# 数据结构

2017秋季　刘鹏远

助教：韩越/卢梦依

# ch.6

查找的概念

---------静态查找

顺序查找

二分查找

静态查找树，索引结构

---------动态查找

二叉排序树/二叉平衡树

B树、 B+树---外查找

键树Trie树

散列/哈希

# 查找(Search)概念



- **查找，就是在数据集合中寻找满足某种条件的数据元素。**
- **通常称用于查找的数据集合为查找结构/查找表，它是由同一数据类型的元素（或记录）组成。**
- **查找的结果通常有两种可能：**
  - 查找成功，即找到满足条件的数据元素。这时，作为结果，可报告该元素在结构中的位置，还可给出该元素中的具体信息。
  - 查找不成功，或查找失败。作为结果，应报告一些信息，如失败标志、位置等。

# 查找(Search)的概念



- **静态查找：**数据集合稳定，不需要添加，删除元素。在这样结合中的查找操作。
- **动态查找：**数据集合在查找的过程中需要添加或删除元素。一般查找不到的元素，将被添加到查找集合中。
- **静态查找**常基于**线性表**，**动态查找**常基于**树**或**字典/散列**。

（原因在于要提高查找效率）

- 若整个查找过程都在内存进行，则称之为**内查找**；反之，若查找过程中需要访问外存，则称之为**外查找**。

# 查找(Search)的概念



- 在每个元素中有若干属性，其中有一个属性，其值可唯一地标识这个元素。称为关键字（key）。使用基于关键字的查找，查找结果应是唯一的。还可能有次关键字，对每个元素不唯一。（举例）

- 衡量查找算法的时间效率的标准(因为关键操作是比较)是：在查找过程中关键字的平均比较次数，也称为平均查找长度ASL(Average Search Length)，通常它是查找结构中元素总数 n 的函数。且对查找成功、不成功分别计算。

# 静态查找表结构的定义



```
#define MAXSIZE 100                 //查找表最大尺寸
typedef int ElemType;               //查找数据的类型
typedef struct {                    //查找表结点定义
    ElemType key;                   //关键字域
    //other;                        //其他数据信息
} SNode;
typedef struct {            //查找表结点定义
    SNode data[MAXSIZE];    //数据存储数组
    int n;                          //数组当前长度
}SSTable//ADT结构定义见P216，利用的是动态顺序表
```

# 顺序查找 (Sequential Search)



- 顺序查找, 又称线性查找, 主要用于在线性结构中进行查找。

- 设若表中有 $n$ 个元素，则顺序查找从表的先端（或后端）开始，依次用各元素的关键字与给定值$x$进行比较，直到找到与其值相等的元素，则查找成功；给出该元素在表中的位置。

- 若整个表都已检测完仍未找到关键字与$x$相等的元素, 则查找失败。给出失败信息。

# 顺序查找 (Sequential Search)



```
int LinearSearch (SStable* L, ElemType x) {
    int i ;
    for(i=0; i< L->n; i++)
        if(x==L->data[i]) return i+1;
    return 0;
}
```

**O(n)**

**对无序表，可否改进是性能提高？**

# 设置“监视哨”的顺序查找算法



```
int LinearSearch (SStable* L, ElemType x) {
//在数据表L->data[1]..L->data[n] 中顺序查找关键字
//值与给定值 x 相等的数据元素, L->data[0].key 作为
//控制搜索自动结束的“监视哨”使用
    L->data[0].key = x;   int i = L->n;
        //将 x 送表头位置设置监视哨
    while (L->data[i].key != x) i--;
        //从后向前顺序查找//整个算法也可以改成正向的
    return i;
}//教材P217，时间效率提高一倍
```

# 算法分析



- 设查找第 $i$ 个元素的概率为 $p_i$，查找到第 $i$ 个元素所需比较次数为 $c_i$，则查找成功的平均查找长度为:

$$\mathrm{ASL}_{\mathrm{succ}} = \sum_{i=1}^{n} p_i \cdot c_i, \quad \left(\sum_{i=1}^{n} p_i = 1\right)$$

- 在顺序查找并设置“监视哨”情形，$c_i = i,\ i = 1, \ldots, n$，因此

$$\mathrm{ASL}_{\mathrm{succ}} = \sum_{i=1}^{n} p_i \cdot i$$

# 算法分析



- 设查找概率相等，即 $p_i = 1/n$ 查找成功的平均查找长度为：

$$ASL_{succ} = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- 而查找不成功时，一定把表中所有元素检测了一遍，一直到监视哨。所以查找不成功的平均查找长度为：

$ASL_{unsucc} = n+1$　综合ASL=1/2U+1/2S，教材P218

- 顺序查找也可以用递归方法实现。当查找表中第一个元素即为所求，查找成功；否则对除第一个元素外的后续表元素构成的查找表使用相同方法递归查找。当后续查找表为空，则查找失败。

# 顺序查找的递归算法



```
int seq_search (SSTable* L, ElemType x, int loc) {
//在数据表L->data[0]..L->data[n-1]中查找其关键字值
//参数 loc 是在表中开始查找位置
    rloc = loc - 1;
    if (rloc  >= L->n) return 0;          //查找失败
    else if (L->data[rloc].key == x) return rloc+1;  //查找成功
        else return seq_search(L, x, rloc+1);        //递归查找
}//这个递归算法意义不大
```

# 基于有序顺序表的顺序查找



- **有序顺序表**限定表中的元素按照其关键字值从小到大或从大到小依次排列。在这类表中进行查找比表中元素任意存放，查找速度会有所提高。
- 在有序顺序表中做**顺序查找**时，若查找不成功，不必检测到表尾才停，只要发现有比它的关键字值大/小的即可停止查找。

# 基于有序顺序表的顺序查找算法



```
int seq_search (SSTable* L, ElemType x) {
//在有序数据表L->data[0]..L->data[n-1] 中顺序查找关键字
//值为 x 的数据元素
    for (int i = 0; i < L->n; i++)
        if (L->data[i].key == x) return i;       //成功成功
        else if (L->data[i].key > x) break;      //查找失败
    return -1;              //顺序查找失败, 返回失败信息
}//顺序表索引从0开始
```

# 基于有序顺序表的顺序查找



- 若设表中有 $n$ 个元素，则查找成功的平均查找长度依旧
- 而查找不成功的平均查找长度请大家自行思考
- 有序顺序表进行顺序查找，**意义不大**

# 性能分析---描述查找过程的判定树



**ASL可统一用描述查找过程的判定树来计算，更直观。**

**判定树来描述查找过程：在判定树中，○表示内结点，它包含关键字集合中的某一个关键字；□表示外结点，代表各关键字间隔中的不在关键字集合中的关键字。它们是查找失败时到达的结点，物理上实际不存在。**

# 性能分析---描述查找过程的判定树



- **查找成功的平均查找长度$ASL_{succ}$定义为该树所有内结点上的权值 p[i]与查找该结点时所需的关键字比较次数c[i] (= l[i] ，树深)乘积之和：**

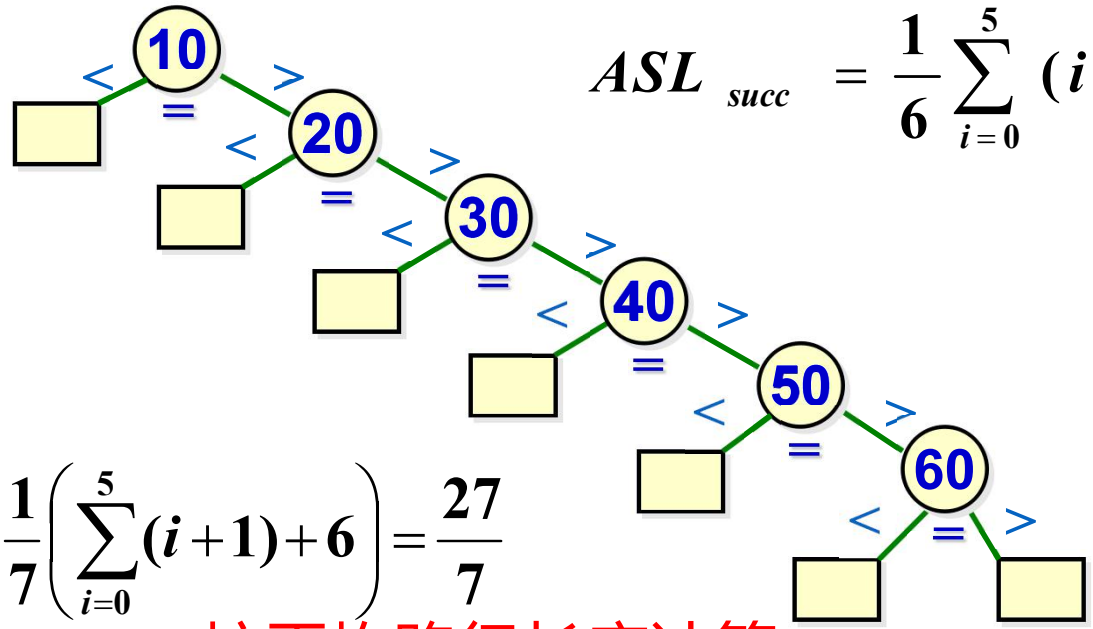
$$ASL_{succ} = \sum_{i=1}^{n} p[i] * l[i].$$

- **查找不成功的平均查找长度$ASL_{unsucc}$为树中所有外结点上权值q[j]与到达该外结点所需关键字比较次数c'[j] (= l'[j]−1)乘积之和：**

$$ASL_{unsucc} = \sum_{j=0}^{n} q[j] * (l'[j] - 1).$$

## ■有序顺序表的顺序查找示例及分析其查找效率的判定树：

( 10, 20, 30, 40, 50, 60 )




$$ASL_{succ} = \frac{1}{6}\sum_{i=0}^{5}(i+1) = \frac{7}{2}$$

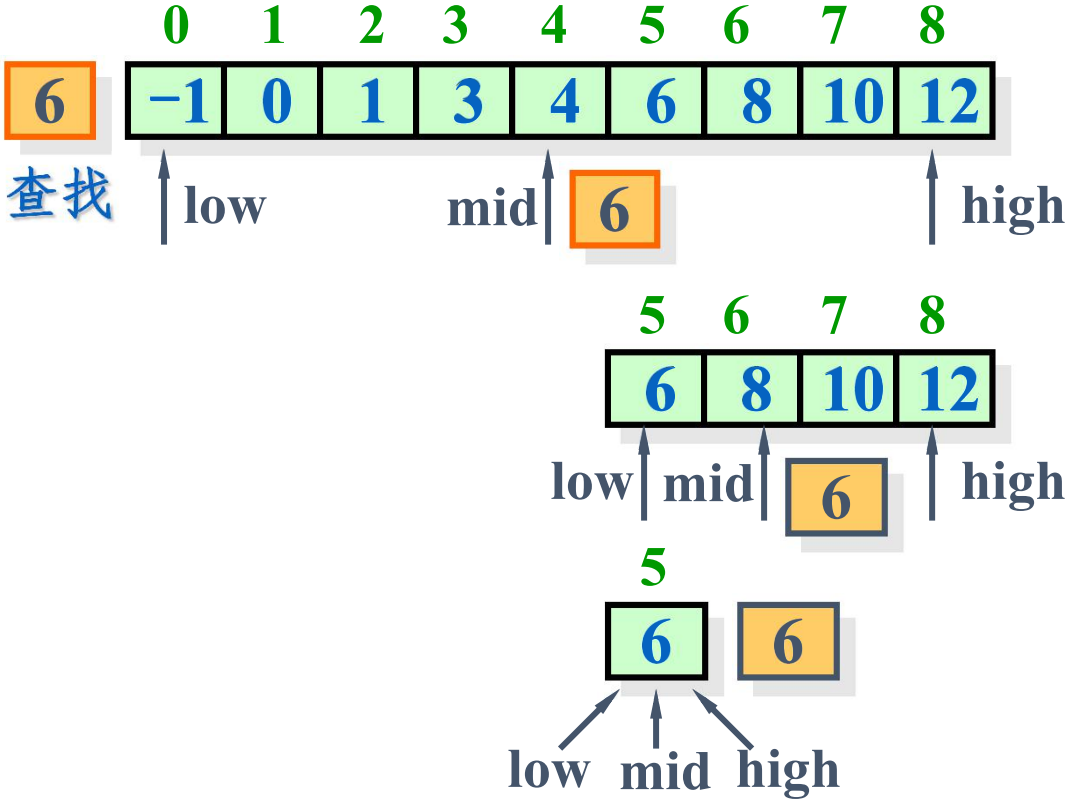
$$ASL_{unsucc} = \frac{1}{7}\left(\sum_{i=0}^{5}(i+1)+6\right) = \frac{27}{7}$$

按平均路径长度计算

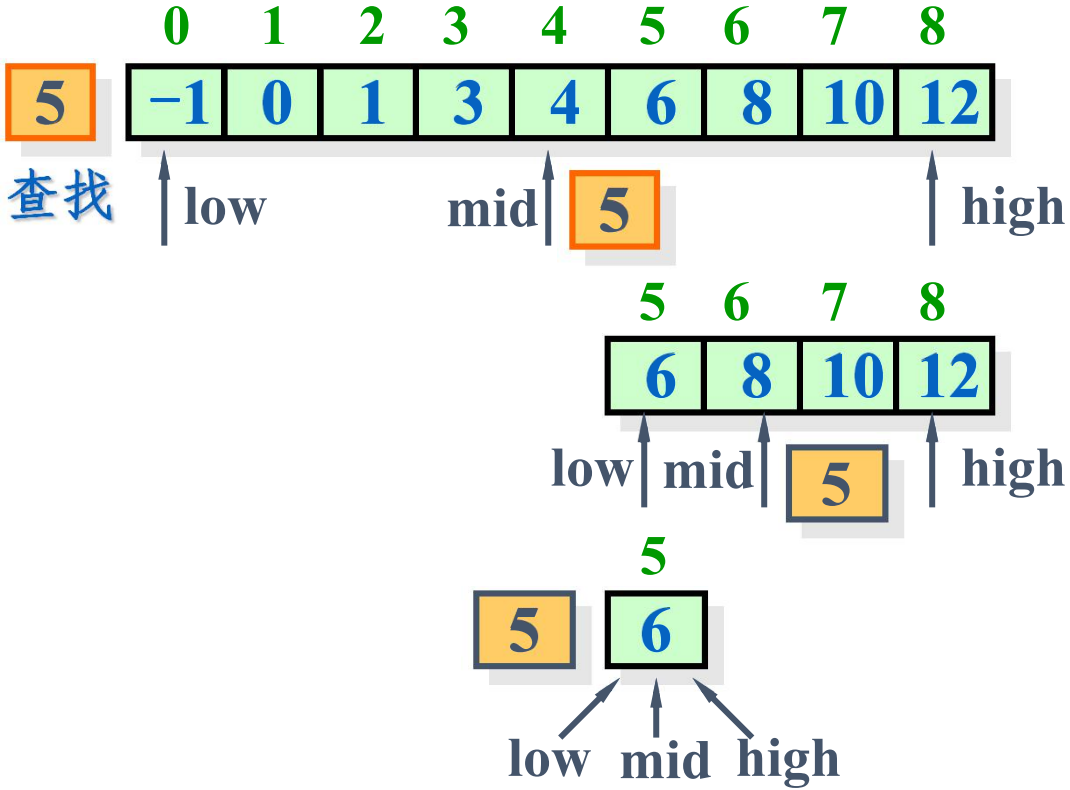
# 基于有序顺序表的查找算法之折半查找



- **折半查找的有序表，应该采取什么存储结构？**
- **答：须采取顺序表存储**
- **算法：**
  - A.data[mid].key == x, 查找成功；
  - A.data[mid].key > x, 把查找区间缩小到　表的前半部分，继续折半查找；
  - A.data[mid].key < x，把查找区间缩小到表的后半部分，继续折半查找。
- **如果查找区间已缩小到一个元素，仍未找到想要查找的元素，则查找失败。**


0 1 2 3 4 5 6 7 8
6
-1 0 1 3 4 6 8 10 12
查找
low
mid
6
high
5 6 7 8
6 8 10 12
low
mid
6
high
5
6
6
low mid high

查找成功的例子


0 1 2 3 4 5 6 7 8
5
−1 0 1 3 4 6 8 10 12
查找
low
mid
5
high
5 6 7 8
6 8 10 12
low
mid
5
high
5
5
6
low mid high

查找失败的例子

# 折半查找的算法



```
int BinSearch(SSTable* L, ElemType x) {
    int high = L->n-1,  low = 0,  mid;
    while (low <= high) {                  //low>high表明失败
        mid = (low + high) / 2;
        if (L->data[mid].key < x)
            low = mid + 1;                 //右缩查找区间
        else if (L->data[mid].key > x)
                high = mid-1;              //左缩查找区间
        else return mid;                   //查找成功
    }
    return -1; }                           //查找失败
```

# 递归的折半查找算法

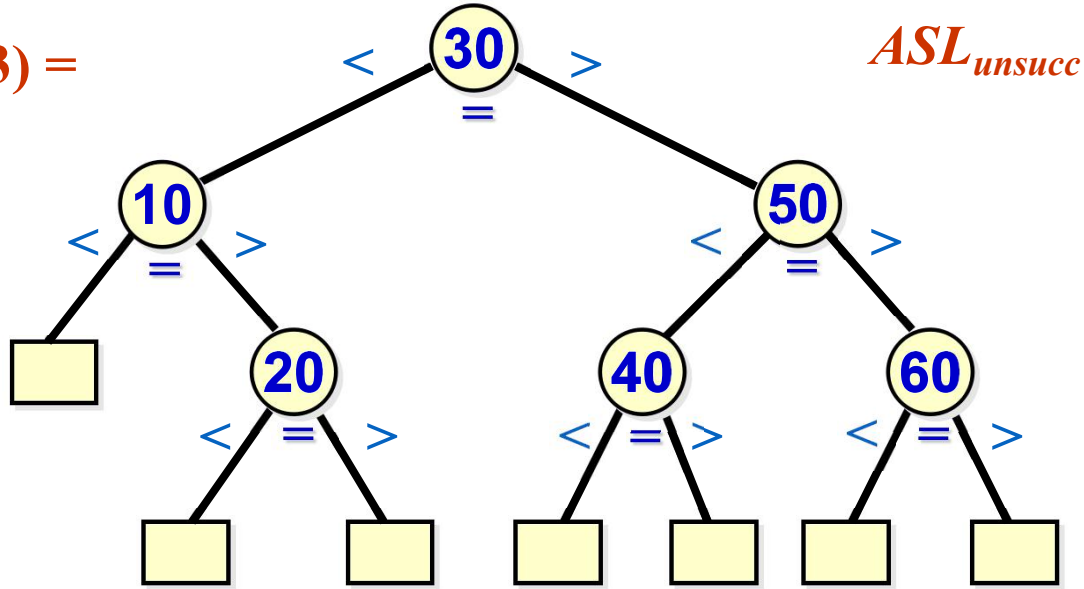


```
int BinSearchRecu(SSTable* L, elem_type x, int low, int high) {
    int mid = -1;
    if (low <= high) {
        mid = (low + high) / 2;
        if (L->data[mid].key < x)
            mid = BinSearchRecu(L, x, mid +1, high);
        else if (L->data[mid].key > x )
            mid = BinSearchRecu(L, x, low, mid-1);
    }
    return mid;
}     //调用方式 BinSearchRecu(L, x, 0, L->n-1);
```

## ■有序顺序表的折半查找的判定树（设成功权值均等，不成功权值也均等）

（10, 20, 30, 40, 50, 60）

$ASL_{succ}$ = 1/6(1+2*2+ 3*3) = 14/6




$ASL_{unsucc}$ = 1/7(2*1+3*6) = 20/7

# 折半查找性能分析



- 若设 $n = 2^h-1$，则描述折半查找的判定树是高度为 $h$ 的满二叉树。

$$2^h = n+1,\ h = \log_2(n+1)$$

- 第 1 层结点有1个，查找第 1 层结点要比较 1 次; 第 2 层结点有2个，查找第 2 层结点要比较 2 次；…, 第 $i$ $(1 \le i \le h)$ 层结点有 $2^{i-1}$ 个, 查找第 $i$ 层结点要比较 $i$ 次，….
- 假定每个结点的查找概率相等，即 $p_i = 1/n$，查找不成功的平均查找长度（自行推导）；查找成功的平均查找长度为：

$$ASL_{succ} = \sum_{i=1}^{n} p_i \cdot c_i = \frac{1}{n}\sum_{i=1}^{n} c_i = \frac{1}{n}(1 * 2^0 + 2 * 2^1 + 3 * 2^2 + \cdots + (h-1) * 2^{h-2} + h * 2^{h-1})$$

■可以用归纳法证明：

$$1 \times 1 + 2 \times 2^1 + 3 \times 2^2 + \cdots + (h-1) \times 2^{h-2} + h \times 2^{h-1} =$$

$$= (h-1) \times 2^h + 1$$

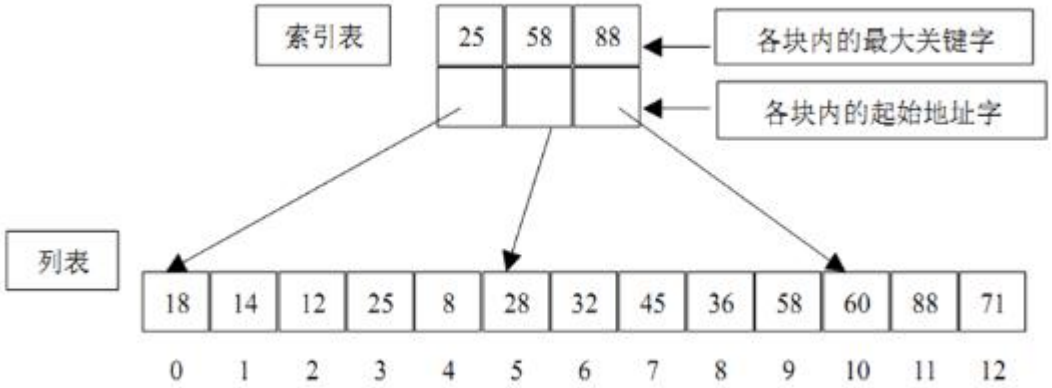
■这样 $ASL_{succ} = \frac{1}{n}((h-1) \times 2^h + 1) = \frac{1}{n}((n+1)\log_2(n+1) - n)$

$$= \frac{n+1}{n}\log_2(n+1) - 1 \approx \log_2(n+1) - 1$$

# 有序表其他分割（介绍）



- 斐波那契查找：按照该数列特点对表进行分割
- 插值查找：一般需要表中数据均匀分布（P222最上面）
- 分块查找/索引顺序查找，分块有序，块内可无序（P225）



# 静态查找树（介绍）

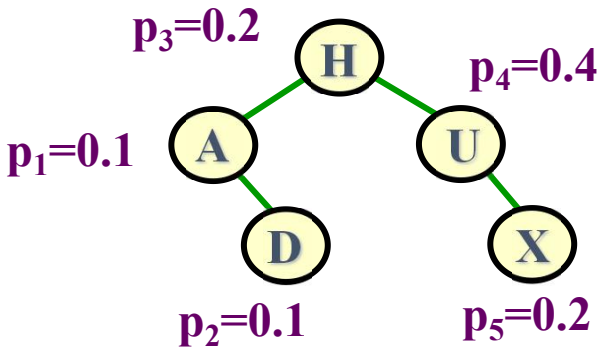


- 当有序顺序表各个结点的查找概率**不等**时，折半查找并非最优，可用静态查找树描述或构造其查找过程。

- 设含有 5 个关键字的有序表 ( A, D, H, U, X )，其查找概率分别为

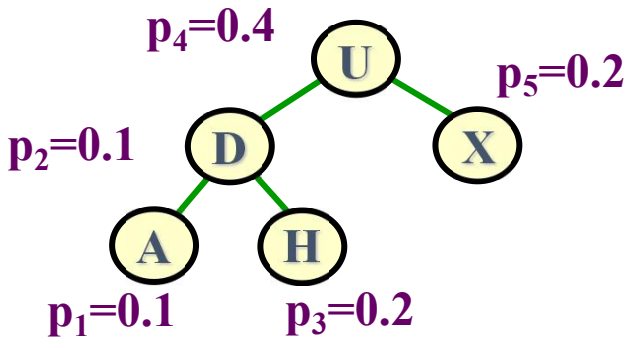
| A | D | H | U | X |
|---|---|---|---|---|
| $p_1 = 0.1$ | $p_2 = 0.1$ | $p_3 = 0.2$ | $p_4 = 0.4$ | $p_5 = 0.2$ |

- 比较相应于折半查找的判定树和用其他方法构成的查找树，其平均查找长度是不同的。




折半查找的判定树

$ASL_{succ}$= 0.2*1+(0.1+0.4)*2
+(0.1+0.2)*3 = 2.1




另一查找树

$ASL_{succ}$= 0.4*1+(0.1+0.2)*2
+(0.1+0.2)*3 = 1.9

- 折半查找在相等查找概率下其查找性能最好；在查找概率不相等的情况下，可以利用构造最/次优查找树的方法建立描述最佳查找过程的查找树。

# 最/次优查找树（介绍）



- 设有序顺序表中关键字为 $k_1, k_2, \ldots, k_n$, 它们的查找概率分别为 $p_1, p_2, \ldots, p_n$, 构成查找树后，它们在树中的层次为 $l_1, l_2, \ldots, l_n$。
- 基于该查找树的查找算法的平均查找长度为

$$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot l_i$$

- 使得 $ASL_{succ}$ 达到最小的静态查找树为静态最优查找树。求最优查找树的算法效率较低，原始算法复杂度达$O(n^3)$, 可动态规划优化成O（$n^2$）。也可求次优查找树，时间代价减低为$O(n\log_2 n)$。构造方法见教材。

# 动态查找

# 二叉搜索树存储

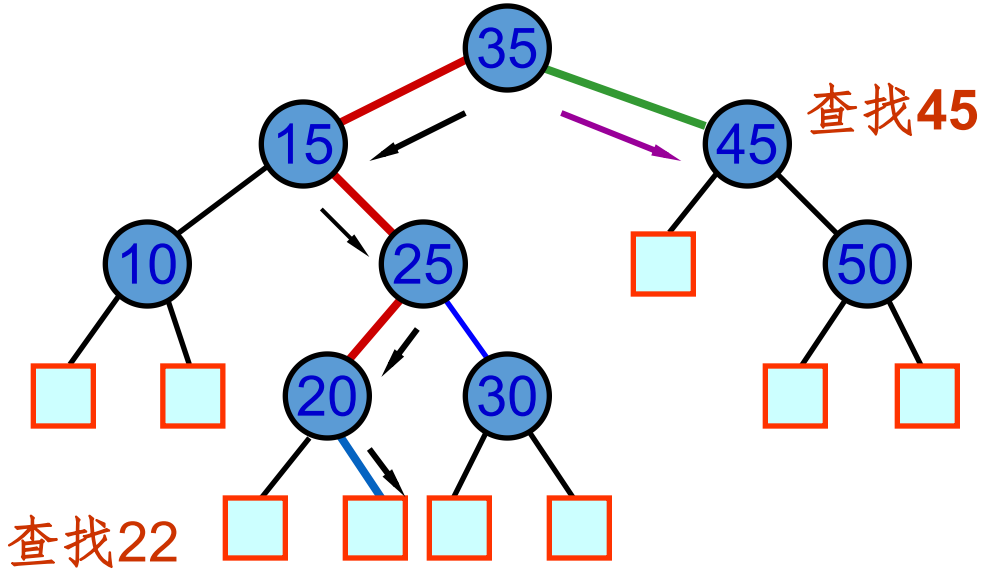


一般用二叉链表存储

```
typedef char ElemType;          //树结点数据类型
typedef struct node {           //二叉排序树结点
    ElemType data;
    struct node *LeftChild, *RightChild;
} BSTNode, *BST;                //二叉排序树定义
```

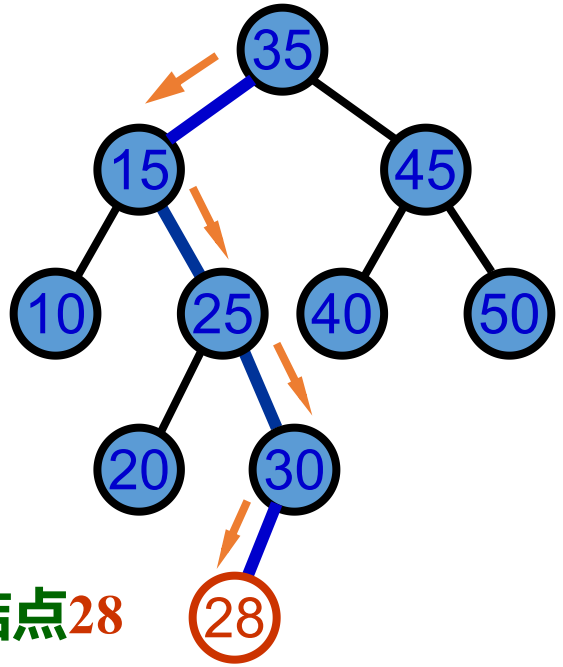
二叉排序树是二叉树的特殊情形，它继承了二叉树的结构，增加了自己的特性，对数据的存放增加了约束

- 可用判定树描述查找过程。内结点是树中原有结点，外结点是失败结点，代表树中没有的数据。
- 查找不成功时检测指针停留在某个失败结点。



**插入之前先使用查找算法在树中检查要插入元素有还是没有。**

- 查找成功：树中已有这个元素,不再插入。
- 查找不成功：树中原来没有关键字等于给定值的结点，把新元素加到查找操作停止的地方。

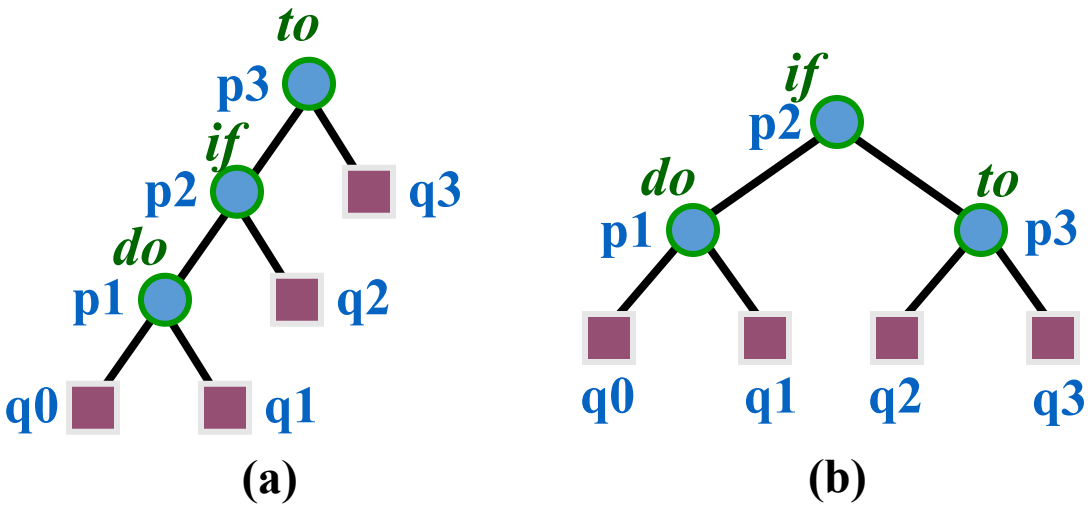



**插入新结点28**

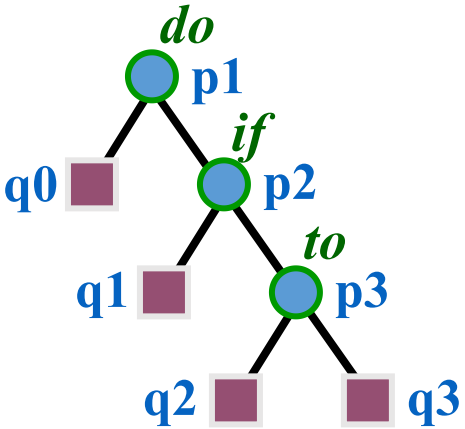
# 二叉排序树性能分析



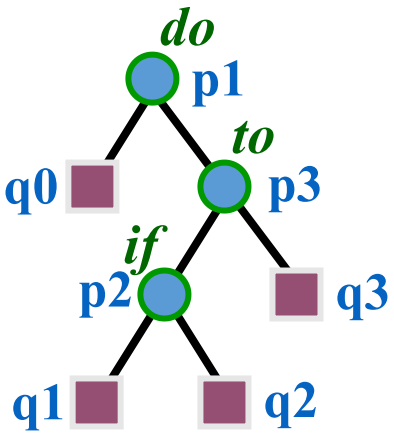
■例，已知关键字集合 { a1, a2, a3 } = { do, if, to }，对应查找概率 p1, p2, p3, 在各查找不成功间隔内查找概率分别为 q0, q1, q2, q3。可能的判定树如下所示。



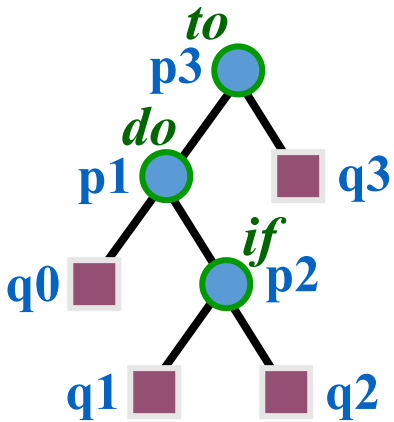
# 二叉排序树性能分析




(c)




(e)




(d)

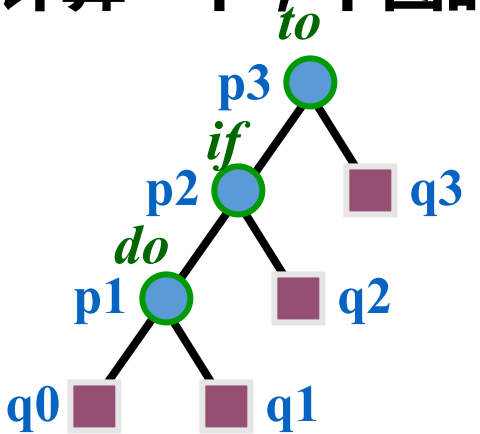
# 二叉排序树性能分析

- 设树中所有内、外结点的查找概率都相等：

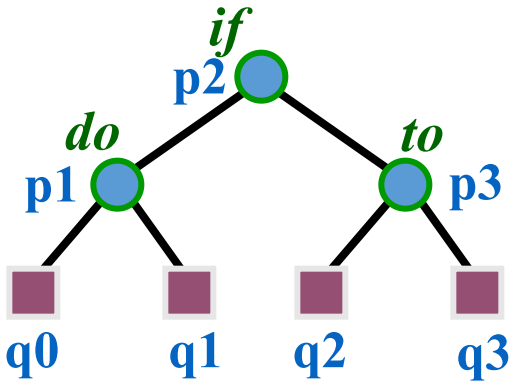
$$p[i] = 1/3,\ 1 \leqslant i \leqslant 3$$

$$q[j] = 1/4,\ 0 \leqslant j \leqslant 3$$

- 请大家计算一下，下图的成功ASL、不成功ASL




(a)




(b)

# 二叉排序树性能分析



- **设树中所有内、外结点的查找概率都相等：**

  $p[i] = 1/3,\ 1 \leqslant i \leqslant 3$

  $q[j] = 1/4,\ 0 \leqslant j \leqslant 3$

- **图(a):** $ASL_{succ} = 1/3*( 3 + 2 + 1 ) = 6/3 = 2$

  $ASL_{unsucc} = 1/4*( 3 + 3 + 2 + 1 ) = 9/4$

- **图(b):** $ASL_{succ} = 1/3*( 2 + 1 + 2 ) = 5/3$

  $ASL_{unsucc} = 1/4*( 2 + 2 + 2 + 2 ) = 8/4$

- **图(c):** $ASL_{succ} = 2,\ ASL_{unsucc} = 9/4$

- **图(d):** $ASL_{succ} = 2,\ ASL_{unsucc} = 9/4$

- **图(e):** $ASL_{succ} = 2,\ ASL_{unsucc} = 9/4$

平均查找长度达到最小的称作最优二叉排序树。

# 平衡二叉树/二叉平衡树



**可以感觉到二叉排序/搜索树的性能关键取决于“平衡”**

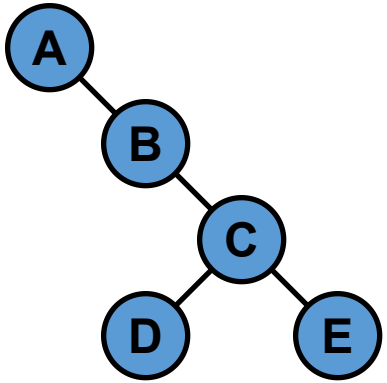
**于是，1962年，G.M. Adelson-Velsky 和 E.M. Landis发明了平衡二叉树/二叉平衡树，一般也叫AVL树。**

# 平衡二叉树/二叉平衡树

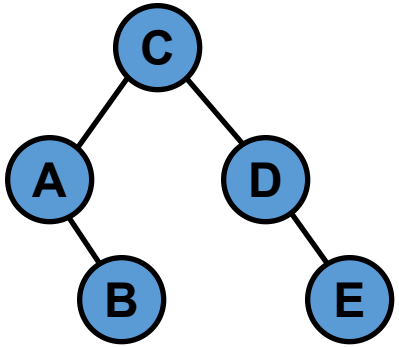


- 一棵AVL树或者是空树，或者是具有下列性质的二叉排序树：它的左子树和右子树都是平衡二叉树，且左子树和右子树的高度之差的绝对值不超过1。

高度是不平衡的




高度是平衡的



# 平衡二叉树/二叉平衡树



如果一棵二叉排序树高度是平衡的, 且有 $n$ 个结点，其高度可保持在$O(\log_2 n)$,平均查找长度也可保持在$O(\log_2 n)$。

为每个结点附加一个数字，给出该结点左子树的高度减去右子树的高度所得的高度差，这个数字即为结点的平衡因子 bf（balance factor）

//即节点的boy-friend，靠谱的bf，决定你人生的平衡^-^

# 平衡二叉树/二叉平衡树



平衡二叉树任一结点平衡因子只能取 –1, 0, 1。

如果一个结点的平衡因子的绝对值大于 1，则这棵二叉排序树就失去了平衡，不再是平衡二叉树。

对失衡的，需要进行平衡二叉树的动态调整

# 平衡化旋转



- **如果在一棵平衡二叉树中插入一个新结点，造成了不平衡。必须调整树的结构，使之平衡化。(注意保持平衡二叉树特性)**
- **平衡化旋转有两类：**
  - 单旋转 (LL和RR)
  - 双旋转 (LR和RL)
- **每插入一个新结点时，平衡二叉树中相关结点的平衡状态会发生改变。因此，在插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点的平衡因子。**

# 平衡化旋转



- 如果在某一结点发现高度不平衡，停止回溯。从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。

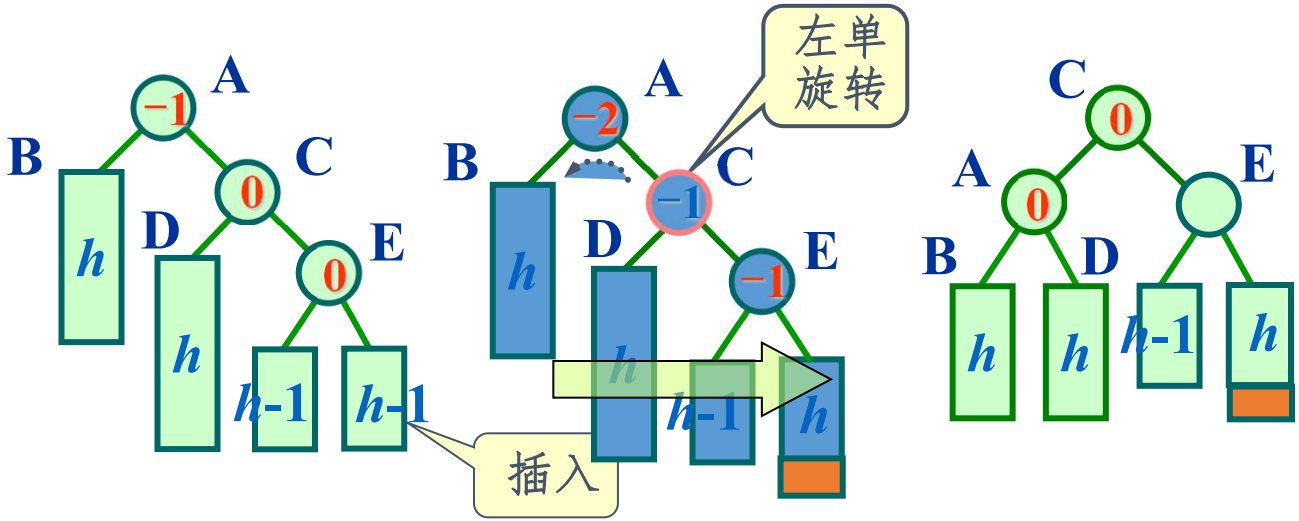
- 如果这三个结点处于一条直线上，则采用单旋转进行平衡化。单旋转可按其方向分为LL和RR，其中一个是另一个的镜像，其方向与不平衡的形状相关。

- 如果这三个结点处于一条折线上，则采用双旋转进行平衡化。双旋转分为LR和RL两类。

# 平衡化旋转



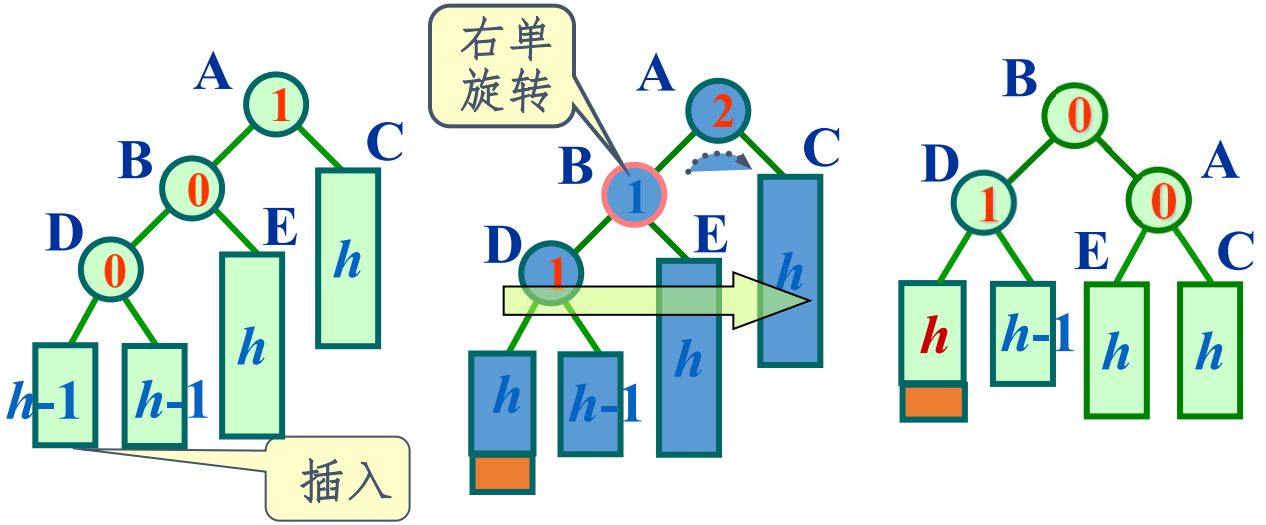
- 在结点A的右子女C的右子树E中插入新结点，该子树高度增1导致结点A的平衡因子变成-2，出现不平衡。为使树恢复平衡，从A沿插入路径连续取3个结点A、C和E，以结点C为旋转轴，让结点A反时针旋转。（如何保持原有特性？）



# 平衡化旋转



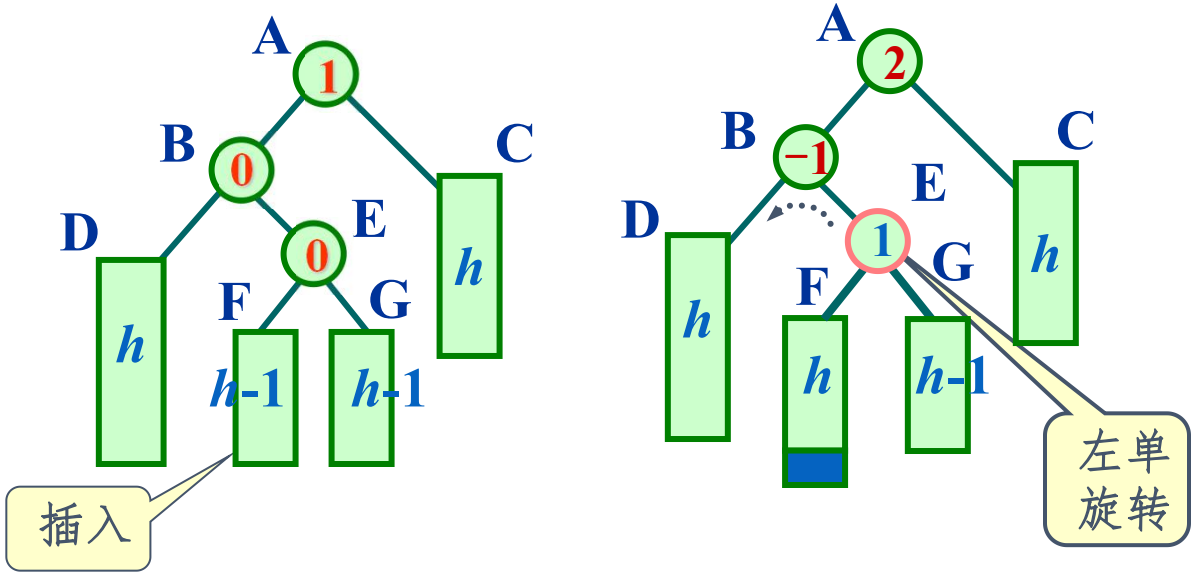
- 在结点A的左子女的左子树D上插入新结点使其高度增1导致结点A的平衡因子增到+2，造成不平衡。为使树恢复平衡，从A沿插入路径连续取3 个结点A、B和D，以结点B为旋转轴，将结点A顺时针旋转。



# 平衡化旋转



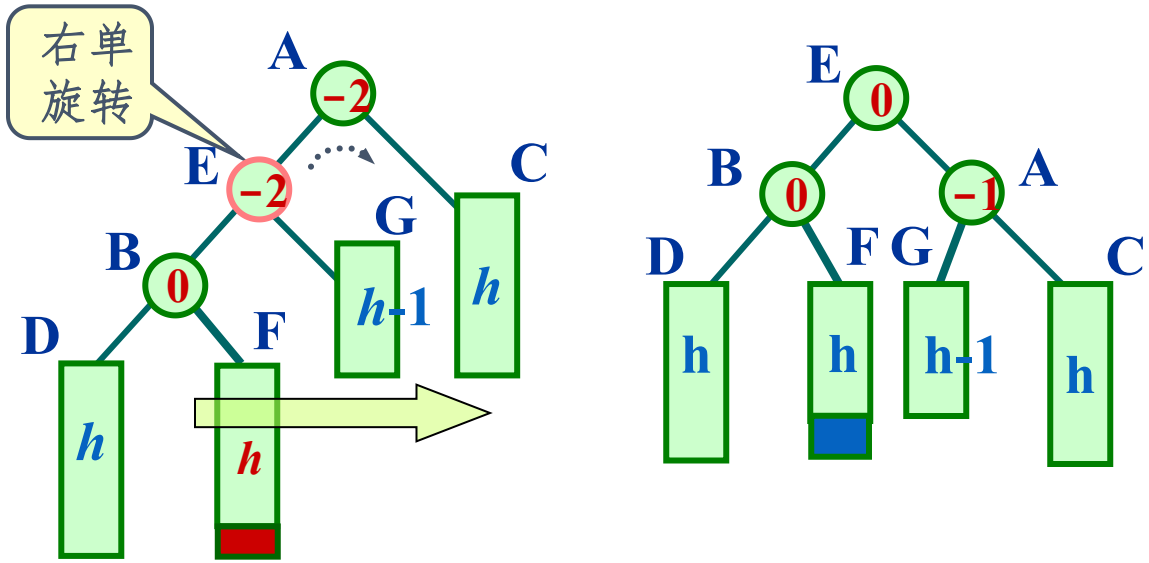
- 在结点A的左子女的右子树中插入新结点，该子树的高度增1导致结点A的平衡因子变为 2，发生了不平衡。



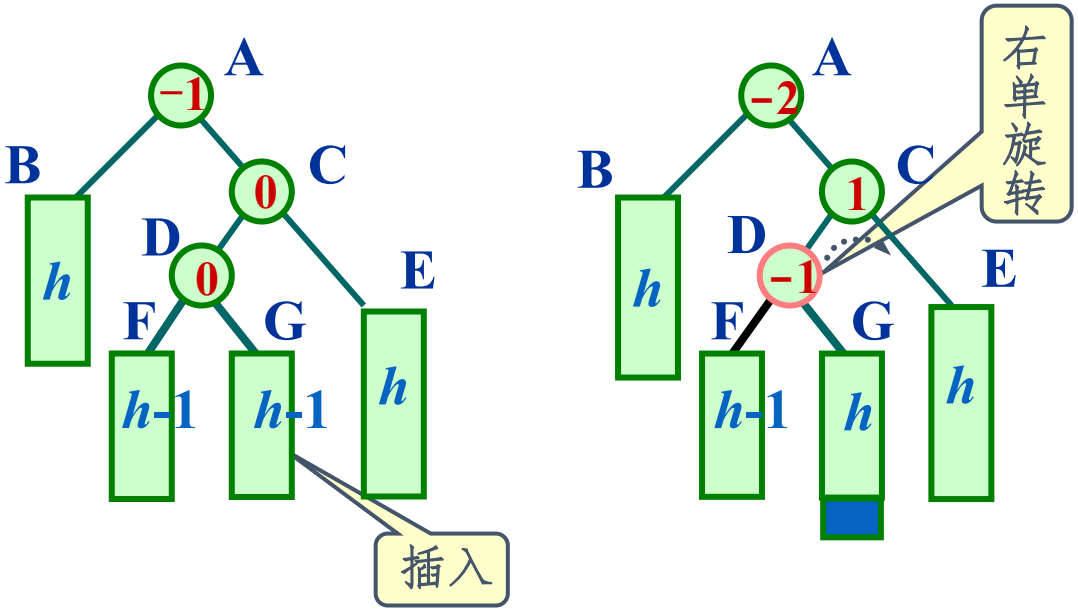
# 平衡化旋转



- 从结点A起沿插入路径选取3个结点A、B和E，先以结点E为旋转轴，将结点B反时针旋转，E顶替原B的位置。再以结点E为旋转轴，将结点A顺时针旋转。



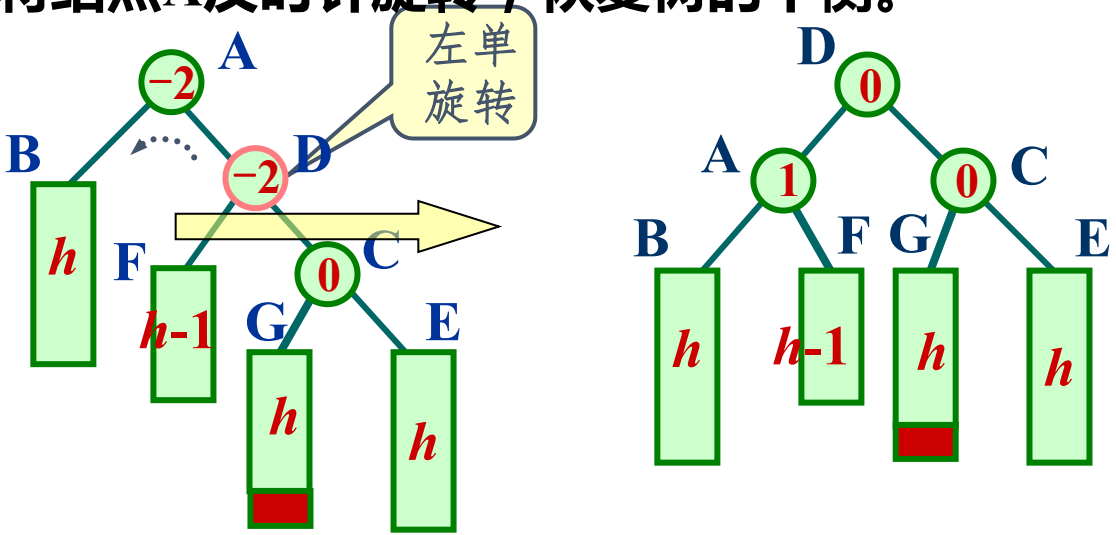
- 在根结点A的右子女的左子树中F或G上插入新结点，该子树高度增1。结点A的平衡因子变为-2，发生了不平衡。



# 平衡化旋转



- 从结点A起沿插入路径选取3个结点A、C和D。首以结点D为旋转轴，将结点C顺时针旋转，以D代替原来C的位置。再以D为旋转轴，将结点A反时针旋转，恢复树的平衡。



# 平衡二叉树的插入与删除：

1、考察当前节点的父节点的bf

2、根据情况看是否向根节点回溯（回溯原则：树高是否有变）

3、根据情况进行平衡化旋转（旋转原则：当前是否失衡）

# 插入对平衡的处理



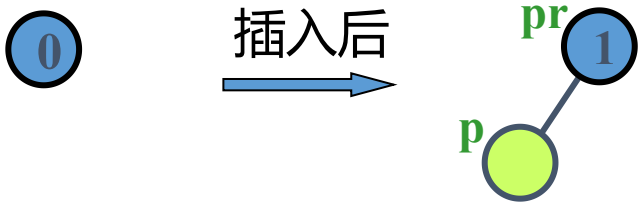
- 从棵空树开始，通过输入一系列元素关键字，逐步建立平衡二叉树。
- 在向一棵本来是高度平衡的平衡二叉树中插入一个新结点时，如果树中某个结点的平衡因子的绝对值 $|bf| > 1$，则出现了不平衡，需要做平衡化处理。
- 在插入新结点后，需从插入结点沿通向根的路径向上回溯，如果发现有不平衡的结点，需从这个结点出发，使用平衡旋转方法进行平衡化处理。

- **设新结点p的平衡因子为0，其父结点为pr。插入新结点后pr的平衡因子值有以下情况：**

1. **结点pr的平衡因子为0。说明刚才是在 pr 的较矮的子树上插入了新结点，此时不需做平衡化处理，返回主程序。子树的高度不变。**


2. **结点pr的平衡因子的绝对值 |bf| = 1。说明插入前pr的平衡因子是0，插入新结点后，以pr为根的子树不需平衡化旋转。但该子树高度增加，**

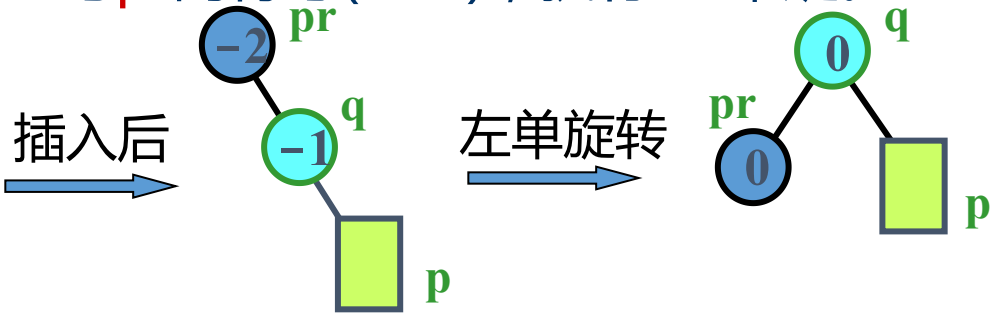
**还需从结点pr向根方向回溯，继续考查结点pr双亲(pr = Parent(pr))的平衡状态。**



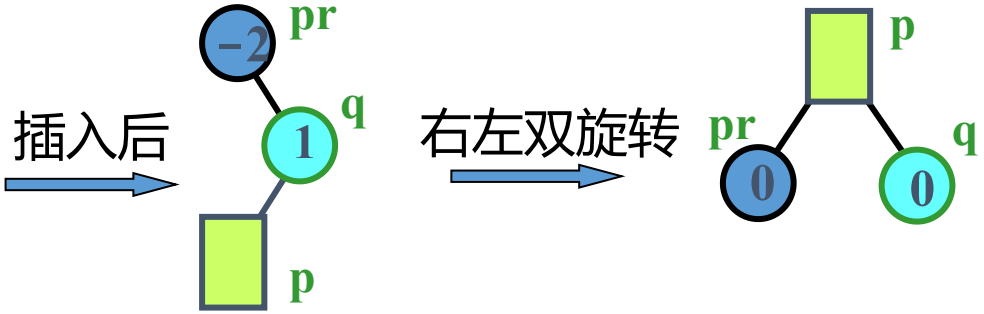

3. **结点pr的平衡因子的绝对值|bf| = 2。说明新结点在较高的子树上插入，造成了不平衡，需要做平衡化旋转。此时可进一步分2种情况讨论：**

① 若结点pr的bf = –2，说明右子树高，结合其右子女q的bf分别处理：

➢若q的bf与pr同符号(=–1)，执行RR单旋。

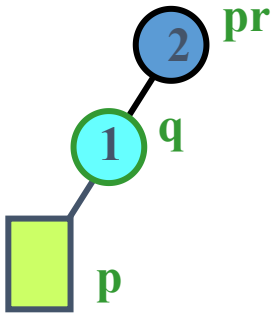



➢若q的bf与pr异符号(=1)，执行RL双旋。

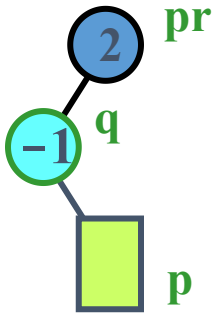


② 若结点pr的bf = 2，说明左子树高，结合其左子女q 的bf分别处理：

- 若q的bf与pr同符号(=1)，执行LL单旋；
- 若q的bf与pr异符号(=–1)，执行LR双旋。

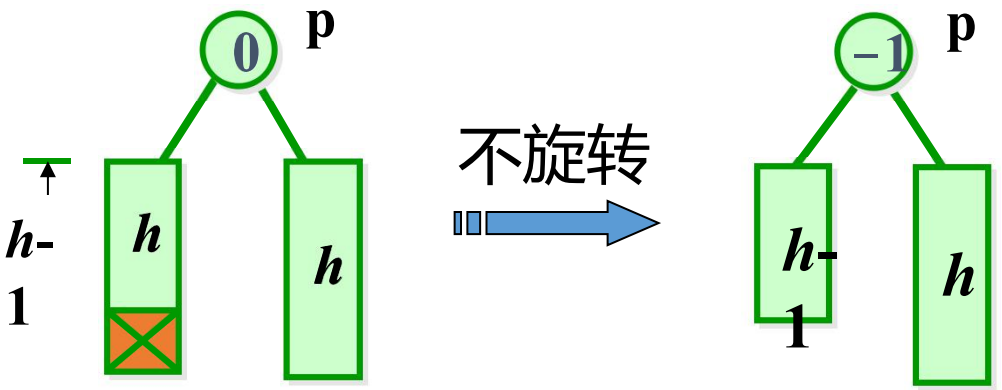


右单旋转

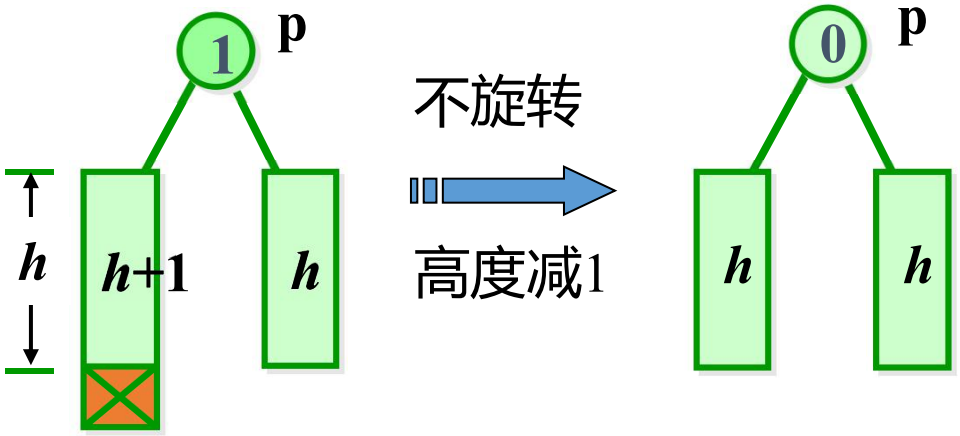


左右双旋转

((1)) 当前结点 p 的bf为0。如果它的左子树或右子树的高度被降低，则它的 bf 改为−1 或1。因该结点的高度未变，不必向上回溯，删除完成。



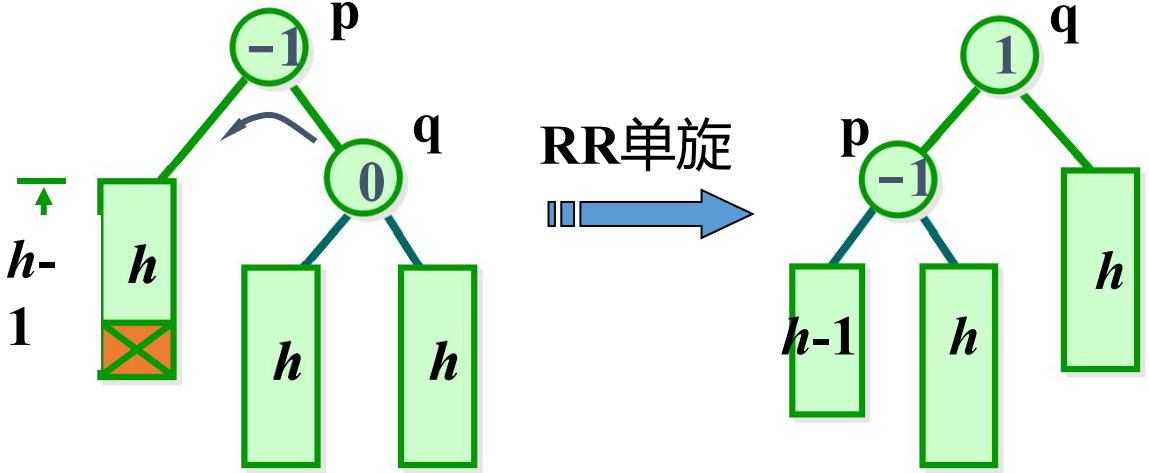
(2)当前结点 p 的bf不为0，且较高子树的高度被降低，则 p 的bf 改为0。因该结点的高度降低，需向上回溯，看其双亲是否失去平衡。



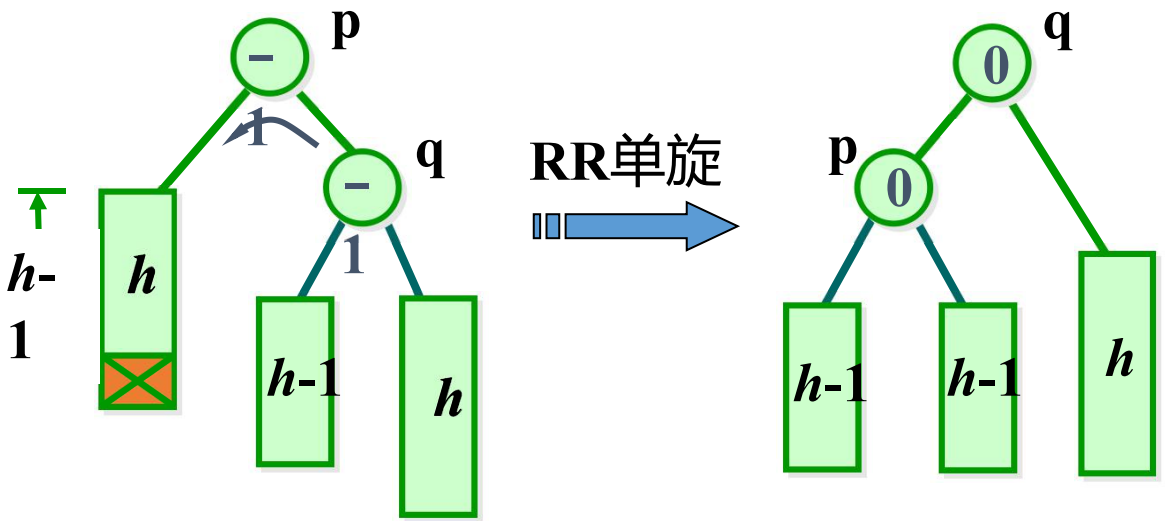
**(3) 当前结点 p 的bf 不为0，且较矮子树的高度又被降低，则在结点 p 发生不平衡。需要进行平衡化旋转来恢复平衡。**

**令 p 的较高的子树的根为 q (该子树高度未被降低)，根据 q 的 bf，有如下 3 种平衡化操作。**

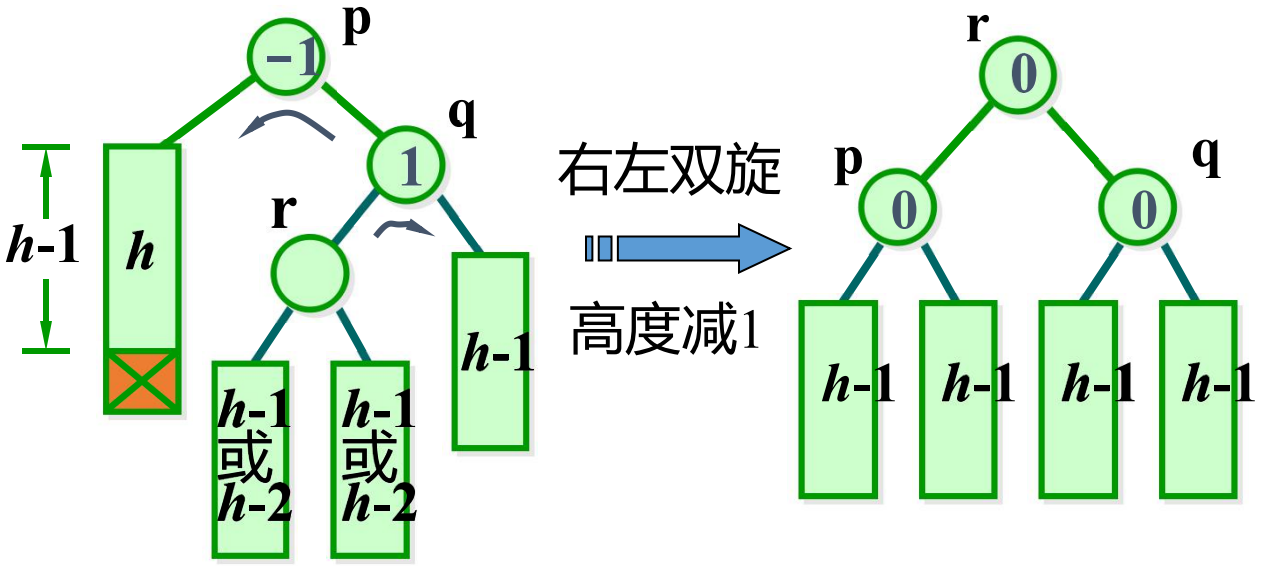
① 如果 q (较高的子树) 的 bf为 0，执行一个单旋转来恢复结点 p 的平衡，由于旋转后子树高度未降低，无需向上回溯。



② 如果 q 的bf与 p 的bf正负号相同，则执行一个单旋转来恢复平衡, 结点 p 和 q 的bf均改为0，由于子树高度降低，需向上回溯判断双亲结点是否失去平衡。



③ 如果 p 与 q 的bf的符号相反, 则执行一个双旋转来恢复平衡, 先围绕 q 转再围绕 p 转。新根结点的bf置为0，其它结点的bf相应处理，由于该子树高度降低，需向上回溯。

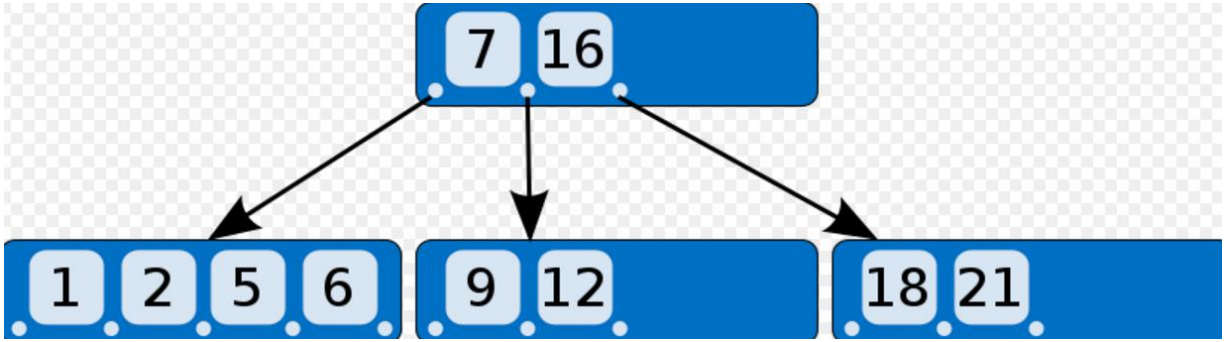


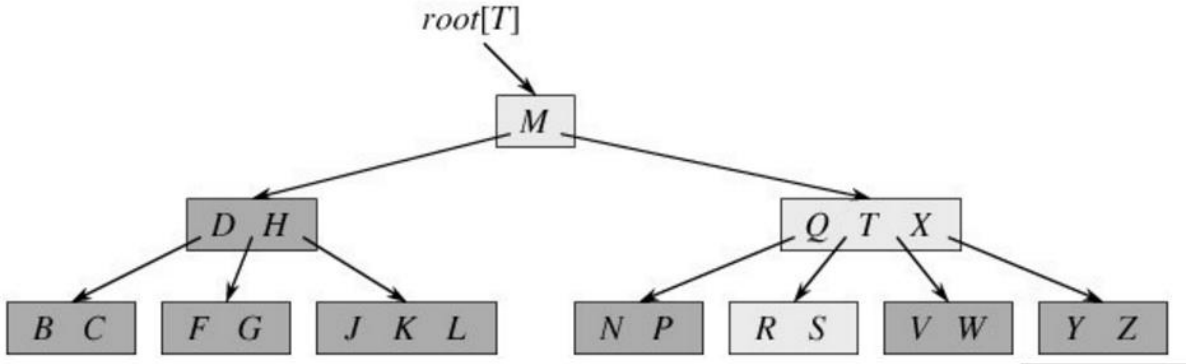
# B树简介



- B树是为了磁盘或其它存储设备而设计的一种多叉（B树每个内结点有多个分支，即多叉）平衡查找树，检查任意一个结点都需要一次磁盘访问，B树每个节点的大小一般设计为一个磁盘的块(block)的大小
- 文件系统、数据库系统都一般使用B树或者B树的各种变形结构来存储信息
- 外查找

# B树简介

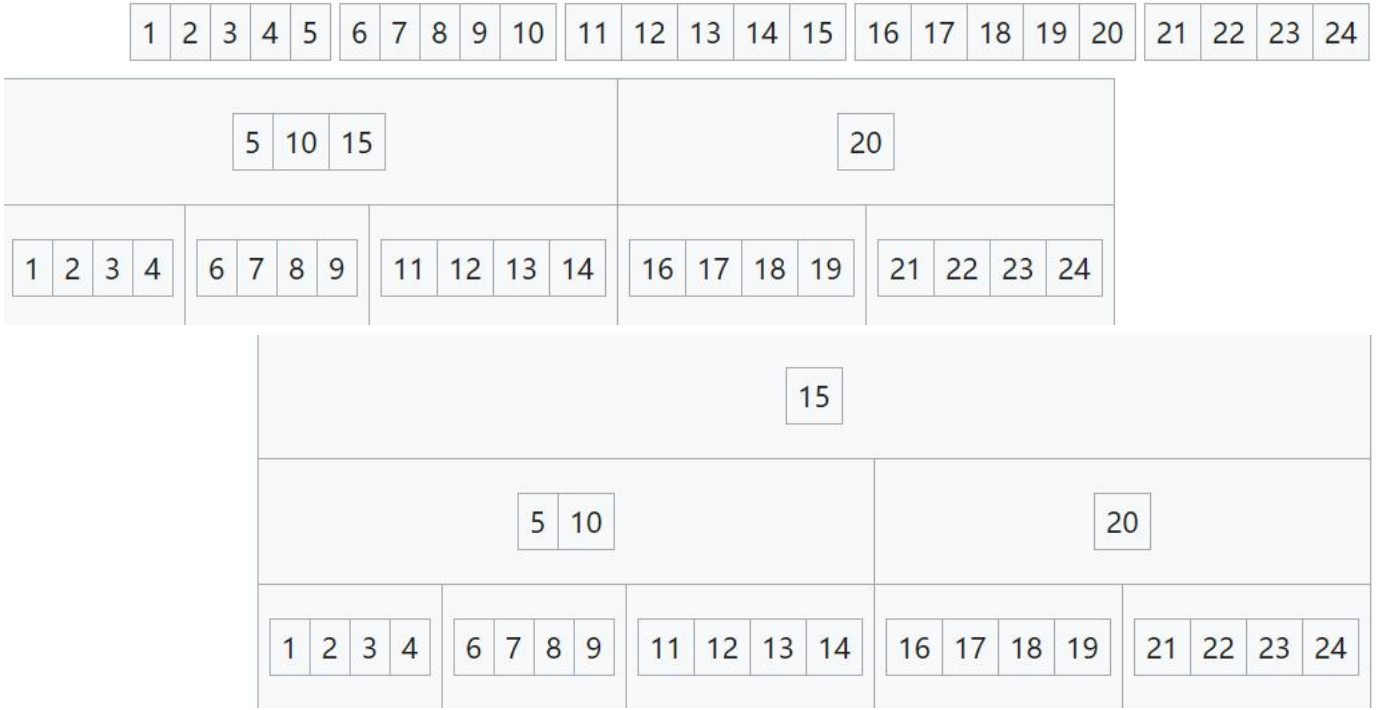


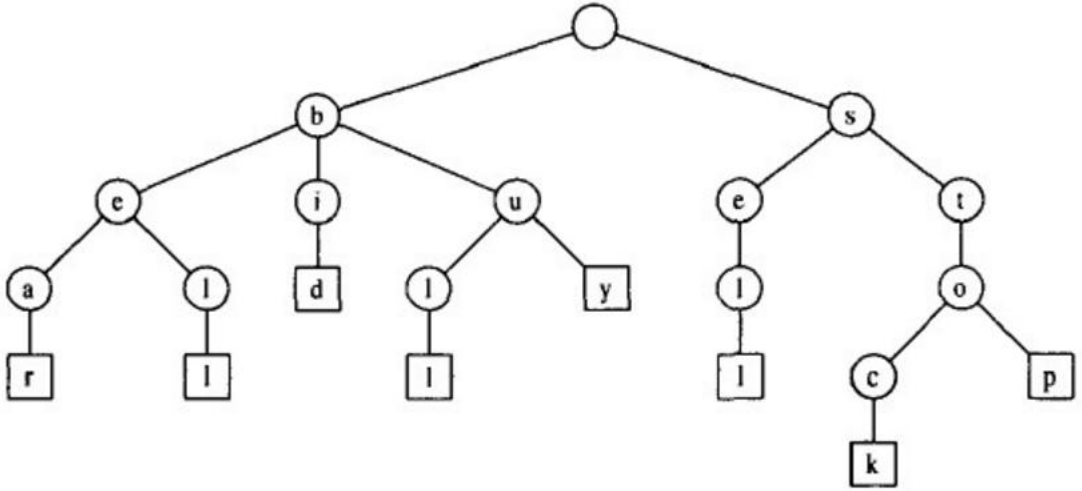
# B树简介



if the leaf nodes have maximum size 4 and the initial collection is the integers 1 through 24



# Trie树（字典树，键树，前缀树）




串{bear,bell,bid,bull,buy,sell,stock,stop}的标准Trie

词频统计，分词词性标注常采用Trie树作为存储结构

- 折半查找（递归、非递归），自行测试
- 静态链表（4.pdf），创建一个空的静态链表，插入1,2,4,5,6,7，删除5，插入9。实现带监视哨的算法分别查找5，查找7。（监视哨设在链表末尾）