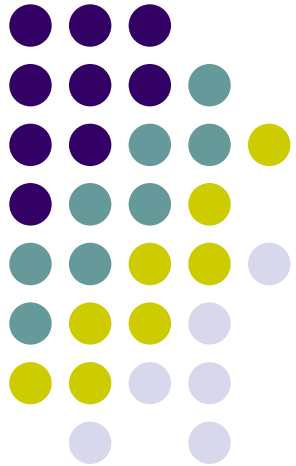
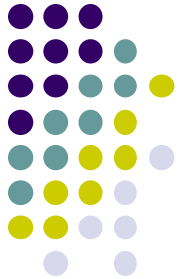
BST和AVL

吉林大学计算机学院
谷方明
fmgu2002@sina.com



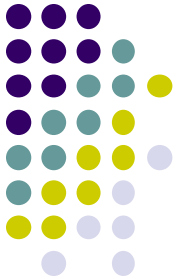
树的引入



- 对半查找、斐波那契查找等都对应一棵隐式的(implicit)二叉树；

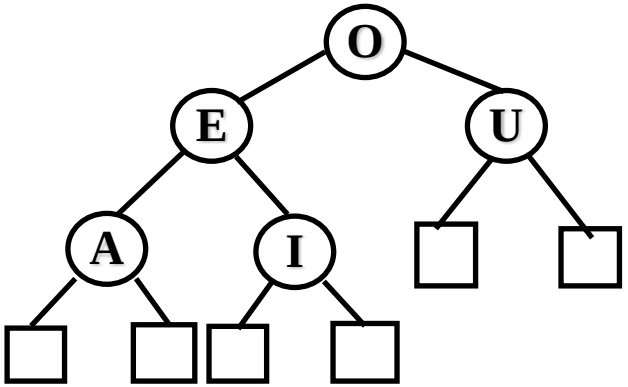
- 用一个显式的(explicit)二叉树结构组织数据进行查找；
  - ✓ 不但能对表进行有效查找，
  - ✓ 还能迅速插入和删除（动态表）。

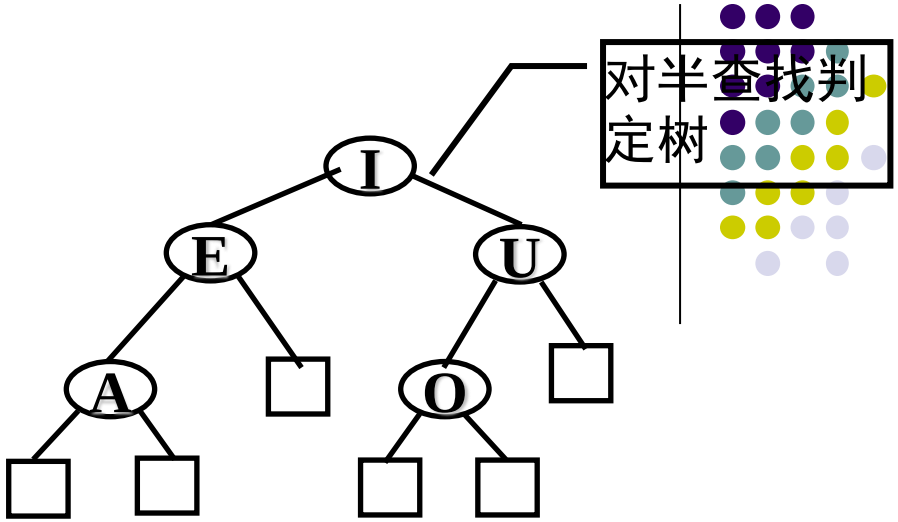
二叉查找树(BST)概念

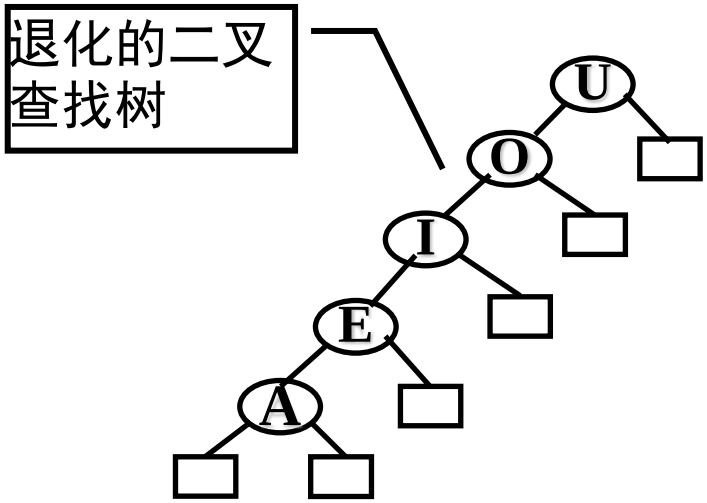


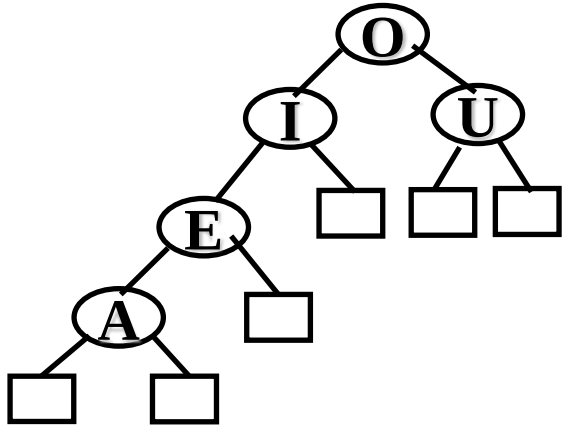
- **BST左子树中结点的关键词都小于KEY(*root*)，其右子树中结点的关键词都大于KEY(*root*)，且左右子树都是二叉查找树**

- **递归版定义，可编写递归算法**

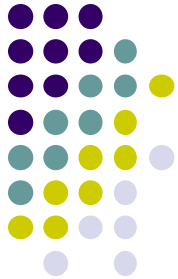

O
E
U
A
I


对半查找判定树
I
E
U
A
O


退化的二叉查找树
U
O
I
E
A


O
I
U
E
A

二叉查找树VS对半查找判定树



- 联系
  - ✓ 对半查找判定树是一种二叉查找树
  - ✓ 都可用于描述查找
- 区别
  - ✓ 二叉查找树是一种显式树结构；对半查找判定树是一种隐式树结构；
  - ✓ 二叉查找树直接用于查找；对半查找判定树用于分析查找

其它定义

- **BST中任一结点*P*，其左子树中结点的关键词都小于KEY(*P*)，其右子树中结点的关键词都大于KEY(*P*)。**
  - ✓ 二叉搜索树性质，可迭代处理。

- **定义8.1 二叉查找树（二叉搜索树、二叉排序树）：BST是一棵可为空的二叉树；若BST非空，则其中根遍历序列按关键词递增排列。**
  - ✓ 中序遍历有序，故又称二叉排序树

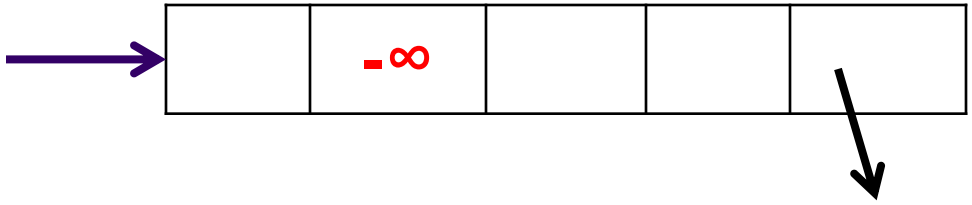
二叉查找树的存储结构

- 链式存储：结点结构如下

LLINK	KEY	DATA	PARENT	RLINK

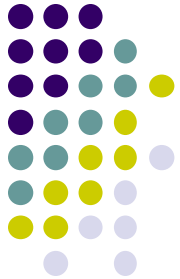
LLINK和RLINK是链接字段，KEY为关键词。

- 可让表头指针root指向一个哨兵变量，最小



LLINK	KEY	DATA	PARENT	RLINK

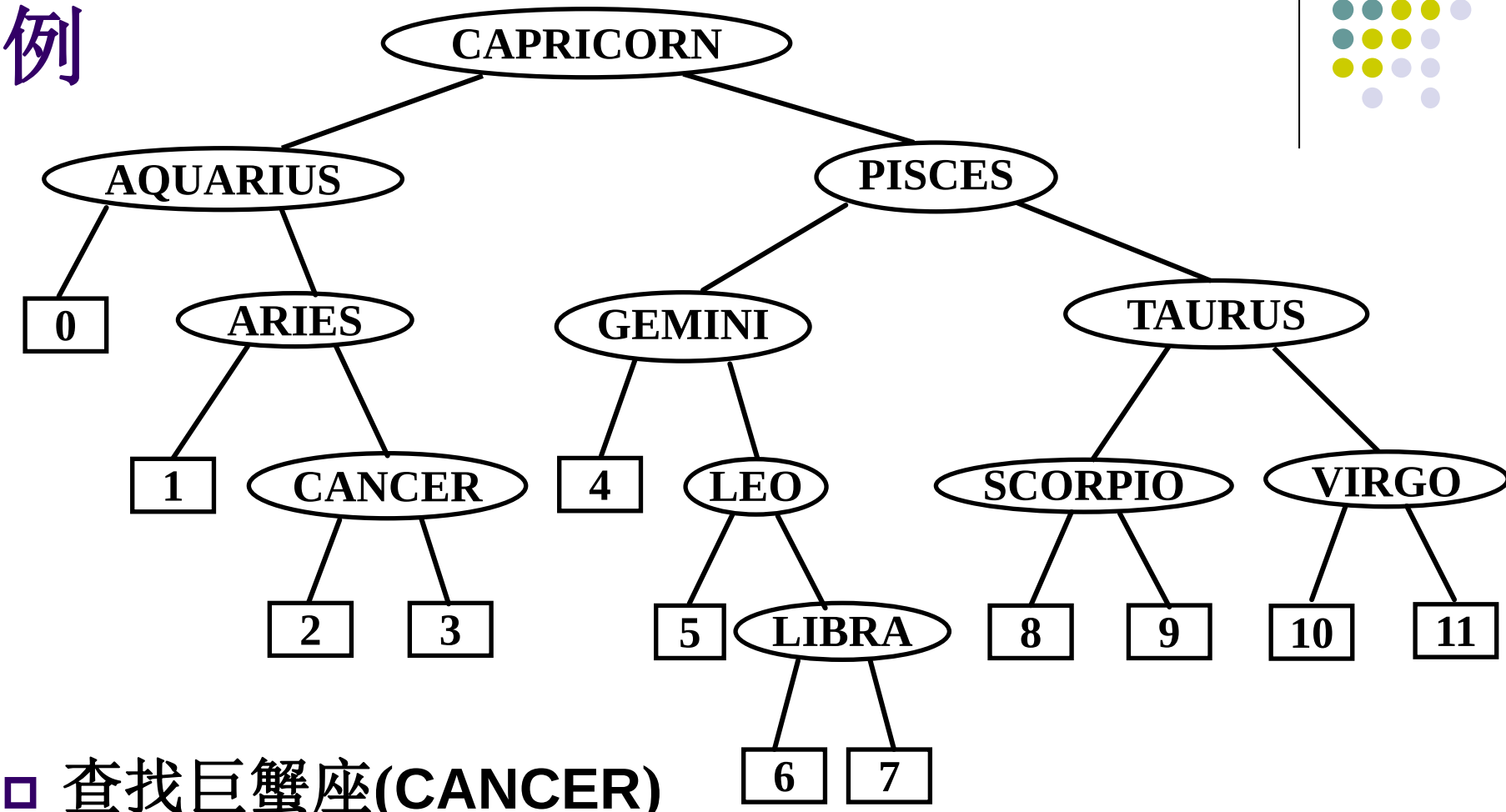
二叉查找树的基本操作



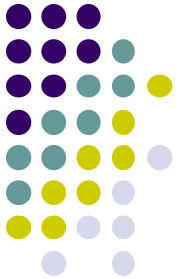
- 查找 （**key-value**）
- **FindMin/FindMax**
- **Predecessor/Successor**
- 插入
- 删除
- 创建
- **……**

//BST既可以作字典，也可以作优先级队列。

例



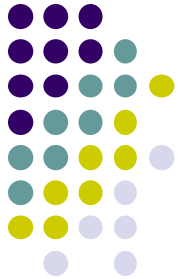
- 查找巨蟹座(CANCER)
- 查找第12个名字人马座（SAGITTARIUS）



- 设表中元素的关键词$K_1<K_2<\ldots<K_N$，则查找成功应该终止于R_i(内结点)，而查找失败应终止在N＋1个记录间隔(或者称外结点，即$K_i<K<K_{N+1}$的情形)。

- 查找失败可以直接返回，也可以在失败位置插入新结点，称为查找与插入操作；

Find (t, k)



算法 Find (t, k)

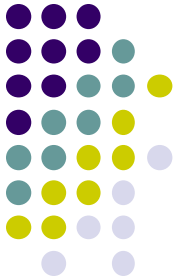
/*在以t为根的树中寻找关键词为k的结点，迭代版*/

F1 [由根迭代比较]

```
while ( t != NULL && t->key != k )
        if ( k < t->key ) t=t->llink;
         else t=t->rlink;
 return t;
```

//时间复杂度： 树高O(h)

FindMin/FindMax

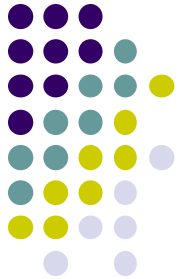


- 算法FindMin(t)

```
/* 在以t为根的BST中找最小元素*/
F1[一路向左]
    if( t != NULL )
        while( t->llink != NULL) t = t->llink;
    return t;

//时间复杂度： 树高O(h)
```

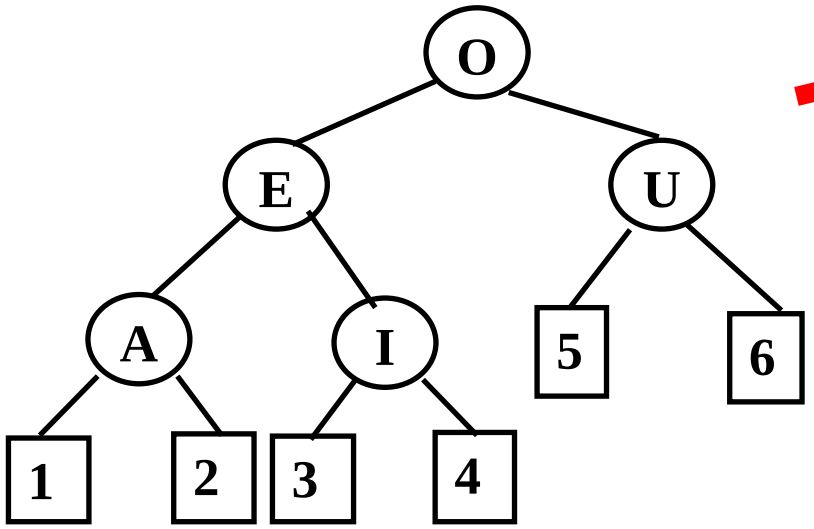
Predecessor/Successor

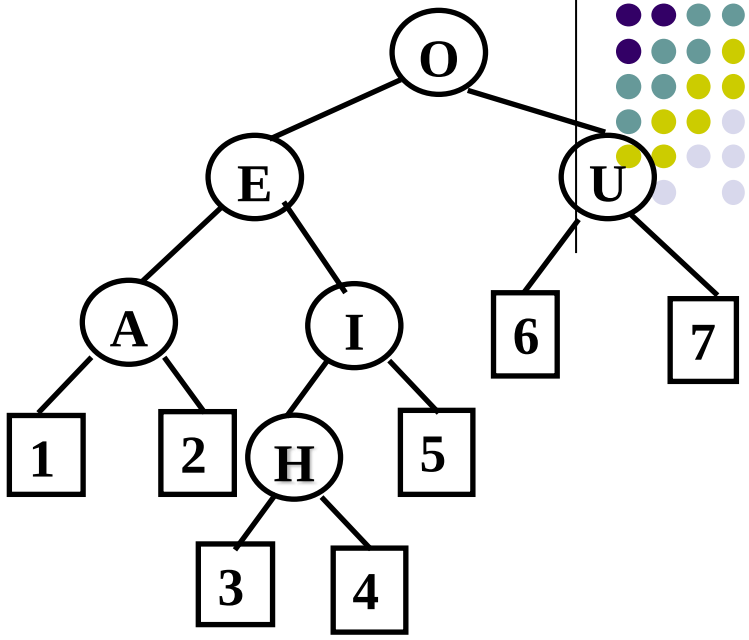


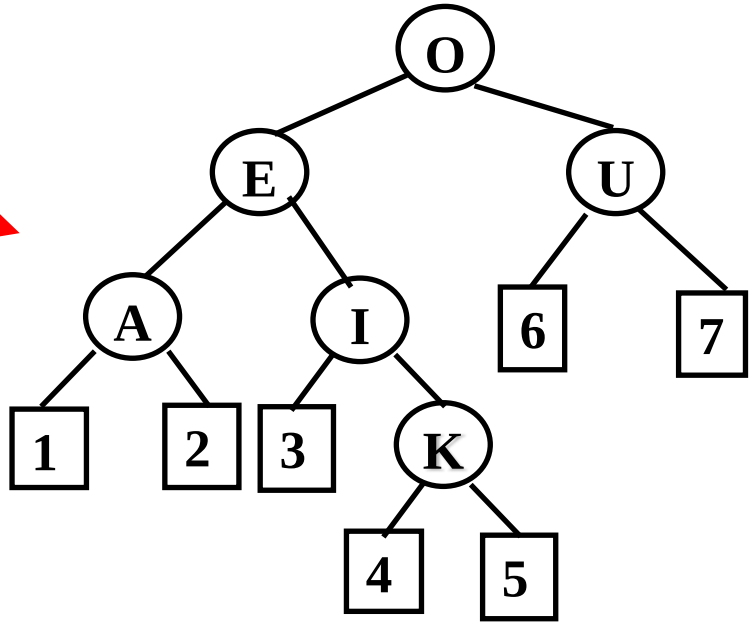
- 算法Successor(x)

```
/*在BST中找x的后继*/
S1[有右儿子，向下找]
    if( x.rlink != NULL ) return FindMin(x.rlink);
S2[否则向上找，找到最近的祖先并在其左子树中]
    y = x.parent;
    while(y!= NULL && x==y.rlink)
        x=y, y=x.parent;
    return y;
//时间复杂度： 向下或向上，O(h)
```

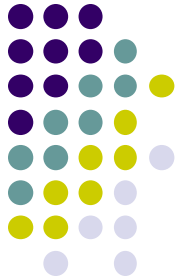
插入







插入



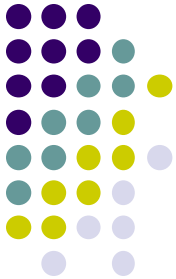
算法 Insert(t, k)

/*在t为根的子树中寻找关键词为k的结点,失败时插入,t是引用，递归版。*/

I1 [递归插入]

```
if(t==NULL){
        t ⇐ AVAIL ;
        t->key = k; t->llink = t->rlink = NULL;
}else if ( k < t->key ) t->llink = Insert(t->llink, k);
 else if ( k> t->key ) t->rlink  = Insert(t->rlink,k);
return t;
```

创建



- 调用 **n**次 **Insert**
- 从空二叉树开始，依次插入包含关键词 **CAPRICORN, AQUARIUS, PISCES, ARIES, TAURUS, GEMINI, CANCER，LEO，VIRGO，LIBRA**和**SCORPIO**之结点所形成的二叉查找树.

- 练习：输入序列为**{45，24，53，12，37，93}**

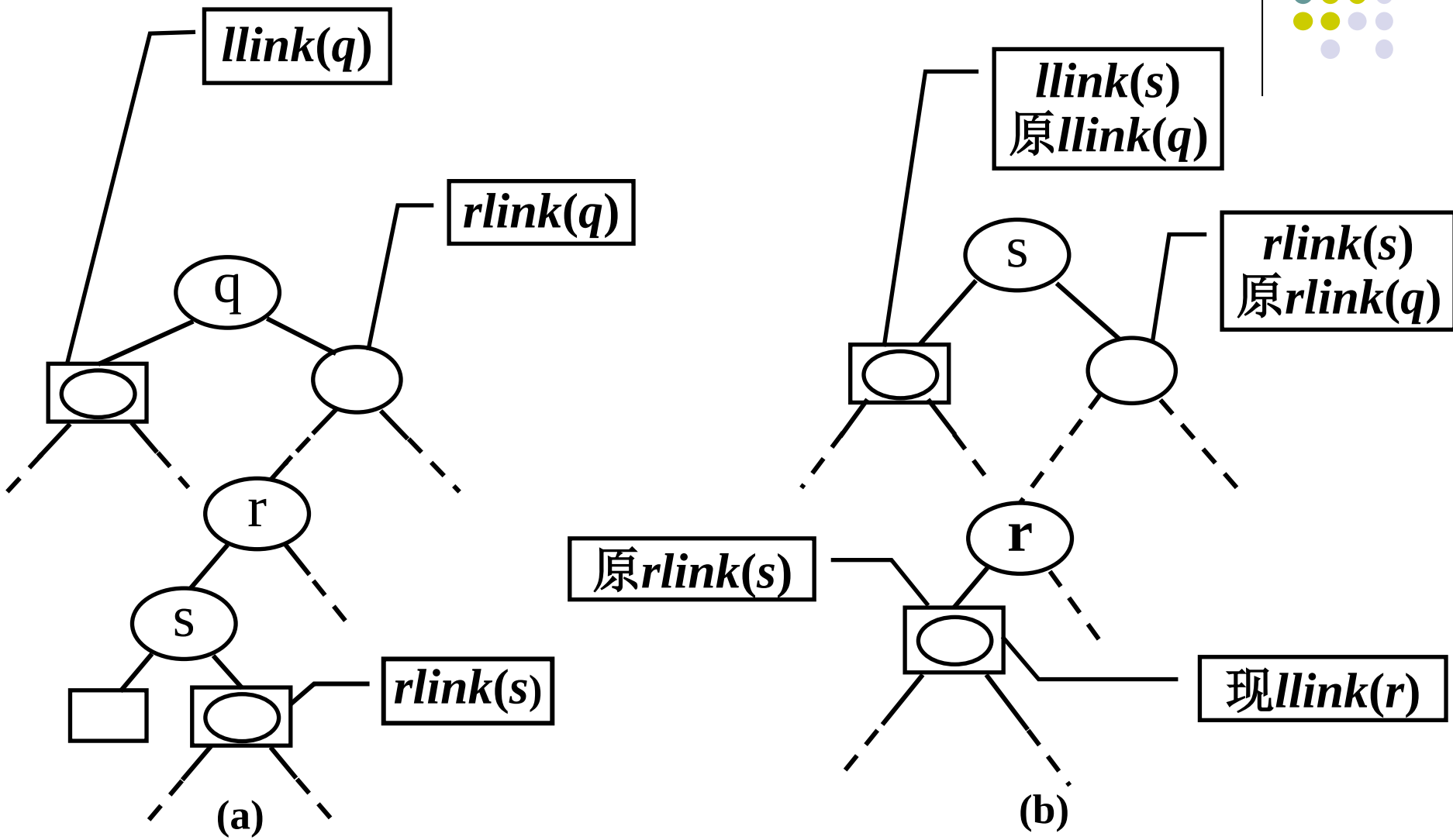
删除

- 假定指针q指向二叉查找树中待删除的结点;
- 删除q后的树仍为二叉查找树。

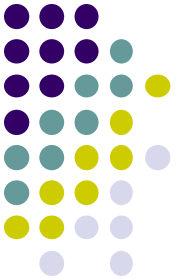
- 删除分三种情况
  - ✓ q的左右儿子都为空；（修改父结点，用空替换）
  - ✓ q只有一个儿子；（用该儿子替换q）
  - ✓ q有两个儿子：为保持中序递增，使用中根后继（前驱）替换；

rlink(*q*) ≠ Λ，且 *llink*(*rlink*(*q*)) ≠ Λ .

使用中根后继替换

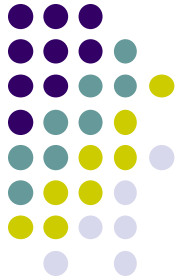


二叉查找树的简单分析



- 一个无序序列，可通过构造一棵二叉查找树而成为有序序列，这被称为二叉查找树的顺序属性。
- 二叉查找树会因输入文件的记录顺序不同，而有不同的形态。对包含*N*个记录的集合，其对应的关键词有*N*!种不同的排列，可构成 $C_{2N}^{N}/(N+1)$ 棵不同的二叉查找树。
- 特殊情况下，会产生退化二叉查找树，从而使最坏情况查找时间达*O*(*N*) .

BST的期望高度



- 当一棵二叉搜索树同时由插入和删除操作生成时，很难精确衡量这棵树的平均高度。
- 当树是由插入操作单独生成时，分析就容易得多

- **定理：一棵n个不同关键字的随机构建的二叉搜索树的期望高度为O(logn)**
  - ✓ n个关键字的一棵随机构建的二叉搜索树为按随机次序插入这些关键字到棵初始的空树中而生成的树，

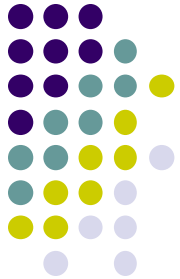
证明I（教材、TAOCP）

- 找到一个关键词所需的比较次数，恰恰比将此关键词插入树中所需的比较次数多1 .

$$S_N = 1 + (U_0 + U_1 + \cdots + U_{N-1})/\mathrm{N}$$

- 由引理8.1
- $S_N = (1+1/N)\, U_N - 1$
- 综合两公式，可得
- $(N+1)U_N = 2\mathrm{N} + (U_0 + U_1 + \cdots + U_{N-1})$
- $N\, U_{N-1} = 2\mathrm{N} - 2 + (U_0 + U_1 + \cdots + U_{N-2})$
- $U_N = U_{N-1} + 2/(\mathrm{N}+1))$
- $U_N = 2\mathrm{H}(N+1) - 2$

证明II （算法导论）



- **Xn:** 一棵具有n个结点的随机构建BST的高度
- **Yn = 2^{Xn} :** 指数高度
- Zi ：指示器变量；i为根结点，Zi = 1，否则0；所有Zi中，恰有一个为1，其余为0；Pr(Zi = 1) = 1/n (1≤ i ≤n)

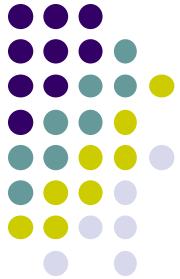
- 假定第 i个结点为根，则
  - ✓ Xn = 1 + max{Xi-1, Xn-i}
  - ✓ Yn = 2 max(Yi-1,Yn-i)
- **Yn = Σ Zi (2 max(Yi-1,Yn-i))**

- $E(Yn) = E(\sum_{i=1}^{n} Zi(2\ \max(Yi-1, Yn-i)))$
- $= \sum_{i=1}^{n} \frac{2}{n} \max(E(Yi-1), E(Yn-i))$
- $\leq \sum_{i=1}^{n} \frac{2}{n}\ (E(Yi-1) + E(Yn-i))$
- $= \frac{4}{n} \sum_{i=0}^{n-1}\ E(Yi)$

- 往证： $E(Yn) \leq \frac{1}{4}\binom{n+3}{3}$
- 基础：n=0和1时，Y0=0， Y1=1，均成立
- 归纳： $E(Yn) \leq \frac{4}{n} \sum_{i=0}^{n-1}\ E(Yi) \leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4}\binom{n+3}{3}$
- $= \frac{1}{n} \sum_{i=0}^{n-1} \binom{n+3}{3} = \frac{1}{n}\binom{n+3}{4} = \frac{1}{4}\binom{n+3}{3}$

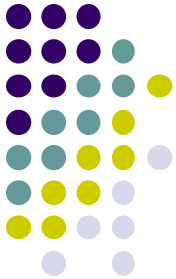
- Jensen不等式：
- 对任意点集{xi}，若λi≥0且Σλi＝1，f是凸函数，
- $\mathrm{f}(\sum \lambda i \cdot xi) \leq \sum \lambda i \cdot f(xi)$
- 对期望和凸函数应用Jensen不等式：
- $\mathrm{f}\big(\mathrm{E}(\mathrm{X})\big) = \mathrm{f}(\sum pi \cdot xi) \leq \sum pi \cdot f(xi) = E(\mathrm{f}(X))$
- 因此：
- $2^{E(Xn)} \leq E\left(2^{Xn}\right) = E(Yn) \leq \frac{1}{4}\binom{n+3}{3}$
- $\mathrm{E(Xn)} = \mathrm{O(logn)}$
- 定理得证。

二叉搜索树的退化的解决方案



- 方案一：随机文件

方案二：最优二叉查找树



- 给定各种查找情况发生的概率α_i和β_i，确定一棵最优(最小加权通路长度)的二叉查找树；**(8-4)**

- 一种自然的最优查找方案
  - ✓ 建立最优二叉查找树；对树动态调整，保证动态插入删除结点后，仍为最优。
  - ✓ 计算复杂度高，有时结点出现的频率无法精确统计。

方案三：AVL树

- 控制树形维护一棵形态较好的二叉查找树（近似最优）

- **1962年，两名俄国数学家G. M. Adelson-Velsky和E. M. Landis，发现了一个非常漂亮的维持一棵好的查找树的方案——AVL 树。仅要求为每个结点增加两个额外的二进制位。**

高度平衡树

- 定义8.2：一棵二叉查找树称为高度平衡二叉查找树（高度平衡树、平衡树），当且仅当由单一的外结点组成，或者由两个子树形T_l和T_r组成，且满足：

(1)｜ $h(T_l) - h(T_r)$ ｜≤1，其中h(T)表示树T的高度；

(2) T_l和T_r都是高度平衡树.

- 定义8.3 ：设 T 为高度平衡树，q 是T任一内结点，q 的平衡系数(平衡因子) BF(q) 定义为$h_R - h_L$.其中， h_L 和 h_R 分别q 的左、右子树的高度。
- 显然BF(q)) = 0, 1或 -1

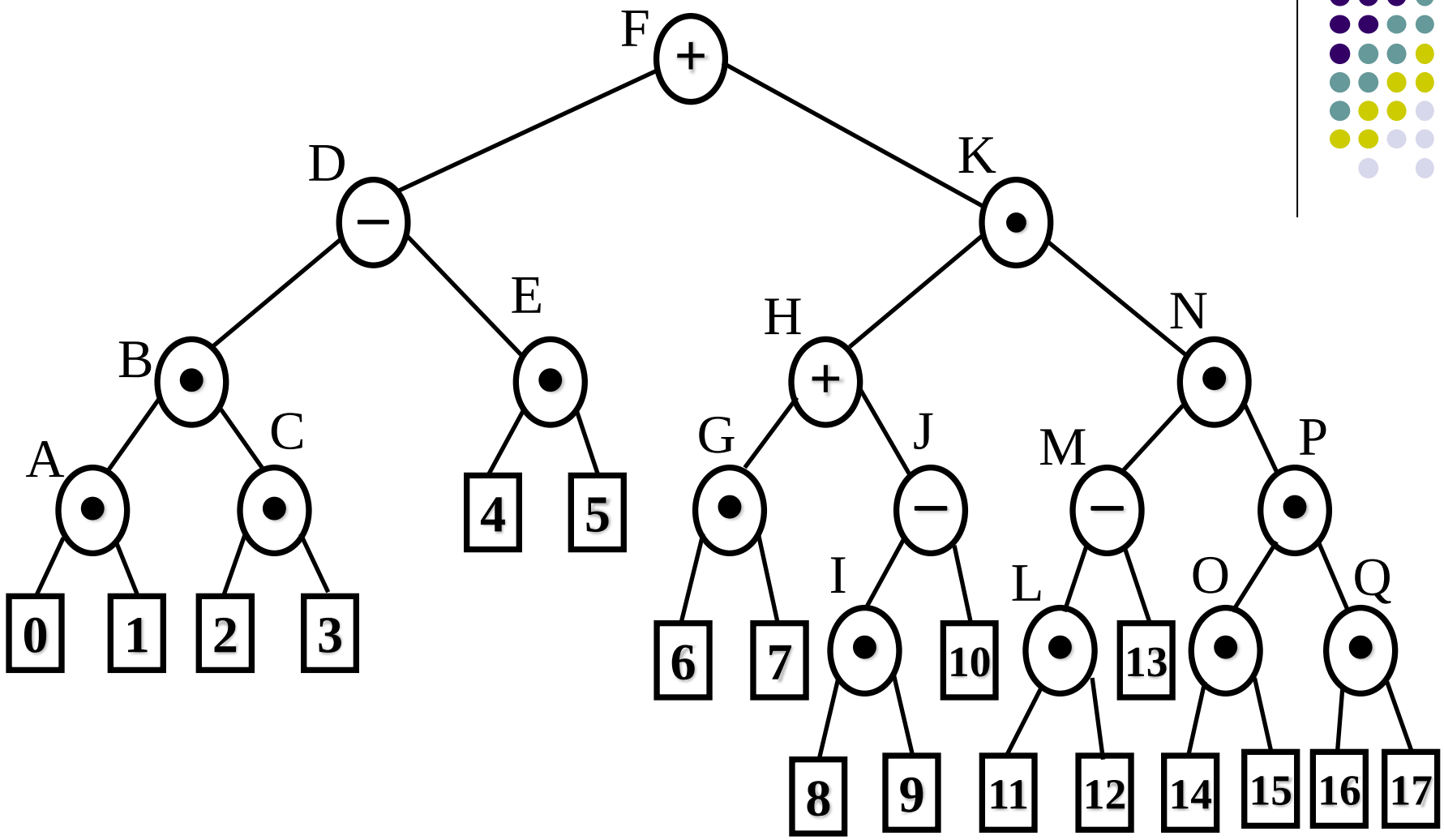
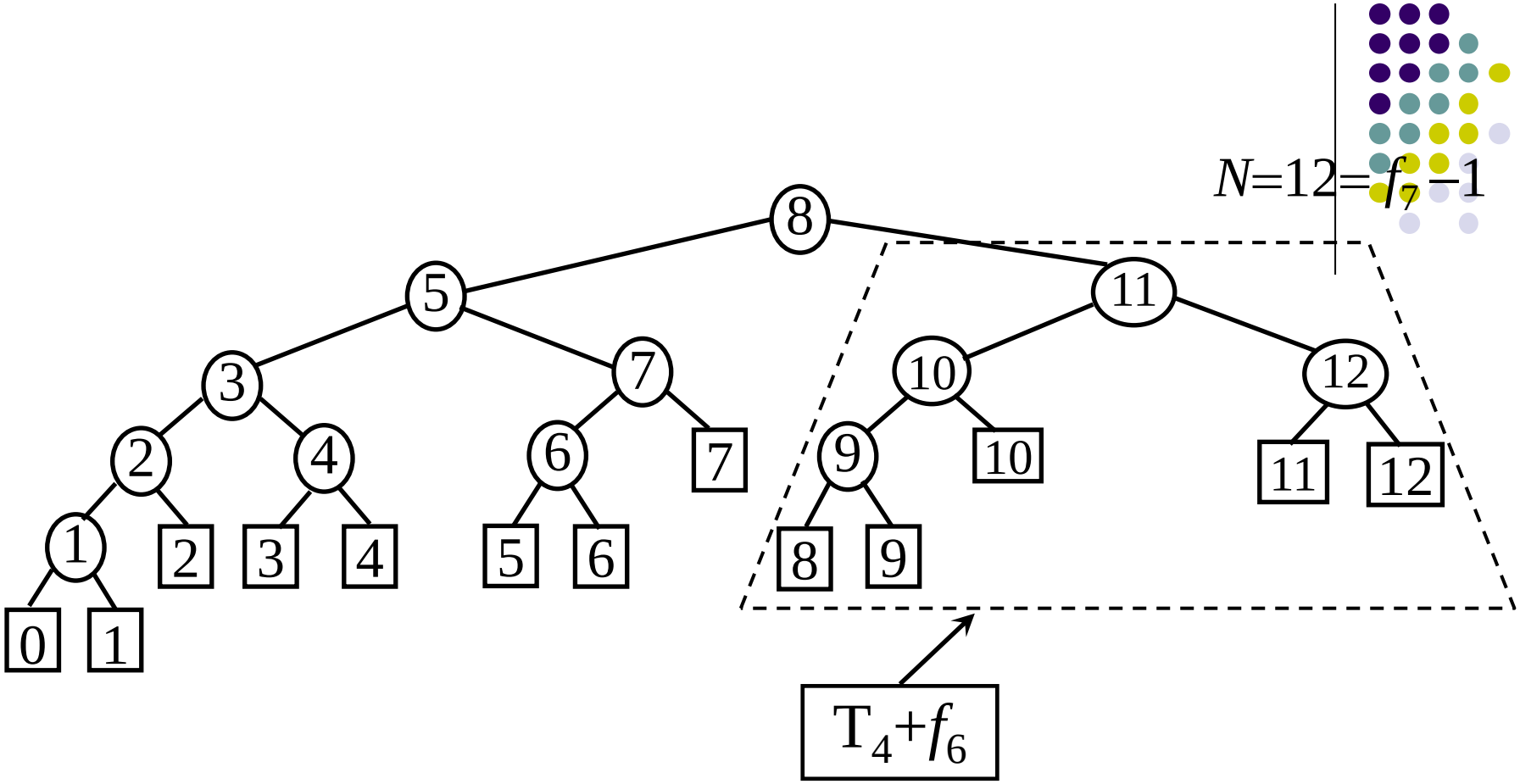


图8.23 一棵平衡二叉树形



- 斐波那契判定树是一棵高度平衡树，且每个结点(以该结点为根的二叉树的内结点数 ≥ 2)的平衡系数为 –1，该判定树在内结点数相同的高度平衡树中具有最大高度.

AVL的高度

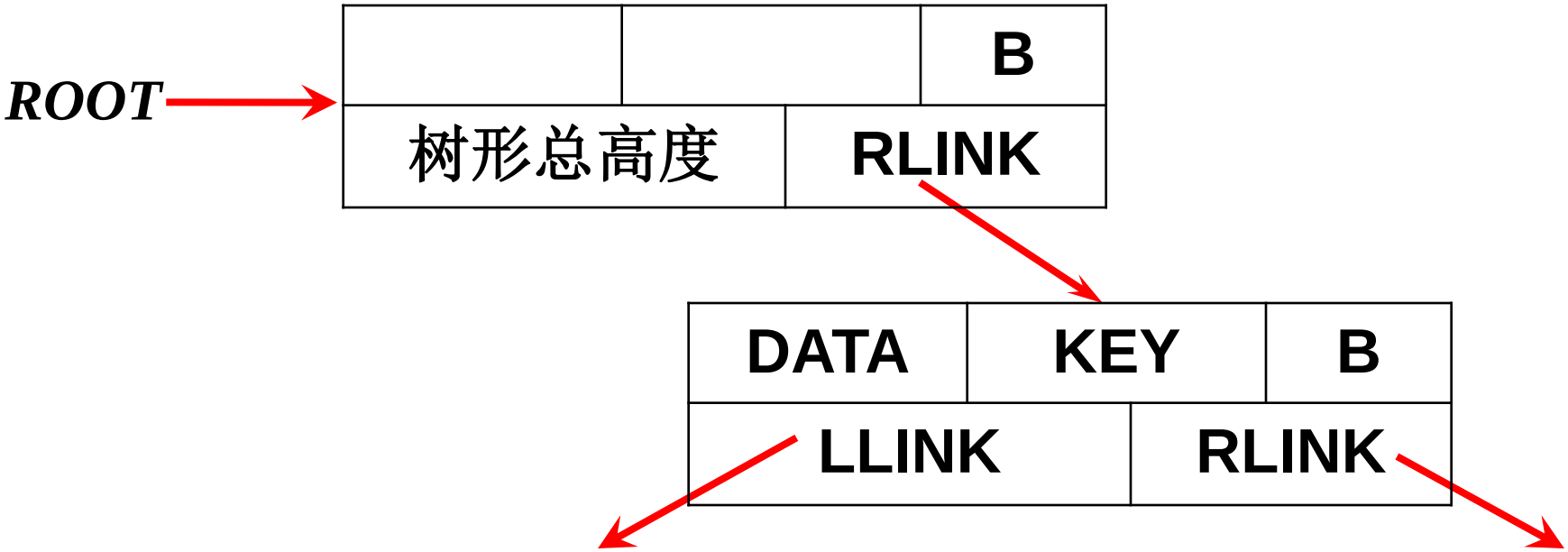
- 定理8.4 (Adelson-Velsky和Landis) 设 T是一棵具有n个内结点的平衡树，T之高度h(T)由下式所限定

$$\log_2(n+1) \le h(T) \le 1.4404 \times \log_2(n+2) - 0.3277$$

- 平衡树的查找路径长度决不会超过最优树的查找路径长度的45% .

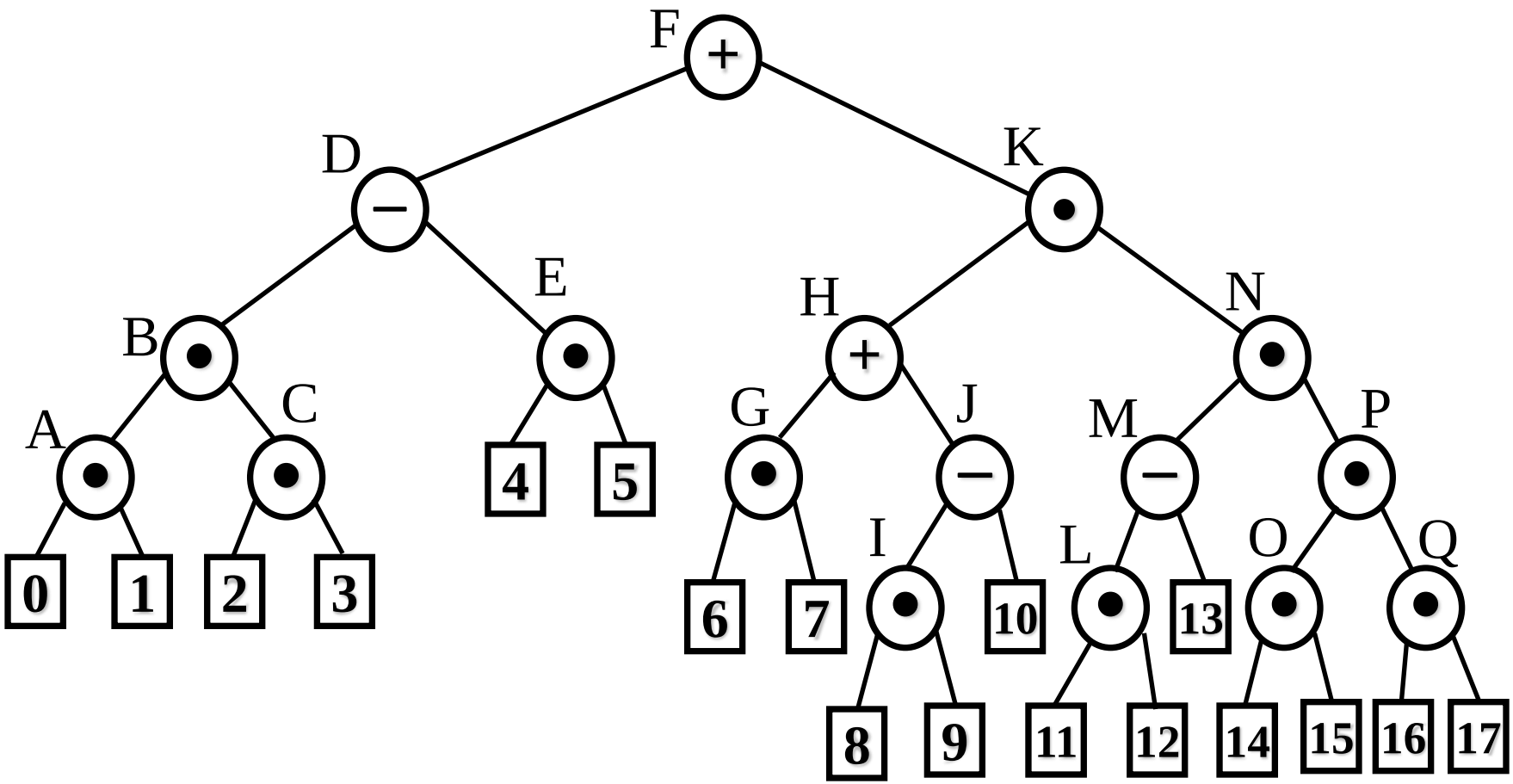
存储结构

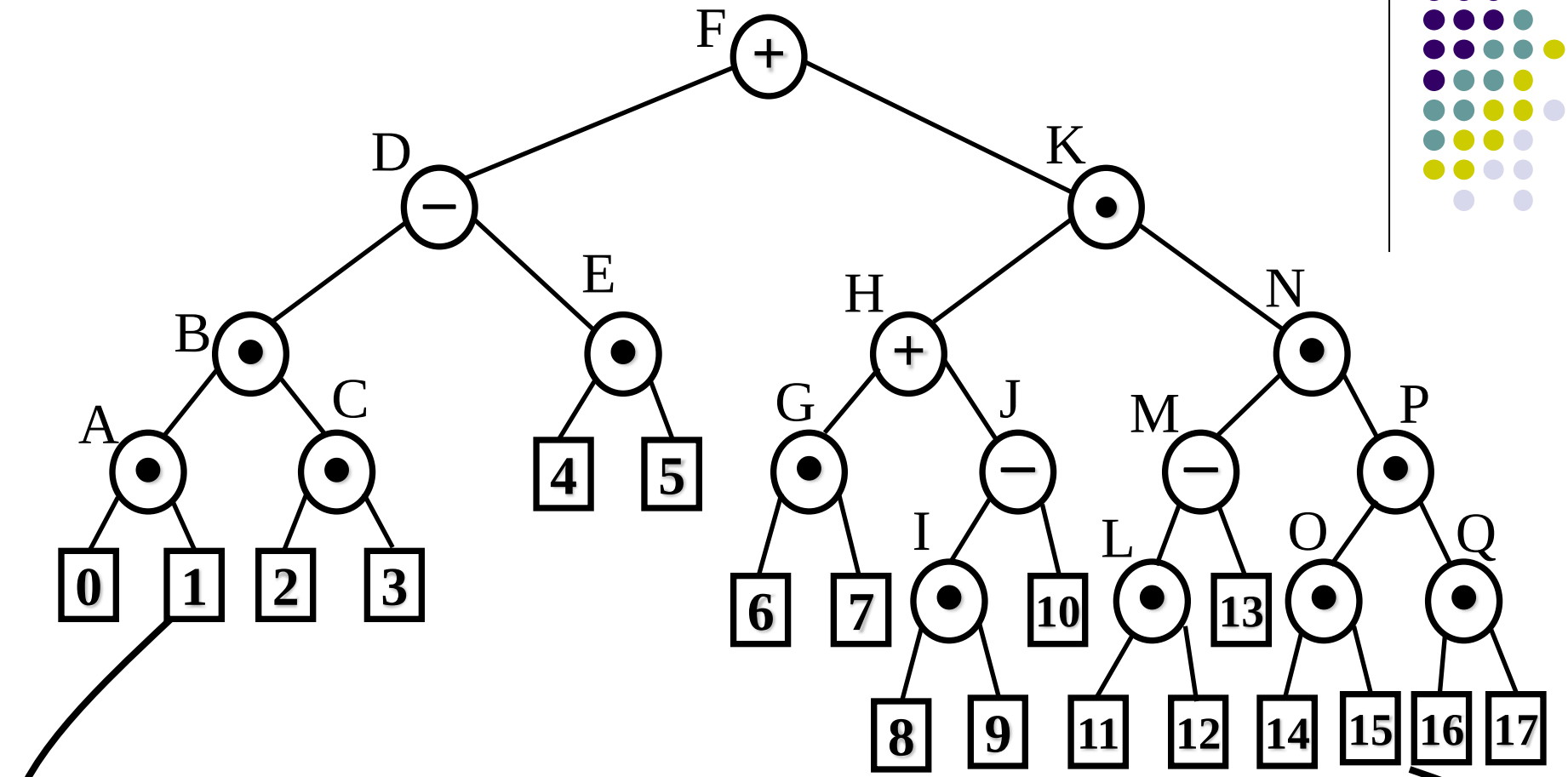

KEY
DATA
B
LLINK
RLINK


ROOT
B
树形总高度
RLINK
DATA
KEY
B
LLINK
RLINK

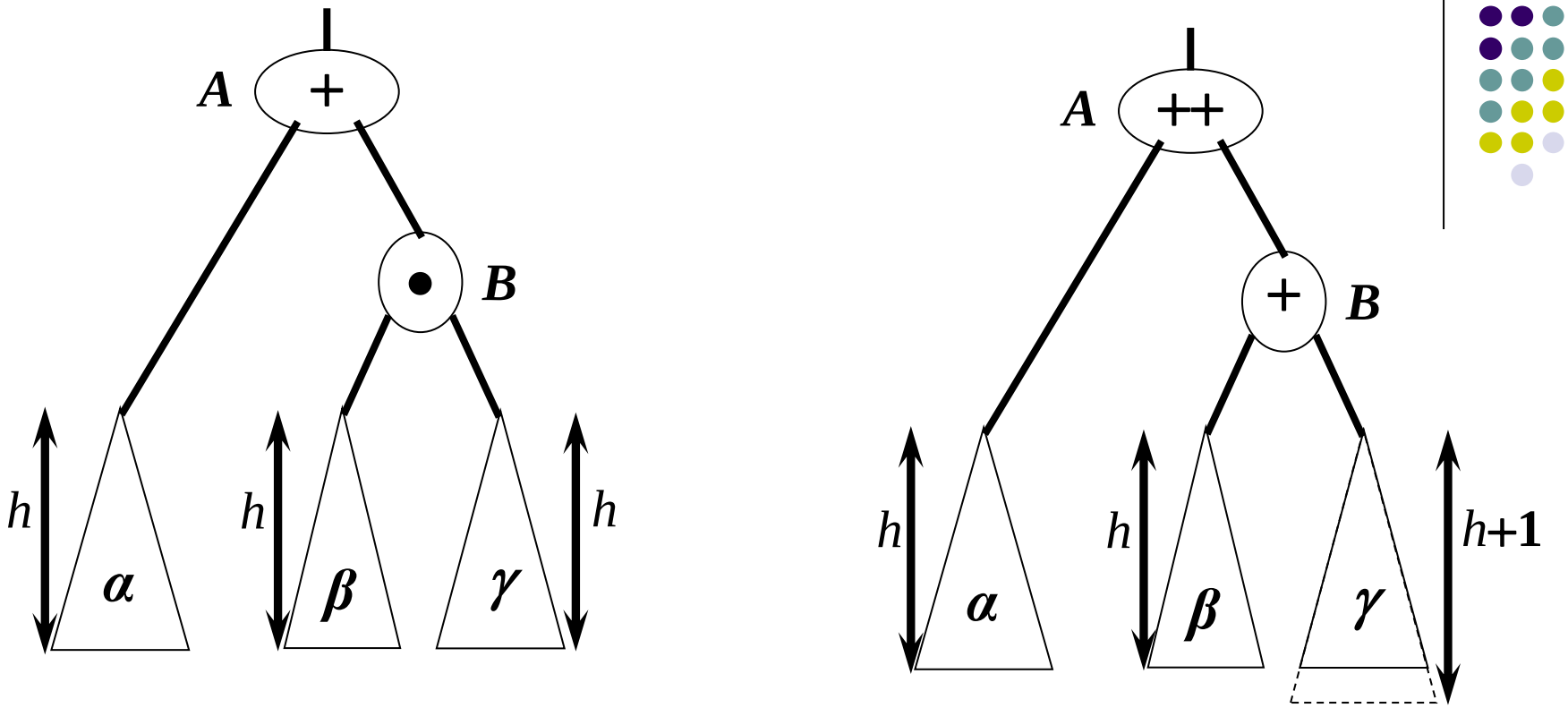
查找和插入操作

在一株高度平衡二叉树上插入结点，可能会破坏其平衡性。如，在图 8.23 中，如果新结点被插入高度平衡树的外结点 **4**，**5**，**6**，**7**，**10**，**13** 等处则无损平衡性。





- 平衡性将遭到破坏：
  - ✓ 平衡系数为**+1**的结点，如果在它的右子树的外结点上插入新结点，使它的右子树变得更高
  - ✓ 平衡系数为 **–1**的结点，如果在它的左子树的叶结点上插入新结点，使它的左子树变得更高



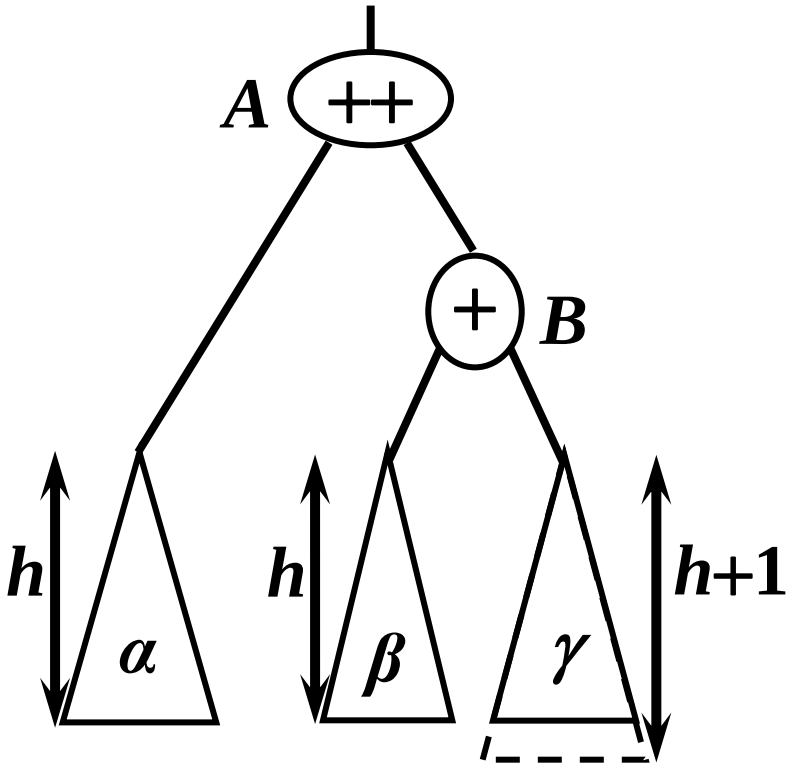
(a) 情况1

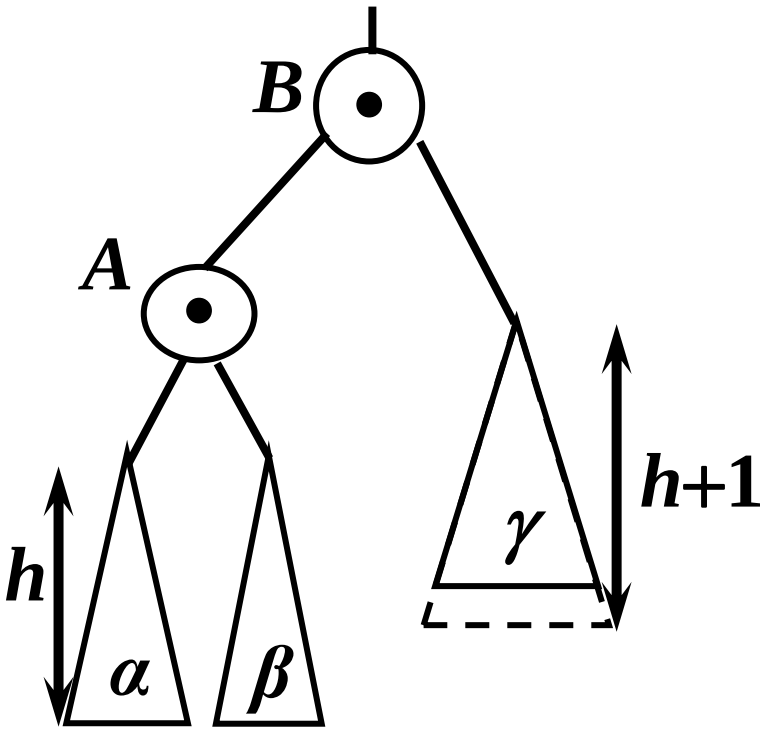
图8.24 破坏平衡的情况1（虚框表示新插入结点的位置）

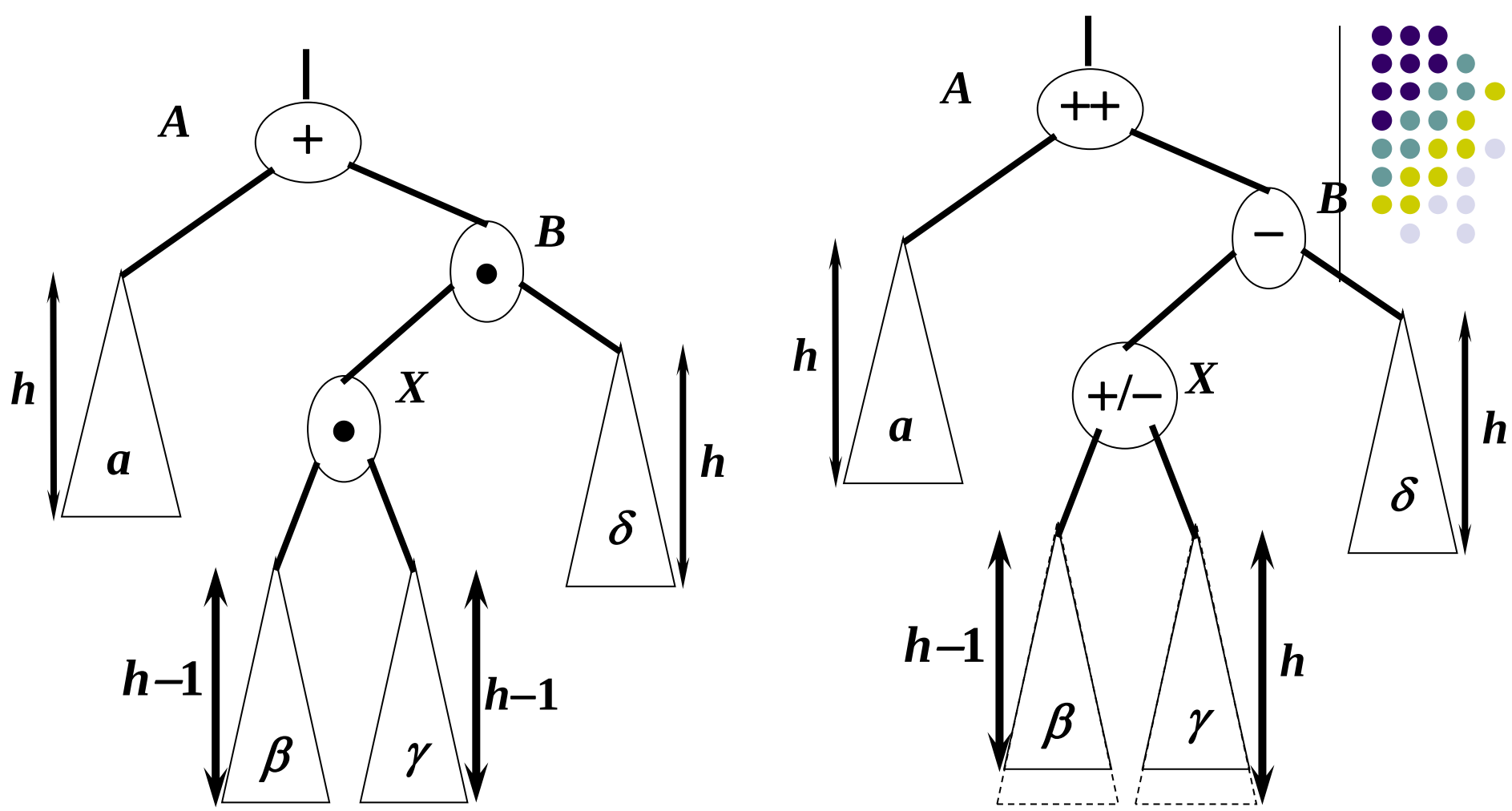
（若反演改图式，左右交换，则会出现另外1种相同的情况）

情况1：“单一转动”。

把结点 *B* 从 *A* 的右下侧**左转**到 *A* 的右上侧，原 *B* 之左子树β 变成了*A* 的右子树，*B* 的新左儿子是结点*A*.







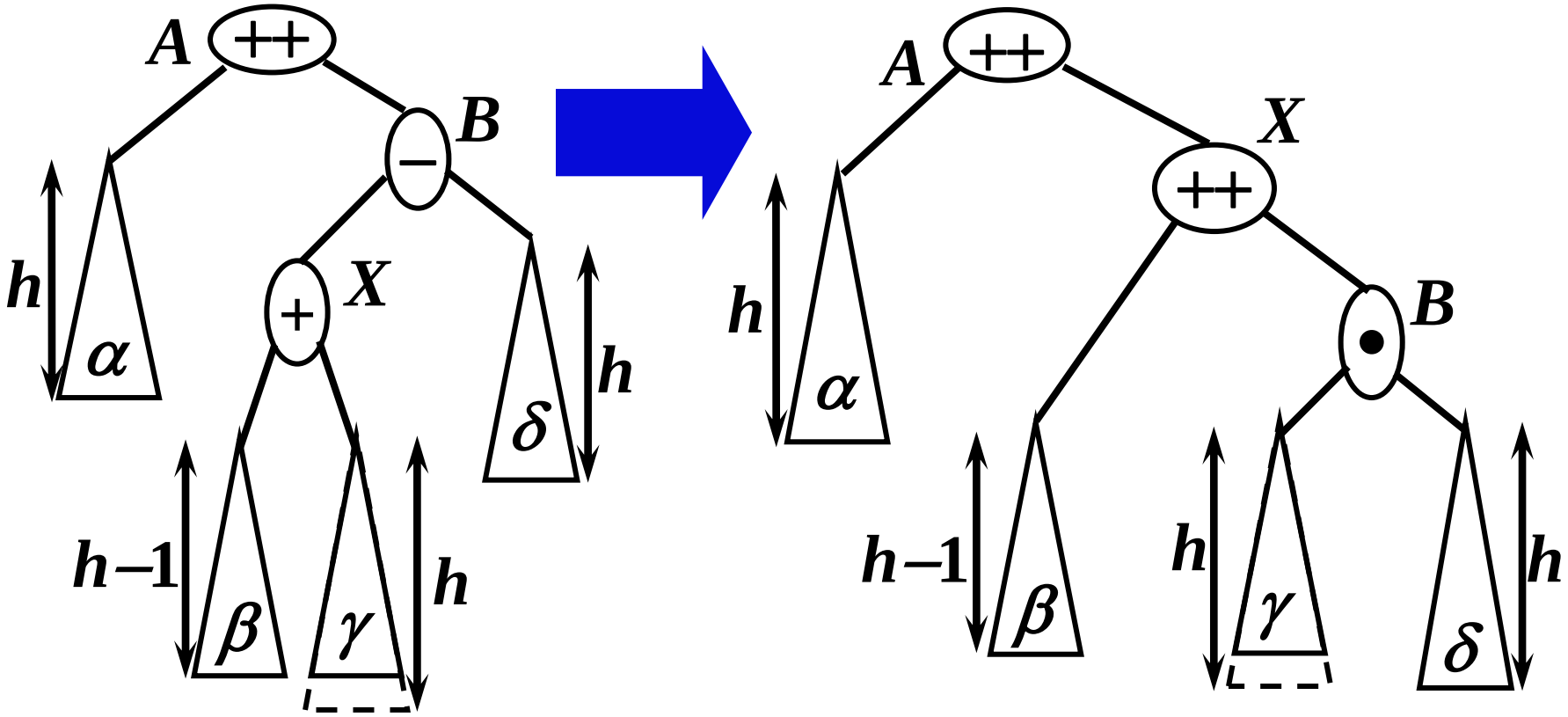
(a) 情况2

图8.24 破坏平衡的情况2（虚框表示新插入结点的位置）

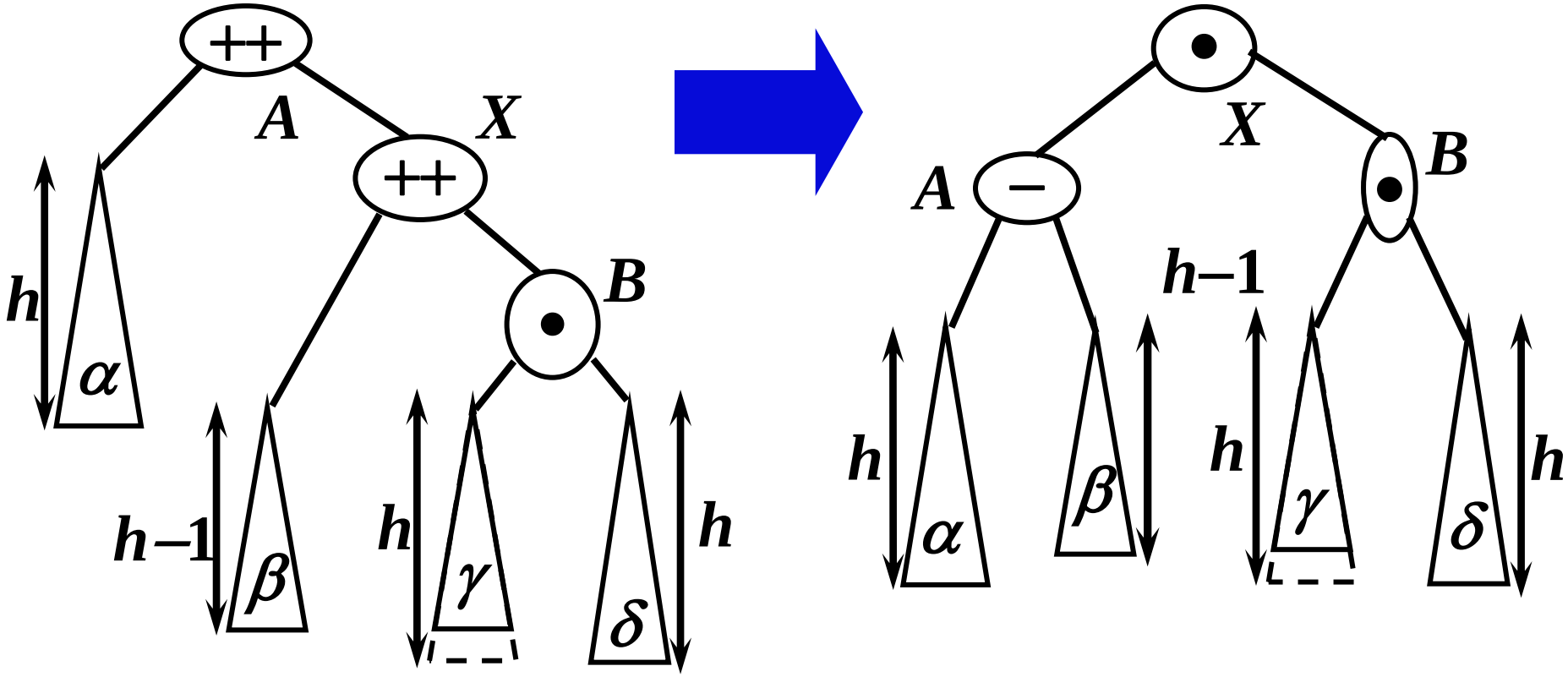
（若反演该图式，左右交换，则会出现另外1种相同的情况）

情况2：“双重转动”。

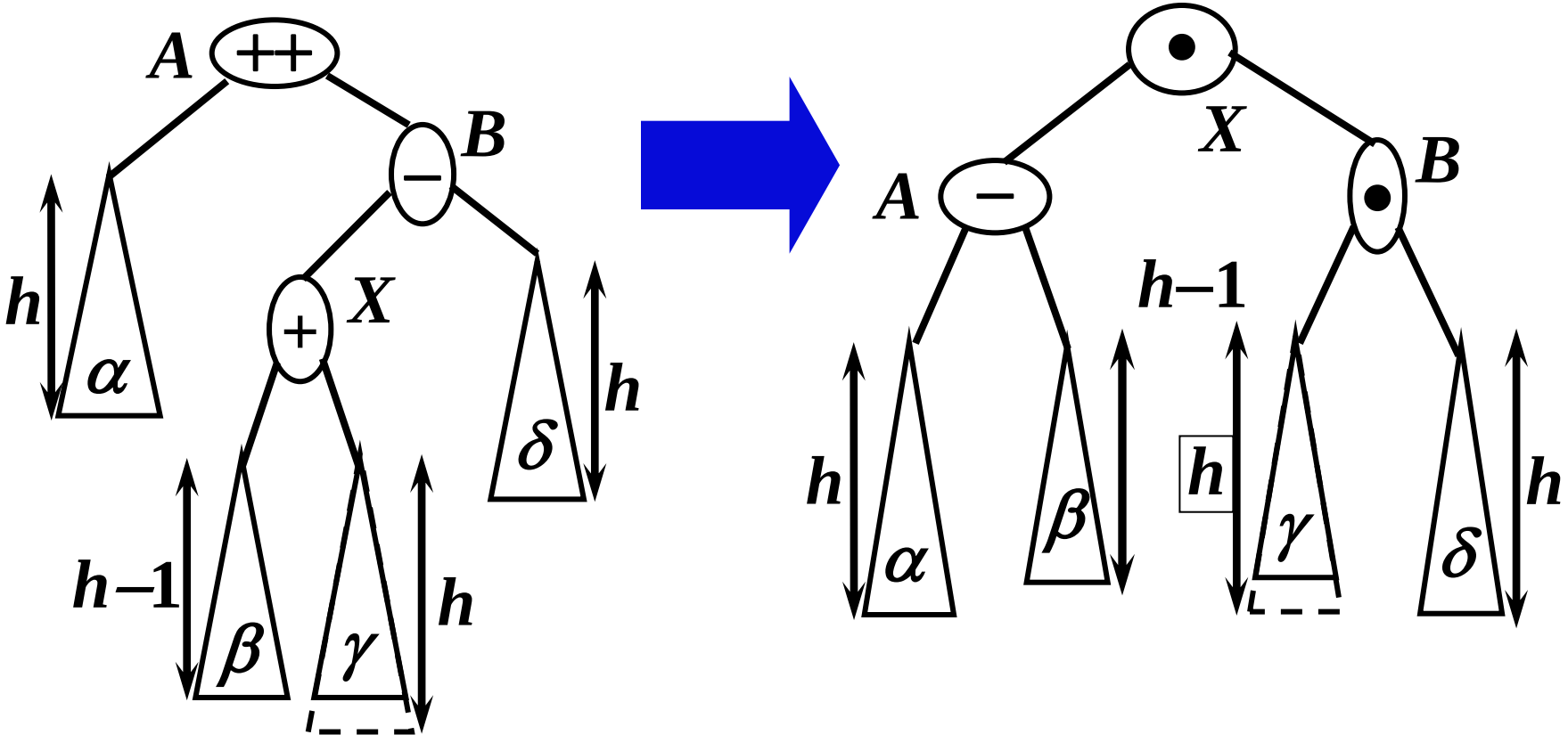
首先，以 X 为轴将 B 从 X 的右上侧**右转**到 X 的右下侧，记为（X，B），从而 A 的右儿子是 X，X 的右儿子是 B，原 X 的右子树 γ 变成了新 B 的左子树；

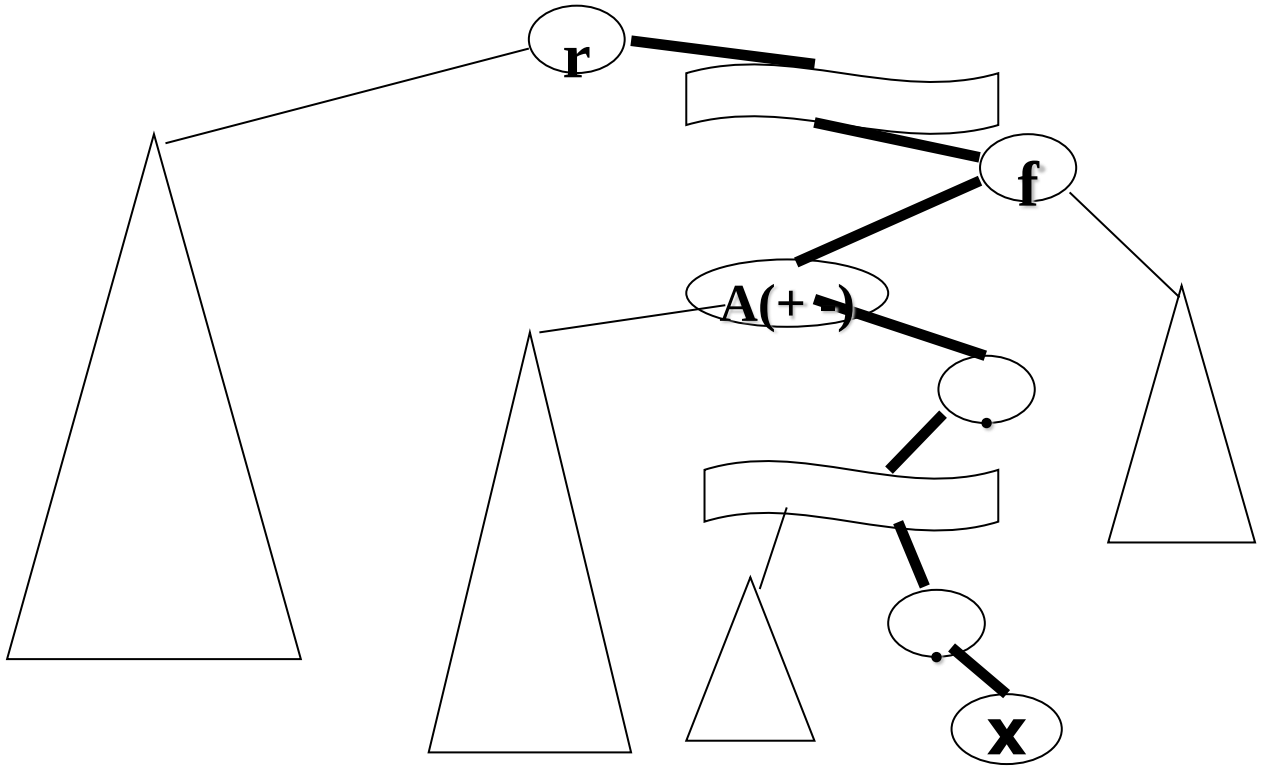


然后，以 X 为轴心，把 A 从 X 的左上方**左转**到 X 的左下侧，记为(A, X)，使 X 的左儿子是 A，右儿子是 B，原 X 的左子树 β 变成了 A 的右子树.



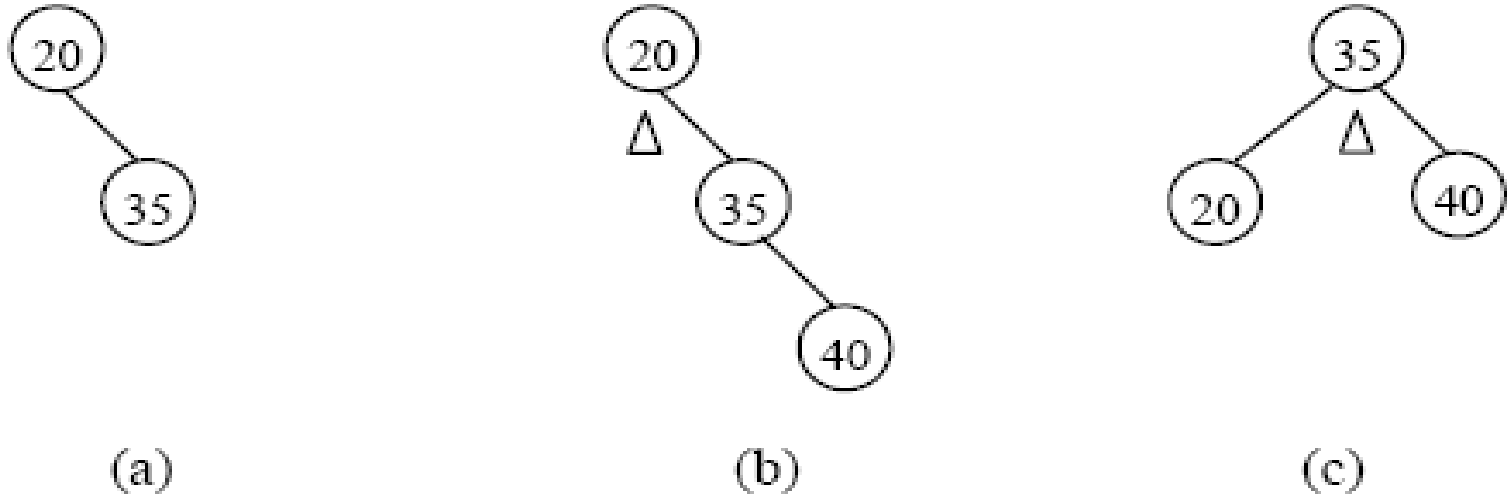
情况2：“双重转动”

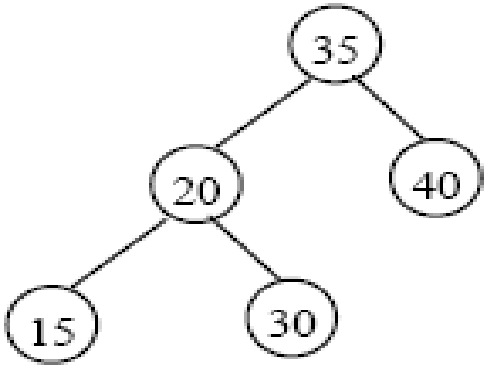


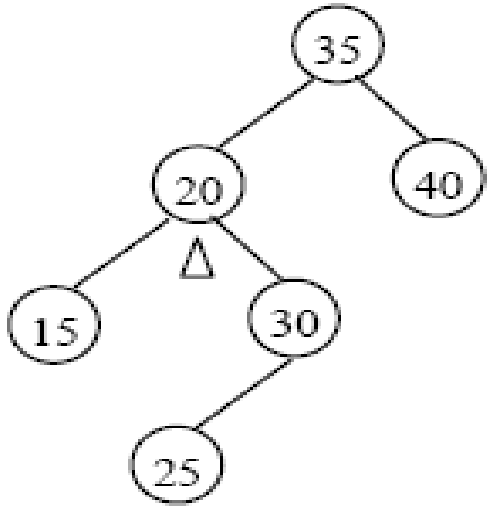


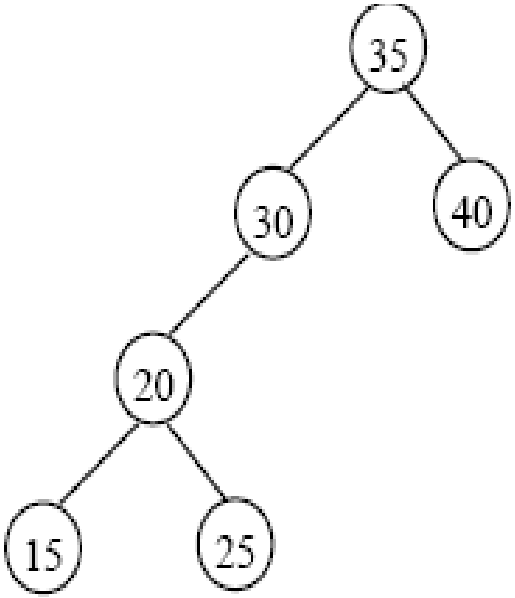
- 插入新结点n后，若树失去平衡，应调整失去平衡的最小子树，即从根结点r到结点n的路径上的最后一个不平衡点A，平衡以 A 为根的子树形可。
- 经过整后，以 A 为根的子树变成了一棵与插入前有同样高度的新树。原先A 子树外的结点不受影响
- 插入算法的效率为O(logN)

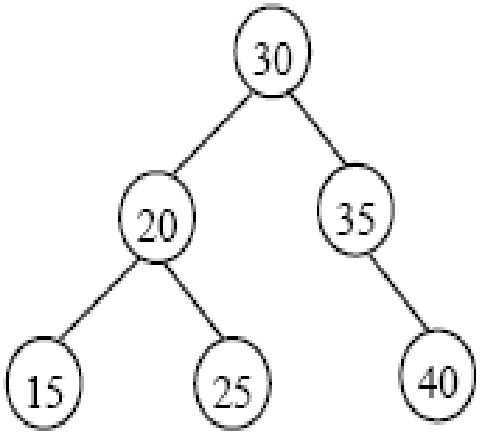
- 例：构造包含关键词{20，35，40，15，30，25} 高度平衡树。



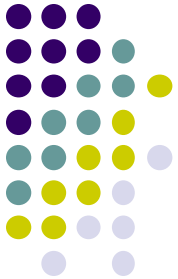

35
20
40
15
30


35
20
40
15
30
25


35
30
40
20
15
25

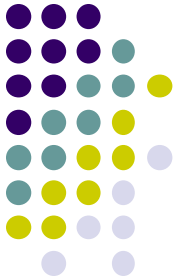

30
20
35
15
25
40

删除操作



- 高度平衡树的删除操作，被删除结点的祖先结点的平衡系数均有可能发生变化。
- 算法将从根结点到被删除结点路径上的所有结点依次入栈，设指针**son**指向被删除结点，**current**是**son**的父结点，则按照如下的规则进行删除：

1）如果B(current)=0，且son=LLINK(current)，则B(current)←+1，否则B(current)←-1. 并终止整个过程.

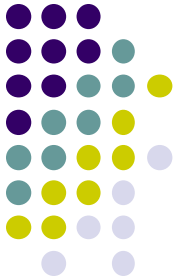


2) 如果 B(current)=+1 且 son=RLINK(current), 或 B(current)=-1 且 son=LLINK(current), 则 B(current)=0, 且current的高度减1. 此时令son←current, current⇐S, 然后继续.

3) 如果 B(current)=+1 且 son=LLINK(current)，则此时current的平衡系数等于+2，因此根据RLINK(current)的平衡系数来进行重新平衡操作，若调整后树高度不变，则算法终止；若调整后树高度减1，则令son←current, current⇐S, 然后继续.

4)如果B(current)=-1且son=RLINK(current)，则此时所发生的情况正好是(3)所示情况的对称情形.

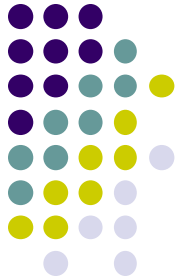
删除分析



- 删除问题可以在$O(\log_2 N)$步骤内解决；

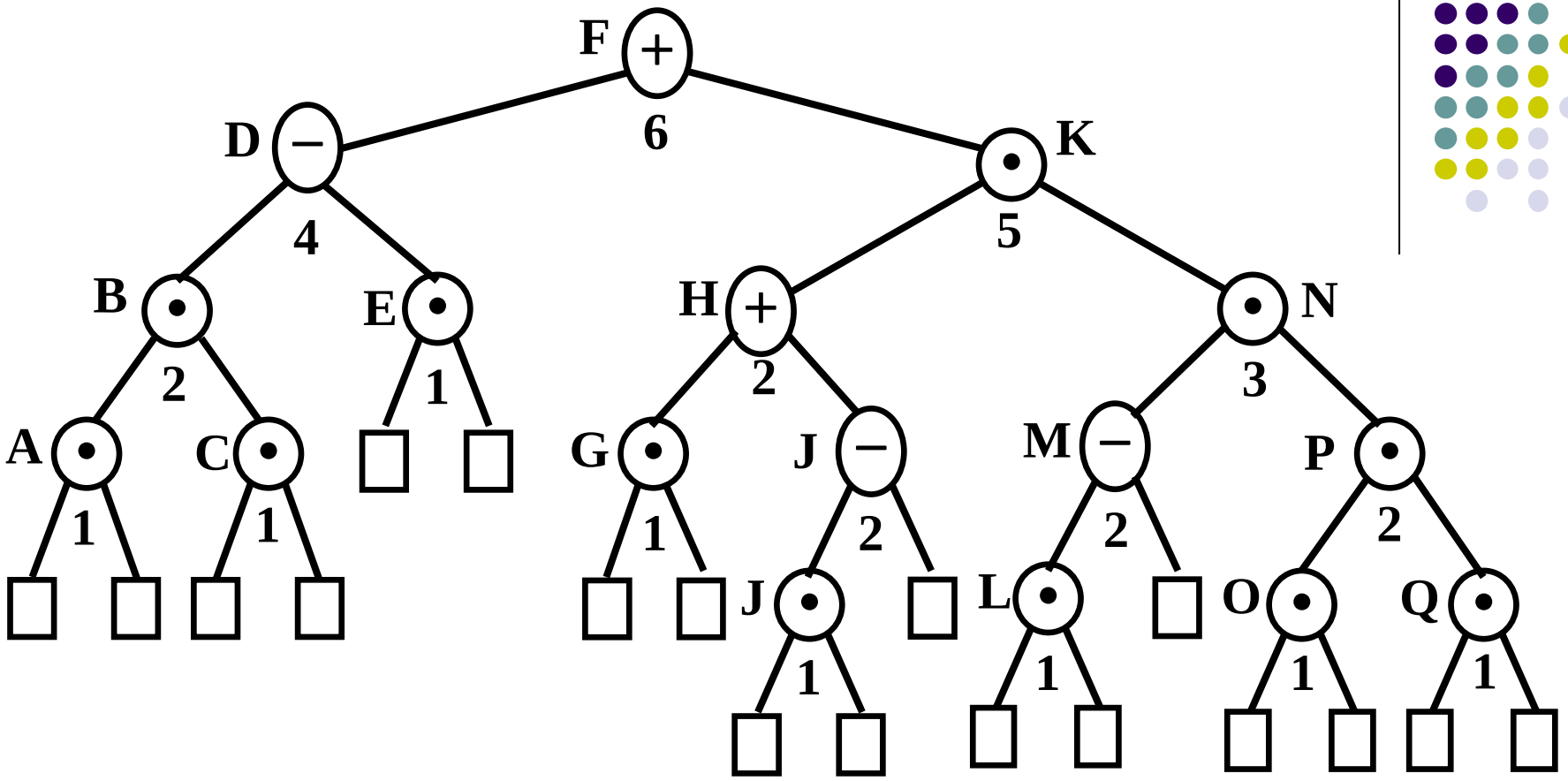
- 删除和插入之间的重要差别是：删除可能需要多达平均 $\log_2 N$ 次调整，而插入所需的调整不多于1次。

平衡树求第k小数



- 用平衡树来表达线性表，既可快速地插入（克服顺序存储的困难），又可对表实施随机存取（克服链接存储的困难）。

- 每一个结点P增加一个被称之为RANK的字段
  - ✓ **RANK (P) = P的左子树形的(内)结点数 + 1**
  - ✓ **RANK域之值确定了结点的相对位置**



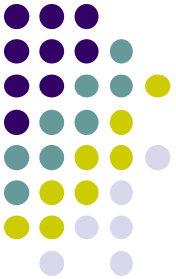
可方便地查找在中根次序下表的第*k**个结点. 假定待查结点的关键词为 *k**. 若*k**< 6, 则向左去查找; 若*k** > 6, 则往右去检索, 比如 *k**= 8, 则NODE (H) 便是所求，因*RANK*(H)=*k**–6=2 . 又如, NODE (E) 在中根次序下的位置是 5，它等于*RANK* (D) + 1；NODE (K) 的中序位置是11, 它等于*RANK*(F)+*RANK*(K)=6+5; 等等.

找第k小元素

算法kth (*t* , *k*)

```
p=t;
while( p!=0 )
    if( k == tr[p].rank ) return p;
    else if(k<tr[p].rank) p=tr[p].l;
    else k-=tr[p].rank ,p=tr[p].r;
}
return p;
```

平衡树的优缺点



- AVL引入平衡因子，很好的限制了树的平均高度为O(logN)，导致插入、删除、查找的最坏时间复杂性均为O(logn)
- AVL插入和删除实现不容易；

- 替代方案
  - ✓ Treap：按随机顺序建立BST
  - ✓ Splay：均摊时间复杂性为MlogN
- 红黑树:一种平衡树；动态集合操作最坏O(logn)