一颗二叉搜索树T中，需要调用过程TREE-INSERT。该过程以结点z作为输入，其中z.key = v, z.left = NIL, z.right = NIL。这个过程要修改T和z的某些属性，来把z插入到树中的相应位置上。

二叉搜索树的插入



插入关键字为13的结点z,首先与根结点12进行比较，13>12，转向右子树，13<18，转向左子树，左子树为空，则插入结点13。

从一棵二叉搜索树T中删除一个结点z

从一棵二叉搜索树T中删除一个结点z的整个策略分为三种基本情况(如下所述)，但只有第三种情况有点辣手。

- 如果z没有孩子结点，那么只是简单地将它删除，并修改它的父结点，用 NIL 作为孩子来替换z。

从一棵二叉搜索树T中删除一个结点z

从一棵二叉搜索树T中删除一个结点z

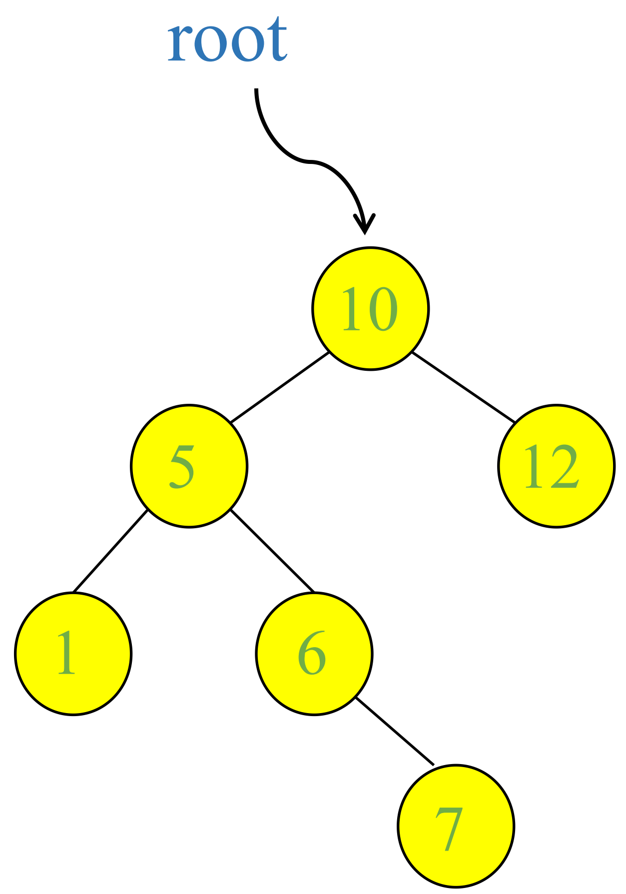
- 如果z只有一个孩子，那么将这个孩子提升到树中z的位置上，并修改z的父结点，用z的孩子來替换z。
- 如果z有两个孩子，那么找z的后继y(一定在z的右子树中），并让y占据树中z的位置。z的原来右子树部分成为y的新的右子树，并且z的左子树成为y的新的左子树。

从一棵二叉搜索树T中删除一个结点z

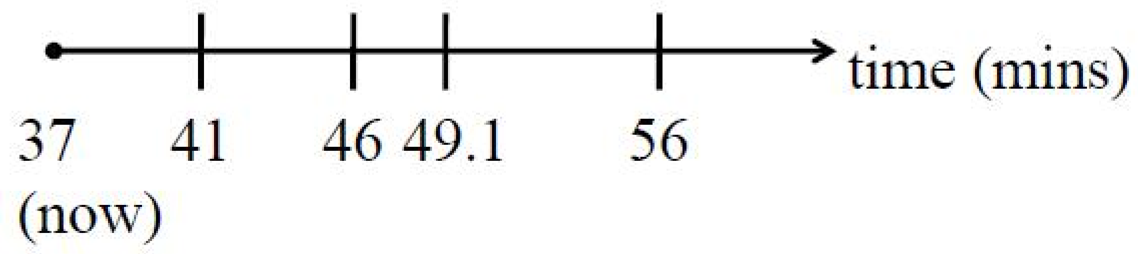
- 如果z有两个孩子，那么找z的后继y(一定在z的右子树中），并让y占据树中z的位置。z的原来右子树部分成为y的新的右子树，并且z的左子树成为y的新的左子树。

BSTs 树的成长

- Insert 10
- Insert 12
- Insert 5
- Insert 1
- Insert 6
- Insert 7



BST 作为数据结构



数据结构的操作

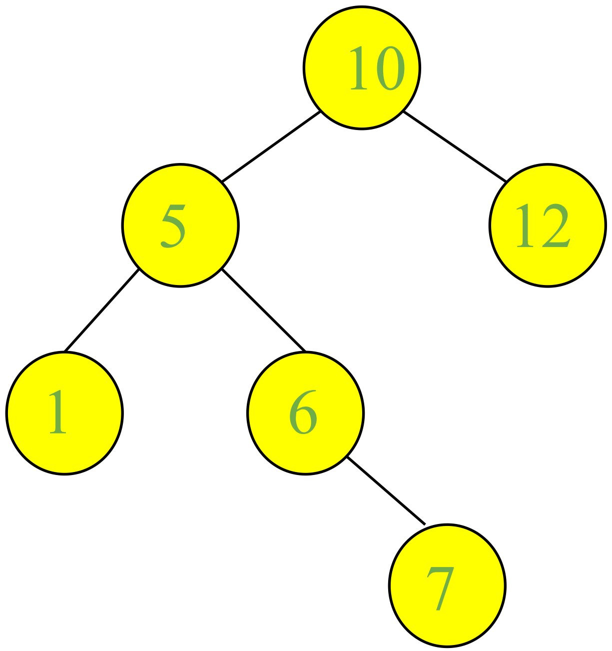
- 插入：插入给定键值k的节点
- 搜索：搜索包含键值k的节点（如果存在）
- 下一个最大节点：找出当前节点x的下一个最大节点
- 找最小节点：找出当前节点x为根节点的子树的最小节点
- 删除节点x

搜索

- 搜索：搜索包含键值k的节点（如果存在）

递归查找左节点或右节点，

- 直到找到k或者不在列表中为止



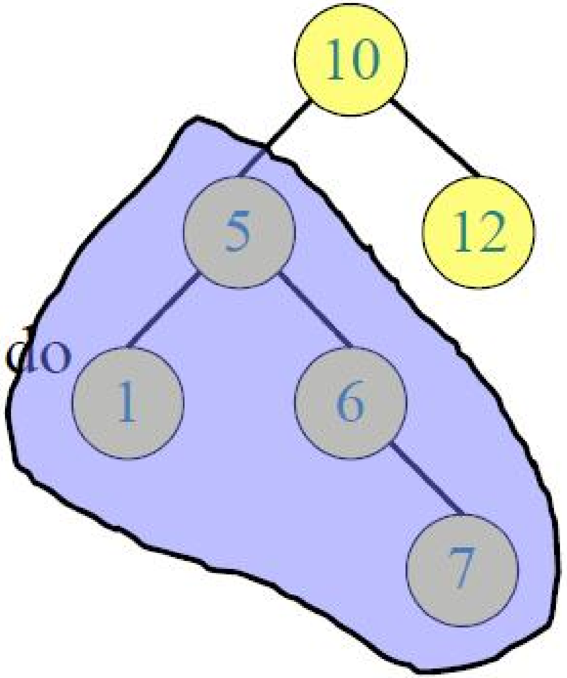
Search(7)
Search(8)

下一个最大节点

下一个最大节点：找出当前节点x的下一个最大节点

next-larger(x):

- If right[x] ≠ NIL then
  return minimum(right[x])
- Otherwise
  y ← p[x]
  While y≠NIL and x=right[y] do
    - x ← y
    - y ← p[y]
  Return y



next-larger(5)

next-larger(7)

最小节点

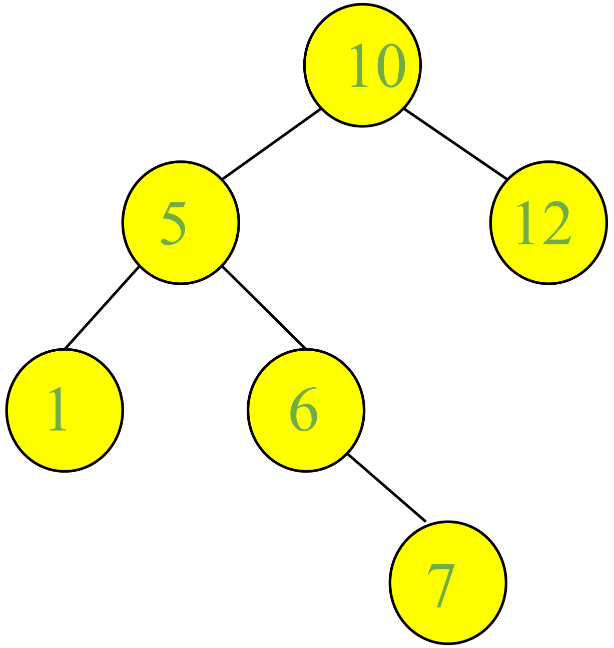
找最小节点：找出当前节点x为根节点的子树的最小节点

Minimum(x)

- While left[x] $\neq$ NIL do

  x $\leftarrow$ left[x]

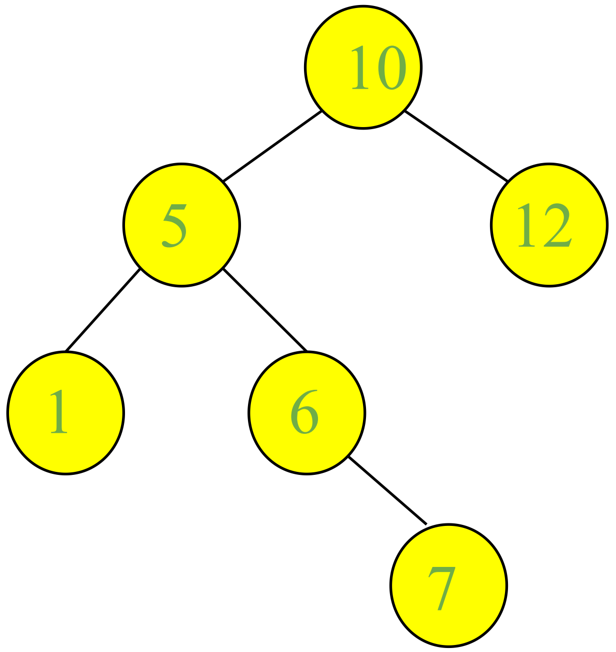
- Return x



minimum(5 **)**

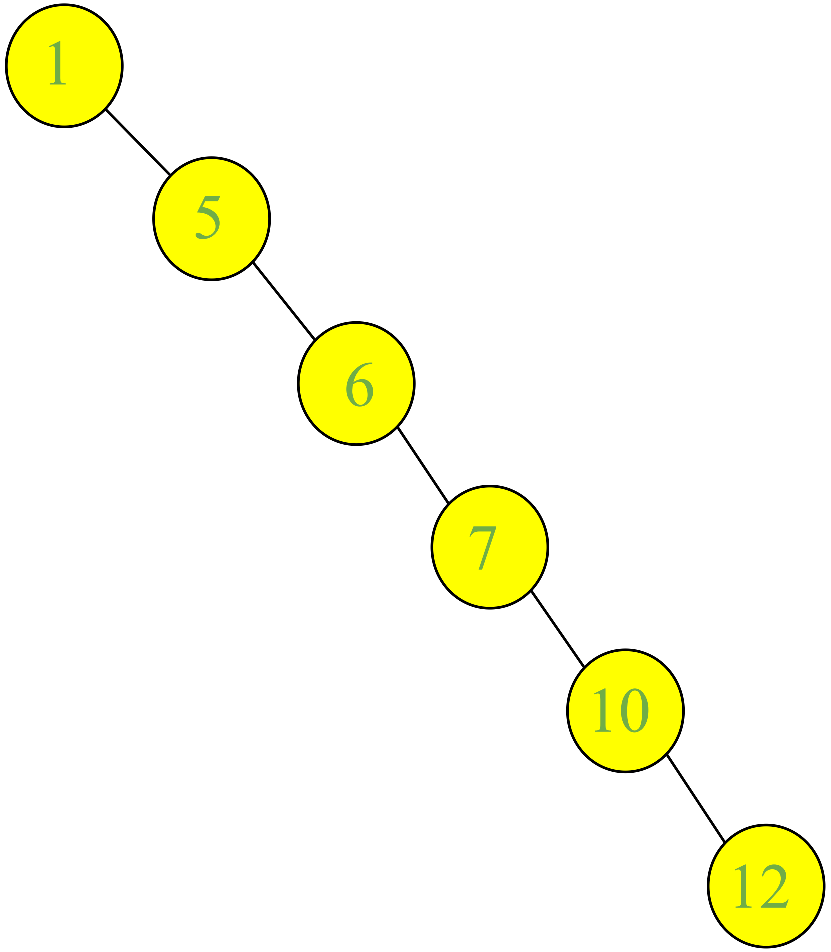
分析

- 我们已经看到了插入、搜索、最小值等操作。
- 这些操作需要多长时间？
- 最坏情况：**O(height)** ➡ 树的高度很重要，所有操作都是O(n）
- 当插入完**n**个元素后，可能的最差二叉搜索树的高度是多少？



分析

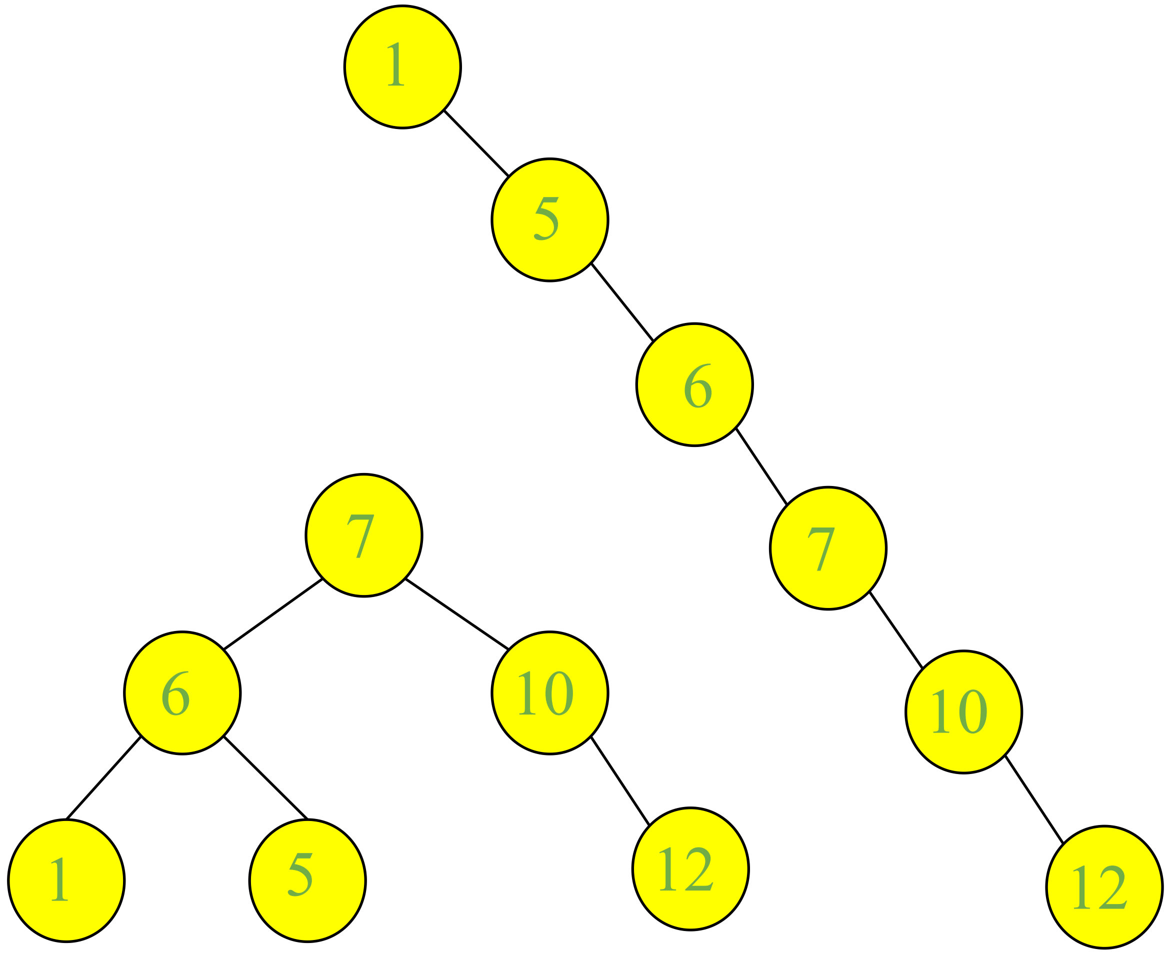
- n-1
- 所以，对于跑道预订系统的操作仍然是O(n)。
- 下一节课：**平衡**二叉搜索树（BST）
- 阅读材料：CLRS 第13.1-2节



回顾

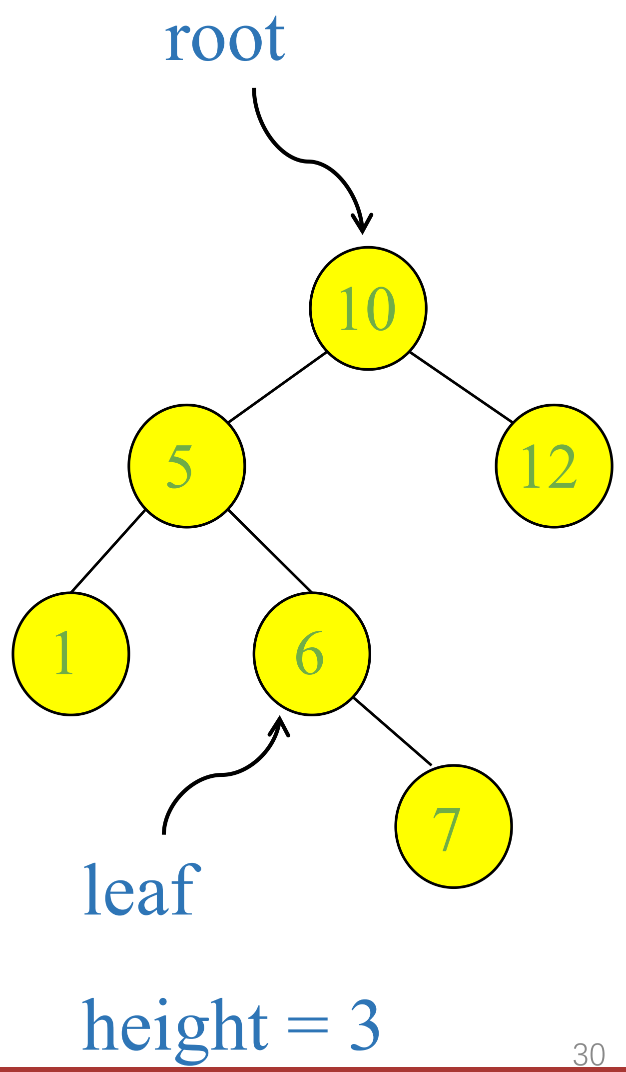
- 二叉搜索树回顾
- 重要性：保持平衡
- 平衡的二叉搜索树（BST）

  -AVL树
  - 定义
  - 旋转，插入



二叉搜索树(BSTs)

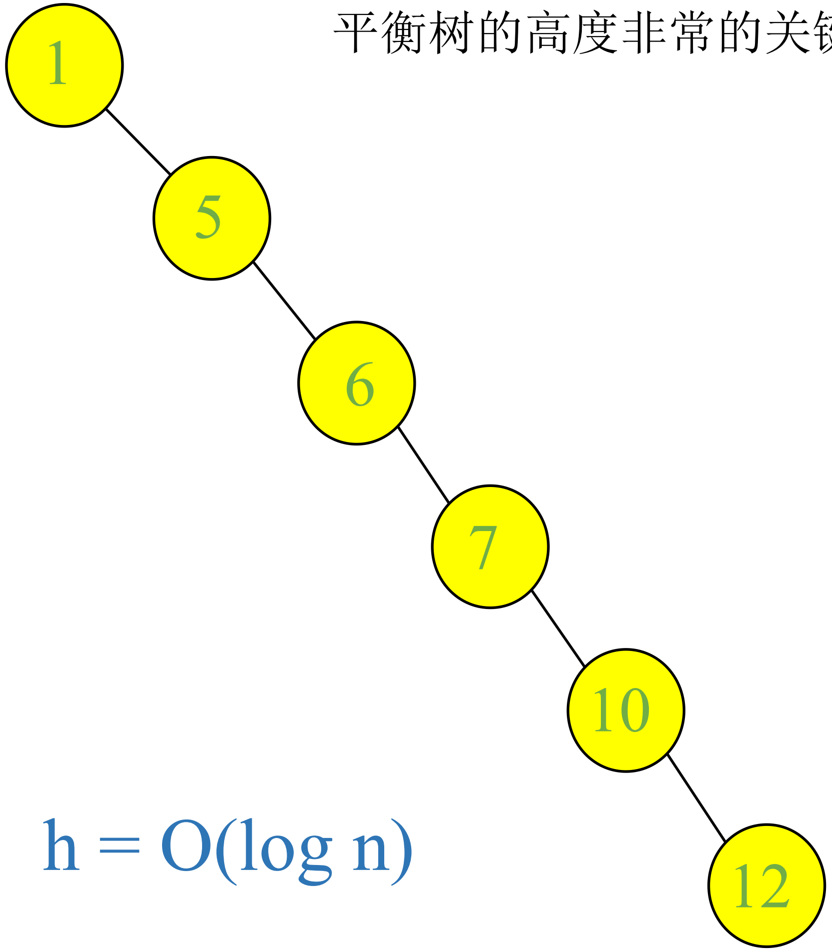
- 每个节点x都有：
  - key[x]
  - 指针：left[x]，right[x]，p[x]
- 属性：对于任何节点x：
  - 对于x的左子树中的所有节点y：**key[y] ≤ key[x]**
  - 对于x的右子树中的所有节点y：**key[y] ≥ key[x]**



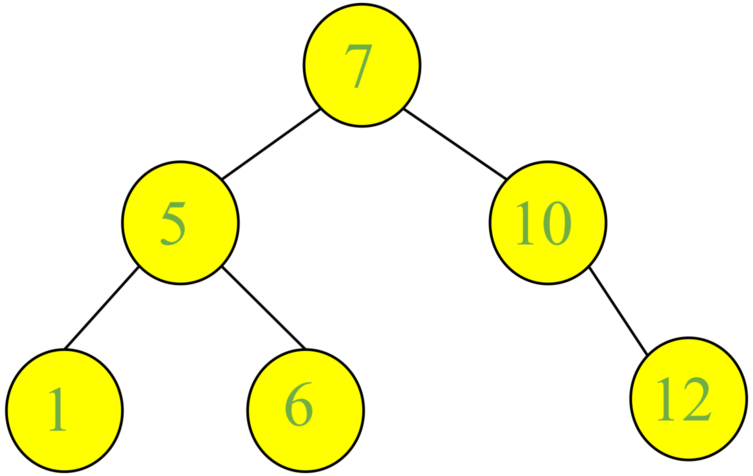
保持平衡的重要性

for n nodes:

平衡树的高度非常的关键



$h = O(\log n)$

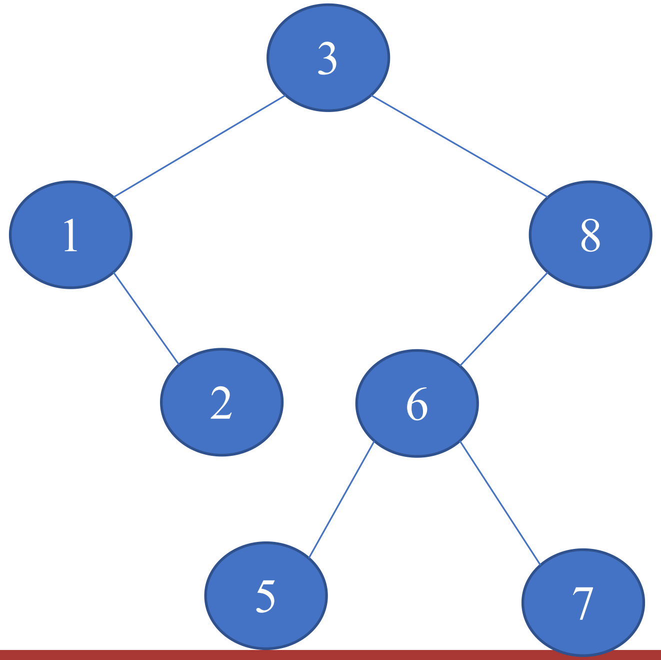


$h = O(n)$

二叉搜索树排序

- 给定一个数组A，构建一个针对A的二叉搜索树（$\Omega(n\log n)$）
- 进行中序遍历

3	1	8	2	6	7	5



时间复杂度

- 给定一个数组A，构建一个针对A的二叉搜索树（$\Omega(n\log n)$）
- 进行中序遍历（$O(n)$）

与快速排序的关系

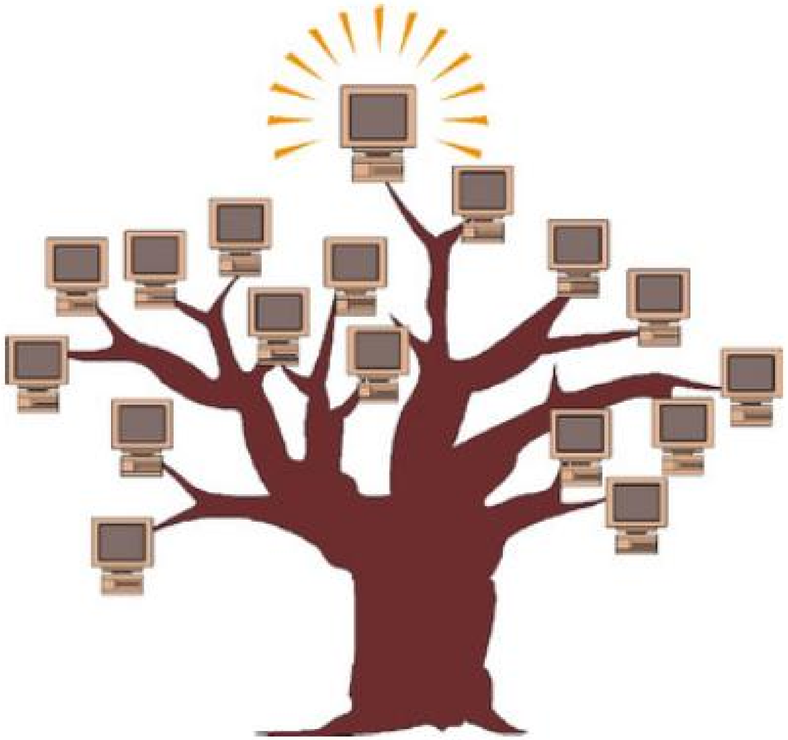
- 二叉搜索树排序中的比较与快速排序中的比较相同。

3	1	8	2	6	7	5

- 我们可以随机化二叉搜索树排序。
- 随机化的二叉搜索树排序具有与随机化快速排序相同的时间复杂度。

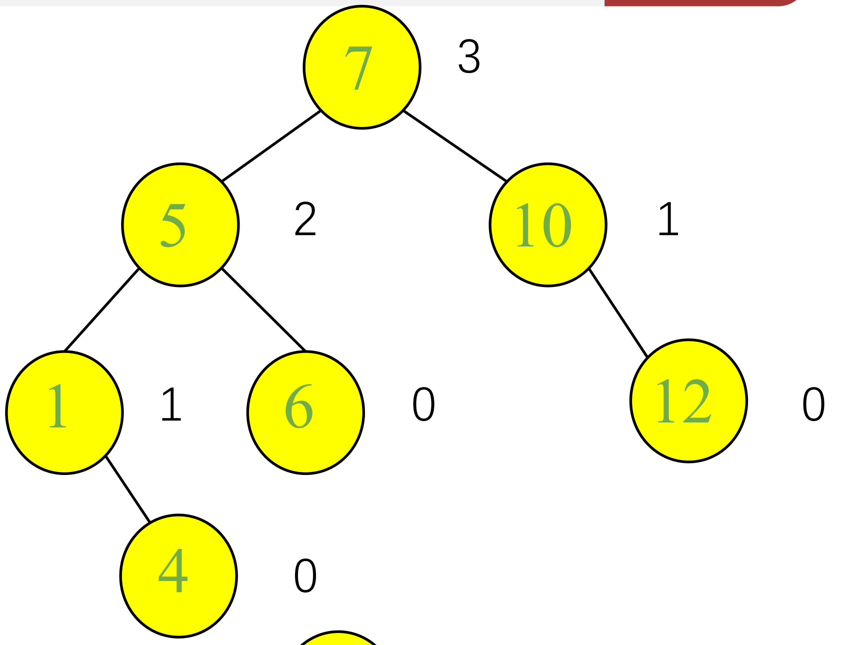
平衡二叉搜索树的策略

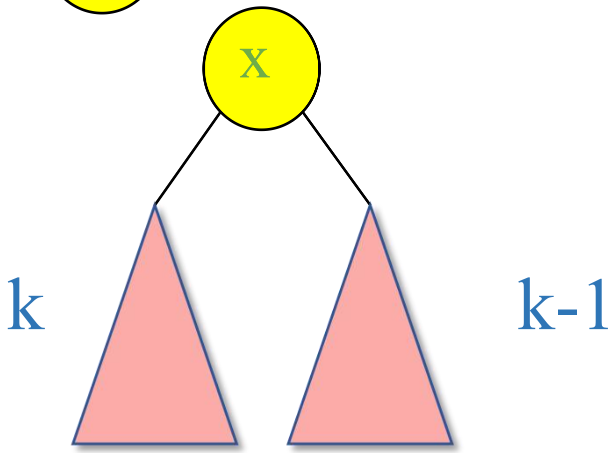
- 给所有的节点增加一些额外的信息
- 给每个本地节点的信息定义一个不变式
- 证明每个本地节点的不变式可以保证树的高度为$O(\log n)$
- 设计算法来维持额外的节点信息和不变式



AVL树：定义

- **信息：**对于每一个节点，维护它的高度（“增加物”）
  - 叶节点高度为0
  - 空节点高度为-1
- **不变式：**对于每一个节点，左子树与右子树的高度差为1





AVL树的高度为 O(log n)

不变式：对于每个节点x，其左子树和右子树的高度差最多为1。

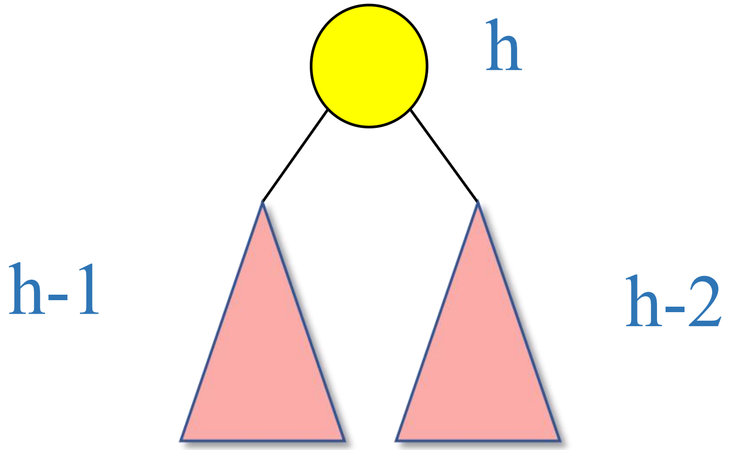
- 设n_h为高度为h的AVL树的最小节点数。
- 我们有$n_h \geq 1 + n_{h-1} + n_{h-2}$

$$n_h > 2n_{h-2}$$

$$n_h > 2^{h/2}$$

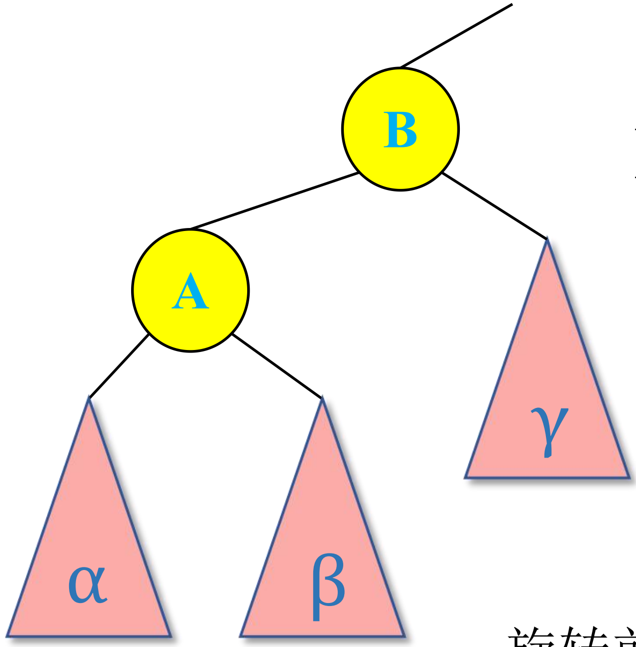
$$h < 2\, lg\, n_h$$

- 常数“2”可以得到改进



每个节点满足不变式的话，上面证明了整棵树的高度必然是$O(\log n)$, 下一步是怎样来维护每个节点的不变式？

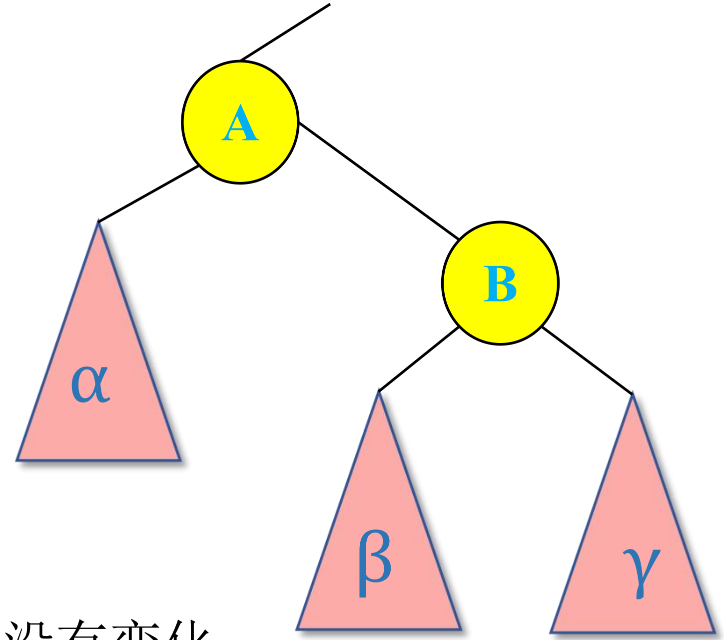
旋转



右旋转(B)
左旋转(A)

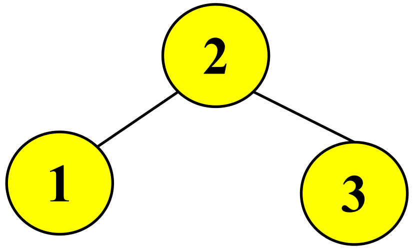
RIGHT-ROTATE(B)

LEFT-ROTATE(A)

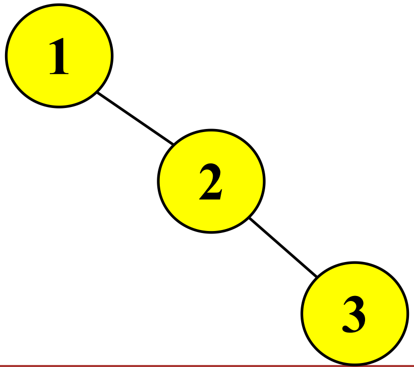


旋转前后节点的键值的排序没有变化

- $a \in \alpha, b \in \beta, c \in \gamma \rightarrow a \leq A \leq b \leq B \leq c$

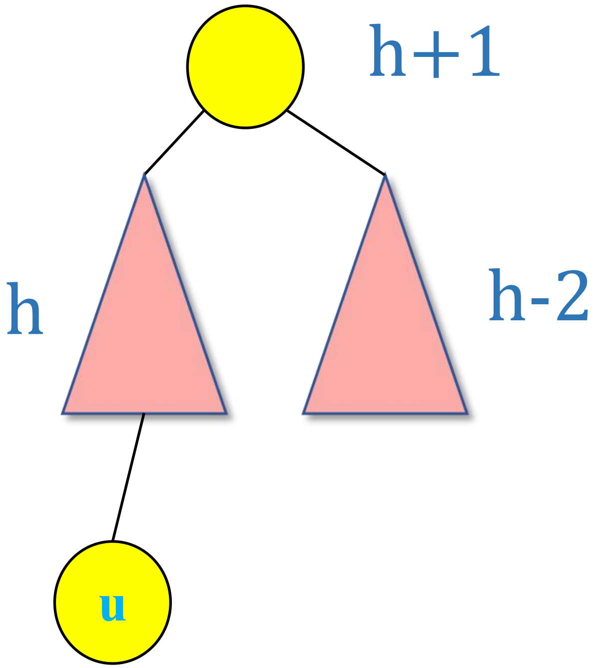


LEFT-ROTATE(1)



插入

- 插入一个新的节点u可能导致不平衡
- 为了解决这个问题，你需要沿着树向上移动，恢复平衡。
- 同样的问题和解决方案也适用于删除节点时的情况。



平衡

- 使X节点是最低的违反属性的节点

  先修复X为根节点的子树，再一步步往上修复

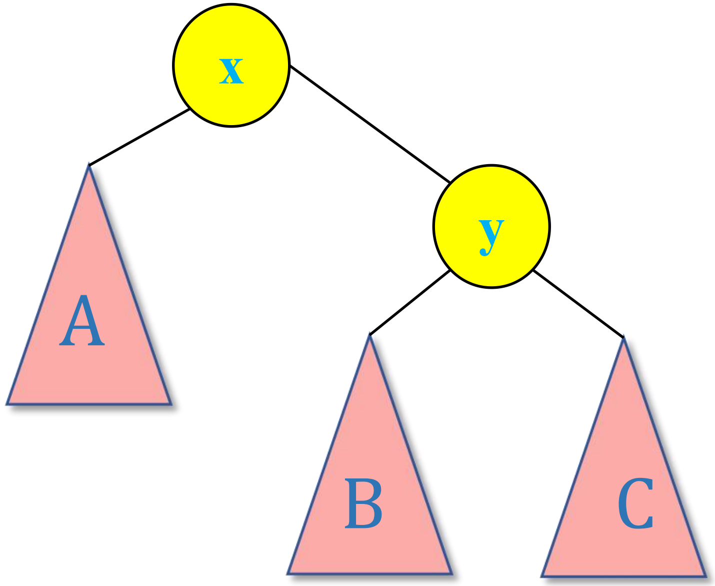
- 右高：假设x右子树比左子树要高

- 可能出现3种情况

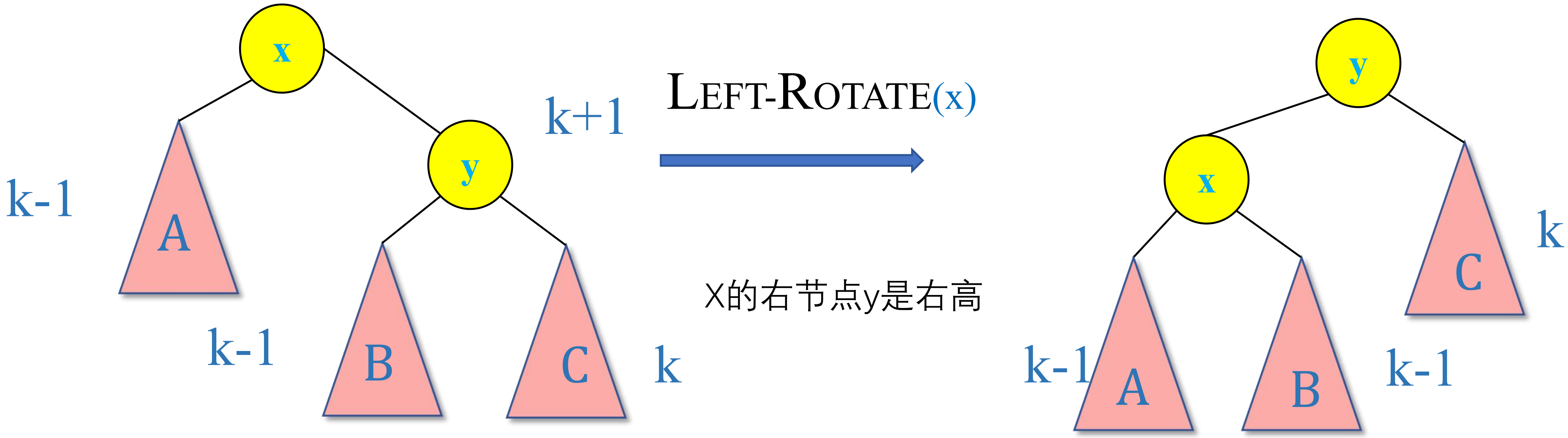
  X的右节点y是右高

  X的右节点y是平衡的

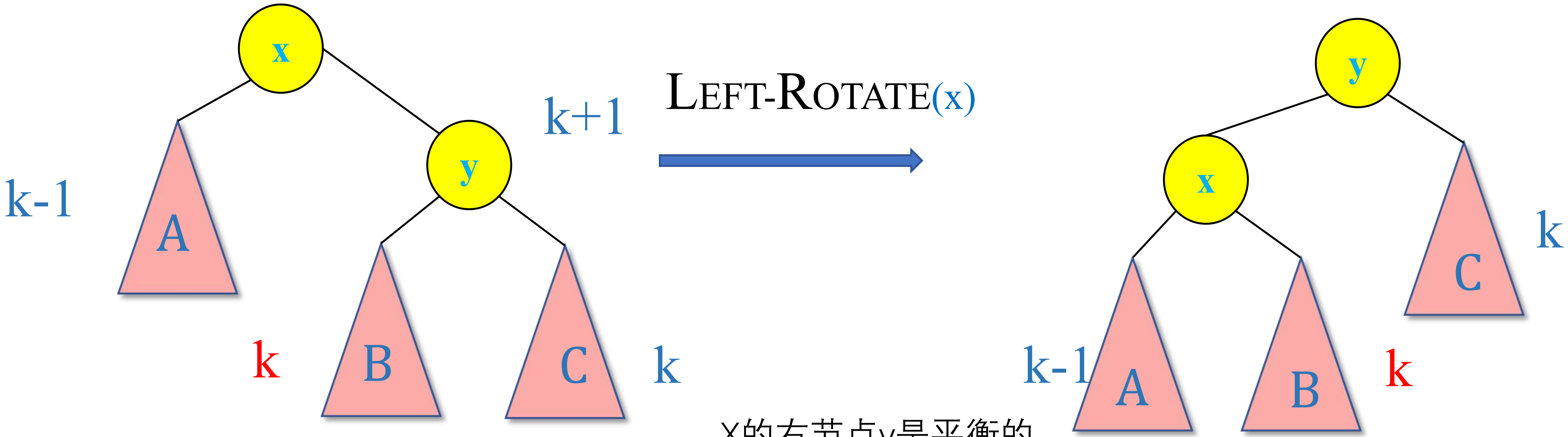
  X的右节点y是左高



情况1：y是右重的

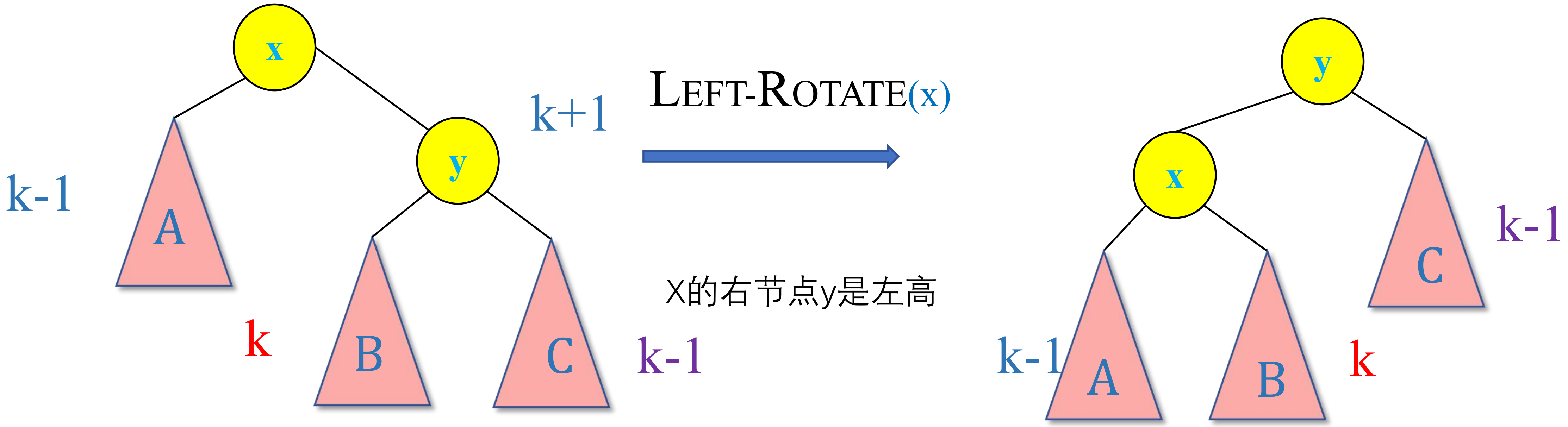


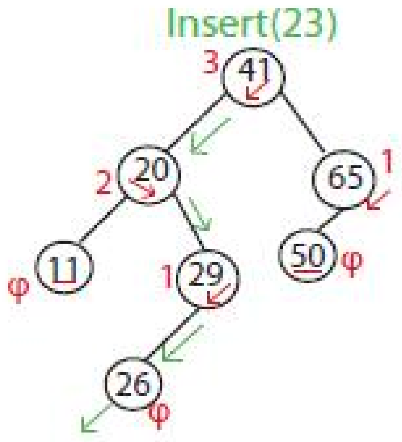
情况2：y是平衡的

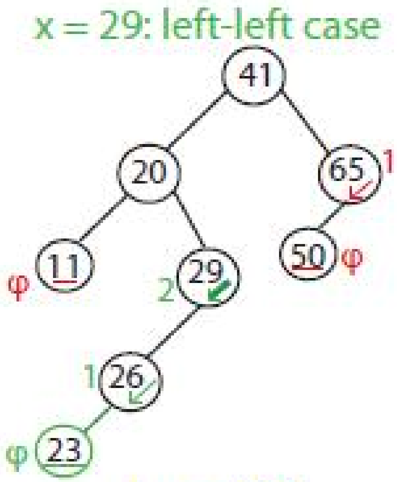


X的右节点y是平衡的

情况3：y是左重的




Insert(23)
3
41
20
2
65
1
11
φ
1
29
50
φ
26
φ

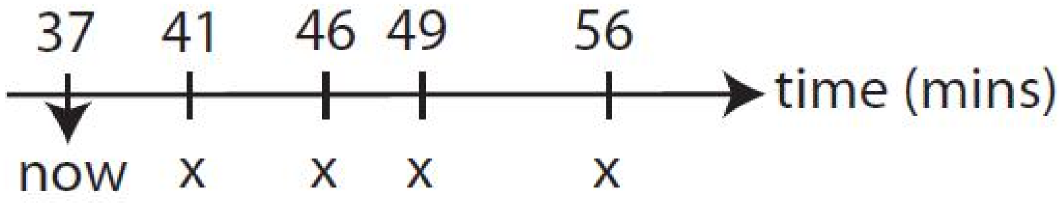

x = 29: left-left case
41
20
65
1
11
φ
2
29
50
φ
1
26
φ
23

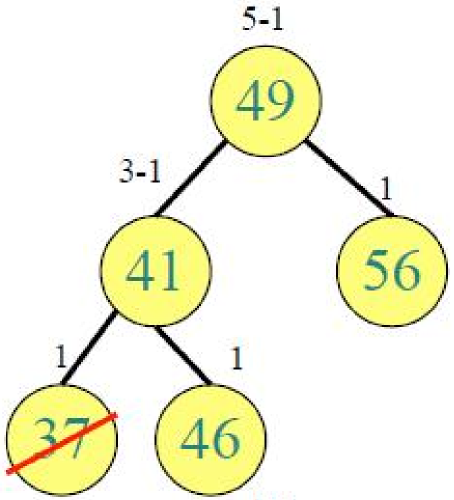
结论

- 可以每次插入时在O(log n)时间内维护平衡BST。
- 搜索等操作需要O(log n)时间。

BST ——跑道预约系统

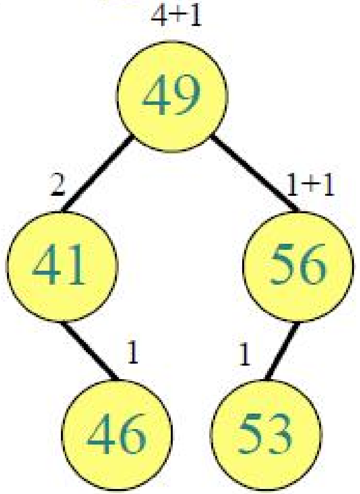
- 当前降落时间集合R={37,41,46,49,56)





- 当一架飞机降落时，从集合中移除时间t

R = (41,46,49,56)



- 如果在距离t不到3分钟的时间内没有其他预定降落，则向集合中添加新的时间t

44=>拒绝（R中的46）　　　53=>好的

- 删除、插入和冲突检查的时间复杂度为O(h)，其中h是树的高度。

平衡搜索树

- AVL树（Adelson-Velsii和Landis于1962年提出）
- 红黑树（参见CLRS第13章）
- 伸展树（Sleator和Tarjan于1985年提出）
- 替罪羊树（Galperin和Rivest于1993年提出）
- Treap（Seidel和Aragon于1996年提出）

红黑树

- 带有额外颜色的BST结构，满足以下条件：

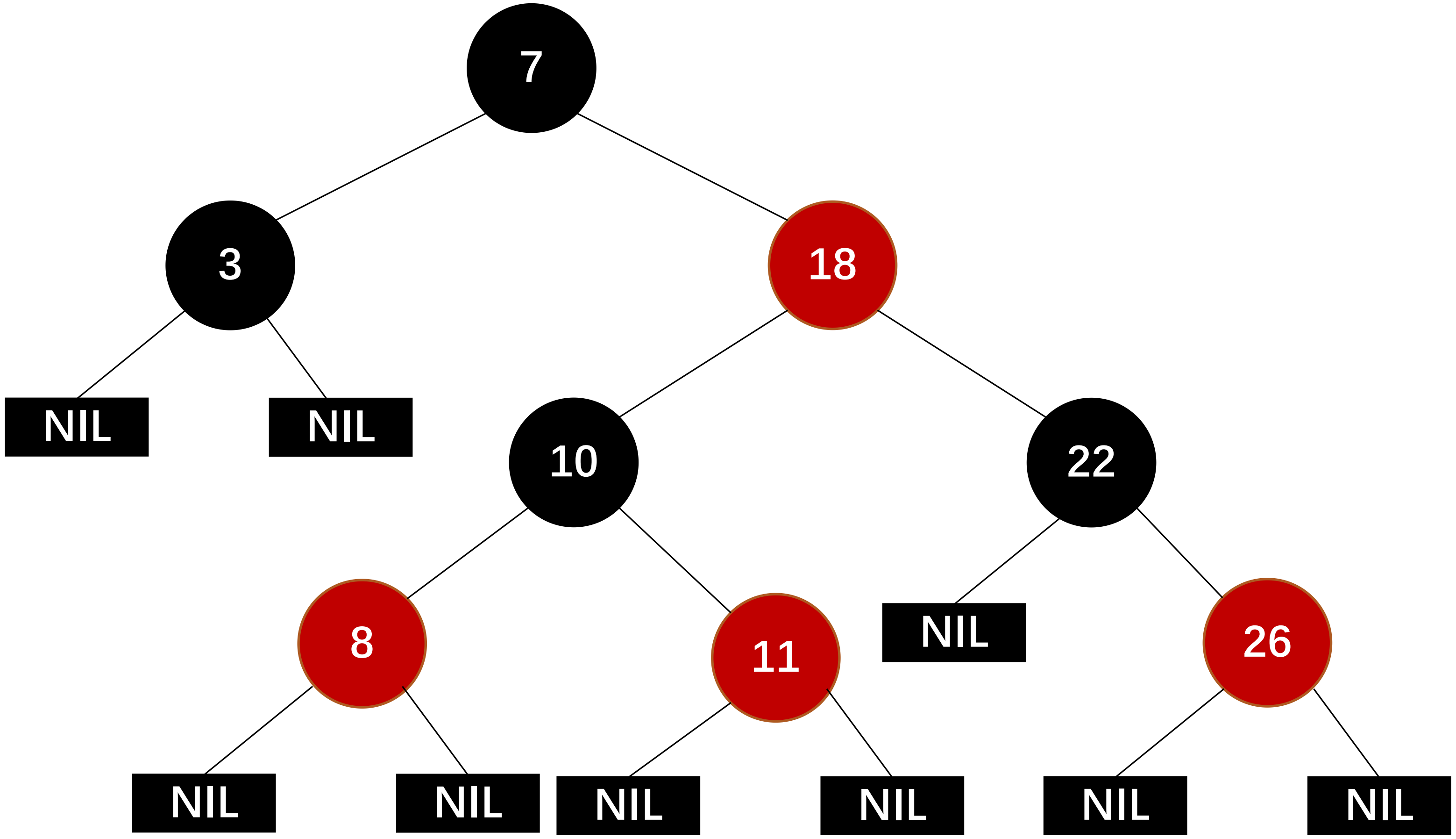
1、每个节点要么是黑色，要么是红色；

2、根和叶子都是黑色的，所有的叶子都是NIL；

3、红色节点的父节点是黑色的；

4、从节点x到其所有后代叶子节点的所有路径中包含相同数量的黑节点。

示例



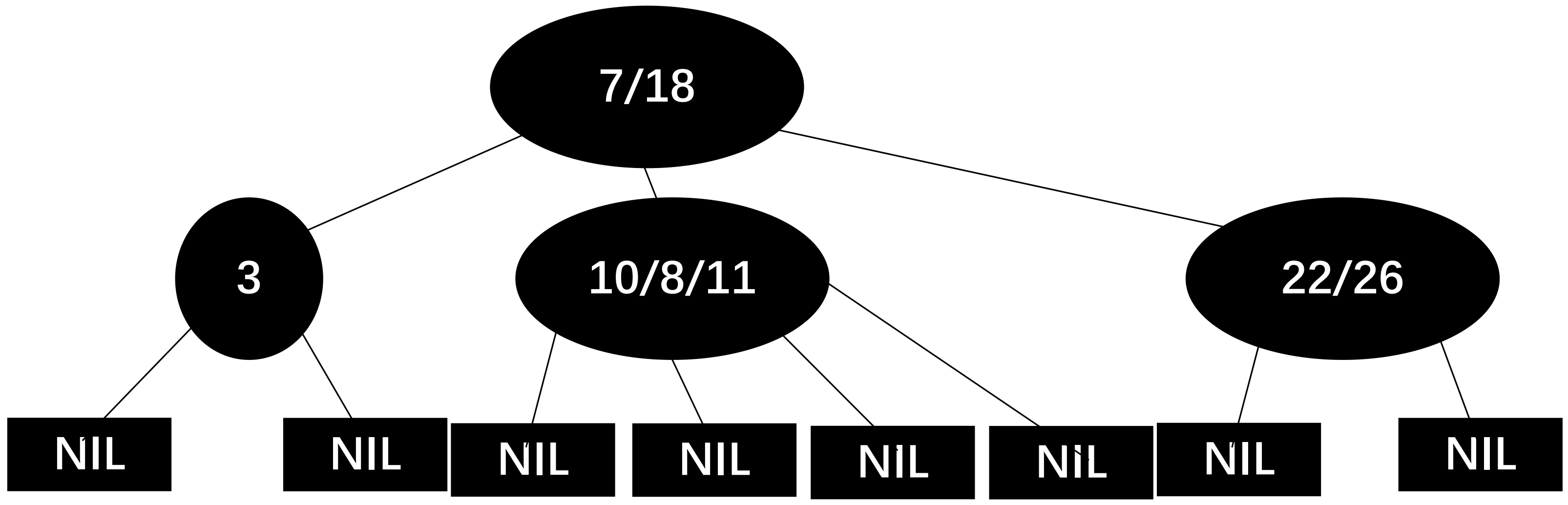
红黑树

- 在红黑树中，从节点x到其所有后代叶子节点的所有路径中相同数量的黑节点被称为x的黑高度。
- 假设红黑树中有n个节点，则有n+1个叶子（通过归纳法证明）。

红黑树的高度

- 红黑树的高度小于 $2\log(n+1)=O(\log n)$
- 证明：将红色节点与其黑色父节点合并。然后，树变成了一个2-3-4树，其中每个节点有2、3或4个子节点，所有叶子具有相同的深度，即黑色高度h'。

证明示例

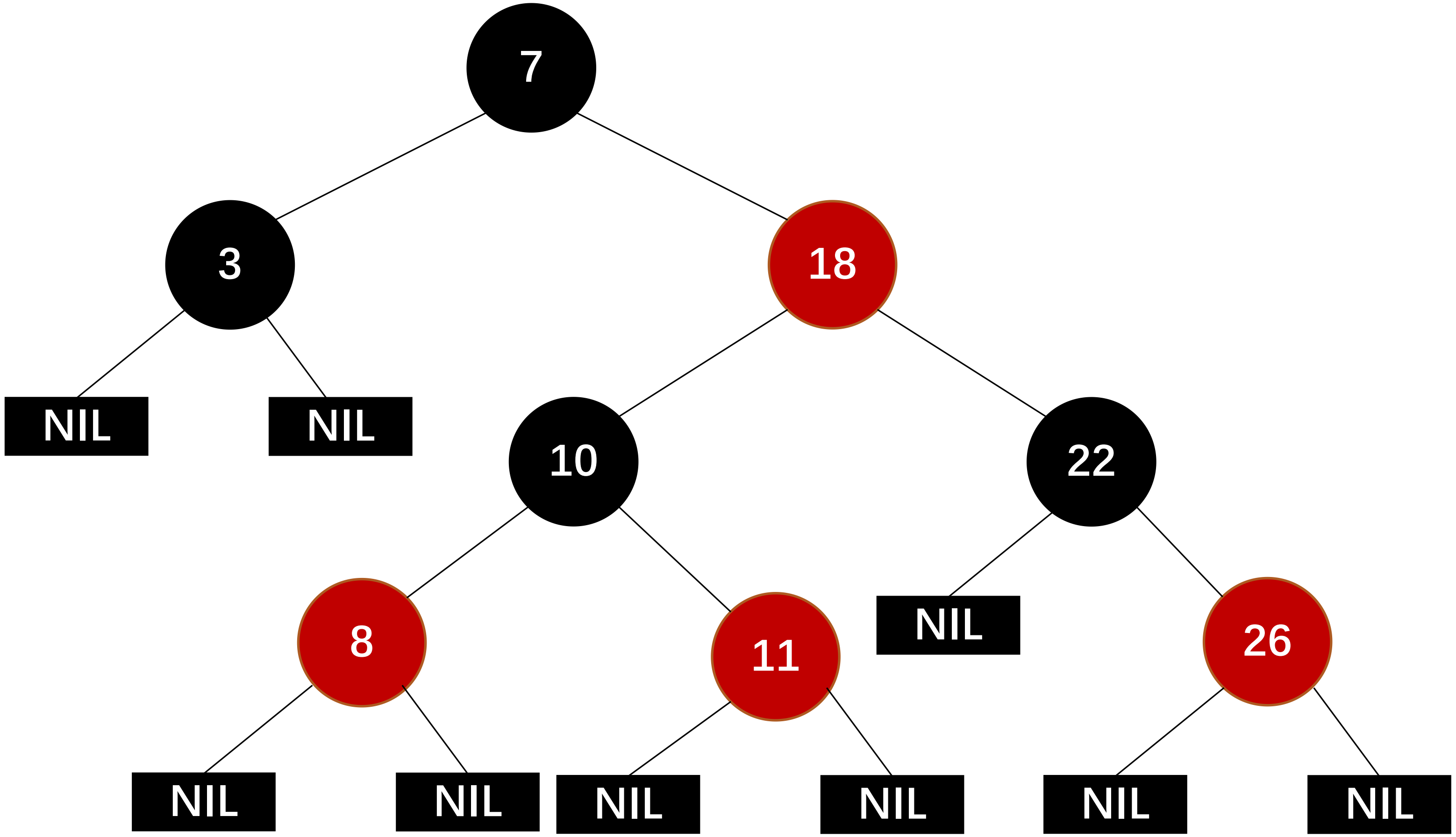


红黑树的高度

- 红黑树的高度小于 2log(n+1)=O(log n)
- 证明: $2^{h'} \leq \#leaves \leq 4^{h'} \Rightarrow 2^{h'} \leq n+1 \Rightarrow h' \leq log(n+1)$
- 因此，红黑树的高度h小于 2h', 所以 $h \leq 2log(n+1)$

红黑树的搜索

- 像BST一样搜索，查询成本为O(log n)



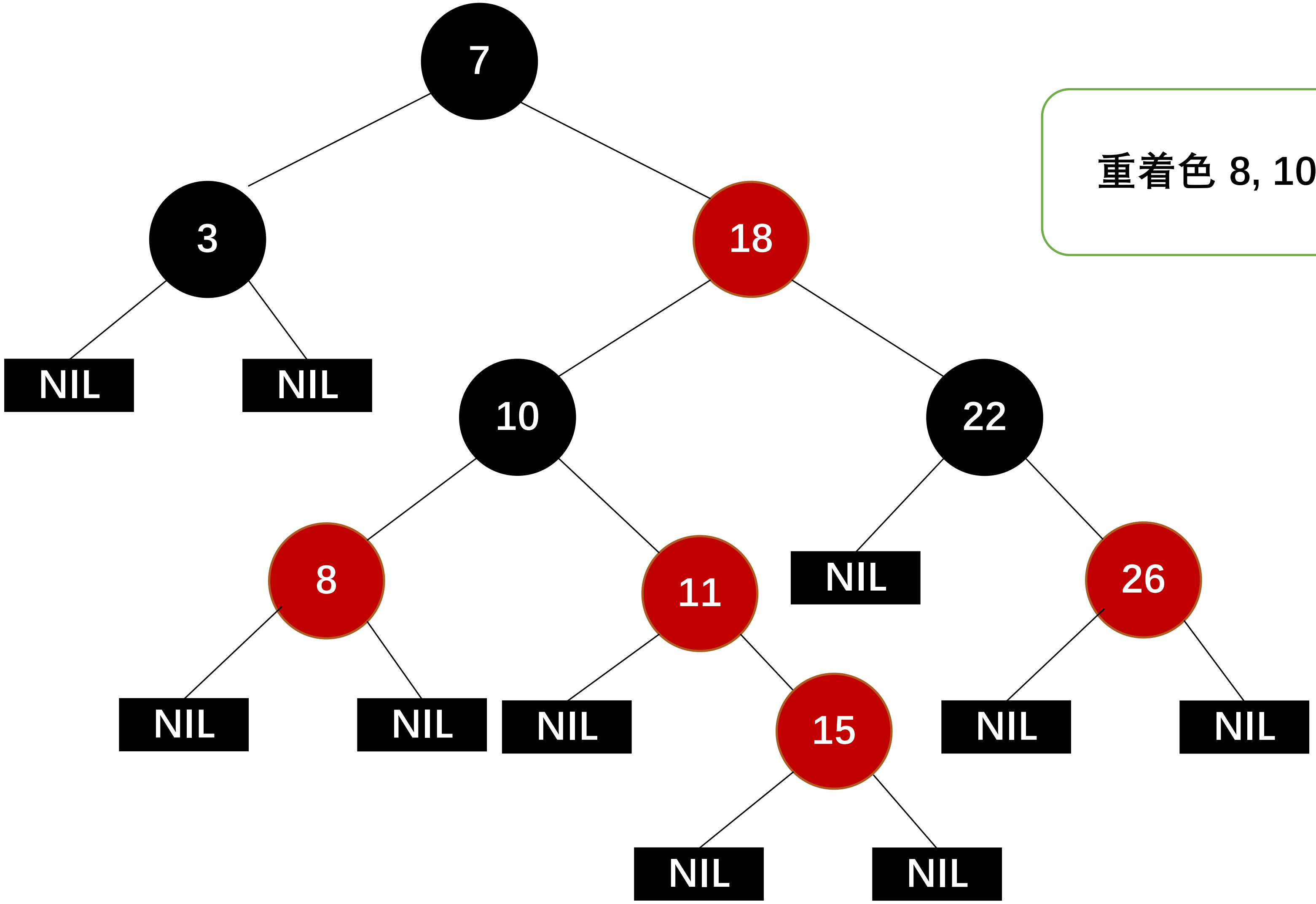
红黑树的插入

```
RB-INSERT(T, z)
    y = T.nil
    x = T.root
    while x ≠ T.nil
        y = x
        if z.key < x.key
        x = x.left
        else x = x.right
    z.p = y
    if y == T.nil
        T.root = z
    elseif z.key < y.key
        y.left = z
    else y.right = z
    z.left = T.nil
    z.right = T.nil
    z.color = RED
    RB-INSERT-FIXUP(T, z)
```

- 更新成本为O(log n)

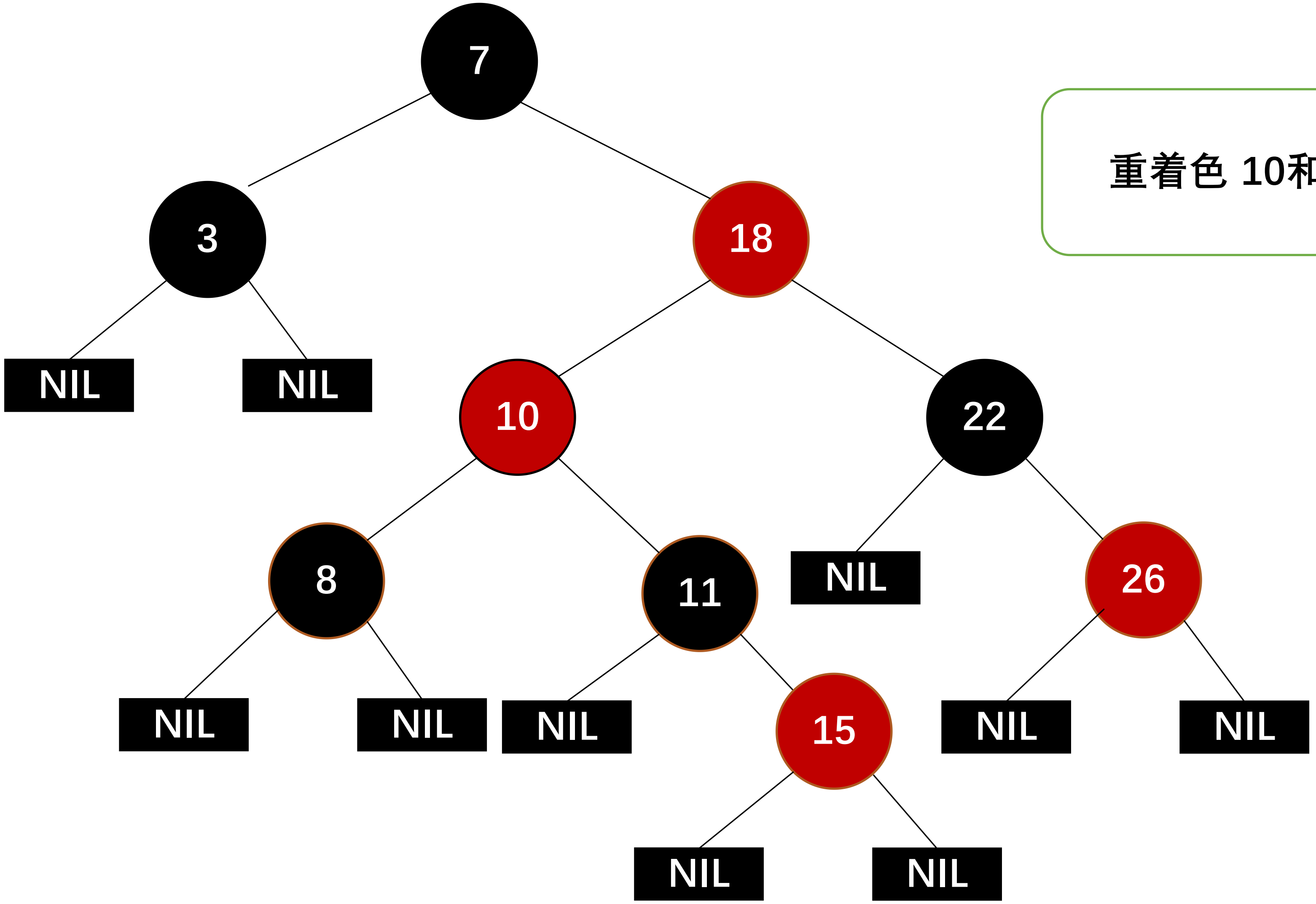
```
RB-INSERT-FIXUP(T, z)
    while z.p.color == RED
        if z.p == z.p.p.left
            y == z.p.p.right
            if y.color ==RED
                z.p.color = BLACK          // case1
                y.color = BLACK            // case1
                z.p.p.color = RED          // case1
                z = z.p.p                  // case1
    else if z == z.p.right
                    z = z.p                // case2
                    LEFT-ROTATE(T, z)      // case2
            z.p.color = BLACK              // case3
            z.p.p.color = RED              // case3
            RIGHT-ROTATE(T,z,p,p)          // case3
    else(same as then clause with "right" and "left" exchanged)
    T.root.color = BLACK
```

示例插入15



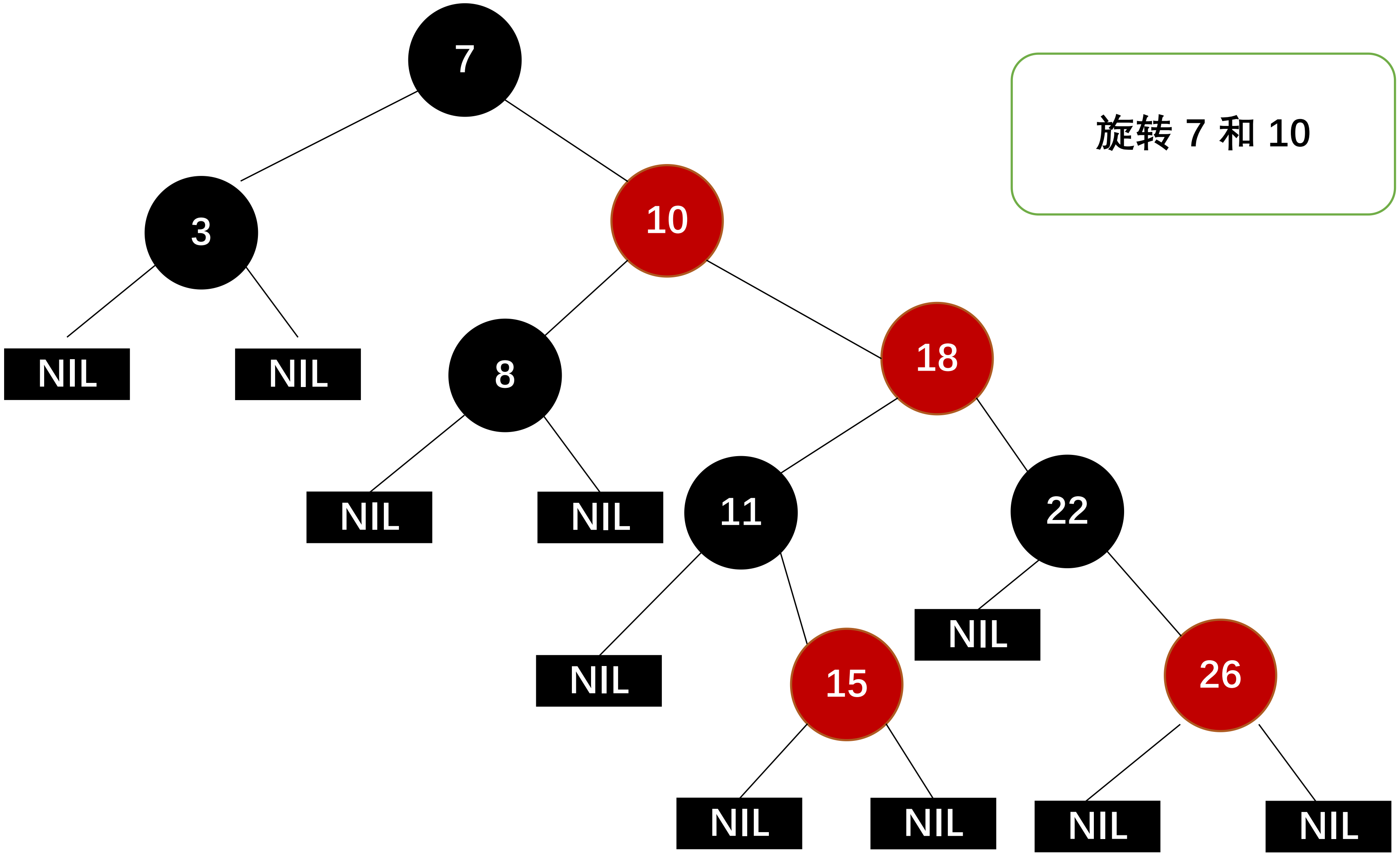
重着色 8, 10, 11

示例插入15



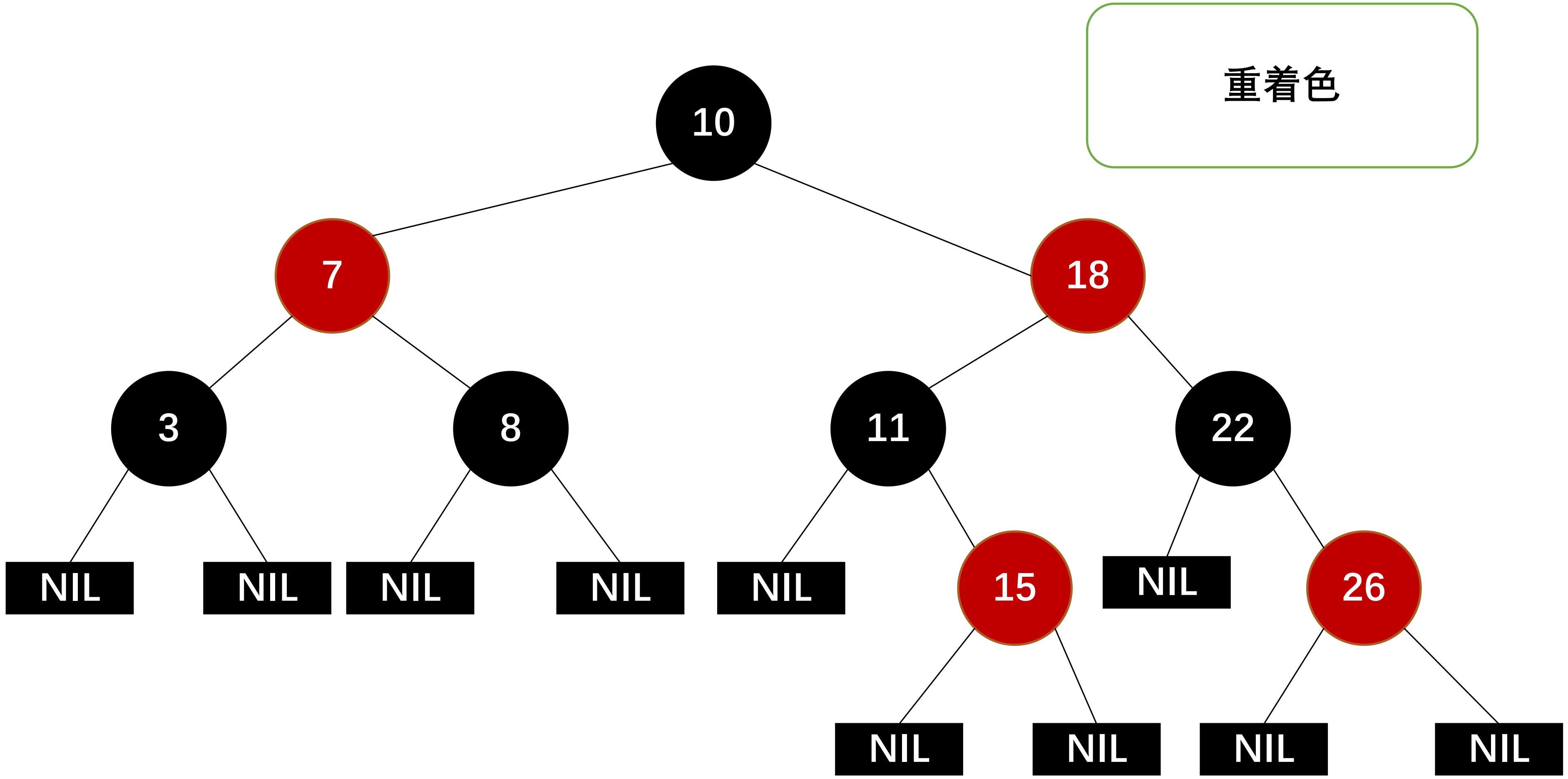
重着色 10和18

示例插入15



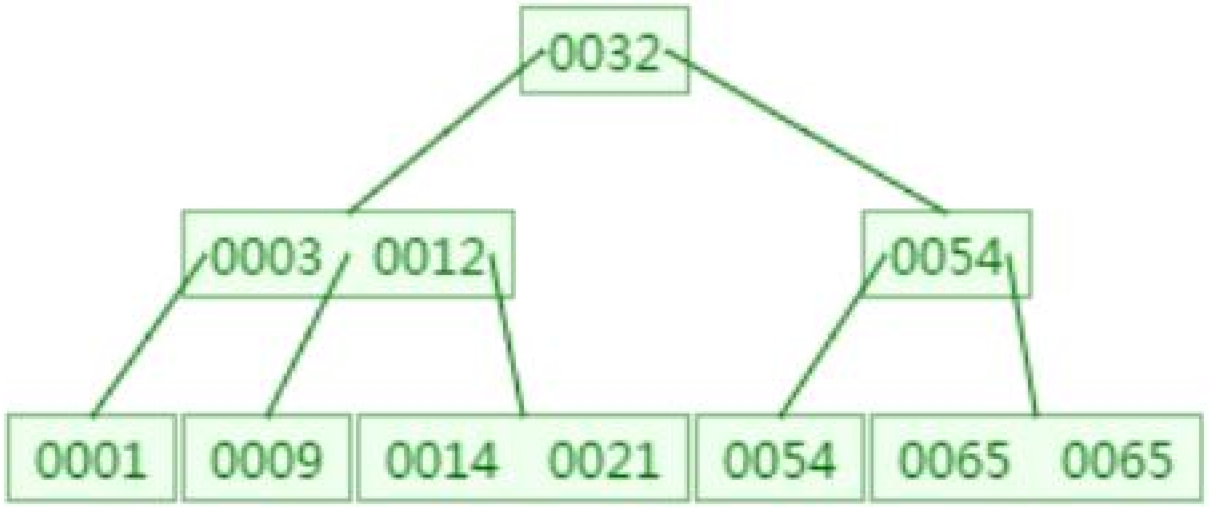
旋转 7 和 10

示例插入15



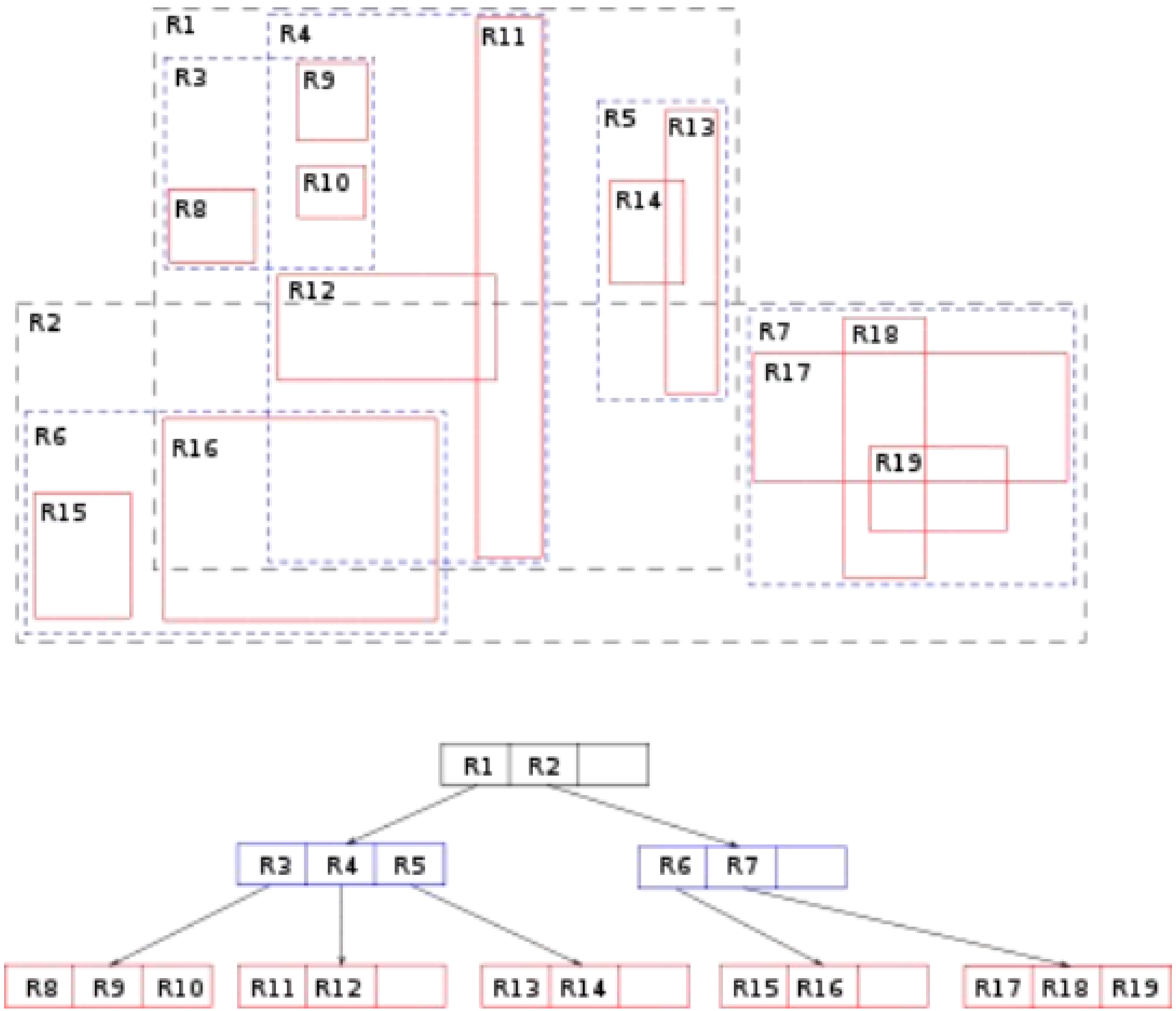
B树

- 在B树中，内部（非叶子）节点可以在预定义的范围[m/2,m]内具有可变数量的子节点。



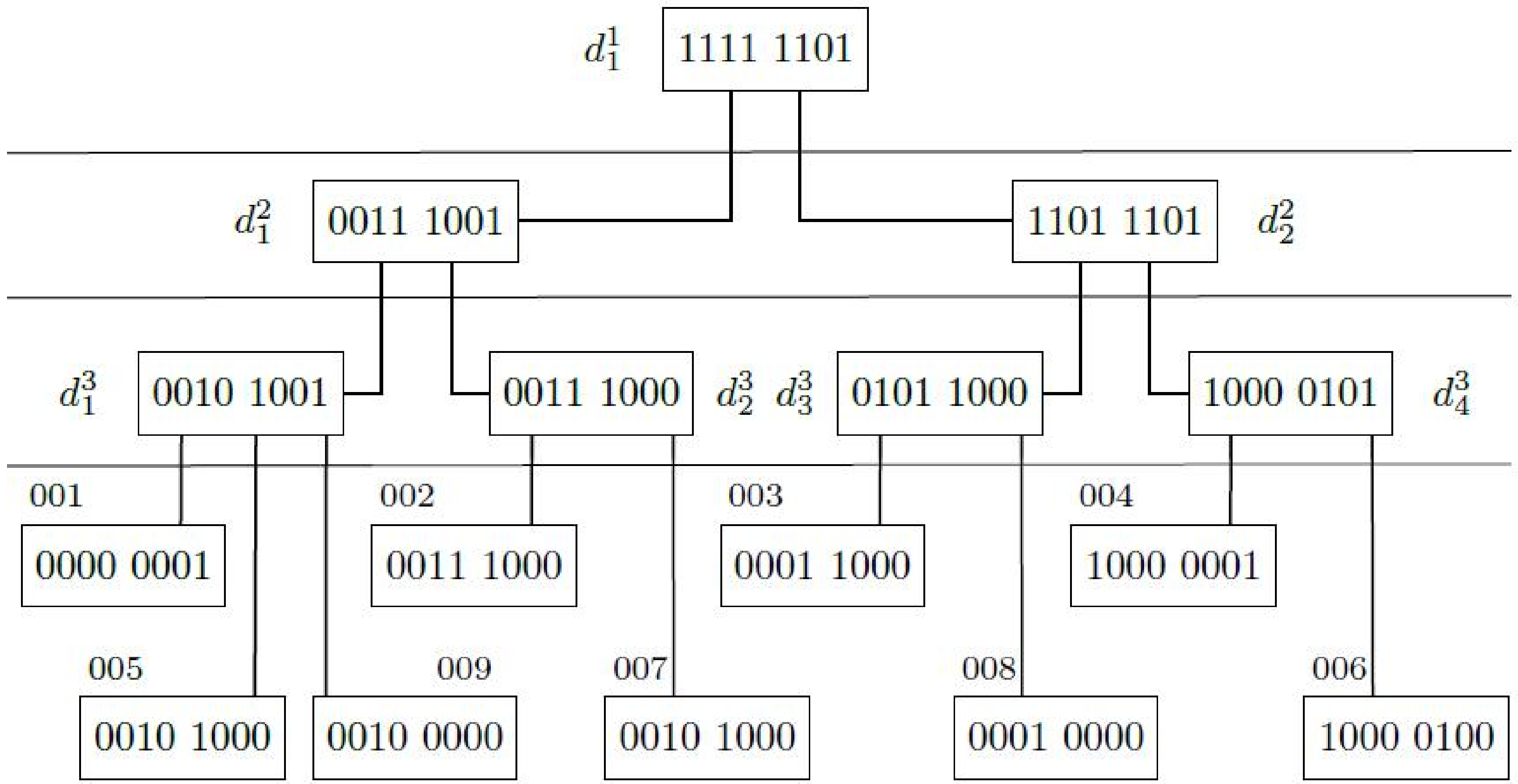
R树

- R树是一种用于空间访问方法的树形数据结构，即用于索引多维信息，如地理坐标、矩形或多边形。


R1
R4
R11
R3
R9
R5
R13
R10
R14
R8
R12
R2
R7
R18
R17
R6
R16
R19
R15
R1 R2
R3 R4 R5
R6 R7
R8 R9 R10
R11 R12
R13 R14
R15 R16
R17 R18 R19

签名树

- 由相应的叶节点指向的签名。



湖南大学
HUNAN UNIVERSITY

HNU

谢谢观赏

实事求是　敢为人先