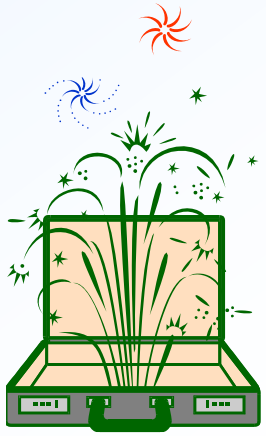
# 第九章 查找

- 查找的概念
- 静态查找表
- 动态查找表
- 哈希表



# 查找的概念

查找表 是由同一类型的数据元素(或记录)构成的集合,由于“集合”中的数据元素之间存在着松散的关系，因此查找表是一种应用灵便的数据结构。

对查找表的操作:

- 查询某个“特定的”数据元素是否在查找表中;
- 检索某个“特定的”数据元素的各种属性;
- 在查找表中插入一个数据元素;
- 从查找表中删去某个数据元素

## 查找表的分类：

静态查找表

仅作查询和检索操作的查找表。

动态查找表

在查找过程中同时插入查找表中不存在的数据元素，或者从查找表中删除已存在的某个数据元素,此类表为动态查找表。

## 关键字

是数据元素（或记录）中某个数据项的值，用以标识（识别）一个数据元素（或记录）。若此关键字可以识别唯一的一个记录，则称之谓“主关键字”。若此关键字能识别若干记录，则称之谓“次关键字”。

# 查找

根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素或（记录）。

若查找表中存在这样一个记录，则称“查找成功”，查找结果：给出整个记录的信息，或指示该记录在查找表中的位置；

否则称“查找不成功”，查找结果：给出

“空记录”或“空指针”。

## 如何进行查找？

查找的方法取决于查找表的结构。

由于查找表中的数据元素之间不存在明显的组织规律，因此不便于查找。

为了提高查找的效率， 需要在查找表中的元素之间人为地 附加某种确定的关系，换句话说， 用另外一种结构来表示查找表。

# 查找方法评价

- 查找速度
- 占用存储空间多少
- 算法本身复杂程度
- 平均查找长度ASL(Average Search Length)：为确定记录在表中的位置，需和给定值进行比较的关键字的个数的期望值叫查找算法的~

对含有$n$个记录的表，$ASL = \sum_{i=1}^{n} p_i c_i$

其中：$p_i$为查找表中第$i$个元素的概率，$\sum_{i=1}^{n} p_i = 1$

$c_i$为找到表中第$i$个元素所需比较次数

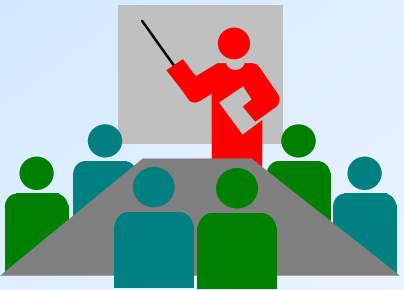
# 静 态 查 找 表

抽象数据类型静态查找表的定义:

**ADT StaticSearchTable {**

数据对象D: D是具有相同特性的数据元素的集合。每个数据元素含有类型相同的关键字，可唯一标识数据元素。

数据关系R：数据元素同属一个集合。



基本操作 P：

Create(&ST, n); //构造一个含 n 个数据元素的静态查找表ST。

Destroy(&ST); //销毁表ST。

Search(ST, key); //查找 ST 中其关键字等于kval 的数据元素。

Traverse(ST, Visit()); //按某种次序对ST的每个元素调用函数Visit()一次且仅一次，

} ADT StaticSearchTable

## ▪顺序表的查找

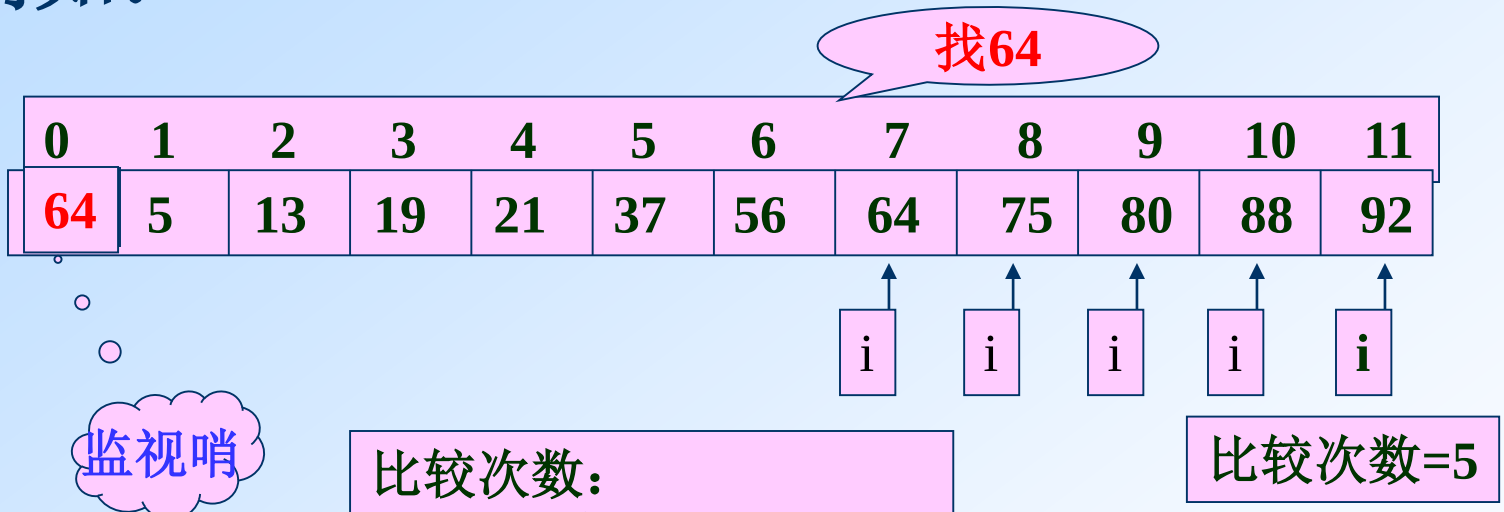
以顺序表表示静态查找表，则Search函数可用顺序查找来实现。其顺序存储结构如下：

```
typedef  struct {
  ElemType *elem;   // 数据元素存储空间基址，建表时
                                按实际长度分配，0号单元留空
   int      length;    // 表的长度
} SSTable;
```



查找过程：从表的一端开始逐个进行记录的关键字和给定值的比较。

例如：




比较次数：

查找第n个元素： 1

查找第n-1个元素： 2

……

查找第1个元素： n

查找第i个元素： n+1-i

查找失败： n+1

## 算法描述：

```
int Search_Seq(SSTable ST,
                                KeyType kval) {
   // 在顺序表ST中顺序查找其关键字等于
   //  key的数据元素。若找到，则函数值为
   // 该元素在表中的位置，否则为0。
   ST.elem[0].key = kval;      // 设置“哨兵”
   for (i=ST.length; ST.elem[i].key!=kval; --i);
                               // 从后往前找
   return i;                   // 找不到时，i为0
} // Search_Seq
```

# 顺序查找性能分析

**查找算法的平均查找长度(Average Search Length)：**

**为确定记录在查找表中的位置，需和给定值进行比较的关键字个数的期望值。**

**其中：n 为表长** $ASL = \sum_{i=1}^{n} P_i C_i$

**$P_i$ 为查找表中第i个记录的概率** $\sum_{i=1}^{n} P_i = 1$

**$C_i$为找到该记录时，曾和给定值比较过的关键字的个数**

对顺序表而言，$C_i = n-i+1$

$ASL = nP_1 +(n-1)P_2 + +2P_{n-1}+P_n$

在等概率查找的情况下，$P_i = \frac{1}{n}$

顺序表查找的平均查找长度为：

$$ASL_{ss} = \frac{1}{n}\sum_{i=1}^{n}(n-i+1) = \frac{n+1}{2}$$

在不等概率查找的情况下，$ASL_{ss}$ 在

$$P_n \geqslant P_{n-1} \geqslant \cdots \geqslant P_2 \geqslant P_1$$

时取极小值。表中记录按查找概率由小到大重新排列,以提高查找效率。

若查找概率无法事先测定，则查找过程采取的改进办法是，在每次查找之后，将刚刚查找到的记录直接移至表尾的位置上。



## ▪有序表的查找

顺序表的查找算法简单，但平均查找长度较大，不适用于表长较大的查找表。

若以有序表表示静态查找表，则查找过程可以基于“折半”进行。

### 折半查找

查找过程：每次将待查记录所在区间缩小一半。

适用条件：采用顺序存储结构的有序表。

# 折半查找算法实现

1.设表长为n，low、high和mid分别指向待查元素所在区间的上界、下界和中点,k为给定值。

2.初始时，令

low=1,high=n,mid=⌊(low+high)/2⌋

让k与mid指向的记录比较

若k==r[mid].key，查找成功

若k<r[mid].key，则high=mid-1

若k>r[mid].key，则low=mid+1

3.重复上述操作，直至low>high时，查找失败。

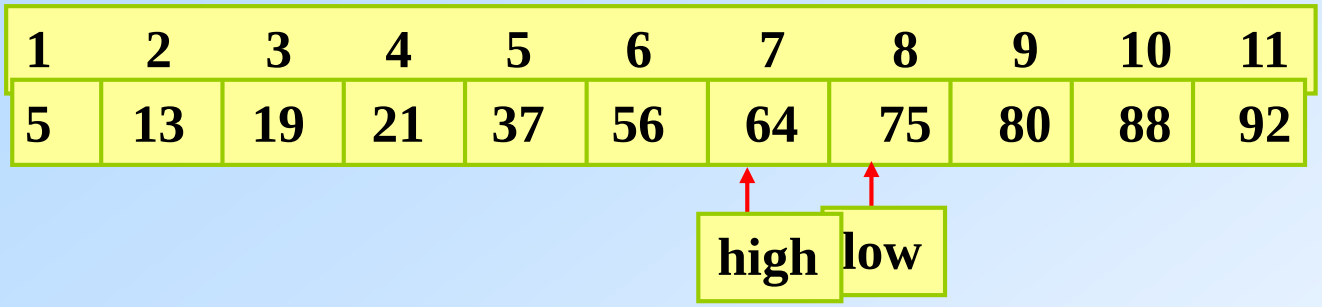
# 例如 key =21 的查找过程

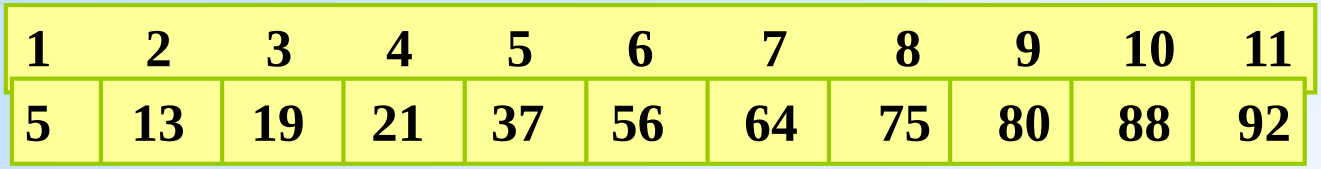

**low** 指示查找区间的下界；

**high** 指示查找区间的上界；

**mid** = (low+high)/2。

# 例key =70 的查找过程




当下界low大于上界high时，则说明表中没有关键字等于Key的元素，查找不成功。

# 折半查找算法

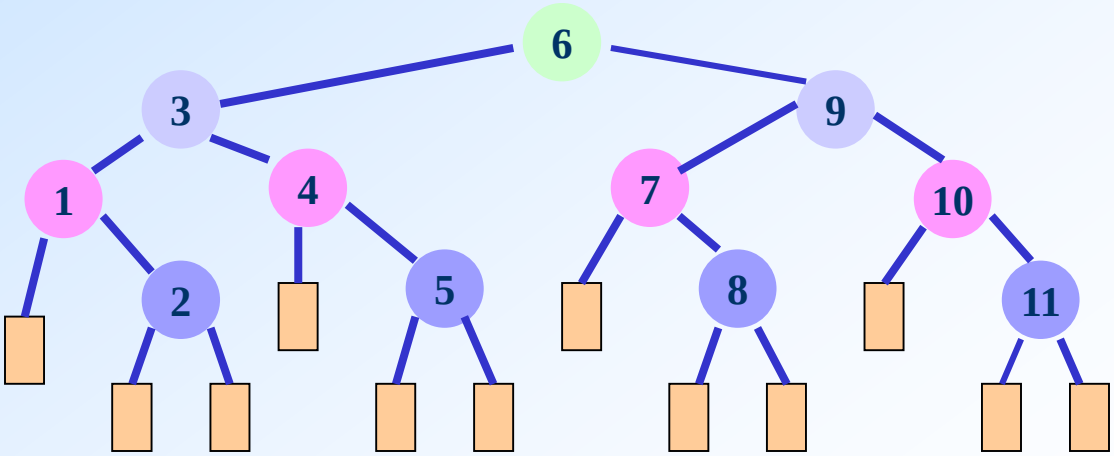
```
int Search_Bin ( SSTable ST, KeyType kval ) {
  low = 1;  high = ST.length;     // 置区间初值
  while (low <= high) {
    mid = (low + high) / 2;
    if （kval == ST.elem[mid].key  )
       return  mid;        // 找到待查元素
    else  if ( kval < ST.elem[mid].key) )
      high = mid - 1;       // 继续在前半区间进行查找
    else  low = mid + 1; // 继续在后半区间进行查找
  }
  return 0;                 // 顺序表中不存在待查元素
} // Search_Bin
```

# 折半查找的性能分析

- 判定树：描述查找过程的二叉树。
- 有n个结点的判定树的深度为$\lfloor \log_2 n \rfloor + 1$
- 折半查找法在查找过程中进行的比较次数最多不超过$\lfloor \log_2 n \rfloor + 1$

先看一个有11个元素的表的例子：n=11

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Ci | 3 | 4 | 2 | 3 | 4 | 1 | 3 | 4 | 2 | 3 | 4 |



假设 有序表的长度$n=2^h-1$（反之$h=\log_2^{(n+1)}$），

则描述折半查找的判定树是深度为h的满二叉树。

树中层次为1的结点有1个，层次为2的结点有2个，层次为h的结点有$2^{h-1}$个。假设表中每个记录的查找概率相等

则查找成功时折半查找的平均查找长度

$$ASL_{bs}=\frac{1}{n}\sum_{i=1}^{n}C_i=\frac{1}{n}\left[\sum_{j=1}^{h}j\times 2^{j-1}\right]=\frac{n+1}{n}\log_2(n+1)-1$$



在 n>50 时，可得近似结果

$$ASL_{bs} \approx \log_2(n+1) - 1$$

可见，

- 折半查找的效率比顺序查找高。
- 折半查找只能适用于有序表，并且以顺序存储结构存储。

# 练习：写出折半查找的递归算法

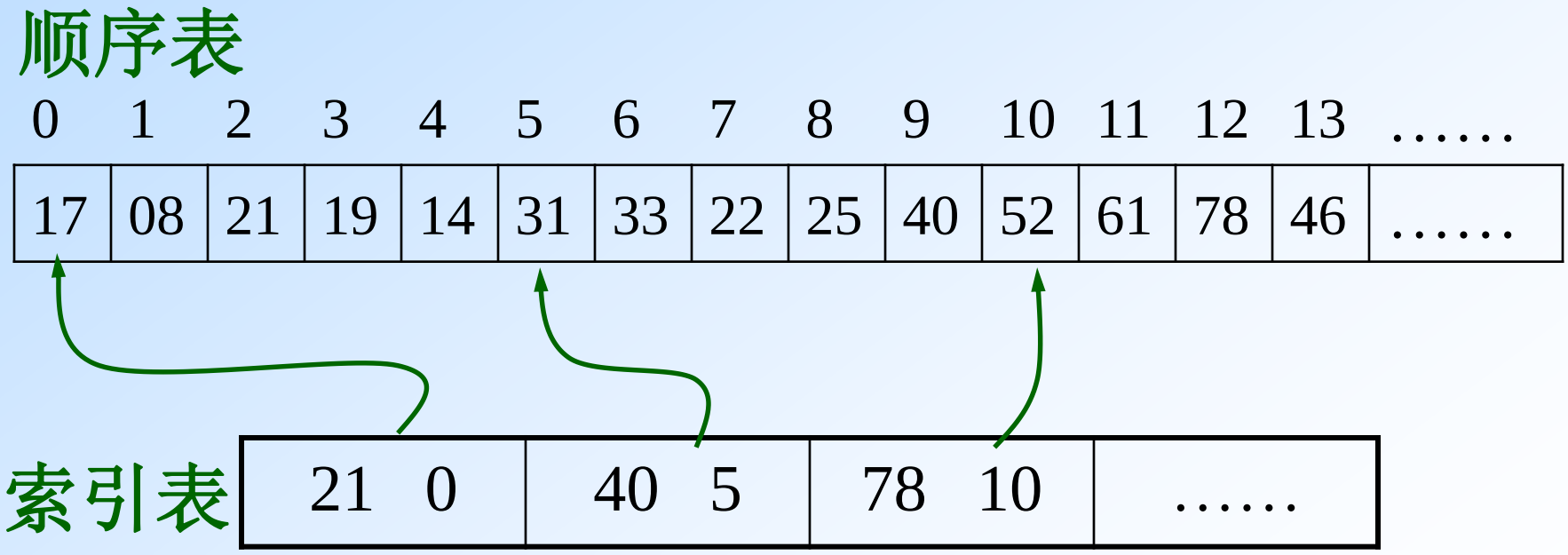
```
template<class T>
int BinSearch(T A[ ], int low, int high, T key)
{
  int midl;
  T midvalue;
  if (low>high)
    return (-1);
   else
    {
     mid = (low+high)/2;
     midvalue = A[mid];
// 终止条件：找到key
      if (key = = midvalue)
        return mid;
      // 若key<midvalue, 则查下子表；否则，查上子表
       else
         if (key < midvalue)
           //
           return BinSearch（A, low, mid – 1, key);
      else
           return BinSearch ( A, mid+1, high, key);
      }
 }
```

# 顺序表和有序表的比较

| | 顺序表 | 有序表 |
|---|---|---|
| 表的特性 | 无序 | 有序 |
| 存储结构 | 顺序 或 链式 | 顺序 |
| 插删操作 | 易于进行 | 需移动元素 |
| ASL的值 | 大 | 小 |

# ■索引顺序表

在建立顺序表的同时，建立一个索引项，包括两项：关键字项和指针项。索引表按关键字有序，表则为分块有序




索引顺序表 = 索引 + 顺序表

# 索引顺序查找

又称分块查找

查找过程：将表分成几块，块内无序，块间有序；先确定待查记录所在块，再在块内查找

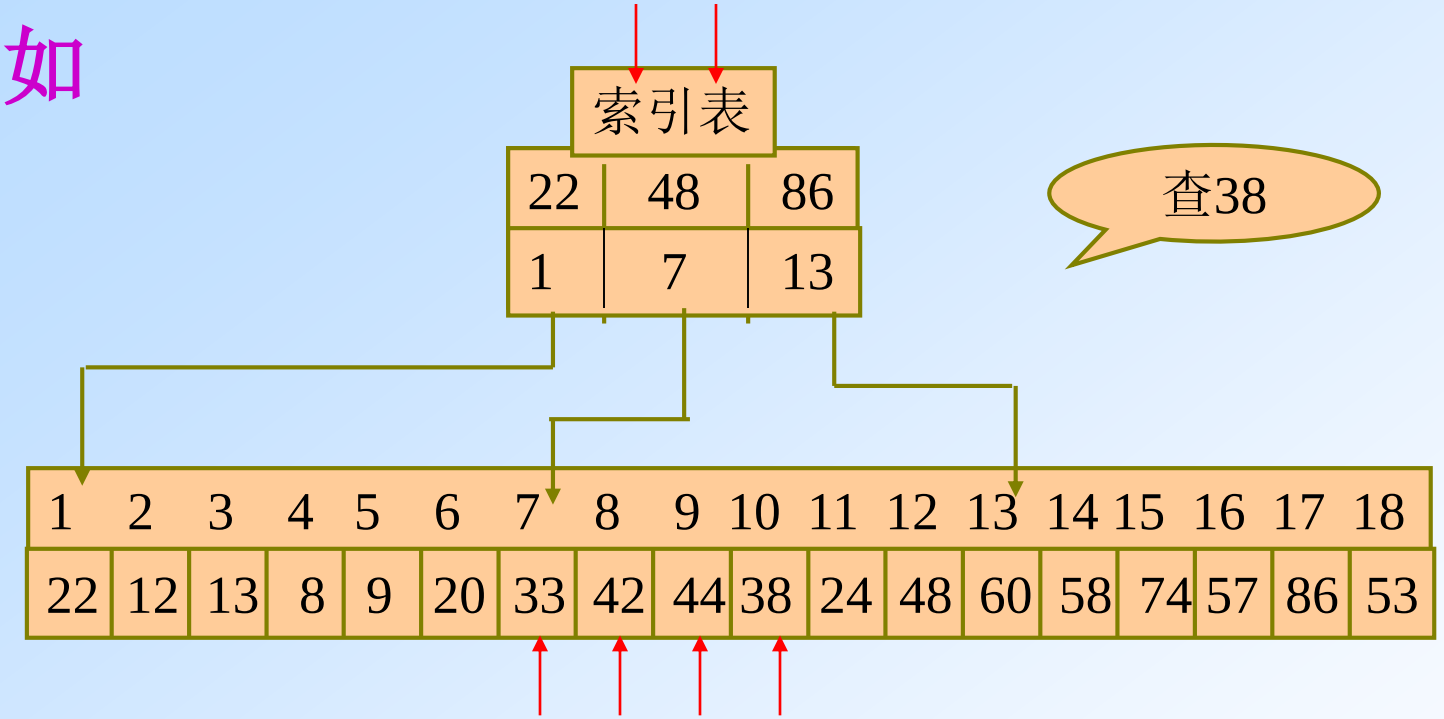
适用条件：分块有序表

算法实现：

用数组存放待查记录,每个数据元素至少含有关键字域

建立索引表，每个索引表结点含有最大关键字域和指向本块第一个结点的指针

# 例如


索引表
22 48 86
1 7 13
查38
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
22 12 13 8 9 20 33 42 44 38 24 48 60 58 74 57 86 53

# 分块查找方法评价

$$ASL_{bs} = L_b + L_w$$

其中：$L_b$ ——查找索引表确定所在块的平均查找长度

$L_w$ ——在块中查找元素的平均查找长度

若将表长为$n$的表平均分成$b$块，每块含$s$个记录，并设表中每个记录的查找概率相等，则：

(1)用顺序查找确定所在块：$ASL_{bs} = \frac{1}{b}\sum_{j=1}^{b} j + \frac{1}{s}\sum_{i=1}^{s} i$

$$= \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2}\left(\frac{n}{s} + s\right) + 1$$

(2)用折半查找确定所在块：$ASL_{bs} \approx \log_2\left(\frac{n}{s} + 1\right) + \frac{s}{2}$

# 查找方法比较

| | 顺序查找 | 折半查找 | 分块查找 |
|---|---|---|---|
| ASL | 最大 | 最小 | 两者之间 |
| 表结构 | 有序表、无序表 | 有序表 | 分块有序表 |
| 存储结构 | 顺序存储结构<br>线性链表 | 顺序存储结构 | 顺序存储结构<br>线性链表 |

# 几种查找表的特性

| | 查找 | 插入 | 删除 |
|---|---|---|---|
| 无序顺序表 | O(n) | O(1) | O(1) |
| 无序线性链表 | O(n) | O(1) | O(1) |
| 有序顺序表 | O(logn) | O(n) | O(n) |
| 有序线性链表 | O(n) | O(1) | O(1) |

# 结论：

- 从查找性能看，最好情况能达O(logn)，此时要求表有序；
- 从插入和删除的性能看，最好情况能达O(1)，此时要求存储结构是链表。

# 动态查找表

动态查找表的特点:表结构本身是在查找过程中动态生成。若表中存在其关键字等于给定值key的记录,表明查找成功；否则插入关键字等于key的记录。

抽象数据类型动态查找表的定义：

**ADT DynamicSearchTable {**

数据对象D：　D是具有相同特性的数据元素的集合。

每个数据元素含有类型相同的关键字可唯一标识数据元素。

数据关系R：　数据元素同属一个集合。

## 基本操作：

InitDSTable(&DT)//构造一个空的动态查找表DT。

DestroyDSTable(&DT)//销毁动态查找表DT。

SearchDSTable(DT, key);//查找 DT 中与关键字key等值的元素。

InsertDSTable(&DT, e);//若 DT 中不存在其关键字等于 e.key 的 数据元素，则插入 e 到 DT。

DeleteDSTable(&T, key);//删除DT中关键字等于key的数据元素。

TraverseDSTable(DT, Visit());//按某种次序对DT的每个结点调用函数 Visit() 一次且至多一次。

}ADT DynamicSearchTable



next

# 二叉排序树（二叉查找树）

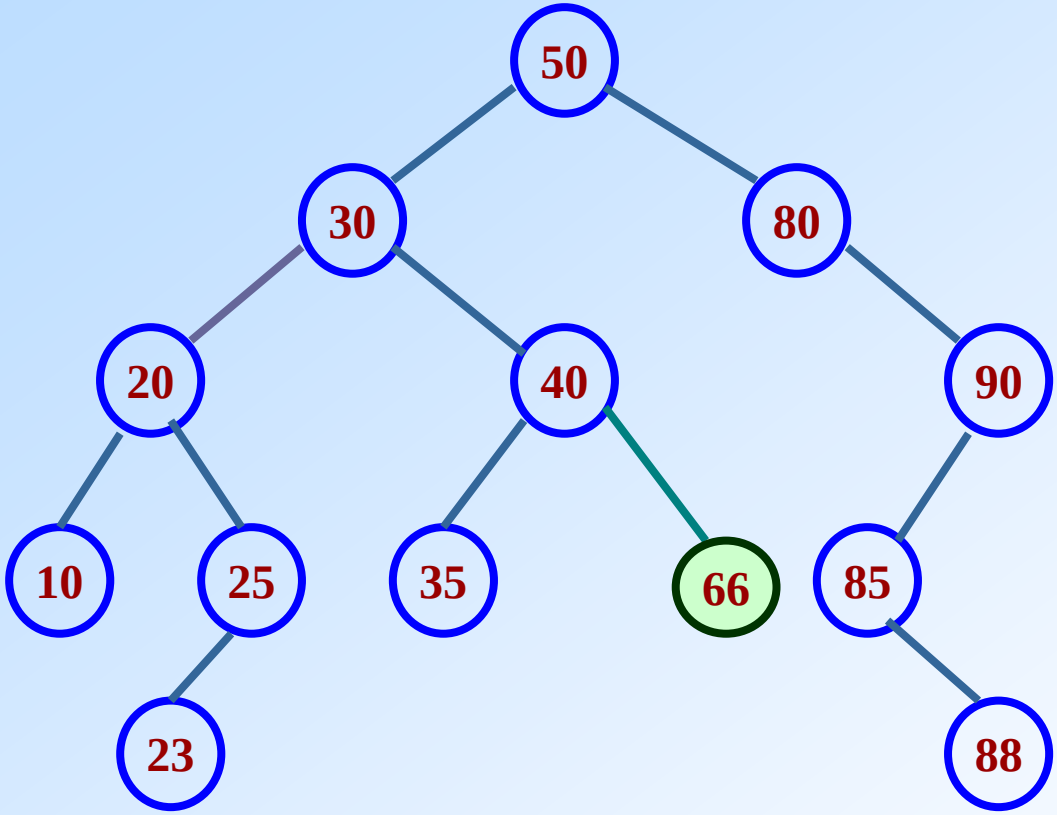
## 定义

二叉排序树或者是一棵空树；或者是具有如下特性的二叉树：

若它的左子树不空，则左子树上所有结点的值均小于根结点的值；

若它的右子树不空，则右子树上所有结点的值均大于根结点的值；

它的左、右子树也都分别是二叉排序树。




不是二叉排序树。

# 二叉排序树的存储结构

以二叉链表形式存储

```
typedef struct BiTNode { // 结点结构
    TElemType      data;
    struct BiTNode  *lchild, *rchild; // 左右指针
 } BiTNode, *BiTree;
```

# 二叉排序树的查找算法

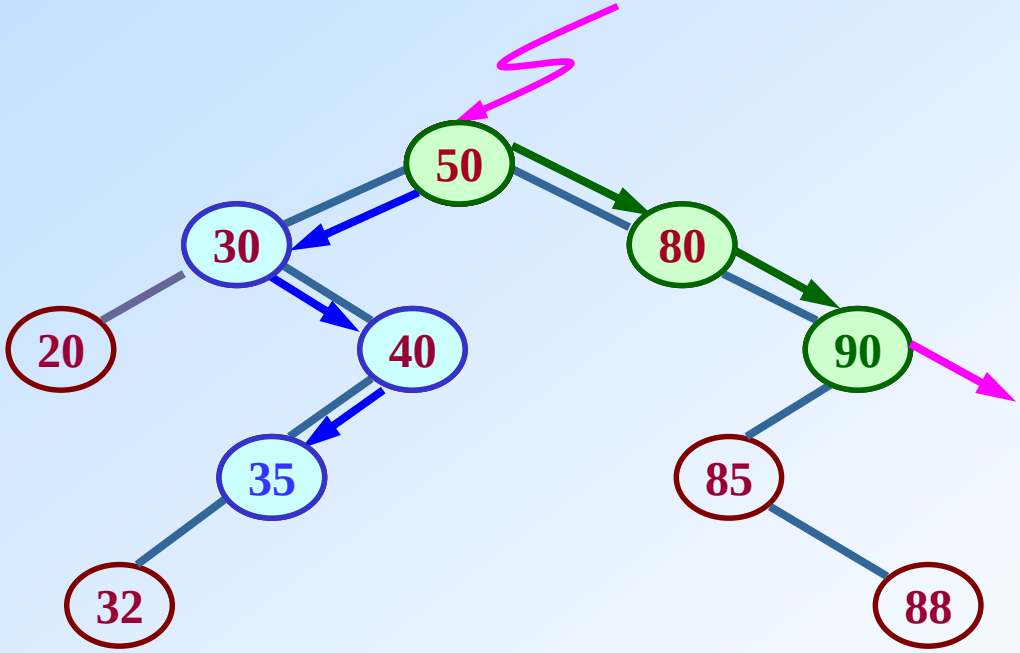
若二叉排序树为空，则查找不成功；
否则

1）若给定值等于根结点的关键字，则查找成功；

2）若给定值小于根结点的关键字，则继续在左子树上进行查找；

3）若给定值大于根结点的关键字，则继续在右子树上进行查找。

# 在二叉排序树中查找关键字值等于50,35,90,95



```
Status SearchBST (BiTree T, KeyType kval, BiTree f,
                                    BiTree &p ) {
  if (!T)
  { p = f;  return FALSE; }  // 查找不成功
  else  if ( EQ(kval, T->data.key) )
  { p = T;  return TRUE; }  // 查找成功
  else  if ( LT(kval, T->data.key) )
  return SearchBST (T->lchild, kval, T, p ); // 在
  左子树中继续查找
  else
  return SearchBST (T->rchild, kval, T, p );
                        // 在右子树中继续查找

} // SearchBST
```



# 二叉排序树的插入算法

- 根据动态查找表的定义，“插入”操作在查找不成功时才进行；
- 若二叉排序树为空树，则新插入的结点为新的根结点；否则，新插入的结点必为一个新的叶子结点，其插入位置由查找过程得到。

```
Status Insert BST(BiTree &T, ElemType e ) {
if (!SearchBST ( T, e.key, NULL, p ))
    {  s =  new  BiTNode; // 为新结点分配空间
       s->data = e;
       s->lchild = s->rchild = NULL;
       if  ( !p )  T = s;     // 插入 s 为新的根结点
       else   if ( LT(e.key, p->data.key) )
                  p->lchild = s;       // 插入 *s 为 *p 的左孩子
       else  p->rchild = s;     // 插入 *s 为 *p 的右孩子
       return TRUE;     // 插入成功
     }
  else return FALSE;
} // Insert BST
```

# 结论

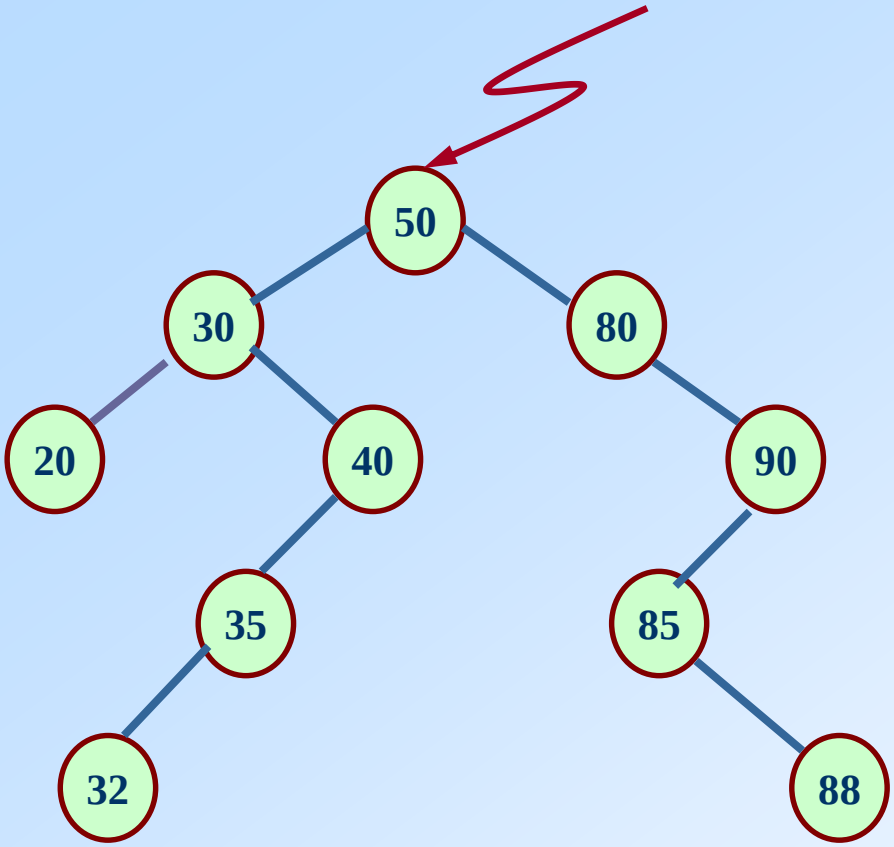
- 一个无序序列可以通过构造一棵二叉排序树而变成一个有序序列
- 每次插入的新结点都是二叉排序树上新的叶子结点
- 插入时不必移动其它结点，仅需修改某个结点的指针

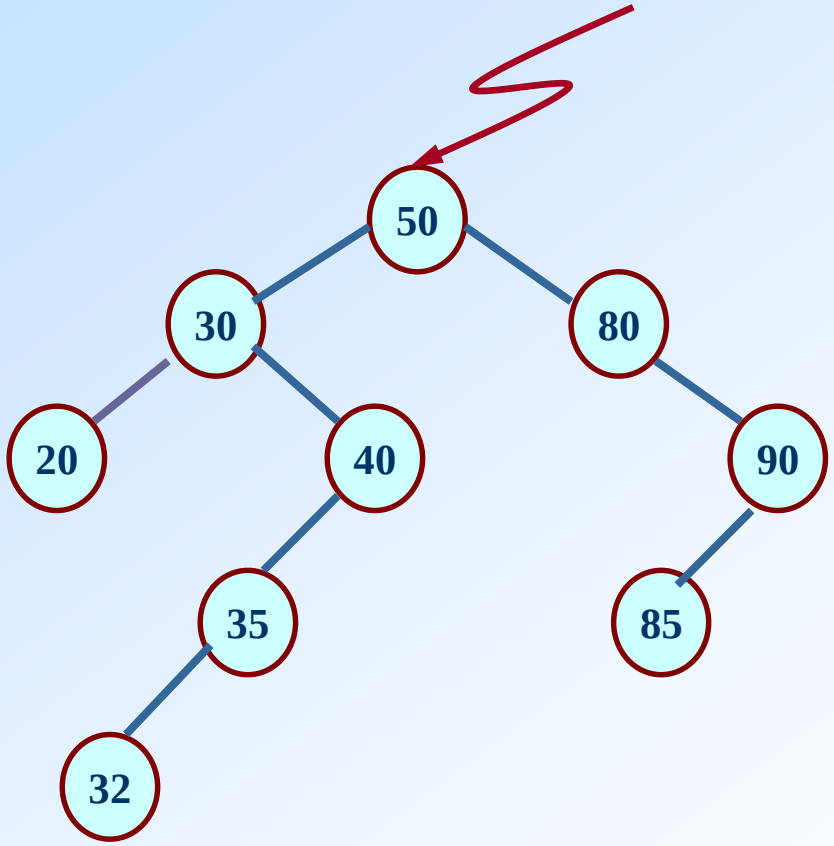
# 二叉排序树的删除算法

和插入相反，删除在查找成功之后进行，并且要求在删除二叉排序树上某个结点之后，仍然保持二叉排序树的特性。

可分三种情况讨论：

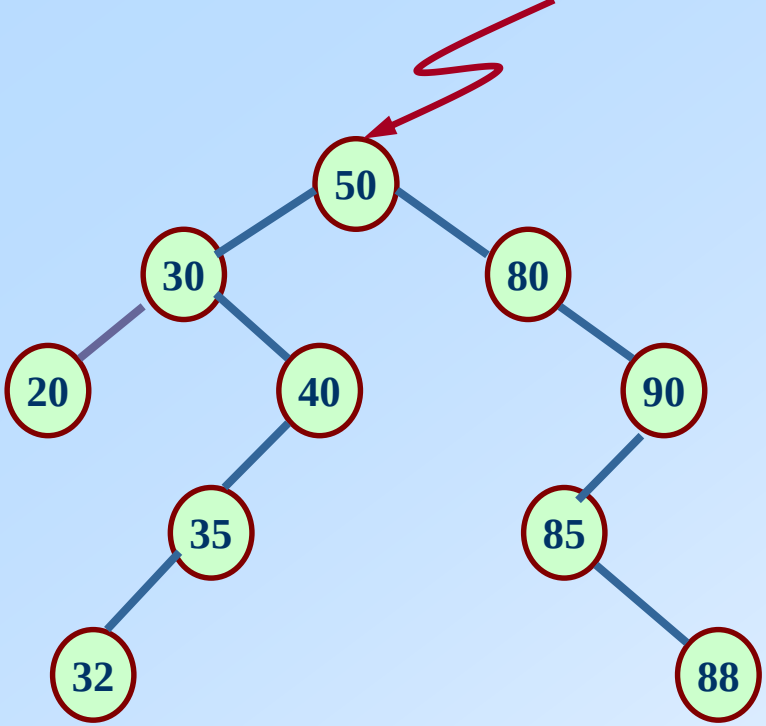
- 被删除的结点是叶子；
- 被删除的结点只有左子树或者只有右子树；
- 被删除的结点既有左子树，也有右子树。

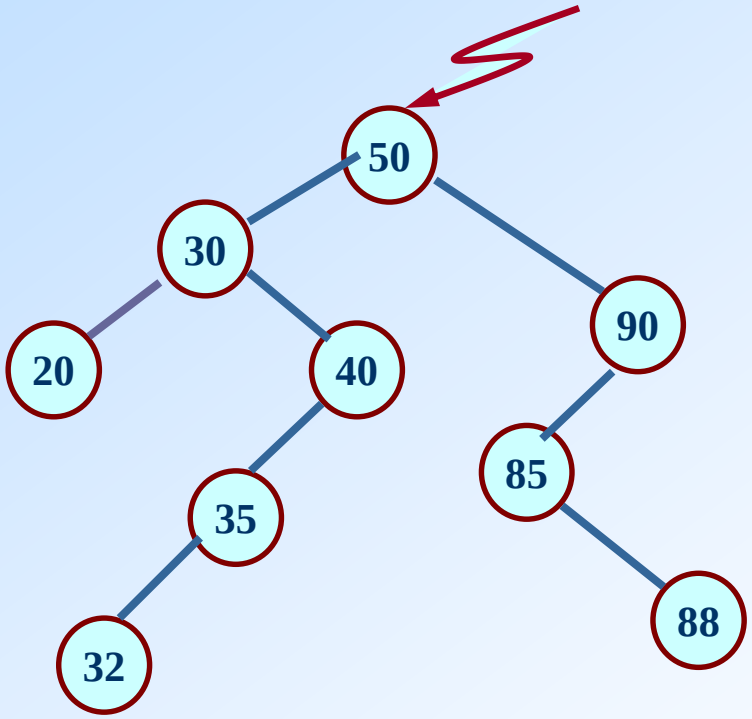



被删除的结点是叶子结点,如Key = 88

结果,其双亲结点中相应指针域的值改为空

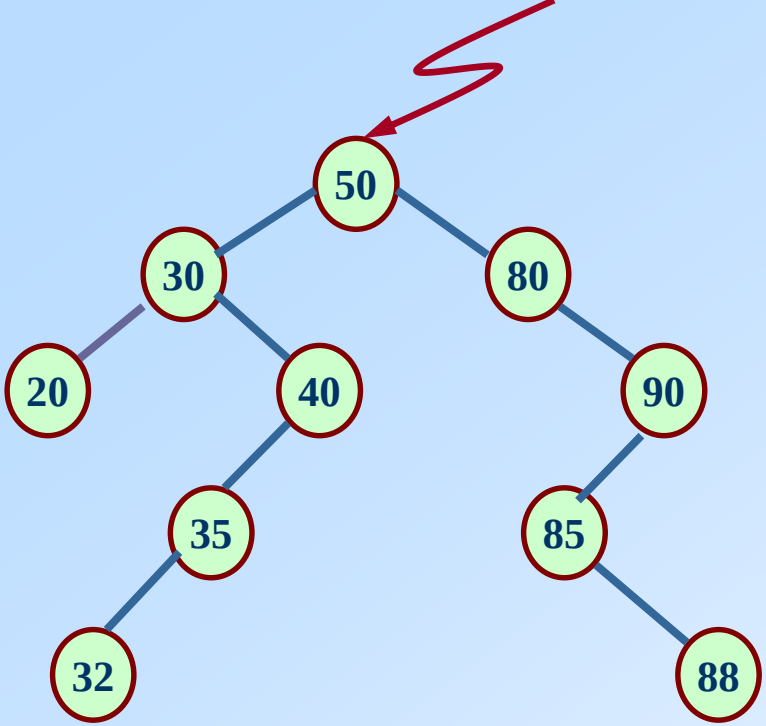


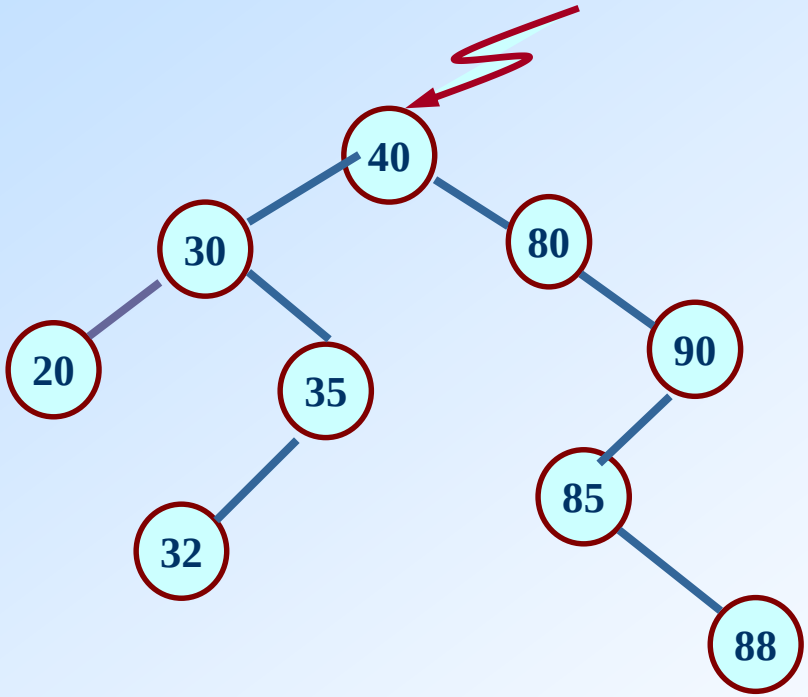




被删除的结点只有左子树或者只有右子树如

**key=80**

结果,其双亲结点的相应指针域的值改为“指向被删除结点的左子树或右子树”。




被删除的结点既有左子树，也有右子树

如被删关键字key = 50

结果,以其前驱替代之，然后再删除该前驱结点

算法

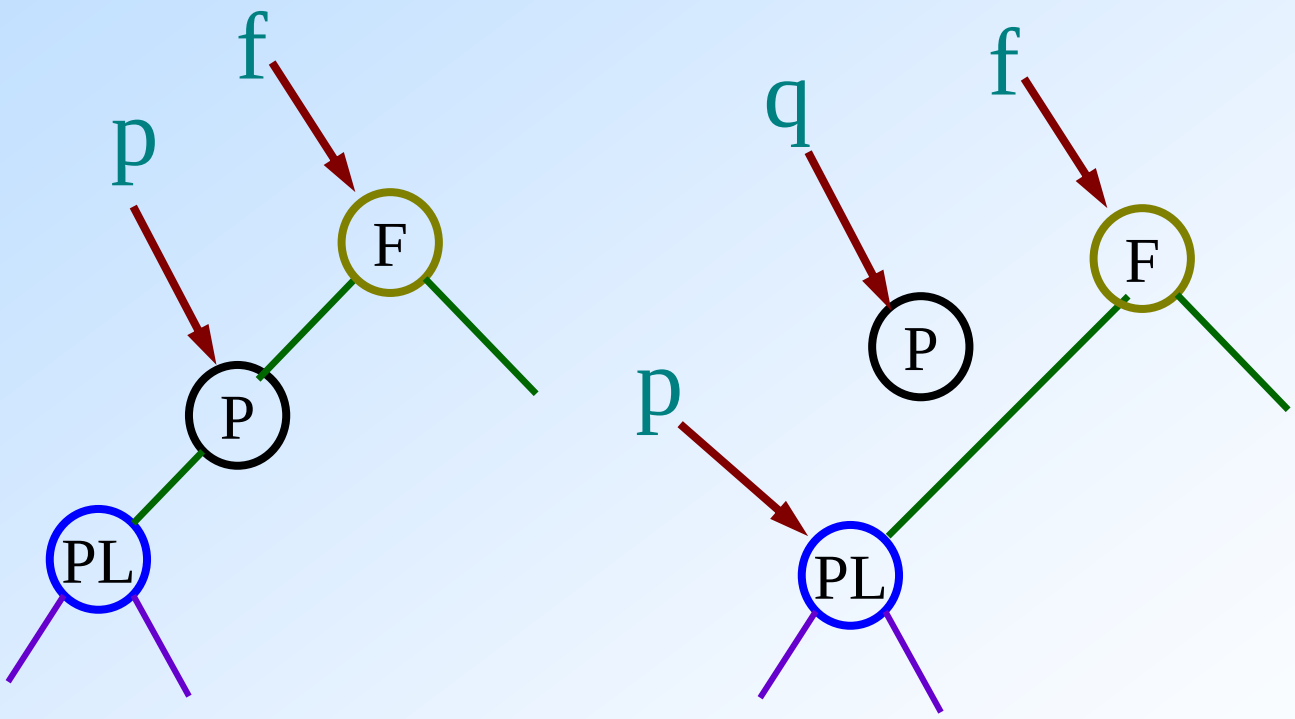
```
Status DeleteBST (BiTree &T,  KeyType kval ) {
    if (!T)  return FALSE;
          // 不存在关键字等于kval的数据元素
  else {if ( EQ (kval, T->data.key) )
{  Delete (T);   return TRUE;  }
          // 找到关键字等于key的数据元素
else if ( LT (kval, T->data.key) )
 return DeleteBST ( T->lchild, kval );
          // 继续在左子树中进行查找
else
return DeleteBST ( T->rchild, kval );
   // 继续在右子树中进行查找
 }
} // DeleteBST
```

其中删除操作过程如下：

```
void Delete ( BiTree &p ){
  // 从二叉排序树中删除结点 p，
  // 并重接它的左子树或右子树
  if (!p->rchild)  {          }
  else if (!p->lchild) {          }
  else {          }
} // Delete
```

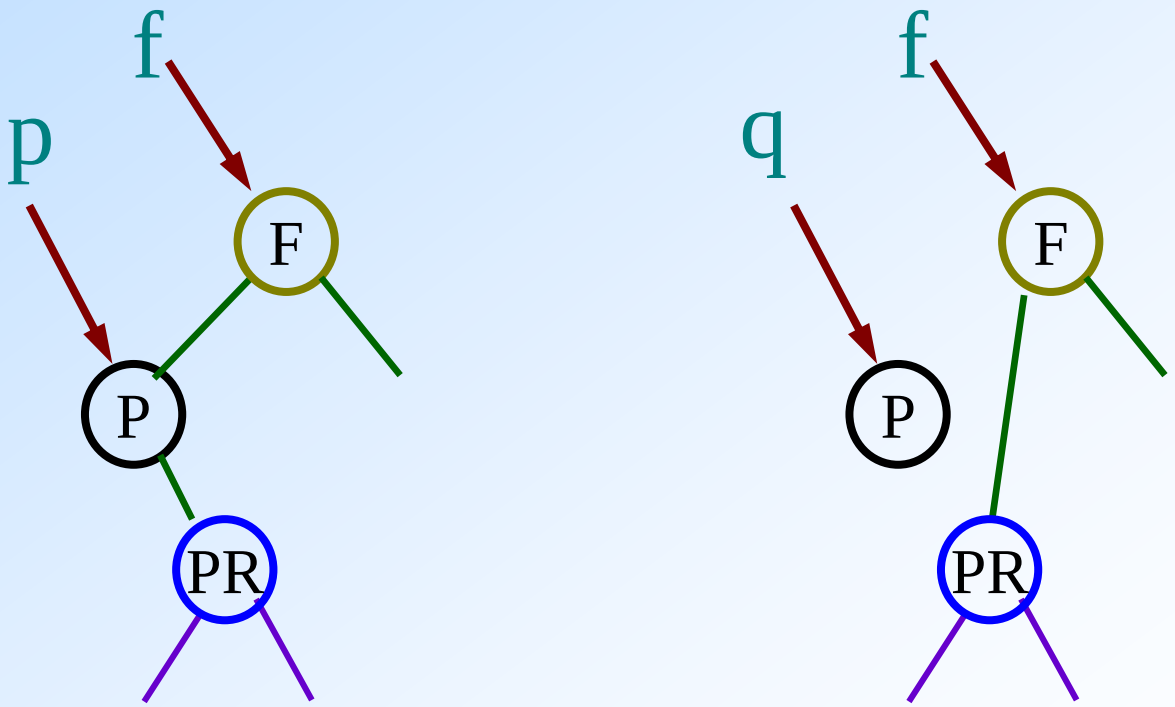
右子树为空树则只需重接它的左子树

**q = p;  f->lchild = p->lchild;  delete(q);**



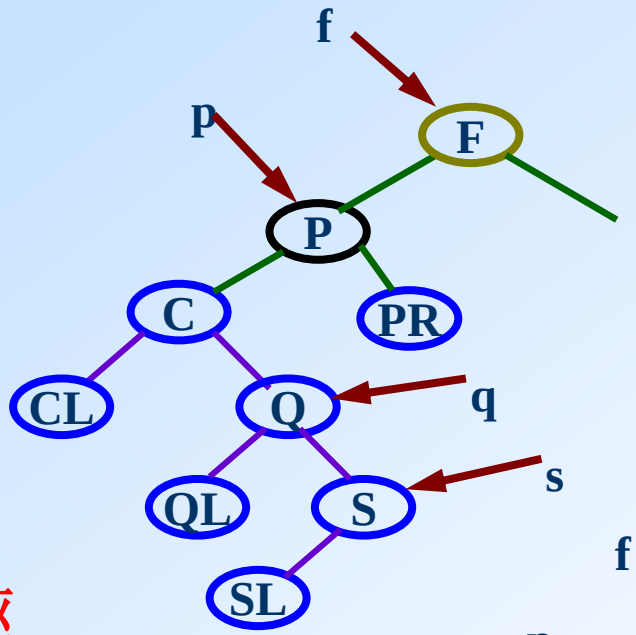
# 左子树为空树只需重接它的右子树

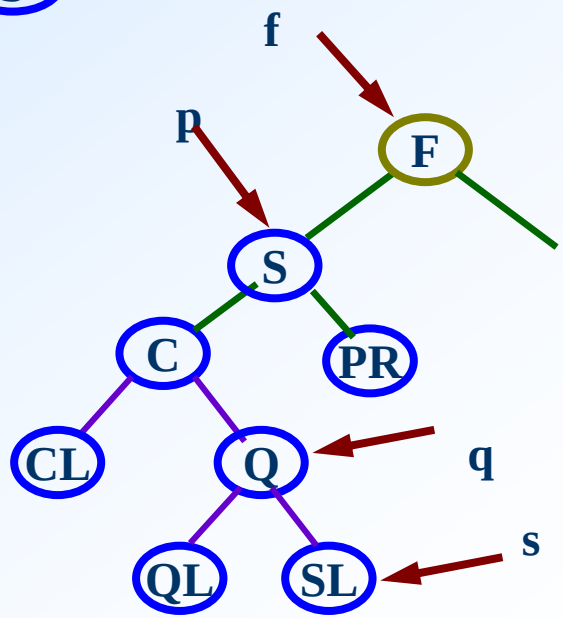
**q = p;  f->lchild = p->rchild;  delete(q);**

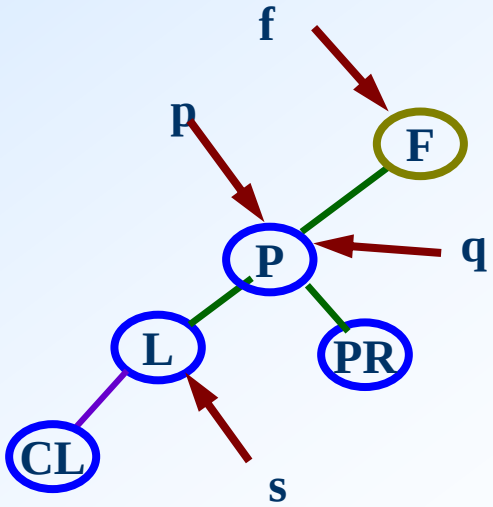


左右子树均不空

```
q = p;  s = p->lchild;
while (s->rchild)
        { q = s;  s = s->rchild; }
         // s 指向被删结点的前驱
p->data = s->data;
if (q != p )  q->rchild = s->lchild;
else  q->lchild = s->lchild;
 // 重接*q的左子树
delete(s);
```



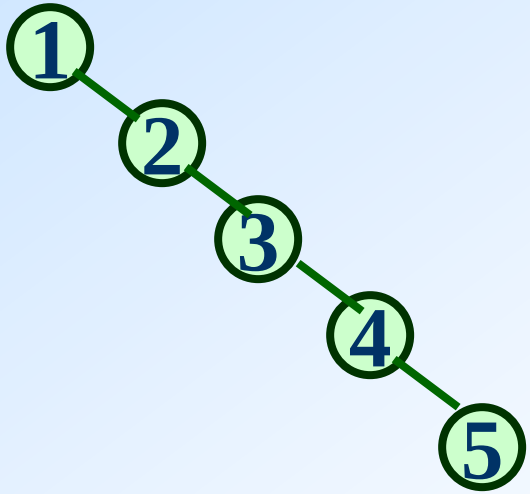



# 二叉排序树查找性能的分析

对于每一棵特定的二叉排序树，均可按照平均查找长度的定义来求它的 ASL 值，显然，由值相同的 n 个关键字，构造所得的不同形态的各棵二叉排序树的平均查找长 度的值不同，甚至可能差别很大。

例如：

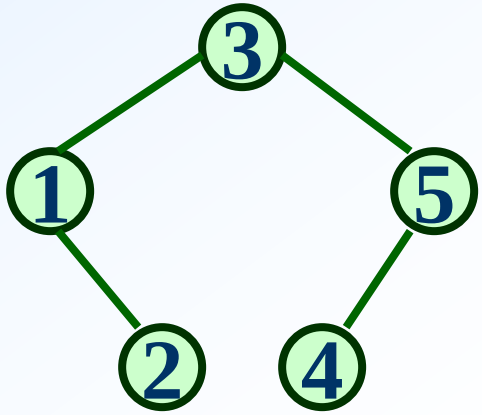
由关键字序列 1，2，3，4，5构造而得的二叉排序树，




ASL =（1+2+3+4+5）/ 5

= 3

由关键字序列 3，1，2，5，4构造而得的二叉排序树

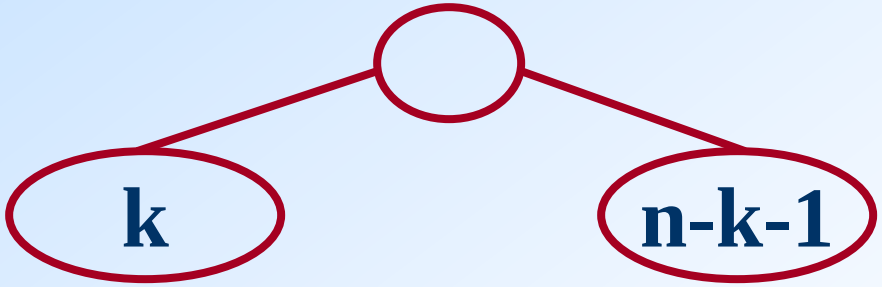



ASL =（1+2+3+2+3）/ 5

= 2.2

下面讨论平均情况:

不失一般性，假设长度为 n 的序列中有 k 个关键字小于第一个关键字，则必有 n-k-1 个关键字大于第一个关键字,由它构造的二叉排序树




的平均查找长度是 n 和 k 的函数

$$P(n, k) \quad (0 \le k \le n-1)$$

假设 n 个关键字可能出现的 n! 种排列的可能性相同，则含 n 个关键字的二叉排序树的平均查找长度

$$ASL = P(n) = \frac{1}{n}\sum_{k=0}^{n-1} P(n,k)$$

在等概率查找的情况下，

$$\boldsymbol{P(n,k) = \sum_{i=1}^{n} p_i C_i = \frac{1}{n}\sum_{i=1}^{n} C_i}$$

$$P(n,k) = \frac{1}{n}\sum_{i=1}^{n} C_i = \frac{1}{n}\left( C_{root} + \sum_{L} C_i + \sum_{R} C_i \right)$$

$$= \frac{1}{n}\left(1 + k(P(k)+1) + (n-k-1)(P(n-k-1)+1)\right)$$

$$= 1 + \frac{1}{n}\left(k \times P(k) + (n-k-1) \times P(n-k-1)\right)$$

由此

$$P(n)=\frac{1}{n}\sum_{k=0}^{n-1}\left(1+\frac{1}{n}\left(k\times P(k)+(n-k-1)\times P(n-k-1)\right)\right)$$

$$=1+\frac{2}{n^2}\sum_{k=1}^{n-1}\left(k\times P(k)\right)$$

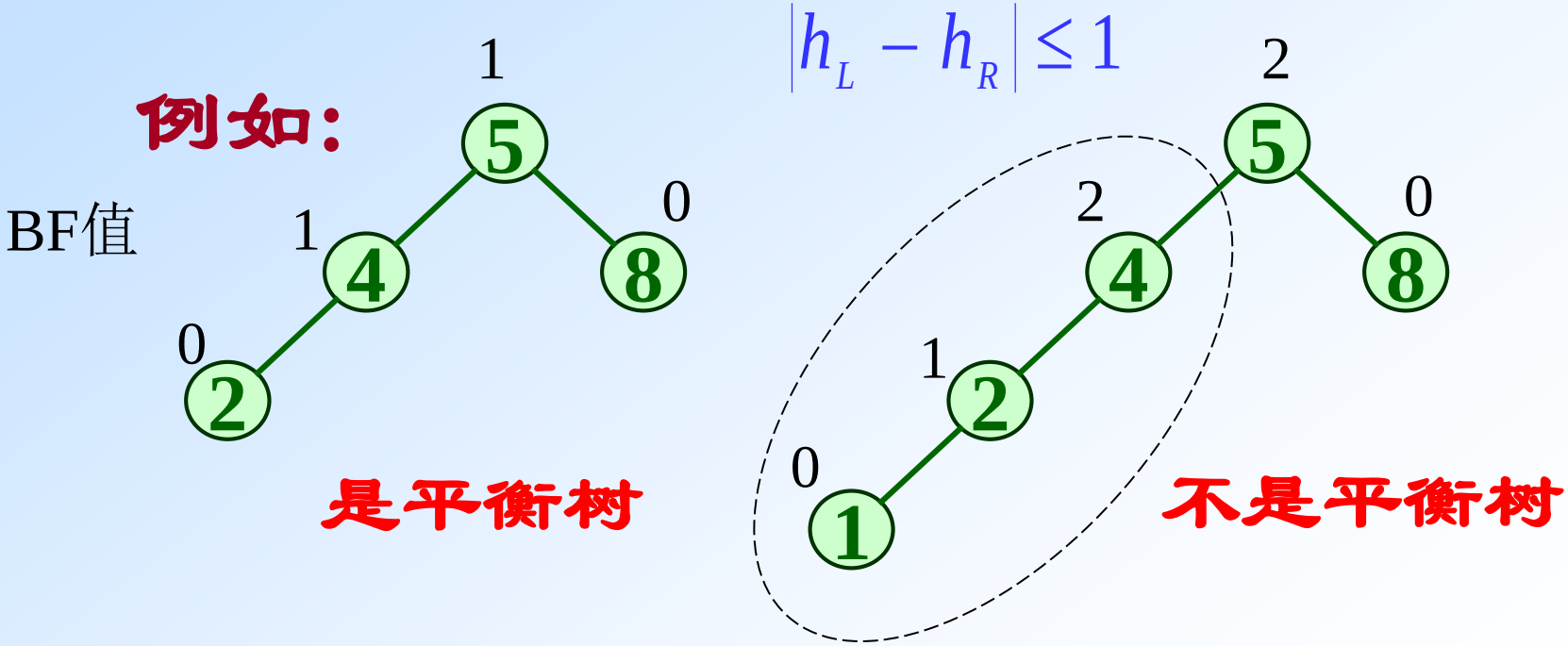
可类似于解差分方程，此递归方程有解：

$$P(n)=2\frac{n+1}{n}\log n+C$$

# 平衡二叉树

二叉平衡树是二叉查找树的另一种形式，其特点为：树中每个结点的左、右子树深度之差（平衡因子BF）的绝对值不大于1

$$|h_L - h_R| \le 1$$

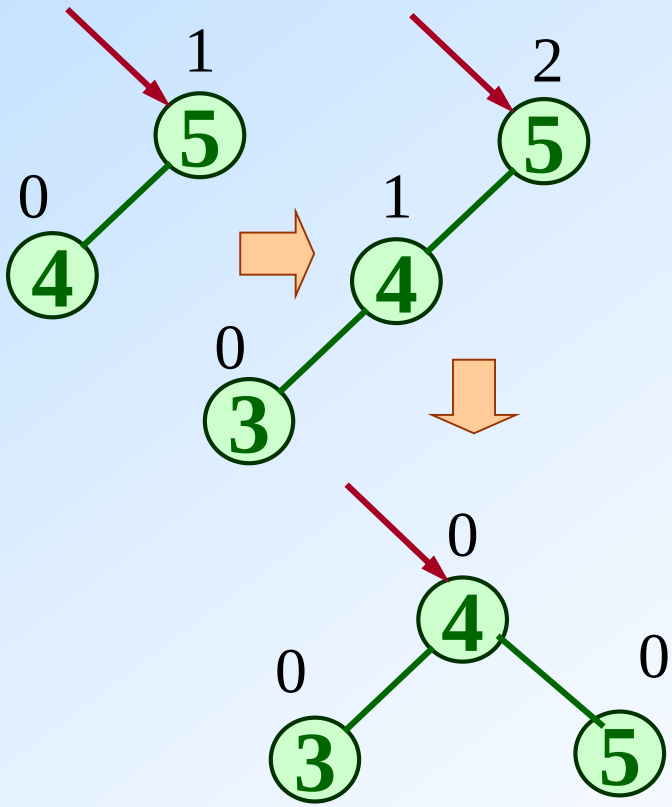


# ❖ 构造二叉平衡（查找）树的方法：

在插入过程中，采用平衡旋转技术。

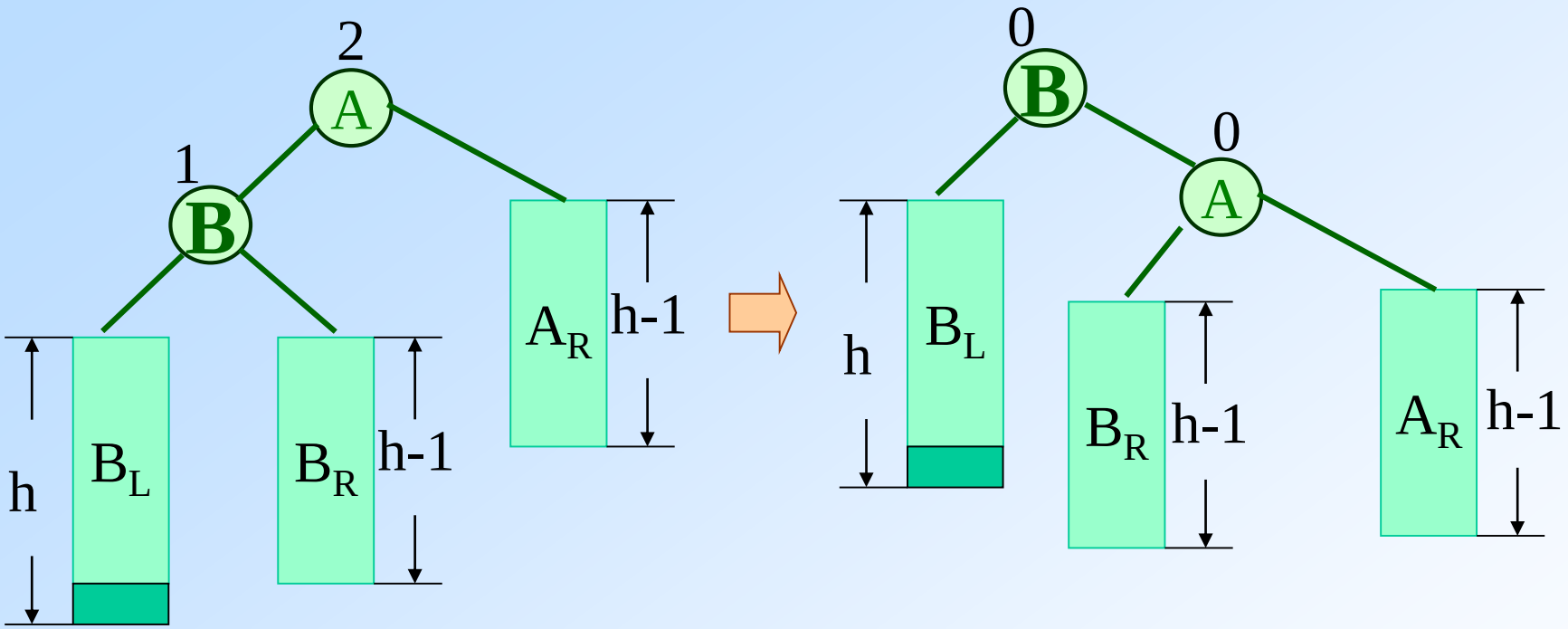
例如:依次插入的关键字为5, 4, 3, 8, 19,1,2，25，23

BF值

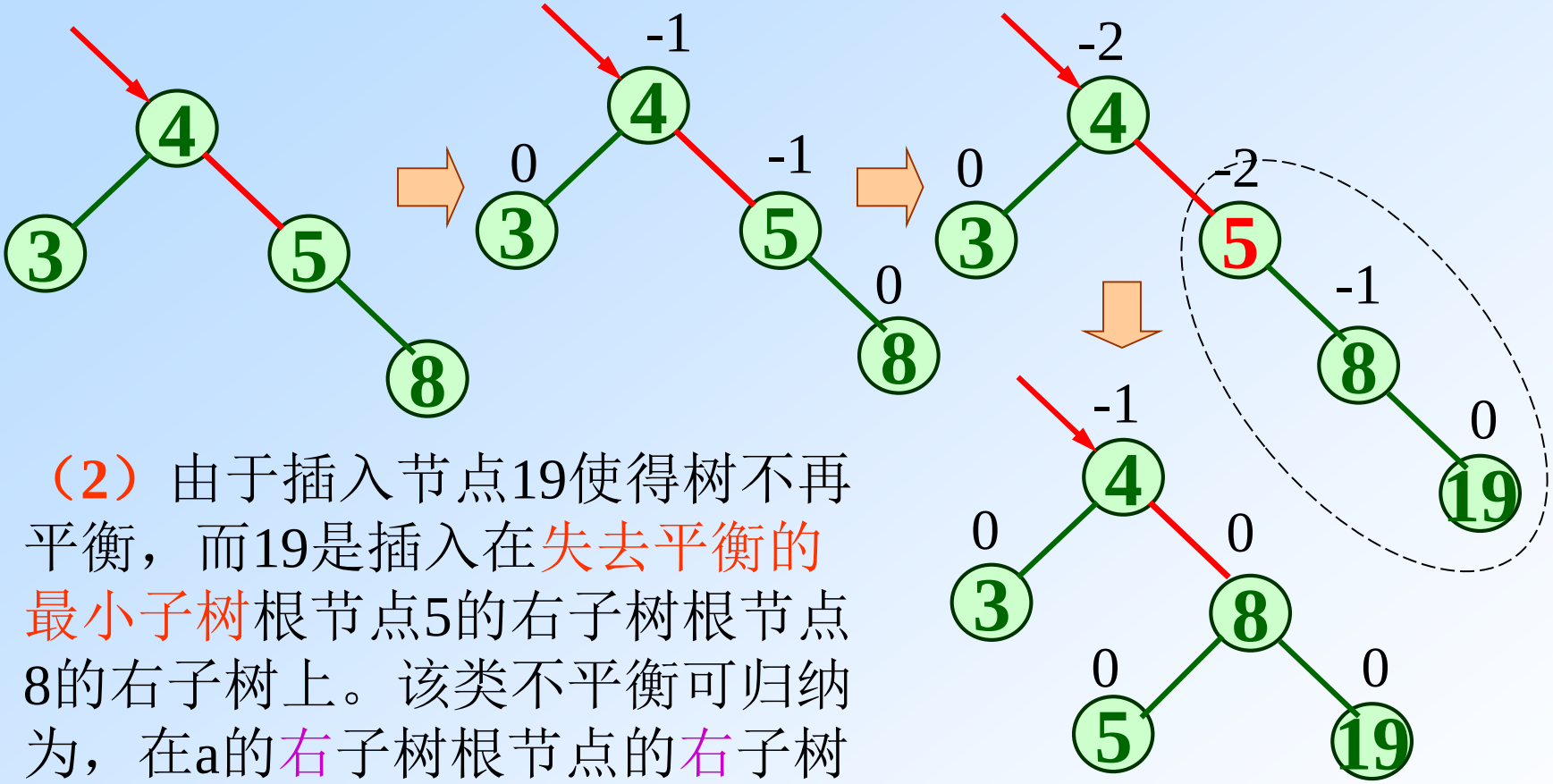



（1）由于插入节点3使得树不再平衡，而3是插入在失去平衡的最小子树根节点5的左子树根节点4的左子树上。定义失去平衡的最小子树根节点为a，则该类不平衡可归纳为，在a的左子树根节点的左子树上插入导致的不平衡可使用单向右旋平衡处理，可以记为左左->右。

可归纳为LL型

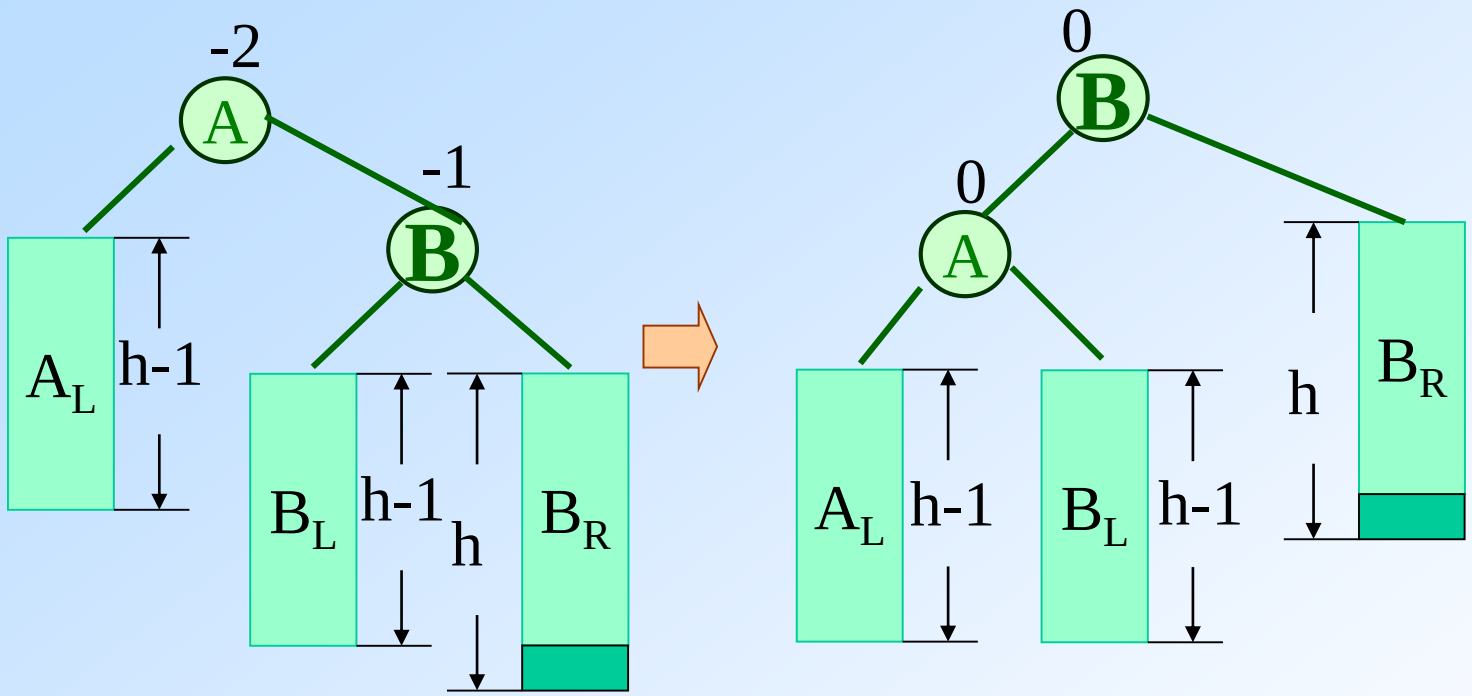



在单向右旋平衡处理后BF(B)由1变为0，BF(A)由2变为0

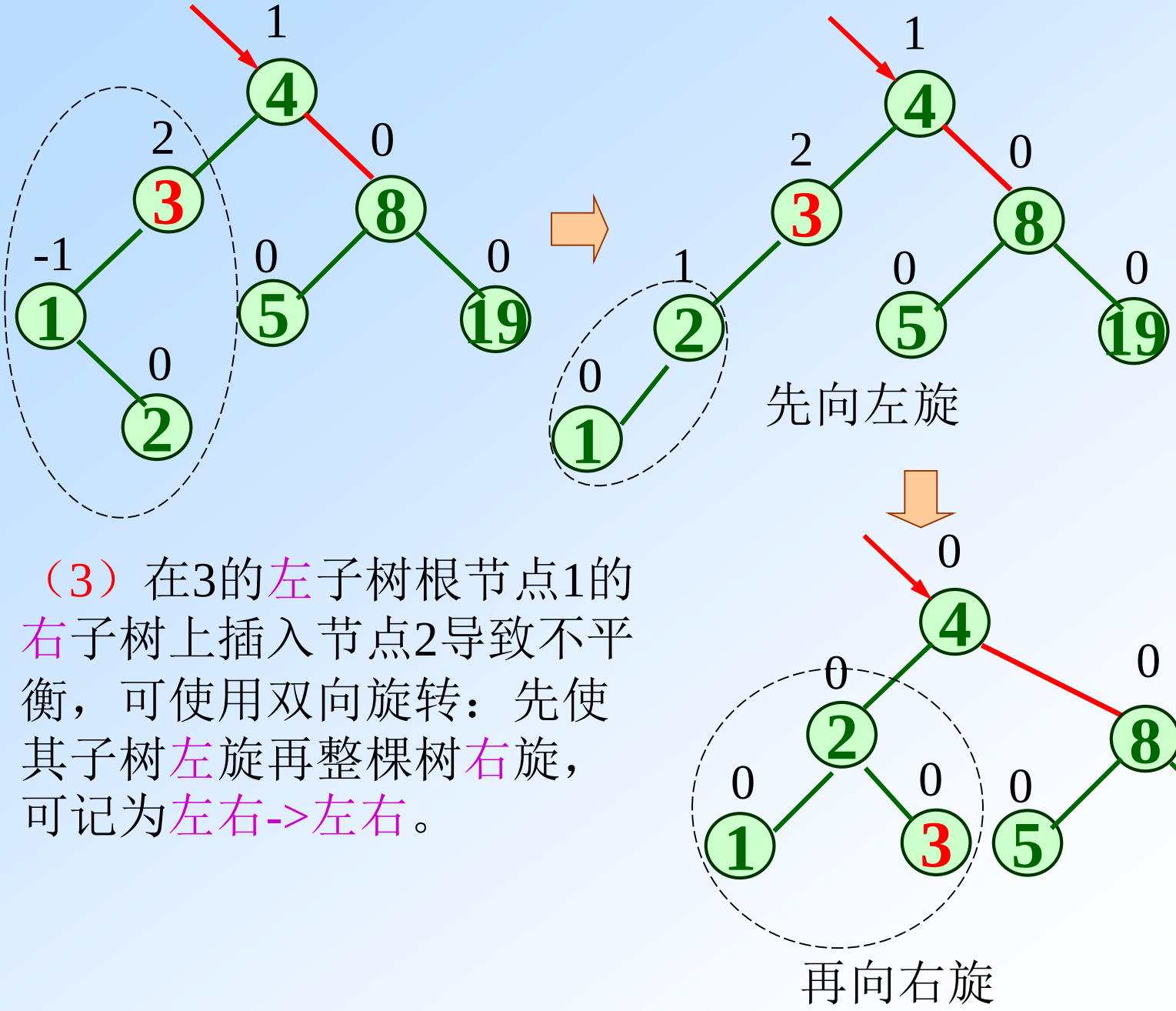



（2）由于插入节点19使得树不再平衡，而19是插入在失去平衡的最小子树根节点5的右子树根节点8的右子树上。该类不平衡可归纳为，在a的右子树根节点的右子树上插入导致的不平衡可使用单向左旋平衡处理，可以记为右右->左。

可归纳为RR型

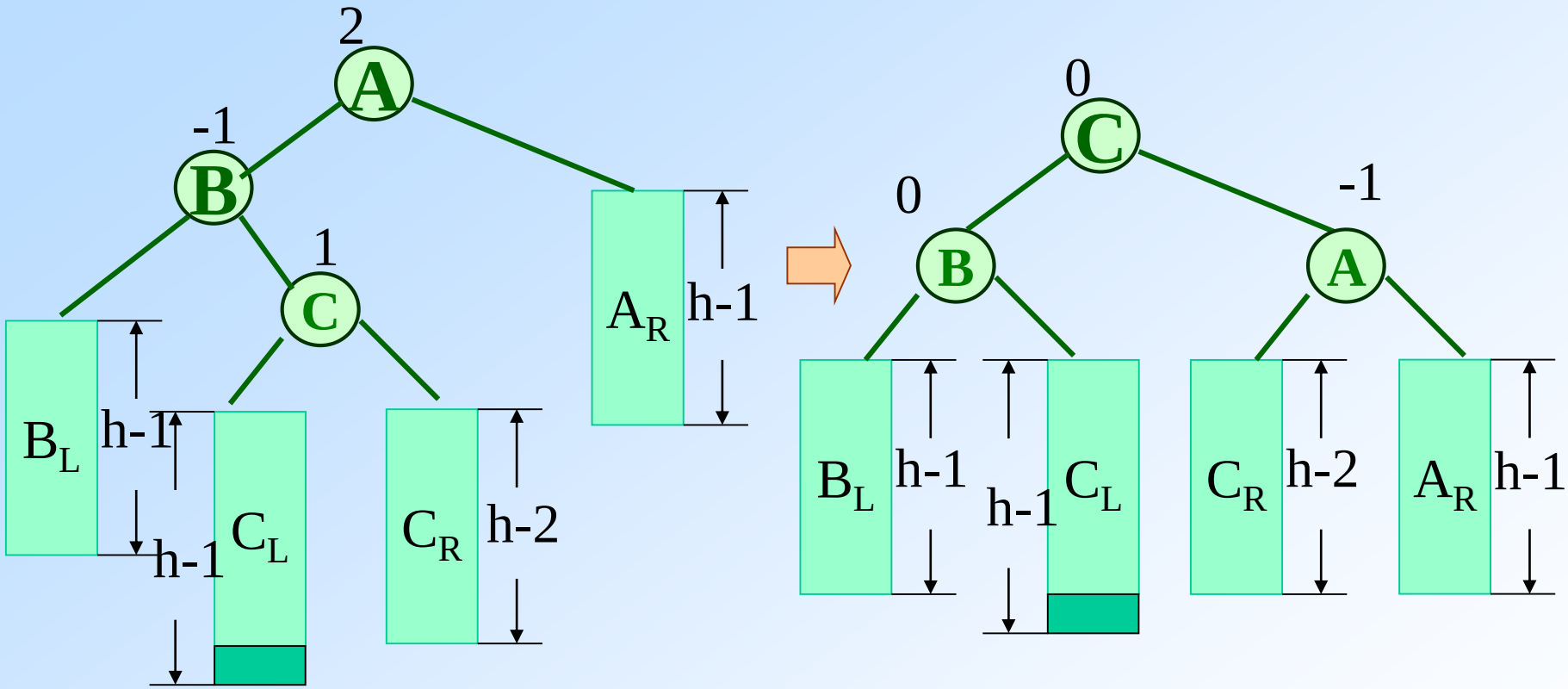



在单向左旋平衡处理后BF(B)由-1变为0，BF(A)由-2变为0



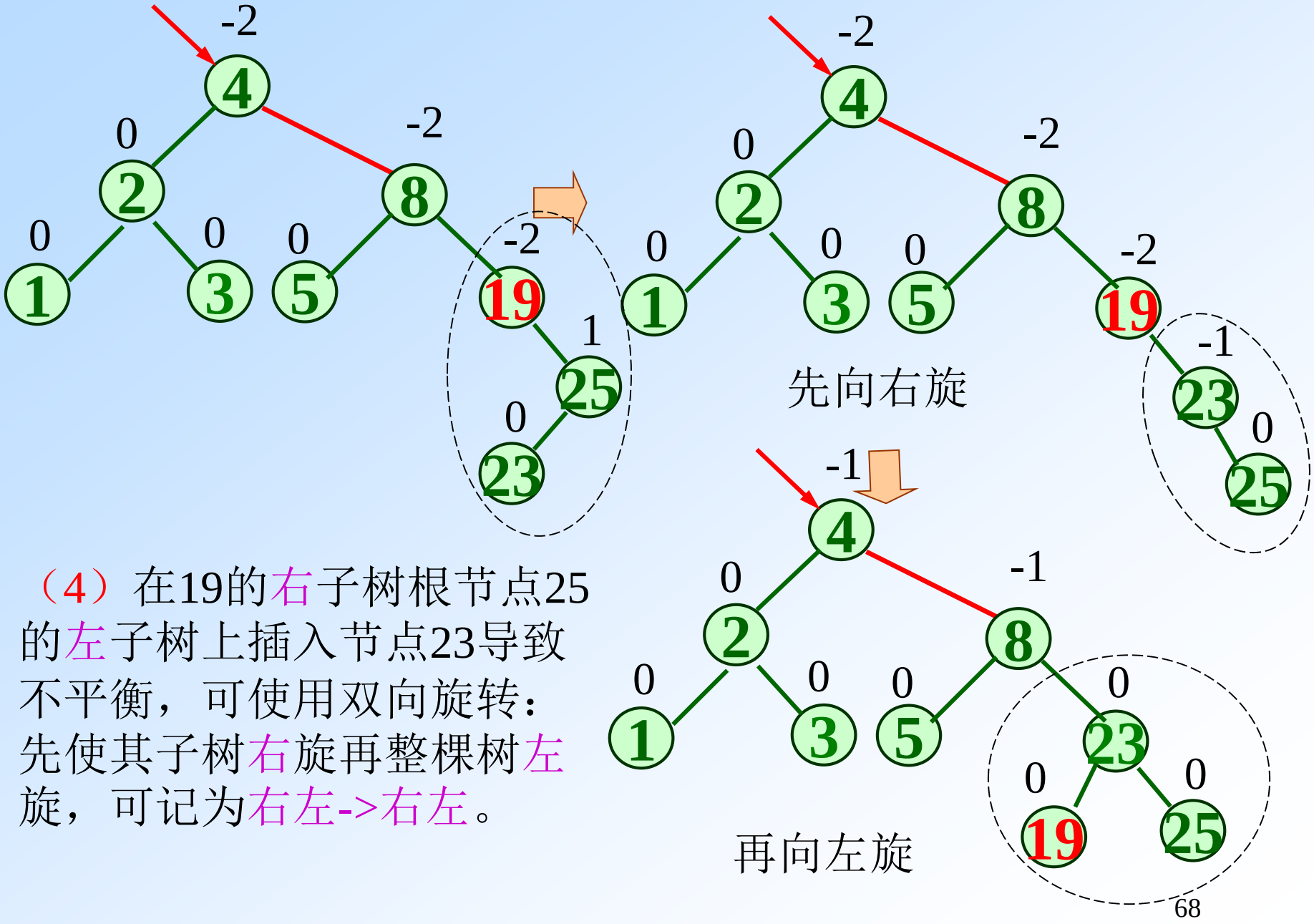

（3）在3的左子树根节点1的右子树上插入节点2导致不平衡，可使用双向旋转：先使其子树左旋再整棵树右旋，可记为左右->左右。

可归纳为LR型



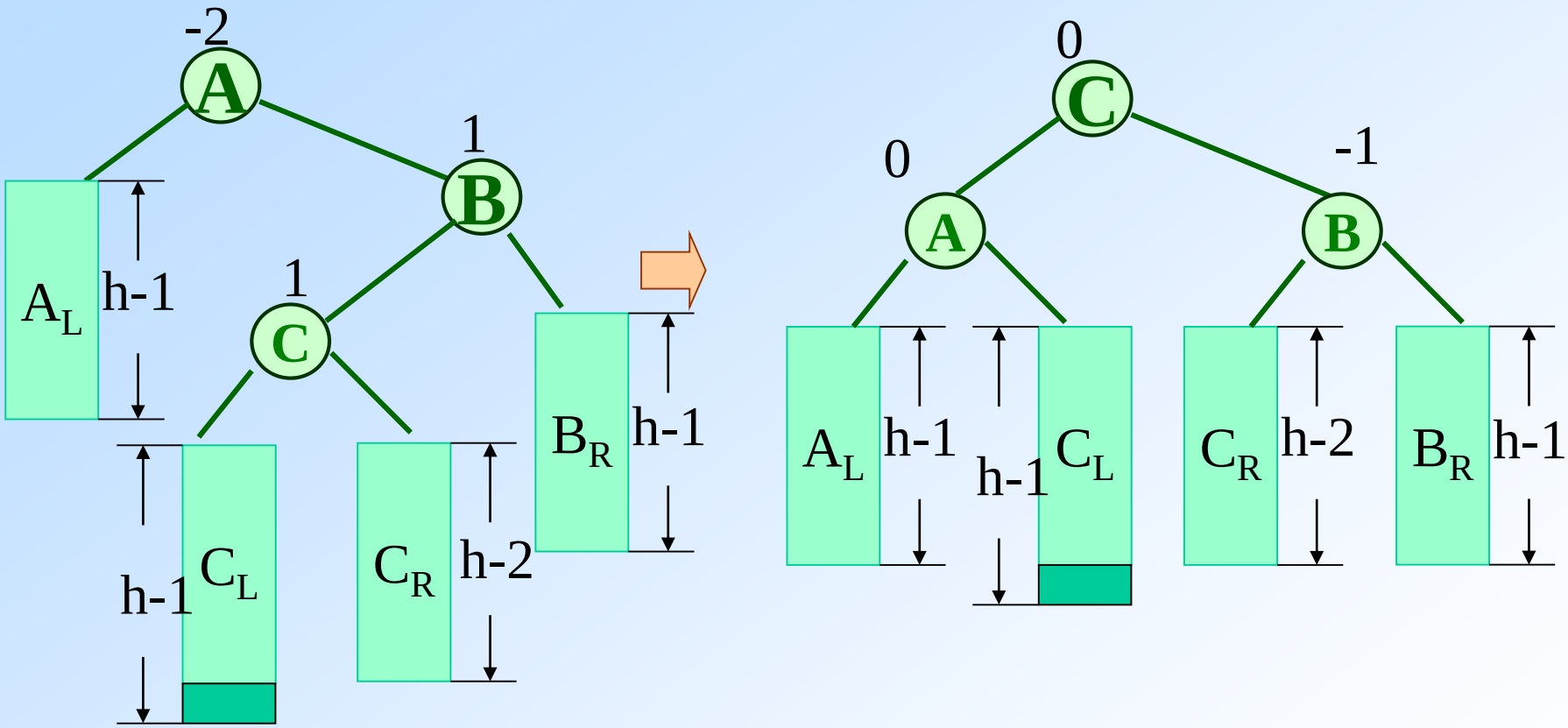

在双向旋转平衡处理后BF(A)由2变为-1，BF(B)由-1变为0

BF(C)由1变为0




（4）在19的右子树根节点25的左子树上插入节点23导致不平衡，可使用双向旋转：先使其子树右旋再整棵树左旋，可记为右左->右左。

可归纳为RL型




在双向旋转平衡处理后BF(A)由-2变为0，BF(B)由1变为-1

BF(C)由1变为0

# 旋转操作特点

1. 对不平衡的最小子树操作
2. 旋转后子树根节点平衡因子为0
3. 旋转后子树深度不变故不影响全树，也不影响插入路径上所有祖先结点的平衡度

# 平衡二叉树插入算法

1. 若是空树，插入节点作为根节点，树深度加1
2. 插入节点key值等于根节点key值，则不插入
3. 插入节点key值小于根节点key值，插入在左子树上，如果左子树深加1并且：
   - ① 若根节点平衡因子为-1，则改为0，树深不变
   - ② 若根节点平衡因子为0，则改为1，树深加1
   - ③ 若根节点平衡因子为1,且其左子的平衡因子为1（左左），则单向右旋，旋转后根节点和其右子的平衡因子改为0，树深不变

④ 若根节点平衡因子为1,且其左子的平衡因子为-1（左右），则先左旋再右旋，旋转后根节点和其左子的平衡因子改为0，右子的平衡因子改为-1,树深不变

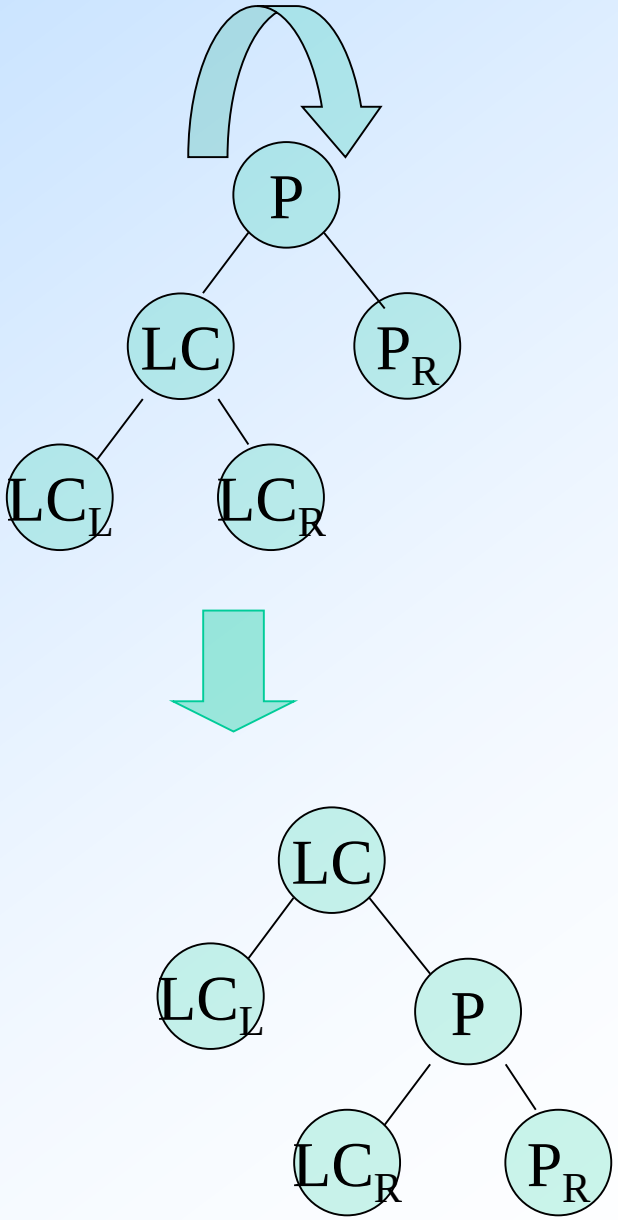
4. 插入节点key值大于根节点key值，插入在右子树上，方法类似第3步

# 二叉树结构：

```
Typedef struct BSTNode{
   ElemType       data;
   int                  bf;    //平衡因子
   struct BSTNode *lchild,*rchild;
}BSTNode,*BSTree;
```

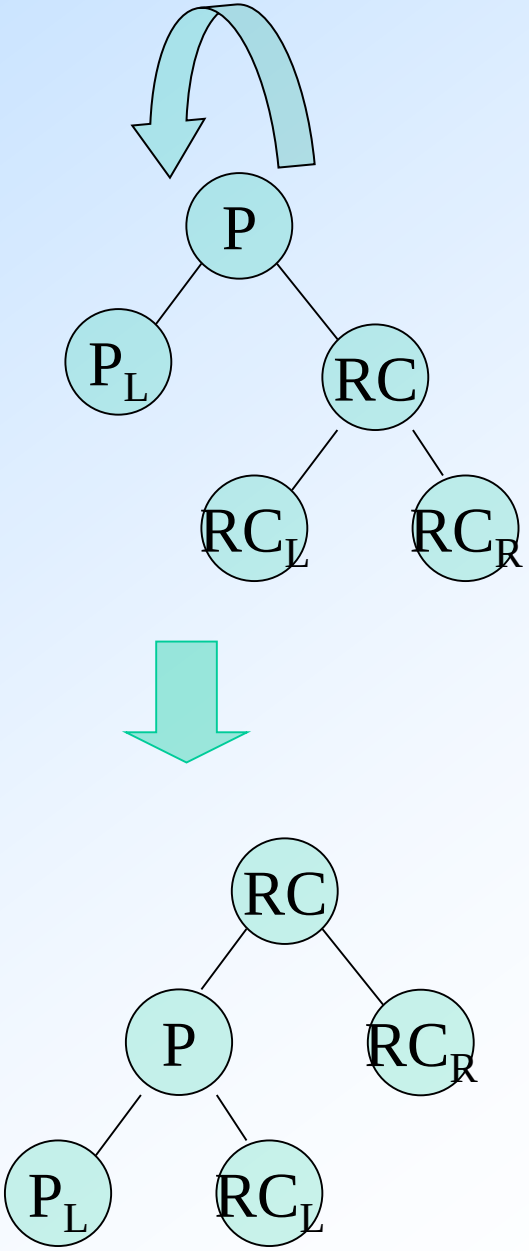
# 右旋：

```
Void R_Rotate(BSTree
   &p){
   lc=p->lchild;
   p->lchild=lc->rchild;
   lc->rchild=p;
   p=lc;
}//R_Rotate
```

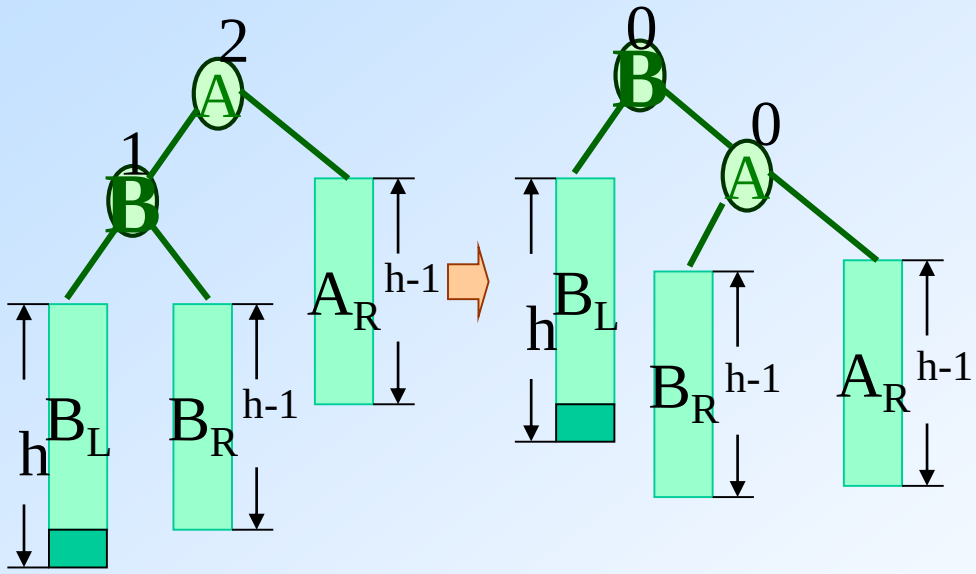


# 左旋:

```
void L_Rotate(BSTree
   &p){
   rc=p->rchild;
   p->rchild=rc->lchild;
   rc->lchild=p;
   p=rc;
}//L_Rotate
```

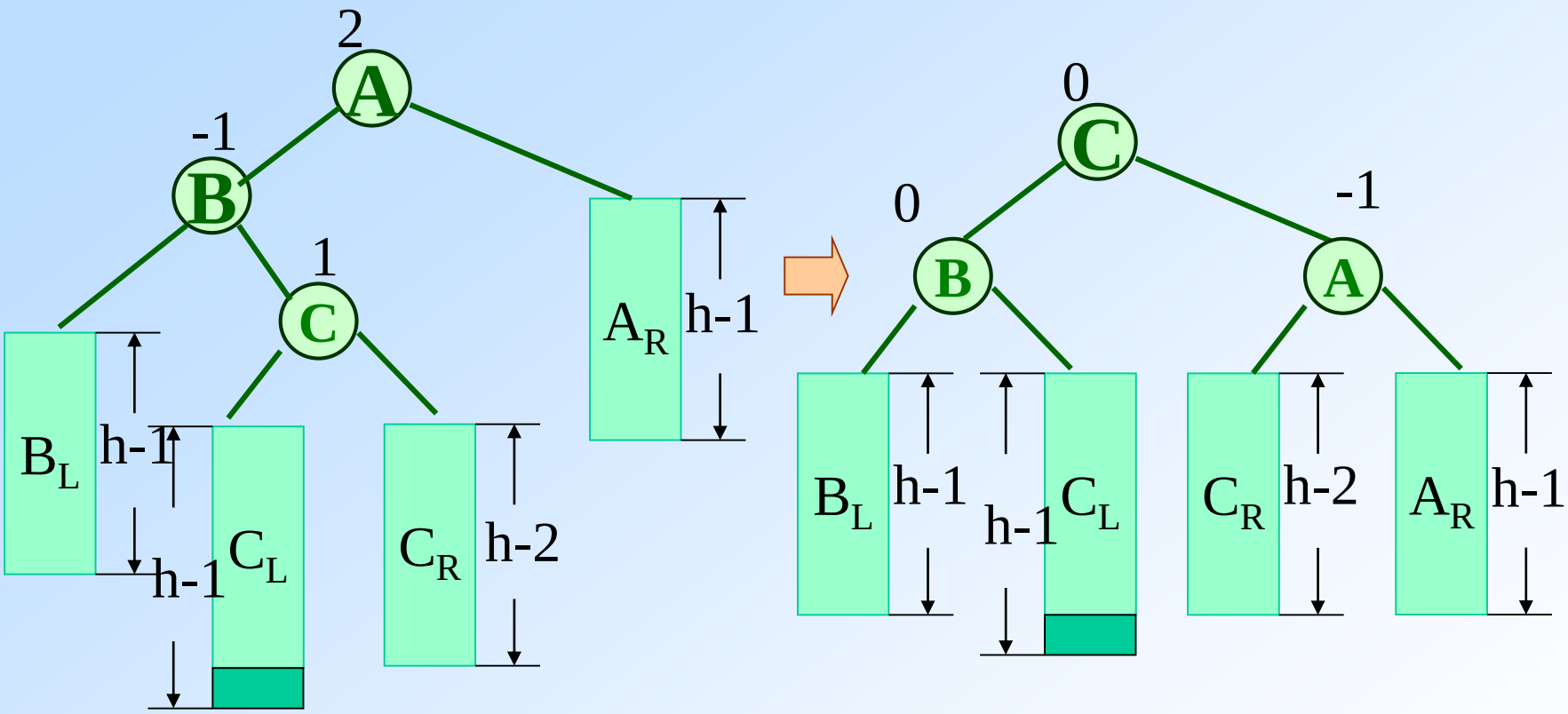


# 左平衡旋转：

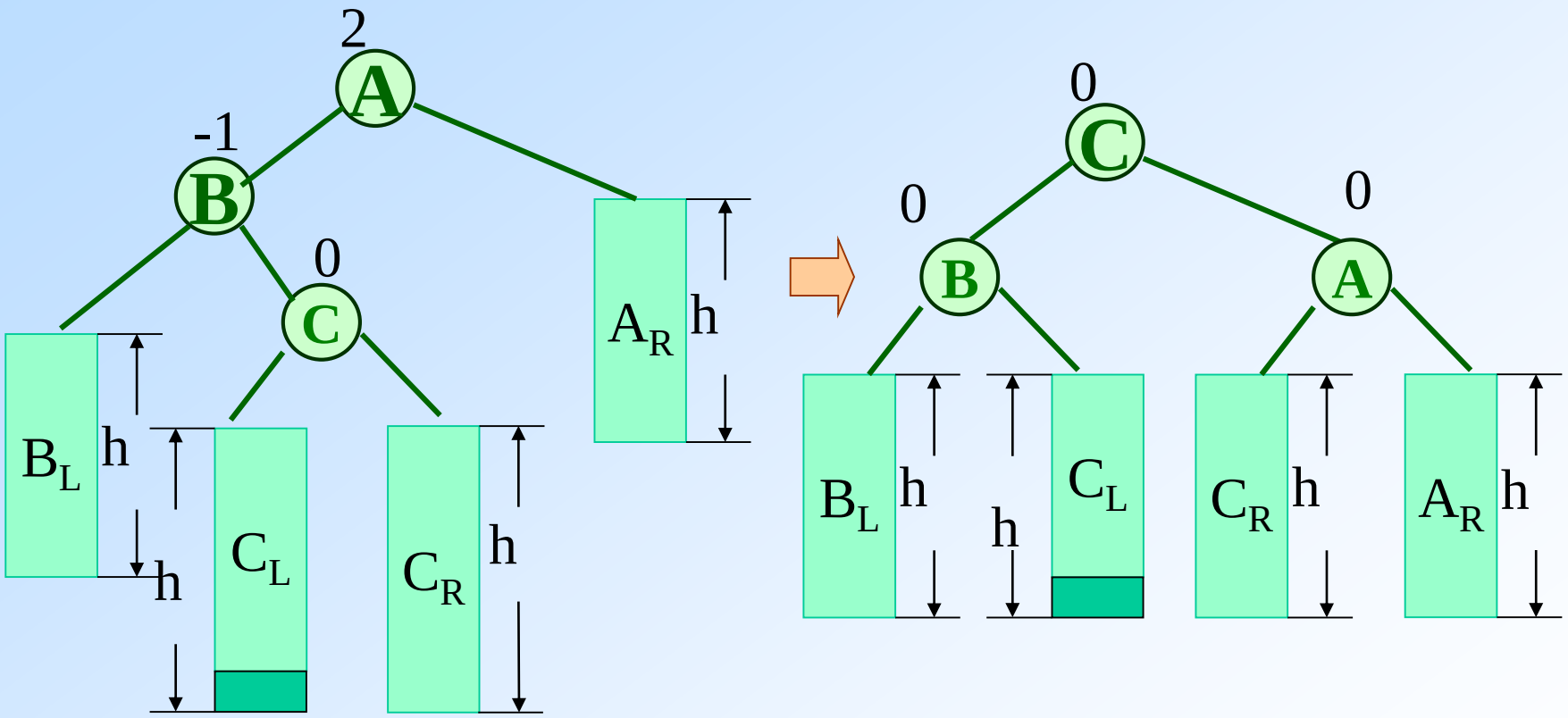



```
#define LH +1
#define EH 0
#define RH -1
Void LeftBalance (BSTree &T) {//T对应图中的A
    lc=T->lchild;//看T的左子(B)，只能是LL或LR型
    switch (lc->bf){
          case LH:        //左左->右旋（LL型）
                    T->bf=lc->bf=EH;
                    R_Rotate(T);break;
```

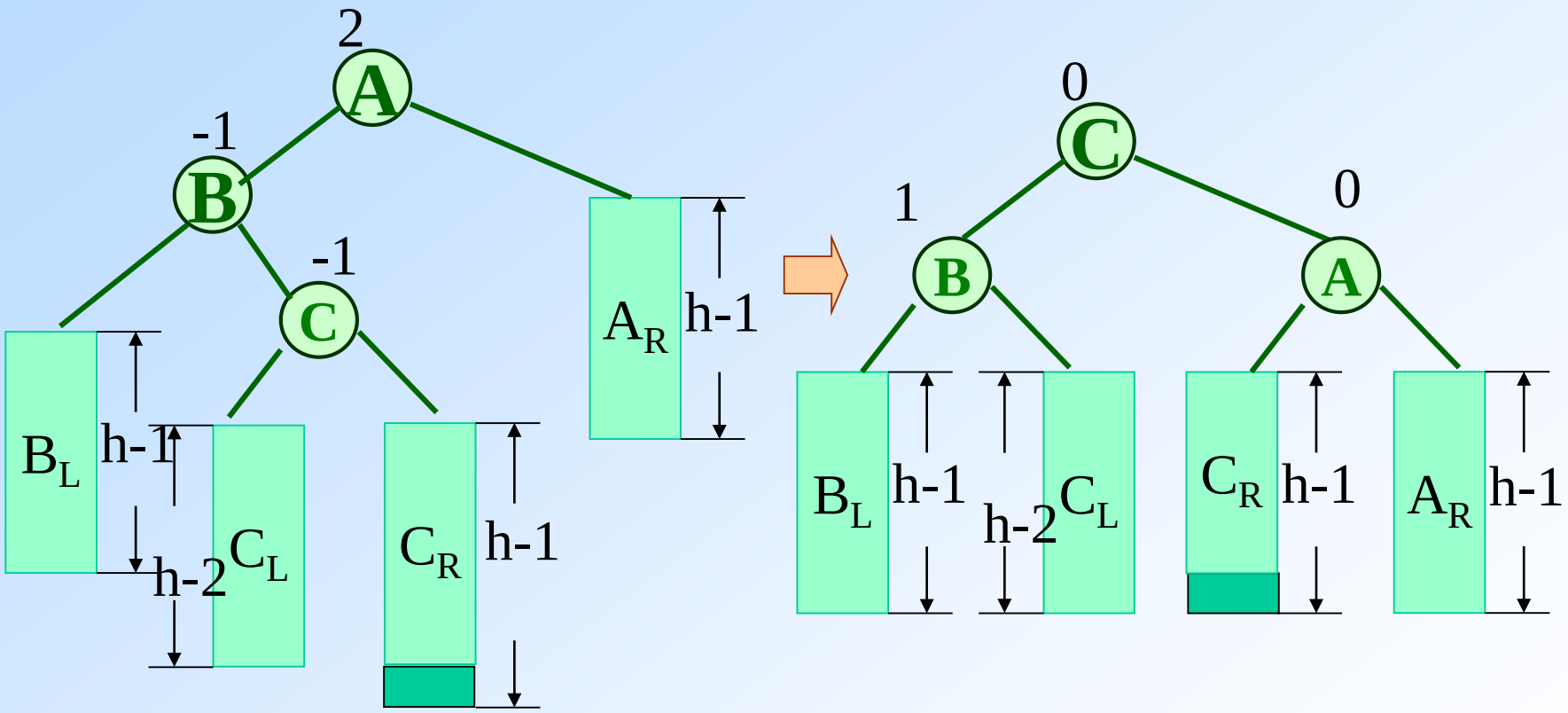
```
        case RH://LR型，先左旋后右旋
                rd=lc->rchild;//节点C
                switch(rd->bf){
                        case LH:T->bf=RH;//见图1
                                lc->bf=EH;break;
                        case EH:T->bf=lc->bf=EH;//见图2
                                break;
                        case RH:T->bf=EH;//见图3
                                lc->bf=LH;break;
                }//switch(rd->bf)
                rd->bf=EH;
                L_Rotate(T->lchild);
                R_Rotate(T);
    }//switch(lc->bf)
}//LeftBalance
```




假设插入点在C的左子树上　　图1




假设插入点在C的左子树上　　图2




假设插入点在C的右子树上　　图3

# 插入节点（主程序）：

```
Status InsertAVL(BSTree &T,ElemType e,Boolean
    &taller){//taller表示树是否长高，布尔型
    if(!T){//如果树不存在则创建
        T=(BSTree)malloc(sizeof(BSTNode));
        T->data=e;T->lchild=T->rchild=NULL;
        T->bf=EH;taller=TRUE;
    }
    else{
```

```
if EQ(e.key,T->data.key)  //找到，不用插入节点
   {taller=FALSE;return=0;}
if LT(e.key,T->data.key){//要插入在左子上
   if(!InsertAVL(T->lchild,e,taller)) return 0;
   //递归调用直至在某个分枝上插入节点作为叶子节点
   if(taller)
   //层层返回，在返回过程中判断是否要旋转来保持平衡
   //并且这样也保证了旋转失衡的最小子树
          switch(T->bf){//检查T的平衡度
             case LH:
                    LeftBalance(T);taller=FALSE;break;
             case EH:
                    T->bf=LH;taller=TRUE;break;
             case RH:
                    T->bf=EH;taller=FALSE;break;
          }//switch(T->bf)
}//if
else{
```

```
            if(!InsertAVL(T->rchild,e,taller)) return 0;
            if(taller)
                    switch(T->bf){
                       case LH;
                              T->bf=EH;taller=FALSE;break;
                       case EH:
                              T->bf=RH;taller=TRUE;break;
                       case RH:
                              RightBalance(T);taller=FALSE;break;
                              //右平衡函数没有给出
                    }//switch(T->bf)
          }//else
    }//else
    return 1;
}//InsertAVL
```

# 平衡树查找的性能

- 查找的时间复杂度为O(logn)
- 分清二叉排序树、平衡二叉树、最优查找树、次优查找树的区别：都是二叉树，前两者是动态查找树，后两者是静态查找树，四种都是二叉树，查找方法也一样。

# 练习

- 12,4,1,7,8,10,9,2,11,6,5

# B_树

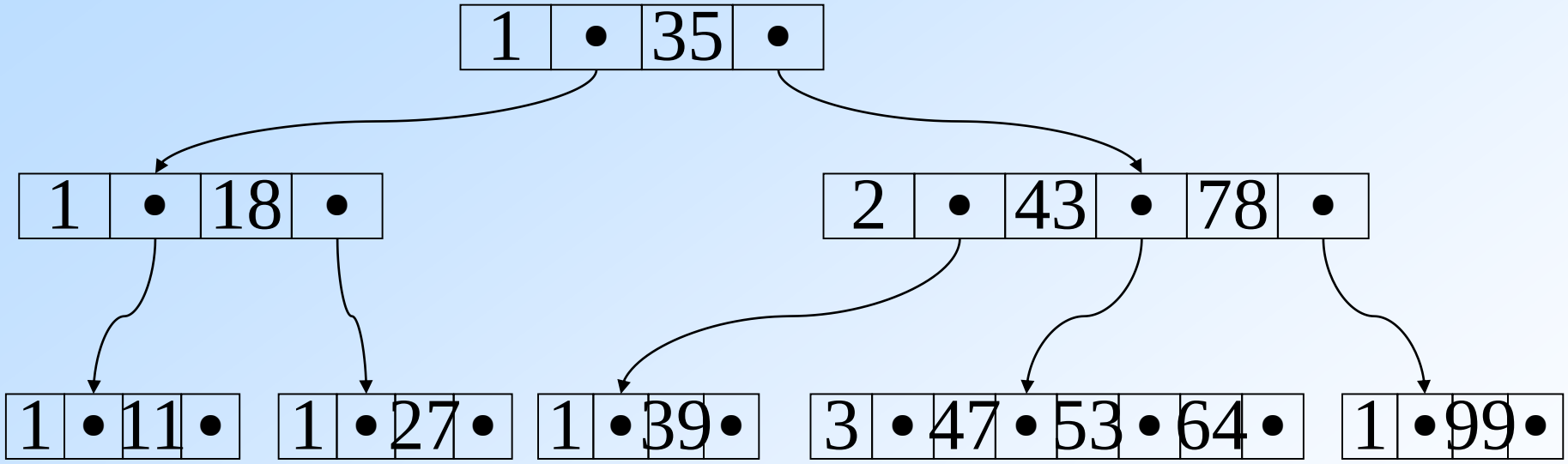
- B_树：一种平衡的多路查找树

一棵m阶的B_树或为空树，或为满足下列特性的m叉树：

1. 树中每个节点至多有m棵子树
2. 若根结点不是叶子结点，则至少有两棵子树
3. 除根结点之外的所有非终端结点至少有⌈m/2⌉棵子树
4. 所有的非终端结点中包含下列信息数据$(n,A_0,K_1,A_1,K_2,A_2,\ldots,K_n,A_n)$其中$K_i(i=1,\ldots,n)$为关键字，且$K_i<K_{i+1},(i=1,\ldots,n-1)$

$A_i(i=0,\ldots,n)$为指向子树根节点的指针，且指针$A_{i-1}$所指子树中所有节点的关键字均小于$K_i(i=1,\ldots,n)$,$A_n$所指子树中所有节点的关键字均大于$K_n$,$n(\lceil m/2\rceil-1)\leq n\leq m-1)$为关键字的个数（或n+1为子树个数）。

5. 所有的叶子结点都出现在同一层次上，并且不带信息（空指针）。




4阶B_树，每个节点最多4个指针3个关键字

一棵m阶B-树每个节点最多有m棵子树m-1个关键字,最少有⌈m/2⌉棵子树⌈m/2⌉-1个关键字

# B-树结构：

```
#define m 3
Typedef struct BTNode{
    int             keynum;
    struct BTNode *parent;
    KeyType         key[m+1];
    struct BTNode *ptr[m+1];
    Record          *recptr[m+1];//记录
}BTNode,*BTree;
Typedef struct{
    BTNode          *pt;
    int             i;//在节点中的关键字序号
    int             tag;//返回查找是否成功(0或1)
}Result;//查找结果
```

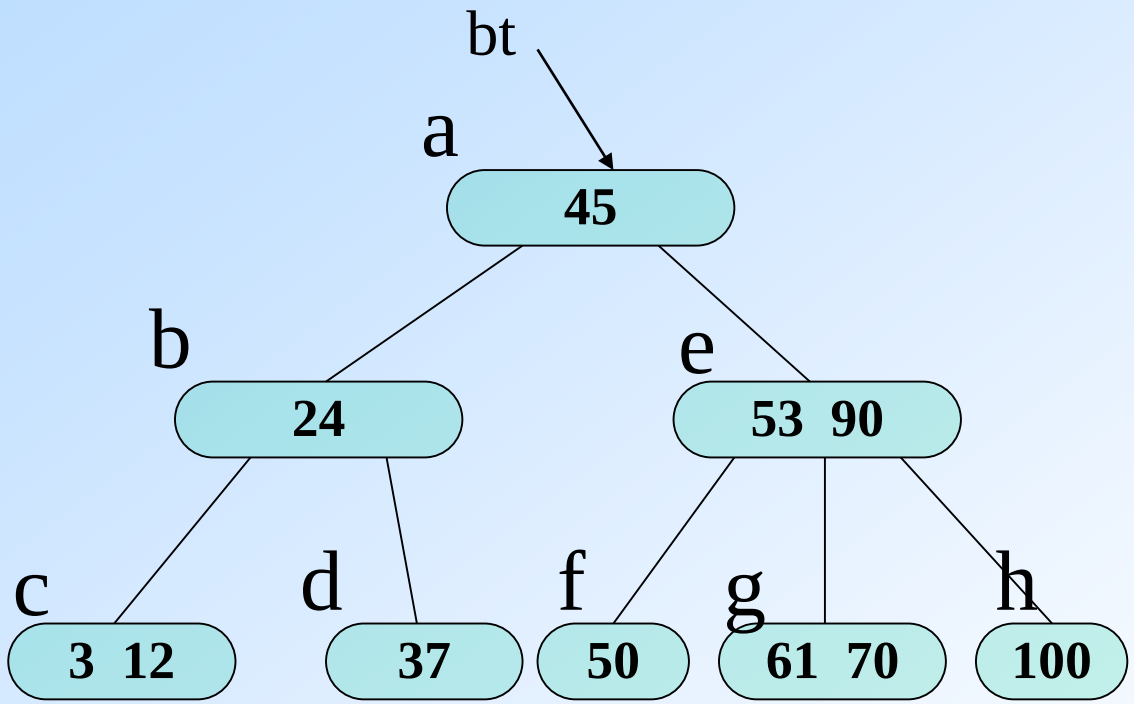
# B-树查找操作：

```
Result SearchBTree(Btree T,KeyType K){
    p=T;q=NULL;found=FALSE;i=0;
    while(p&&!found){
          n=p->keynum;i=Search(p,K);//在p->key[1..keynum]中查找i使
    得：p->key[i]<=K<p->key[i+1]
          if(i>0&&p->key[i]==K)found=TRUE;
          else{q=p;p=p->ptr[i];}
    }
    if(found)return(p,i,1);
    else return(q,i,0);
}//SearchBTree
```

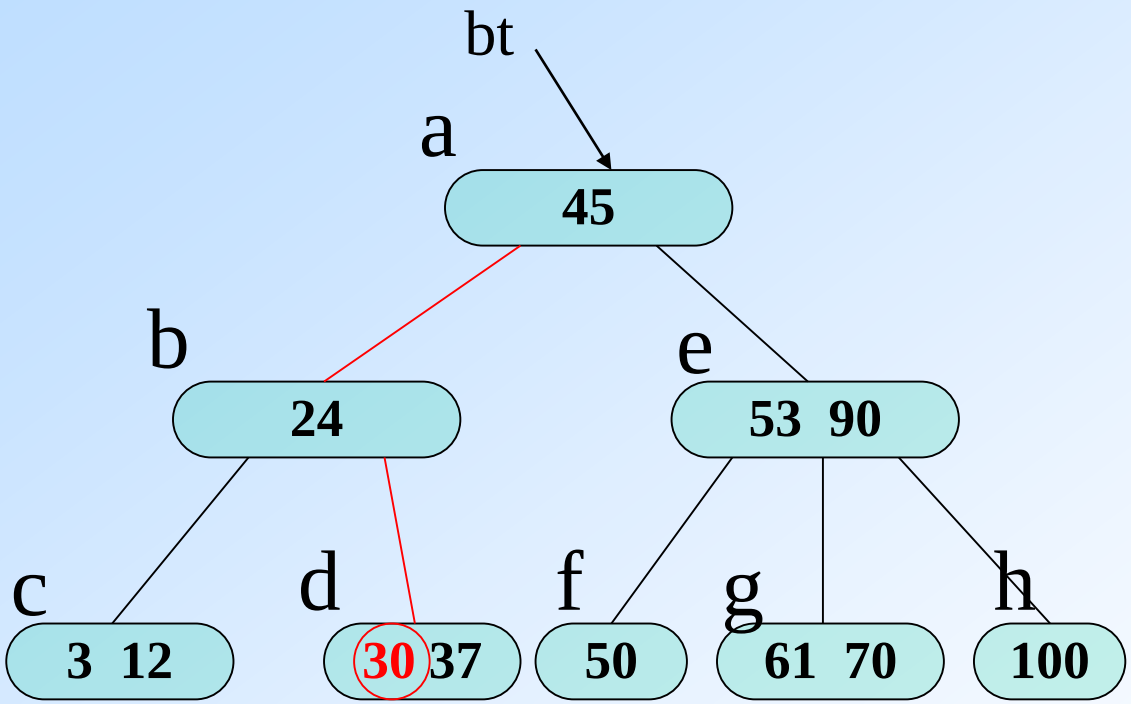
# B-树的插入：

- B-树⌈m/2⌉-1≤节点中的关键字个数≤ m-1,并且整个B-树可以看成全部由关键字组成的树,每次插入一个关键字不是在树中添加一个叶子节点，而是在查找的过程中找到叶子节点所在层的上一层（叶子节点是记录，上一层是关键字最后一层），在某个节点中添加一个关键字，若结点的关键字个数不超过m-1，则插入完成，否则产生节点的分裂。

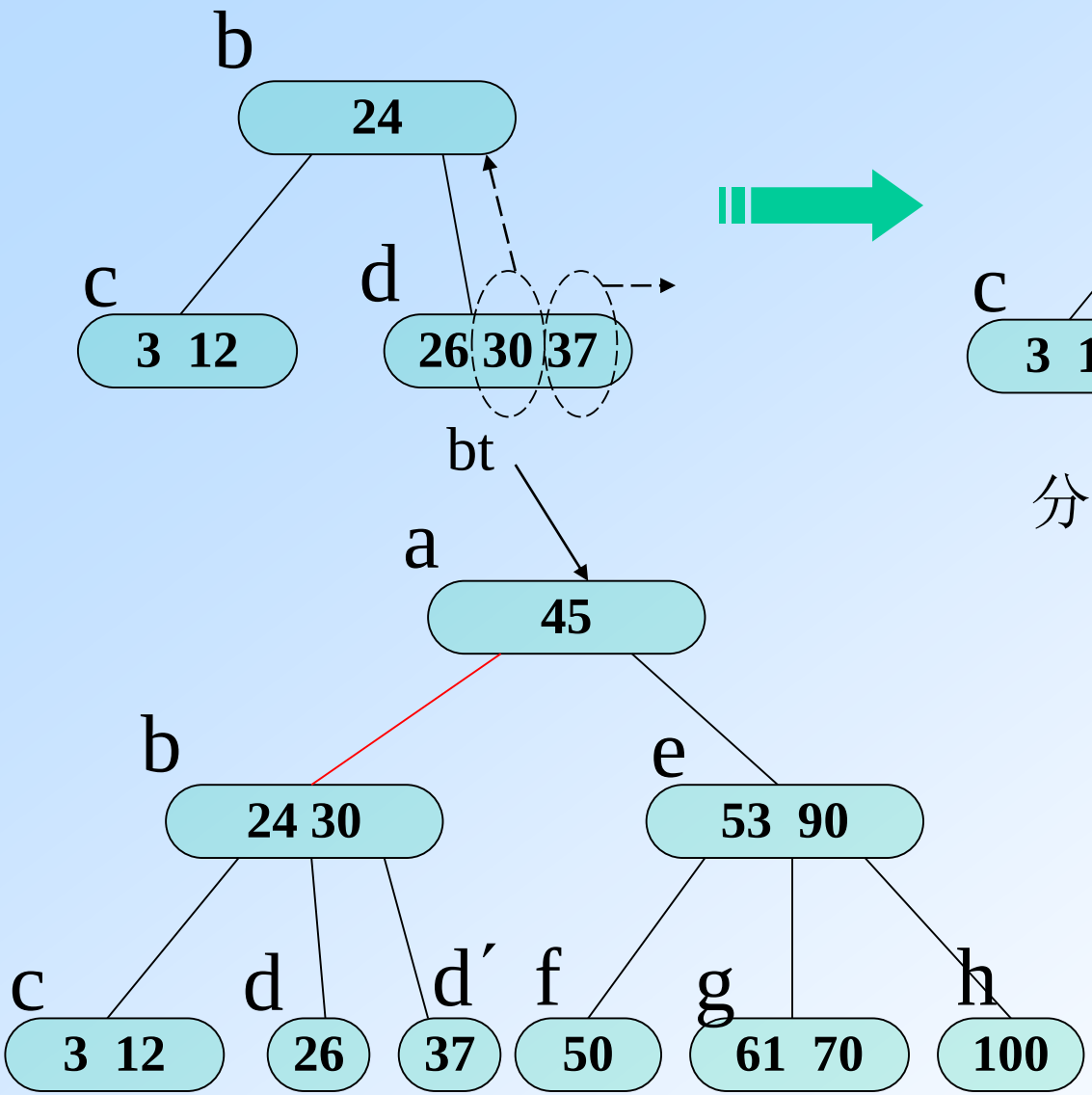
如在3阶B-树中依次插入关键字30，26，85，7

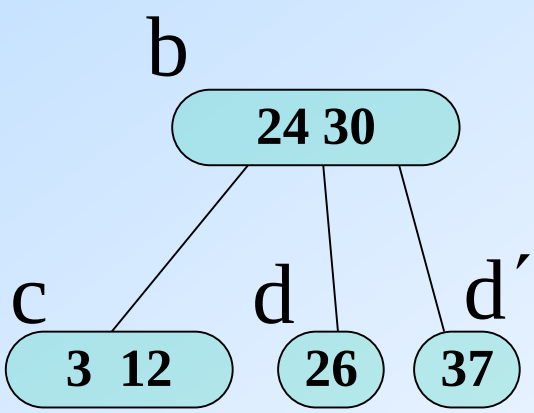



3阶B-树也叫2-3树



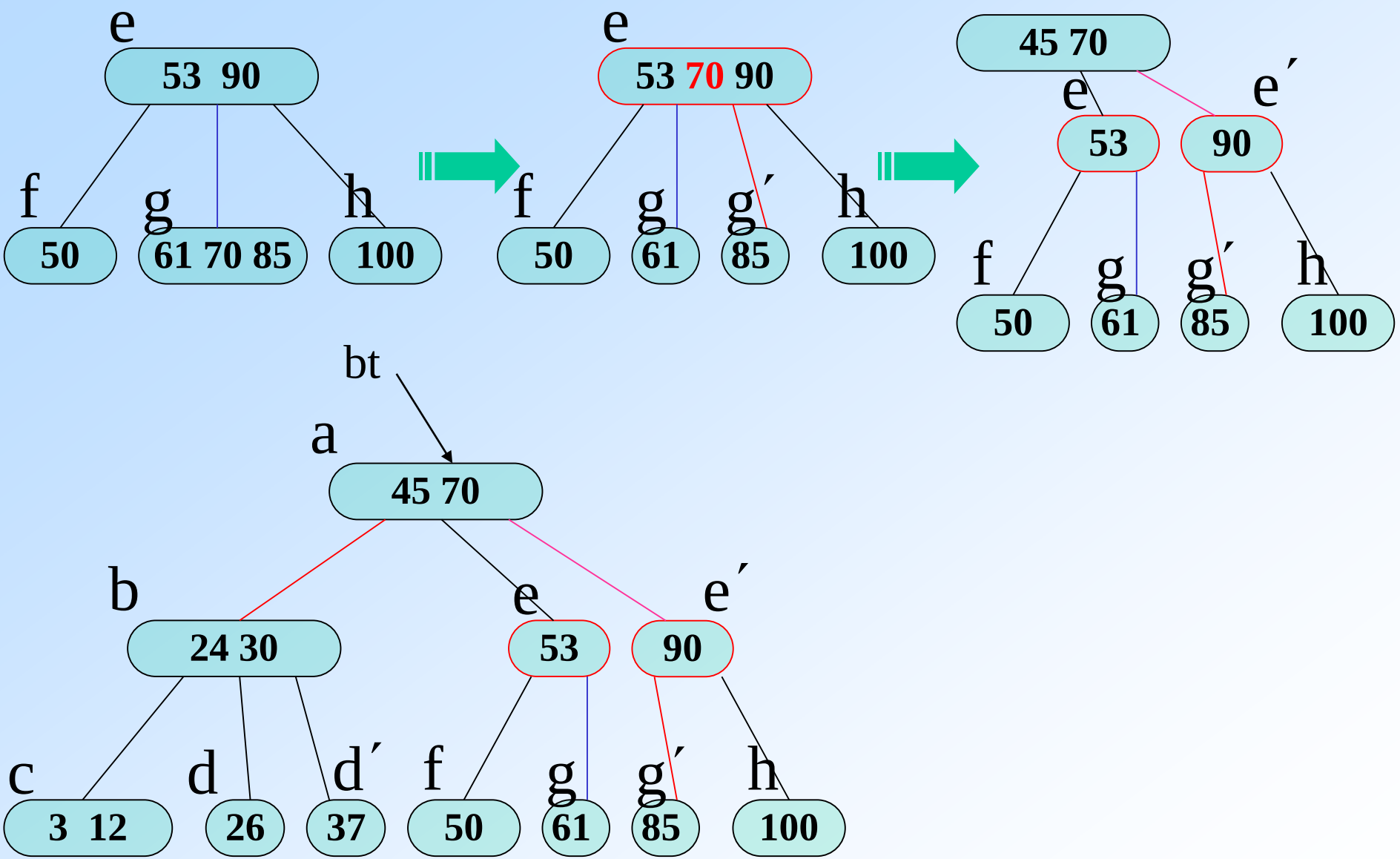

插入30



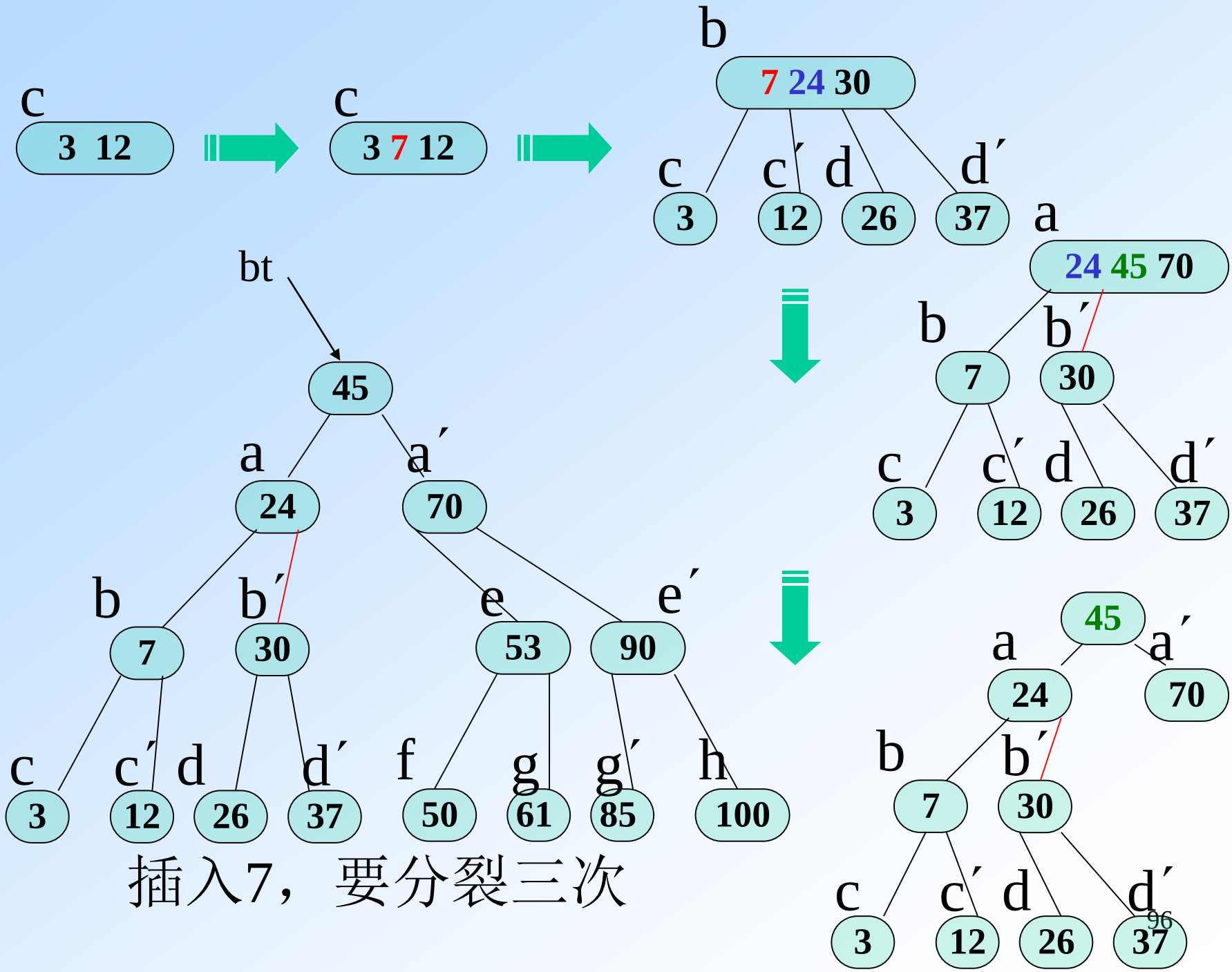




插入26

分裂时26及其前后指针保留在d中，37及其前后指针创建到d′中，30及指向d′的指针插入到双亲结点b中,如果30的插入导致b中关键字个数等于3，则按同样方法分裂


e
53 90
f
g
h
50
61 70 85
100
e
53 70 90
f
g
g′
h
50
61
85
100
45 70
e
e′
53
90
f
g
g′
h
50
61
85
100
bt
a
45 70
b
e
e′
24 30
53
90
c
d
d′
f
g
g′
h
3 12
26
37
50
61
85
100

插入85，要分裂两次


c
3 12
c
3 7 12
b
7 24 30
c
c′
d
d′
3
12
26
37
a
24 45 70
b
b′
7
30
c
c′
d
d′
3
12
26
37
bt
45
a
a′
24
70
b
b′
e
e′
7
30
53
90
c
c′
d
d′
f
g
g′
h
3
12
26
37
50
61
85
100
45
a
a′
24
70
b
b′
7
30
c
c′
d
d′
3
12
26
37

插入7，要分裂三次

# 节点分裂的一般规律：

假设p节点中已有m-1个关键字，当插入一个关键字之后，节点中信息为：

$m,A_0,(K_1,A_1),(K_2,A_2),\ldots,(K_m,A_m)$其中$K_i < K_{i+1}$ $1 \le i < m$,先将p分裂成p和p′两个节点

$p=\lceil m/2 \rceil-1,A_0,(K_1,A_1),\ldots,(K_{\lceil m/2 \rceil-1},A_{\lceil m/2 \rceil-1})$

$p' = m-\lceil m/2 \rceil,A_{\lceil m/2 \rceil},(K_{\lceil m/2 \rceil+1},A_{\lceil m/2 \rceil+1}),\ldots,(K_m,A_m)$

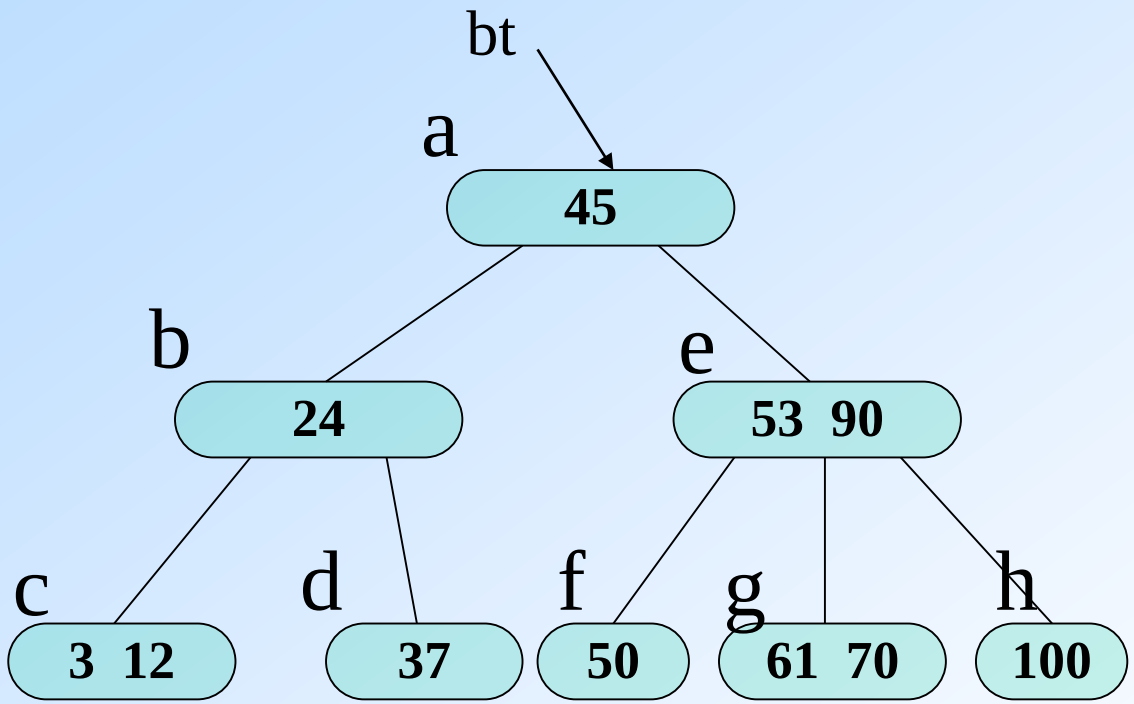
# B-树插入节点算法：

```
Status InsertBTree(BTree &T,KeyType K,BTree q,
    int i){  //T:B-树 K:关键字 q:插入节点 i:插入位置
    x=K;ap=NULL;finished=FALSE;
    while(q&&!findished){
        Insert(q,i,x,ap);//插入节点
        if(q->keynum<m) finished=TRUE;//判断是否要分裂
        else{
            s=⌈m/2⌉;split(q,ap);x=q->key[s];
            //分裂，用s指向要插入父结点的关键字位置
            q=q->parent;
            if(q) i=Search(q,x);//搜索父结点的插入位置
        }//else
    }//while
    if(!finished) NewRoot(T,q,x,ap);//空树则创建新根节点
}//InsertBTree
```

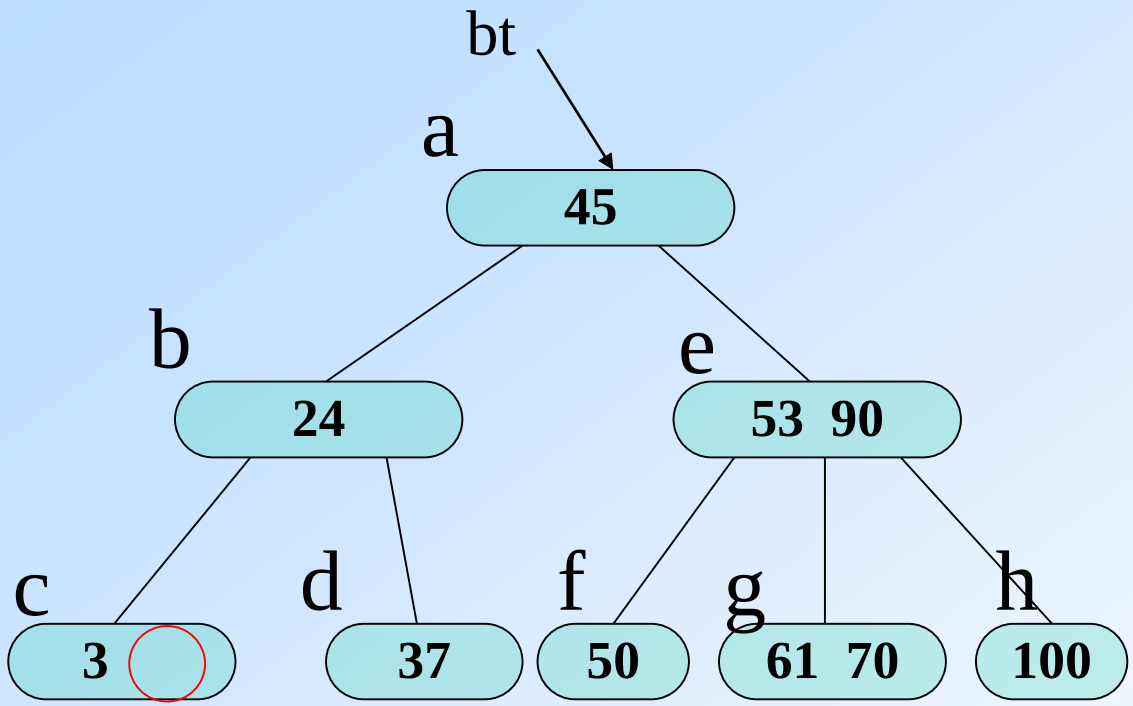
# B-树删除

- 假设删除关键字不在最下层，设关键字为$K_i$，则可以用$A_i$指向子树的最小关键字或$A_{i-1}$指向子树的最大关键字替换$K_i$，再删去这个关键字即可，而该关键字必定在最下层，所以只需讨论删除最下层节点关键字的情况。
- 假设删除节点在最下层，删除后仍满足B-树定义则删除结束，否则要进行合并节点的操作，合并可能自下向上层层进行。

如在3阶B-树中依次删除关键字12,50，53，37，分为三种情况

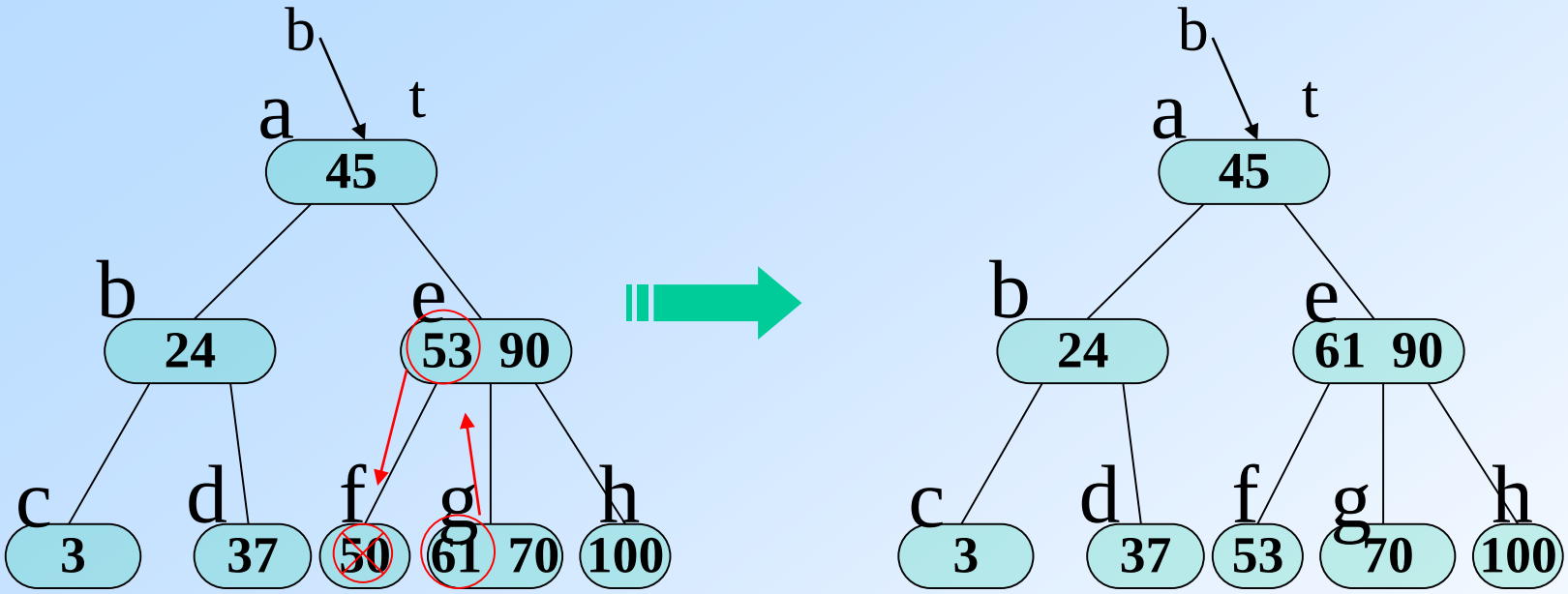



3阶B-树



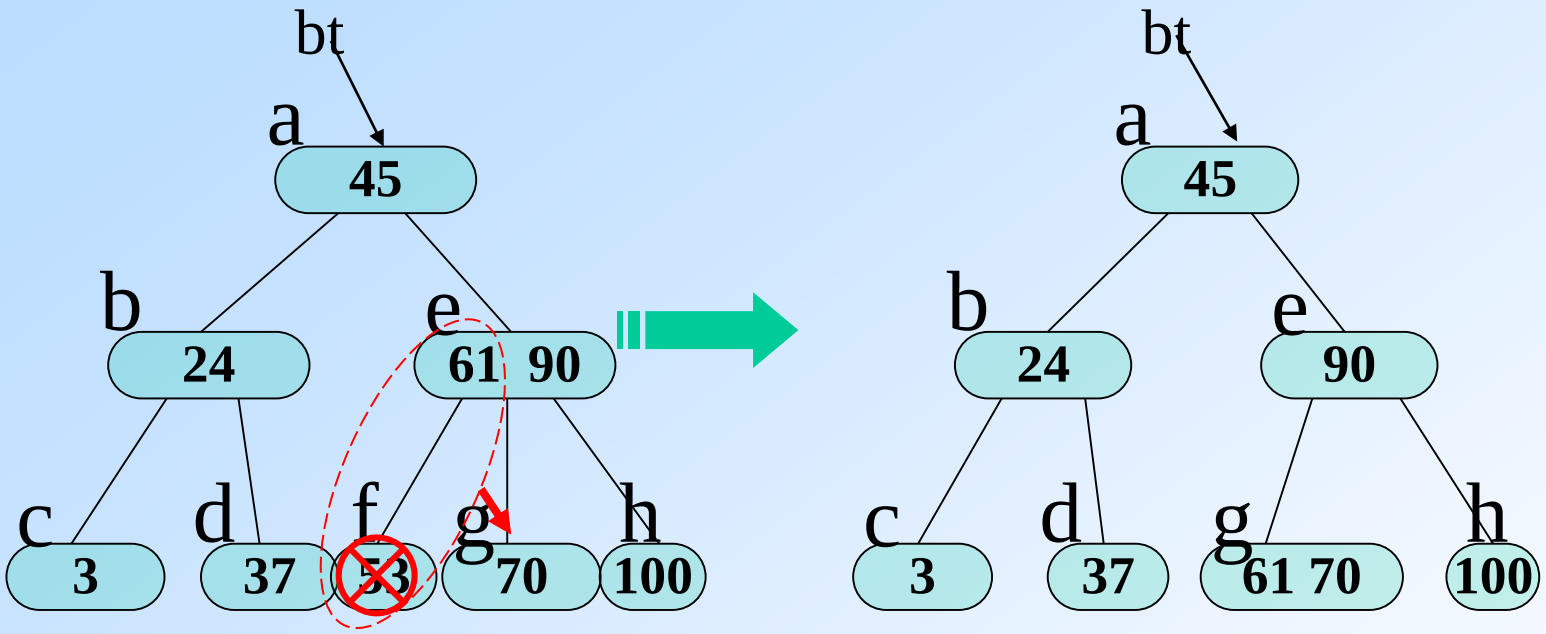

删除关键字12

(1)被删关键字(12)所在节点(c)中的关键字数目不小于⌈m/2⌉则只须从该节点(c)中删去该关键字(12)和相应指针，树的其他部分不变。



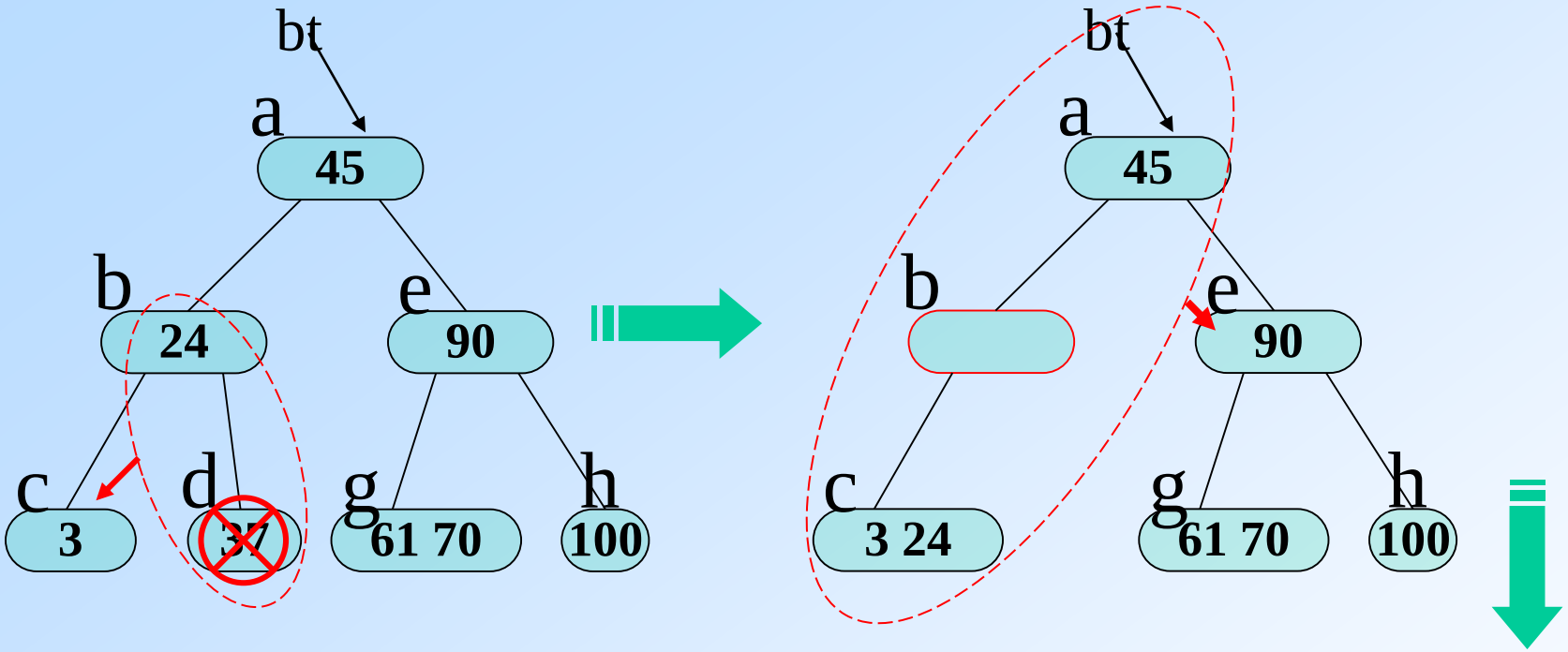

删除关键字50

（2）被删关键字(50)所在节点(f)中的关键字数目等于⌈m/2⌉-1,其相邻的某个兄弟结点(g)（左兄弟或右兄弟）中的关键字数目大于⌈m/2⌉-1，则将其兄弟结点(g)中的最小（或最大）关键字(61)上移至双亲结点中，而将双亲结点中小于（或大于）且紧靠该上移关键字(61)的关键字(53)下移至被删关键字(50)所在节点(f)中。



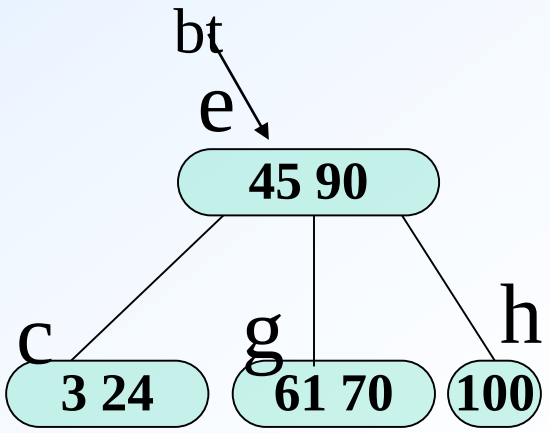

删除关键字53

（3）被删关键字(53)所在节点(f)和其相邻的兄弟结点(g)中的关键字数目均等于⌈m/2⌉-1(1), 假设该节点有右兄弟(g),在删去关键字(53)后，他所在结点(f)中剩余的关键字和指针加上双亲结点(e)中的关键字(61)一起合并到右兄弟结点(g)中。




删除关键字37

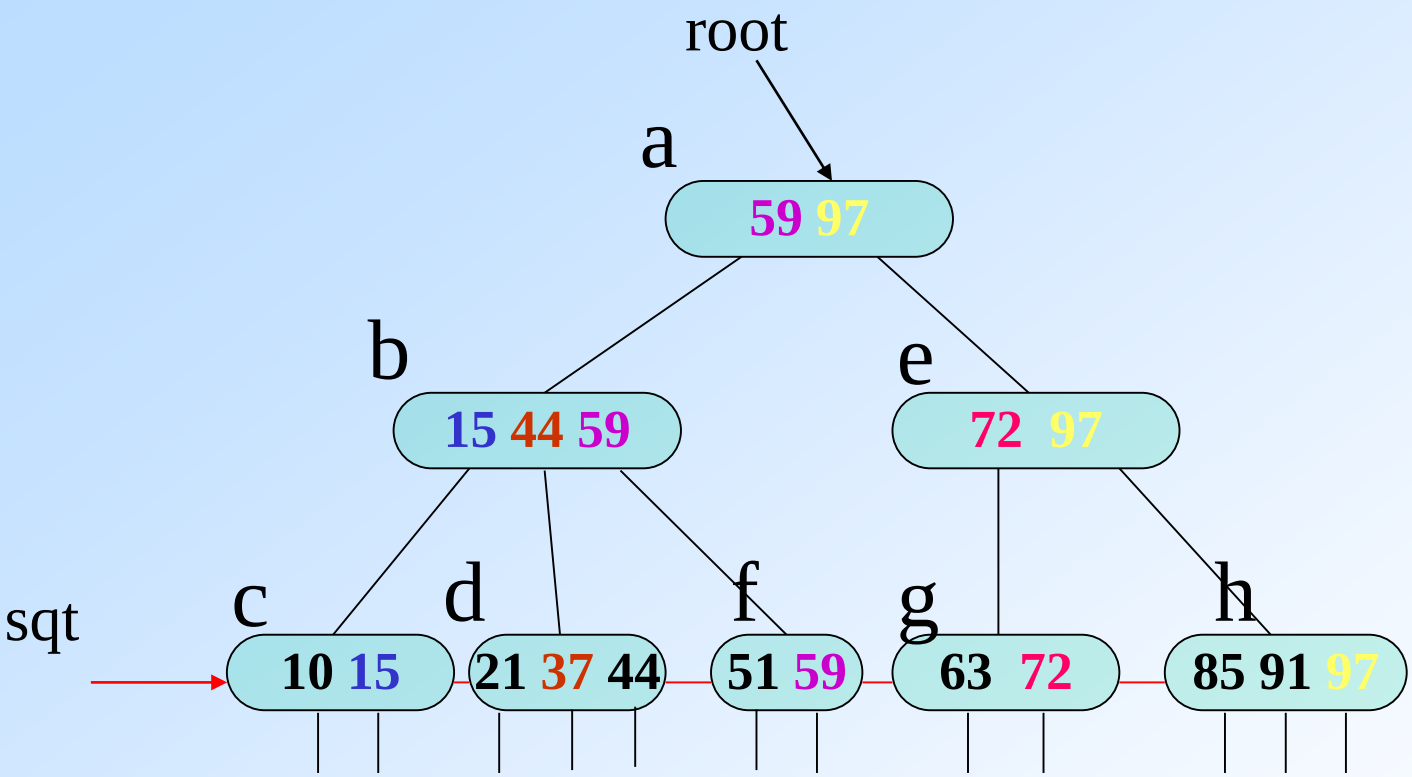
如果删除后使双亲结点中的关键字数目小于$\lceil m/2 \rceil - 1$，则依此类推层层向上合并。



# $B^+$树

$B^+$树是$B_-$树的变型，m阶的$B^+$树和m阶的$B_-$树的区别是：

- 关键字个数和子树个数一样多
- 所有叶子结点中包含全部关键字信息及指向含这些关键字记录的指针，且叶子节点本身依关键字的大小自小而大顺序链接。
- 所有非终端结点可看成索引，节点中仅含有其子树中的最大（或最小）关键字。


root
a
59 97
b
15 44 59
e
72 97
sqt
c
d
f
g
h
10 15
21 37 44
51 59
63 72
85 91 97

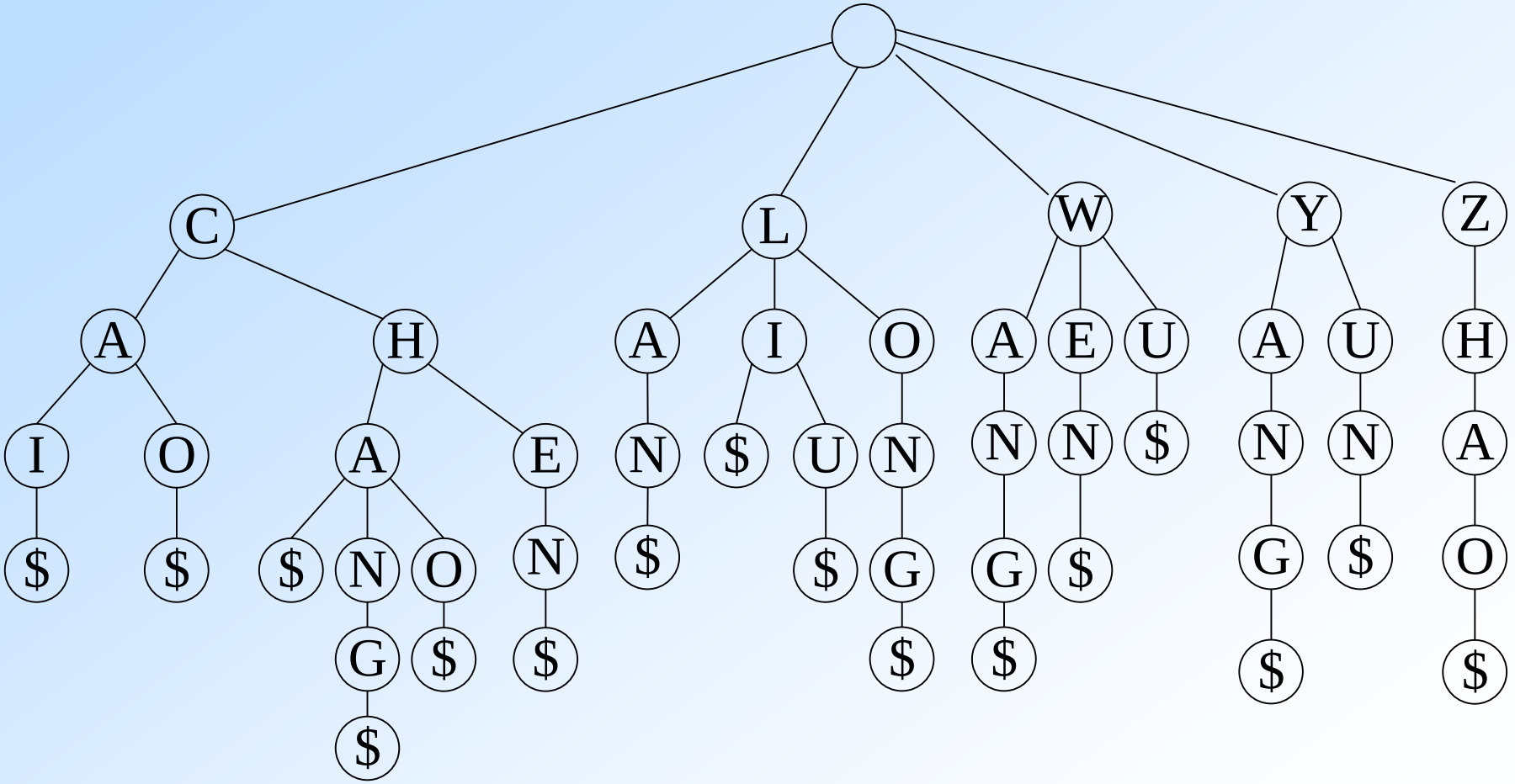
3阶B+树

- B+树的查找可以从根节点开始，若非终端结点上的关键字等于给定值，并不终止，二是继续向下直到叶子节点（因为非终端结点中只包含关键字而不是整个记录）；也可以从最小关键字的叶子节点开始顺序查找。

# 键树

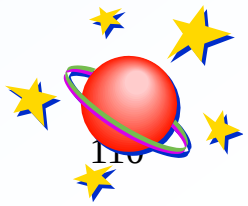
- 键树（数字查找树）：是一棵度≥2的树，树中每个结点不是包含一个或几个关键字，而是只含有组成关键字的符号。如数值中的每一位，单词中的每个字母。

假设关键字集合为
{CAI,CAO,LI,LAN,CHA,CHANG,WEN,CHAO,YUN,YANG,LONG,WANG,ZHAO,LIU,WU,CHEN}



# 哈 希 表

哈希表的相关定义
哈希函数的构造方法
处理冲突的方法
哈希表的查找
哈希表的插入
哈希查找分析



# 哈希表的相关定义

## 哈希查找

又叫散列查找，利用哈希函数进行查找的过程。

基本思想：在记录的存储地址和它的关键字之间建立一个确定的对应关系；这样，不经过比较，一次存取就能得到所查元素的查找方法。

## 哈希函数

在记录的关键字与记录的存储地址之间建立的一种对应关系。哈希函数是一种映象，是从关键字空间到存储地址空间的一种映象。

可写成，$addr(a_i)=H(k_i)$

其中：$a_i$是表中的一个元素

$addr(a_i)$是$a_i$的存储地址

$k_i$是$a_i$的关键字

## 哈希表

根据设定的哈希函数 **H(key)** 和所选中的处理冲突的方法，将一组关键字映象到一个有限的、地址连续的地址集 (区间) 上，并以关键字在地址集中的“象”作为相应记录在表中的存储位置，如此构造所得的查找表称之为“哈希表”。

# 例 30个地区的各民族人口统计表

| 编号 | 地区别 | 总人口 | 汉族 | 回族…… |
|---|---|---|---|---|
| 1 | 北京 | | | |
| 2 | 上海 | | | |
| ⋮ | ⋮ | | | |

以编号作关键字，
构造哈希函数：H(key)=key
H(1)=1
H(2)=2

以地区别作关键字，取地区名称第一个拼音字母的序号作哈希函数：H(Beijing)=2
H(Shanghai)=19
H(Shenyang)=19

从例子可见：

哈希函数只是一种映象，所以哈希函数的设定很灵活，只要使任何关键字的哈希函数值都落在表长允许的范围之内即可。

冲突：key1≠key2，但H(key1)=H(key2)的现象

- 一般来说，只能尽量减少冲突而不能完全避免冲突，这是因为通常关键字集合比较大，其元素包括所有可能的关键字，而地址集合的元素仅为哈希表中的地址值，如上例中城市和地区的可能出现值不下几百几千而地址集合的大小只有26，所以哈希函数是一个压缩映象，就不可避免产生冲突。在定义哈希表时既要定义好哈希函数又要给出处理冲突的方法。

# 哈希函数构造的方法

- 直接定址法
- 数字分析法
- 平方取中法
- 折叠法
- 除留余数法
- 随机数法



# 直接定址法

哈希函数为关键字的线性函数

**H(key) = key 或者 H(key) = a × key + b**

此法仅适合于：

地址集合的大小 = = 关键字集合的大小

其中a和b为常数



# 数字分析法

假设关键字集合中的每个关键字都是由 s 位数字组成 ($u_1$, $u_2$, …, $u_s$)，分析关键字集中的全体，并从中提取分布均匀的若干位或它们的组合作为地址。此法适于能预先估计出全体关键字的每一位上各种数字出现的频度。

例 有80个记录，关键字为8位十进制数，哈希地址为2位十进制数

①② ③ ④⑤⑥ ⑦⑧

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 8 | 1 | 3 | 4 | 6 | 5 | 3 | 2 |
| 8 | 1 | 3 | 7 | 2 | 2 | 4 | 2 |
| 8 | 1 | 3 | 8 | 7 | 4 | 2 | 2 |
| 8 | 1 | 3 | 0 | 1 | 3 | 6 | 7 |
| 8 | 1 | 3 | 2 | 2 | 8 | 1 | 7 |
| 8 | 1 | 3 | 3 | 8 | 9 | 6 | 7 |
| 8 | 1 | 3 | 6 | 8 | 5 | 3 | 7 |
| 8 | 1 | 4 | 1 | 9 | 3 | 5 | 5 |

分析：①只取8
②只取1
③只取3、4
⑧只取2、7、5
④⑤⑥⑦数字分布近乎随机

所以：取④⑤⑥⑦任意两位或两位与另两位的叠加作哈希地址



# 平方取中法

以关键字的平方值的中间几位作为存储地址。求“关键字的平方值”的目的是“扩大差别”，同时平方值的中间各位又能受到整个关键字中各位的影响。

此方法适合于:

关键字中的每一位都有某些数字重复出现频度很高的现象。



# 折叠法

将关键字分割成若干部分，然后取它们的叠加和为哈希地址。两种叠加处理的方法：

移位叠加:将分割后的几部分低位对齐相加

间界叠加:从一端沿分割界来回折送，然后对齐相加

此法适于关键字的数字位数特别多。

例　关键字为：0442205864，哈希地址位数为4

```
  5 8 6 4
  4 2 2 0
    0 4
-----------
  1 0 0 8 8
```

H(key)=0088

移位叠加

```
  5 8 6 4
  0 2 2 4
    0 4
-----------
  6 0 9 2
```

H(key)=6092

间界叠加



# 除留余数法

设定哈希函数为:

$$H(key) = key \ MOD \ p \quad (p \le m)$$

其中， m为表长

p 为不大于 m 的素数

或是

不含 20 以下的质因子

## 为什么要对 p 加限制？

**例如：**

给定一组关键字为: 12, 39, 18, 24, 33, 21，若取 p=9, 则他们对应的哈希函数值将为:

3, 3, 0, 6, 6, 3

可见，若 p 中含质因子 3, 则所有含质因子 3 的关键字均映射到“3 的倍数”的地址上，从而增加了“冲突”的可能



# 随机数法

设定哈希函数为:

H(key) = Random(key)

其中，Random 为伪随机函数

此法用于对长度不等的关键字构造哈希函数。

选取哈希函数考虑的因素：

计算哈希函数所需时间

关键字长度

哈希表长度（哈希地址范围）

关键字分布情况

记录的查找频率



# 处理冲突的方法

“处理冲突” 的实际含义是：

为产生冲突的地址寻找下一个哈希地址。

- 开放定址法
- 再哈希法
- 链地址法



# 开放定址法

为产生冲突的地址 H(key) 求得一个地址序列： $H_0, H_1, H_2, \ldots, H_s$ $1 \le s \le m-1$

$H_i = ( H(key) + d_i ) \text{ MOD } m$

其中： $i=1, 2, \ldots, s$

H(key)为哈希函数;m为哈希表长;

$d_i$为增量序列,有下列三种取法:

对增量 $d_i$ 的三种取法：

1) 线性探测再散列

$d_i = c \times i$ 最简单的情况 $c=1$

2) 二次探测再散列

$d_i = 1^2, -1^2, 2^2, -2^2, \ldots,$

3) 随机探测再散列

$d_i$ 是一组伪随机数列 或者

$d_i = i \times H_2(key)$ (又称双散列函数探测)

注意：增量 $d_i$ 应具有“完备性”

即：产生的 $H_i$ 均不相同，且所产生的 $s(m-1)$个 $H_i$ 值能覆盖哈希表中所有地址。则要求：

※ 平方探测时的表长 m 必为形如 4j+3 的素数（如: 7, 11, 19, 23, … 等）；

※ 随机探测时的 m 和 $d_i$ 没有公因子。

例 表长为11的哈希表中已填有关键字为17，60，29的记录，H(key)=key MOD 11,现有第4个记录，其关键字为38，按三种处理冲突的方法，将它填入表中


(1) H(38)=38 MOD 11=5 冲突
H1=(5+1) MOD 11=6 冲突
H2=(5+2) MOD 11=7 冲突
H3=(5+3) MOD 11=8 不冲突

(2) H(38)=38 MOD 11=5 冲突
H1=(5+1²) MOD 11=6 冲突
H2=(5-1²) MOD 11=4 不冲突

(3) H(38)=38 MOD 11=5 冲突
设伪随机数序列为9，则：
H1=(5+9) MOD 11=3 不冲突

例如：给定关键字集合构造哈希表

{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }

设定哈希函数 H(key) = key MOD 11 ( 表长=11 )

若采用线性探测再散列处理冲突

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 55 | 01 | 23 | 14 | 68 | 11 | 82 | 36 | 19 | | |
| 1 | 1 | 2 | 1 | 3 | 6 | 2 | 5 | 1 | | |

若采用二次探测再散列处理冲突

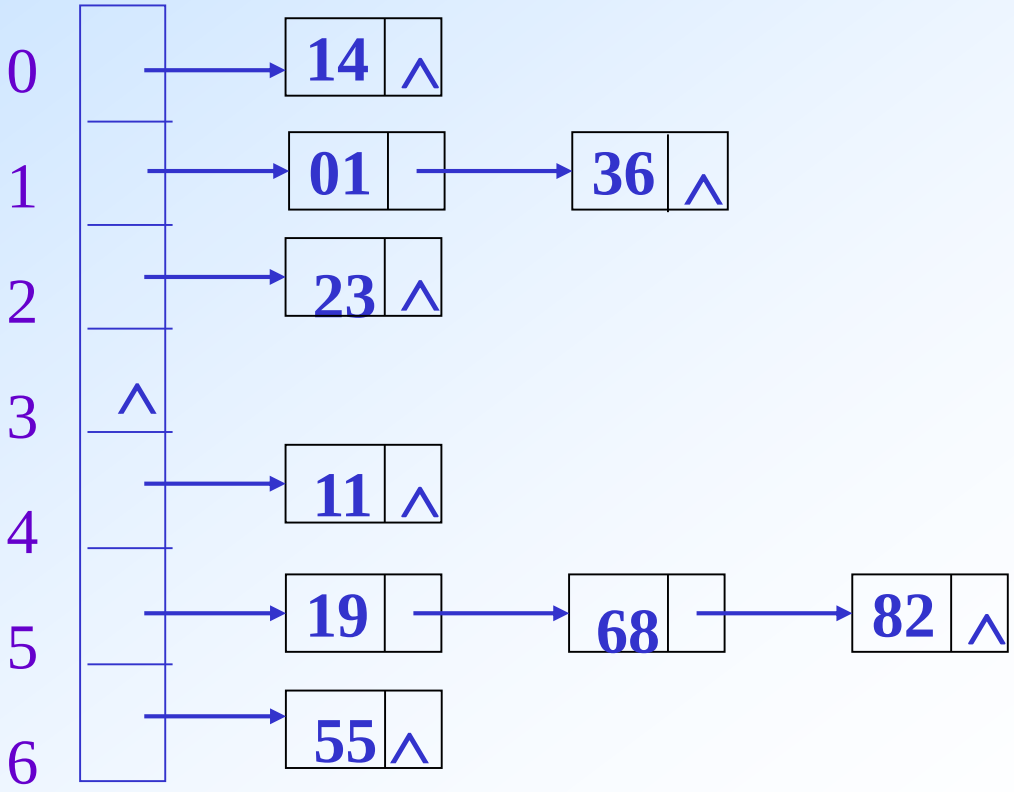
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 55 | 01 | 23 | 14 | 36 | 82 | 68 | | 19 | | 11 |

# 链地址法

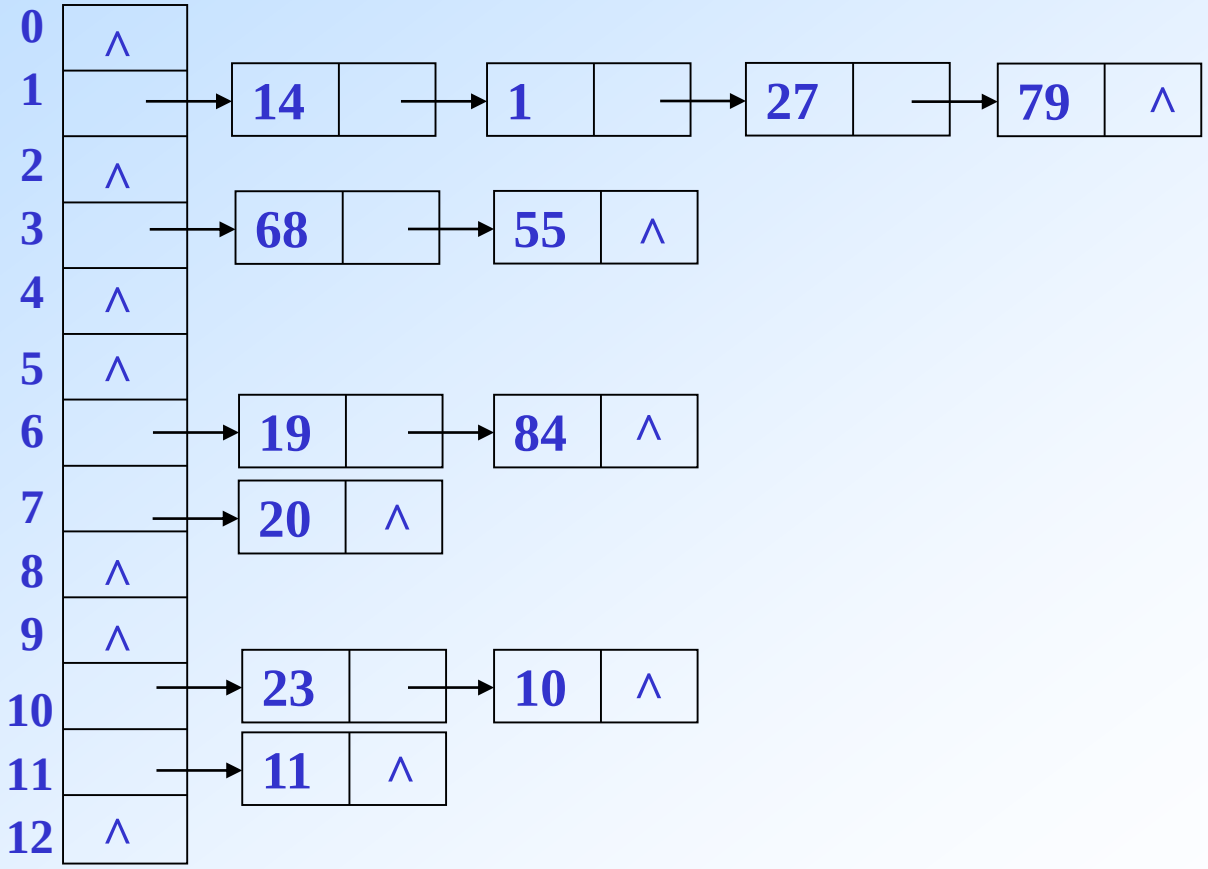
将所有哈希地址相同的记录都链接在同一链表中。

例:给定关键字{ 19, 01, 23, 14, 55, 68, 11, 82,36 }

哈希函数为 H(key)=key MOD 7



例　已知一组关键字(19,14,23,1,68,20,84,27,55,11,10,79)
哈希函数为：H(key)=key MOD 13,
用链地址法处理冲突



# 再哈希法

方法：构造若干个哈希函数，当发生冲突时，计算下一个哈希地址，直到冲突不再发生。

即：$H_i=Rh_i(key)$　　$i=1,2,\ldots\ldots k$

其中：$Rh_i$——不同的哈希函数

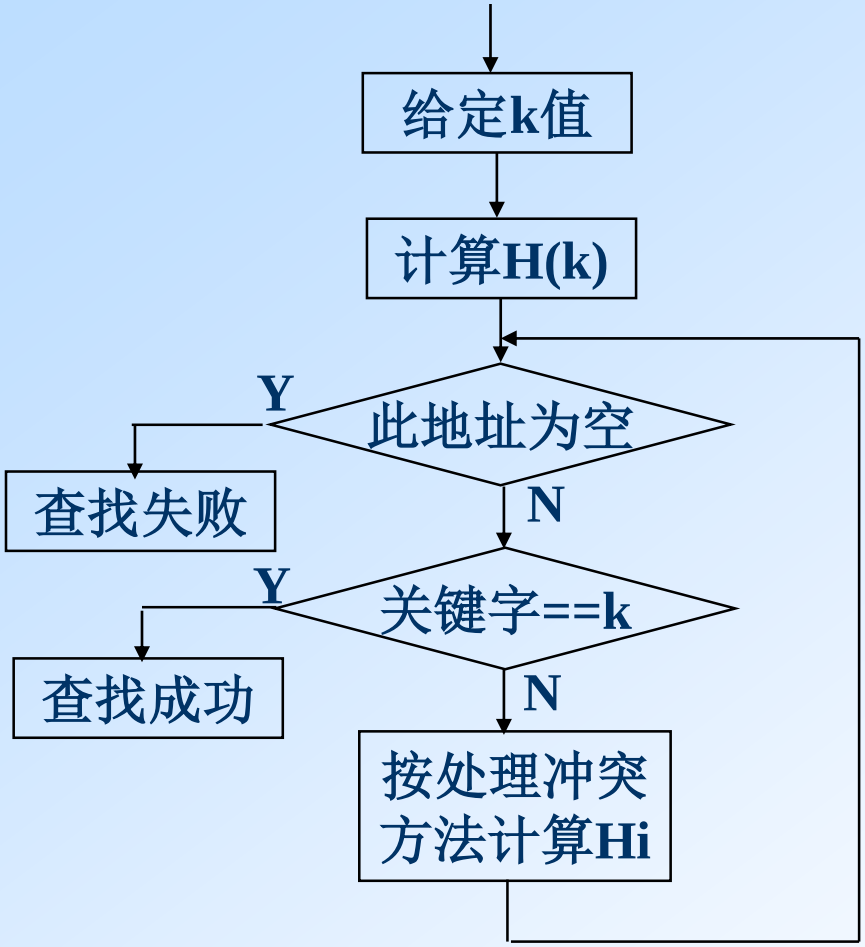
特点：计算时间增加

# 建立公共溢出区

- 设哈希函数的值域为[0,m-1]，则设向量HashTable[0..m-1]为基本表，每个分量存放一个记录，另设立向量OverTable[0..v]为溢出表。所有关键字和基本表中关键字为同义词的记录，不管他们由哈希函数得到的哈希地址是什么，一旦发生冲突，都填入溢出表。

# 哈希表的查找

## 哈希查找过程




对于给定值 K,

计算哈希地址 i = H(K)

若 r[i] = NULL 则查找不成功

若 r[i].key = K 则查找成功

否则 “求下一地址 Hi”,

直至r[Hi] = NULL (查找不成功)
或r[Hi].key = K (查找成功) 为止。

# 开放定址哈希表的存储结构

```
int  hashsize[] = { 997, ... };
typedef struct {
  ElemType  *elem;
  int  count;              // 当前数据元素个数
  int  sizeindex;
                  // hashsize[sizeindex]为当前容量
} HashTable;
#define  SUCCESS  1
#define  UNSUCCESS  0
#define  DUPLICATE  -1
```

# 查找算法

```
Status SearchHash (HashTable H, KeyType K,
                                  int &p, int &c) {
  // 在开放定址哈希表H中查找关键码为K的记录
      p = Hash(K);        // 求得哈希地址
      while ( H.elem[p].key != NULLKEY &&
                          !EQ(K, H.elem[p].key)）
          collision(p, ++c);      // 求得下一探查地址 p
      if (EQ(K, H.elem[p].key)) return SUCCESS;
       // 查找成功，返回待查数据元素位置 p
      else return UNSUCCESS;   // 查找不成功
} // SearchHash
```



# 哈希表的插入

```
Status InsertHash (HashTable &H, Elemtype e){
if ( HashSearch ( H, e.key, p, c ) == SUCCESS )
    return DUPLICATE;
                        // 表中已有与 e 有相同关键字的元素
elseif ( c < hashsize[H.sizeindex]/2 ) {
                // 冲突次数 c 未达到上限，（阀值 c 可调）
     H.elem[p] = e;  ++H.count;  return OK;
                        // 查找不成功时，返回 p为插入位置
}
else  RecreateHashTable(H);        // 重建哈希表
 } // InsertHash
```

例　已知一组关键字(19,14,23,1,68,20,84,27,55,11,10,79)
哈希函数为H(key)=key MOD 13, 哈希表长为m=16，用线性探测再散列处理冲突得哈希表

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 14 | 1 | 68 | 27 | 55 | 19 | 20 | 84 | 79 | 23 | 11 | 10 | | | |

给定值K=84的查找过程为:

H(84)=6　不空且不等于84, 冲突

H1=(6+1)MOD16=7不空且不等于84, 冲突

H2=(6+2)MOD16=8不空且等于84, 查找成功, 返回记录在表中的序号。

给定值K=38的查找过程为:

H(38)=12　不空且不等于38,冲突

H1=(12+1)MOD16=13空记录

表中不存在关键字等于38 的记录,查找不成功。

# 哈希表查找的分析

从查找过程得知，哈希表查找的平均查找长度实际上并不等于零。

决定哈希表查找的ASL的因素：

1) 选用的哈希函数;

2) 选用的处理冲突的方法;

3) 哈希表饱和的程度，装载因子 α =n/m 值的大小（n—表中填入的记录数，m—表的长度）

一般情况下，可以认为选用的哈希函数是“均匀”的，则在讨论ASL时，可以不考虑它的因素。

因此，哈希表的ASL是处理冲突方法和装载因子的函数。

例如：前述例子

线性探测处理冲突时，　ASL = 22/9

双散列探测处理冲突时，ASL =14/9

链地址法处理冲突时，　ASL = 13/9

可以证明：查找成功时有下列结果：

线性探测再散列

$$S_{nl} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

随机探测再散列

$$S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$$

链地址法

$$S_{nc} \approx 1+\frac{\alpha}{2}$$

从以上结果可见，

哈希表的平均查找长度是 α 的函数，

而不是 n 的函数。

这说明，用哈希表构造查找表时，

可以选择一个适当的装填因子 α ，

使得平均查找长度限定在某个范围内。

—— 这是哈希表所特有的特点。