数据结构与控制算法分析

专题三

查找与排序

学习内容与要求

- 学习和掌握顺序查找和折半查找算法的原理和实现；
- 学习和掌握二叉排序树的概念及其构造方法、二叉排序树的查找算法原理。
- 学习和掌握选择排序、交换排序、插入排序、归并排序和快速排序方法的原理。

1 Search
(查找/搜索)

所谓查找（或搜索），就是在数据集合中寻找满足某种条件的数据对象：

1. 查找成功　即找到满足条件的数据对象时，作为结果，可报告该对象在结构中的位置，还可给出该对象中的具体信息。

2. 查找不成功　或搜索失败。作为结果，应报告一些信息，如失败标志、位置等。

- 通常称用于查找的数据集合为查找结构，它是由同一数据类型的数据(或记录)组成。
- 每个对象有若干属性，其中有一个属性，其值可唯一地标识这个对象，称为关键字。使用基于关键字的搜索，查找结果应是唯一的。但在实际应用时，查找条件是多方面的，可以使用基于属性的查找方法，但查找结果可能不唯一。

实施查找时有两种不同的环境

- 静态环境　查找结构不需进行插入和删除操作。　　— 静态查找表

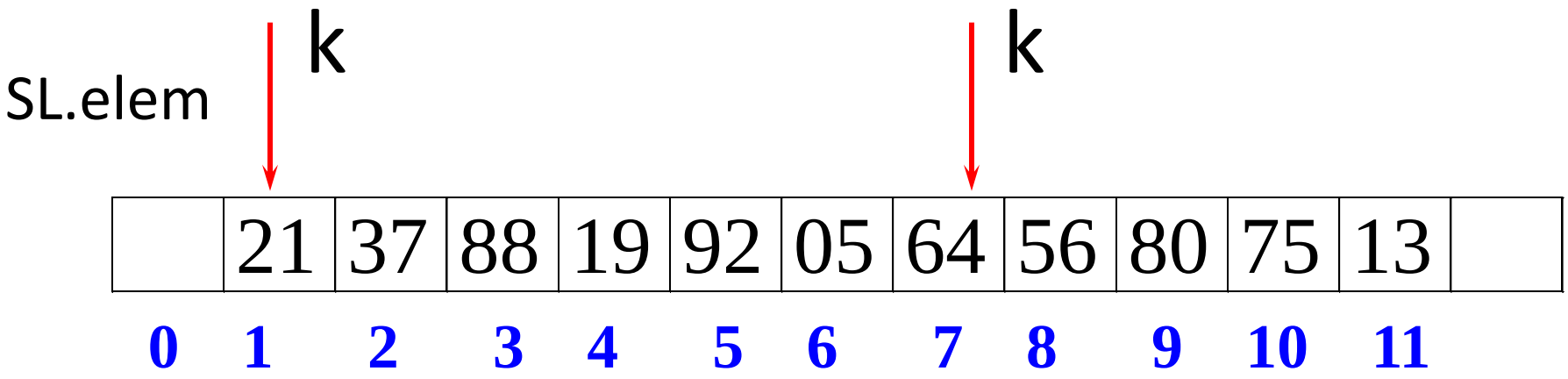
- 动态环境　查找过程中可能要对查找结构执行数据插入、删除或修改等操作，并对查找结构进行调整，结构可能发生变化。　　— 动态查找表

- 动态查找表的表结构是在查找过程中动态生成的。

1.1 顺序查找（Sequential Search)

- 以**线性结构**表示静态查找表。
- 基本原理：将待查找记录依次逐个与表中记录进行比较。

顺序查找过程（从前向后查找）



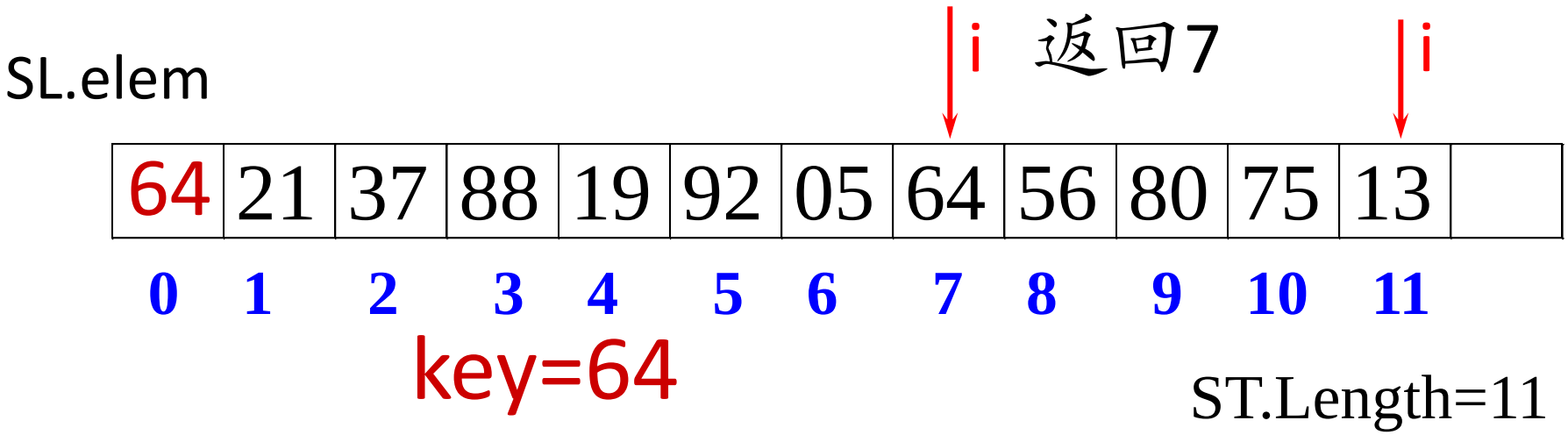
ST.Length=11

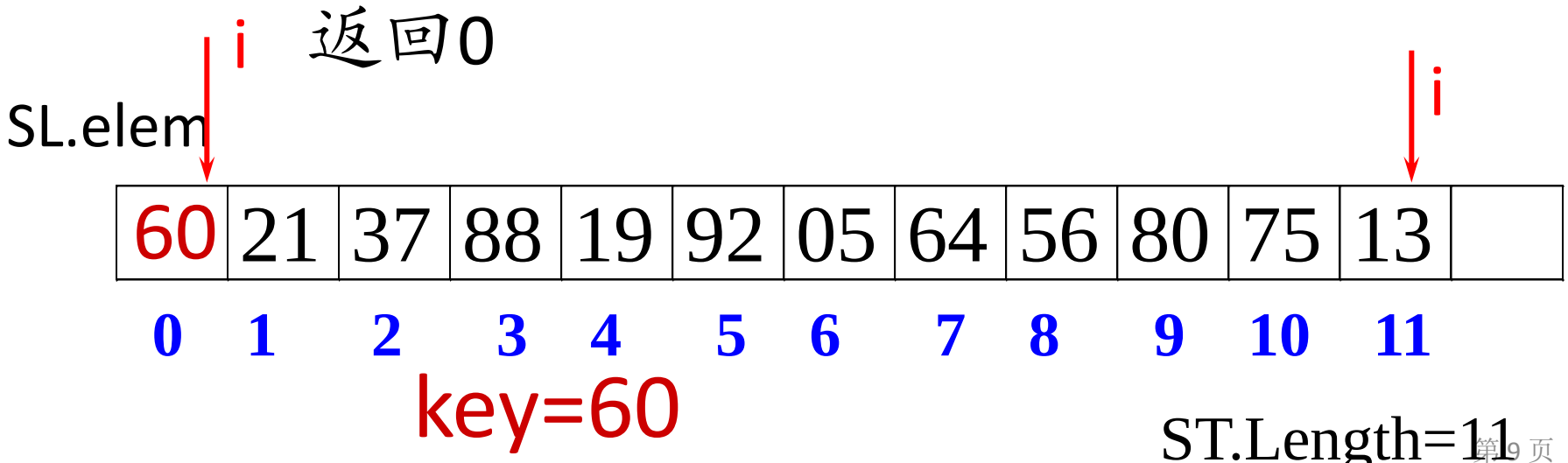
假设给定查找值 e=64,

要求 SL.elem[k] = e, 问: k = ?

顺序查找过程（从后向前查找）

0单元用于存放待查找关键字，作为“哨兵”(guard)





顺序查找算法实现（从后向前查找）

```
int Search_Seq( TB SL, TYPE key )
{
  SL.elem[0].key = key;     //  “哨兵”
  for (i=SL.length; SL.elem[i].key!=key;
                          --i); // 从后往前找
  return i; // 查找成功时i为有效下标,否则i为0
}
```

查找算法的评价指标

查找成功: 最少比较次数

最多比较次数

平均比较次数

查找失败: 最少比较次数

最多比较次数

平均比较次数

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

指为了确定记录在查找表中的位置，需要和给定值进行关键字比较的次数的期望值:

$$ASL = \sum_{i=1}^{n} P_i C_i$$

其中，n为表长，P_i为查找表中第i个记录的概率，且$\sum_{j=1}^{n} P_i = 1$；C_i为查找到该记录时，曾和给定值进行关键字比较的次数。

在等概率情形，查找任一记录的概率相等：$p_i = 1/n,\ i = 1, 2, \ldots, n$

以从前向后顺序查找算法为例，

- 查找成功时，

$$ASL_{succ} = \sum_{i=1}^{n} \frac{1}{n} \cdot i = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

- 查找不成功时，$ASL_{unsucc} = \sum_{i=1}^{n} \frac{1}{n} \cdot n = n$

查找算法时间复杂度？ $O(n)$

顺序查找算法总结

- 查找长度：

  查找成功: 最少比较次数　　1

  最多比较次数　　n

  平均比较次数　(n+1)/2

  查找失败: 最少比较次数　n

  最多比较次数　n

  平均比较次数　n

- 优点：查找结构无特殊要求（线性结构均适用）；算法简单；
- 缺点：查找效率较低，不适于大表查找。

1.2 折半查找（二分查找）（Binary Search)

上述按顺序查找表的查找算法简单，但平均查找长度较大，不适用于表长较大的查找结构。

若静态查找表为有序表，则查找过程可以基于“折半”进行。

基于有序顺序表的折半查找

- 设*n*个数据对象存放在一个有序顺序表中，并按其关键字值从小到大（或从大到小）排好序。
- 原理：折半查找时，每次都先求出位于查找区间正中的对象的下标mid，用其关键字与给定值x比较，然后根据比较结果将查找区间缩小一半，直至找到被查找对象。

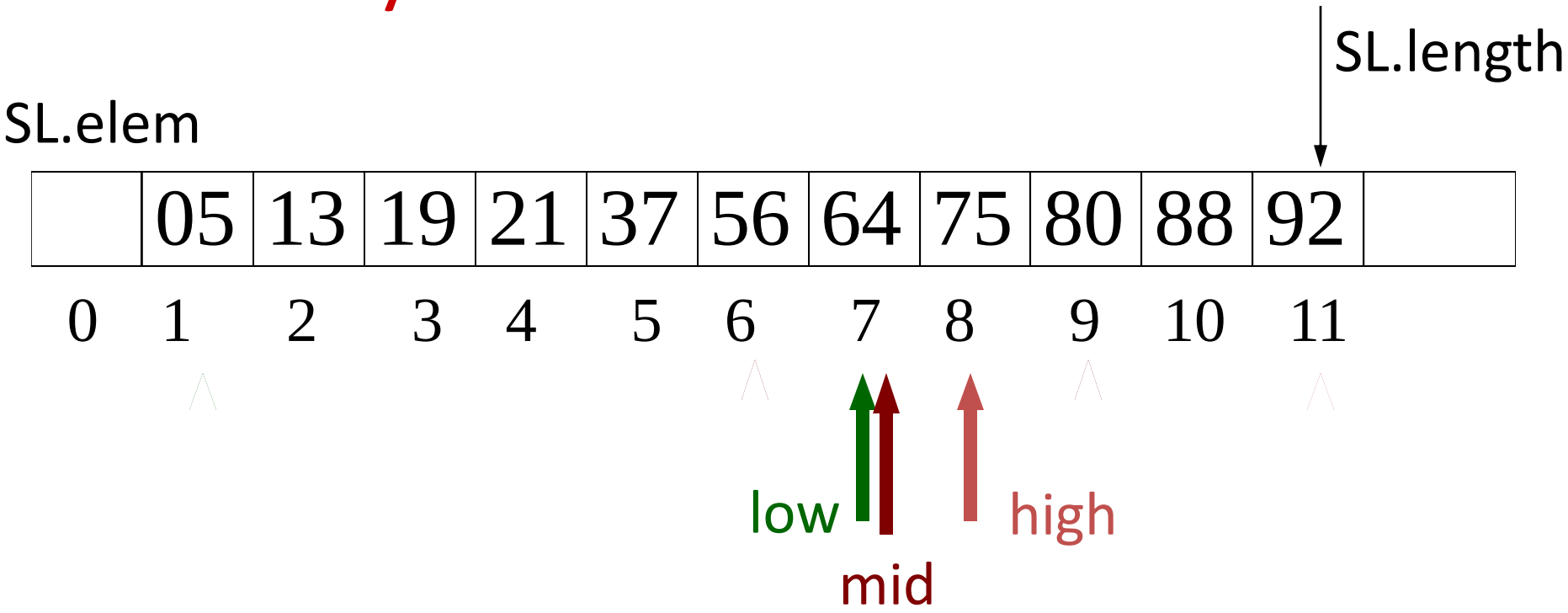
折半查找算法

（设数据按关键字从小到大有序排列）

1. Element[mid].key==x 查找成功；
2. Element[mid].key>x 把查找区间缩小为表的前半部分，继续折半查找；
3. Element[mid].key<x 把查找区间缩小为表的后半部分，继续折半查找；

- 如果查找区间缩小到一个对象，且仍未找到想要查找的对象，则查找失败。

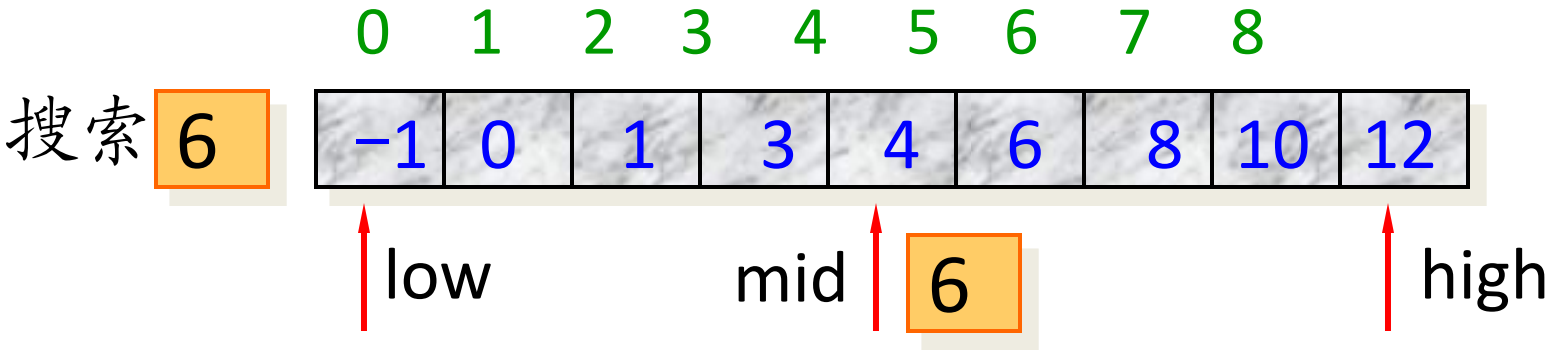
例如：key=64 的查找过程如下



mid = ⌊(low+high)/2⌋ （向下取整）

Low 指示查找区间的下界

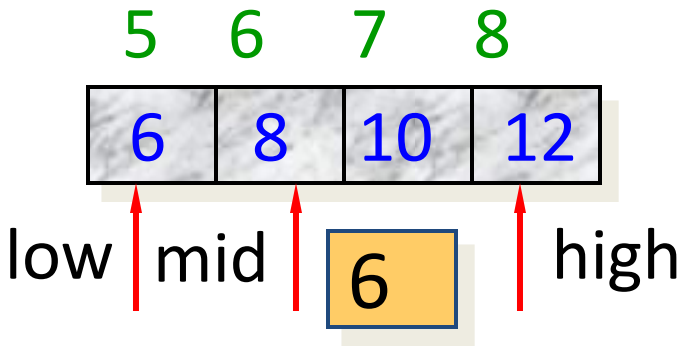
high 指示查找区间的上界

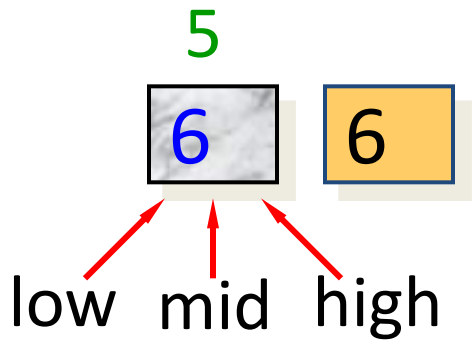


(1) low=0,high=8,mid=4;由于Elem[mid]<Key, 因此， low = mid+1=5;

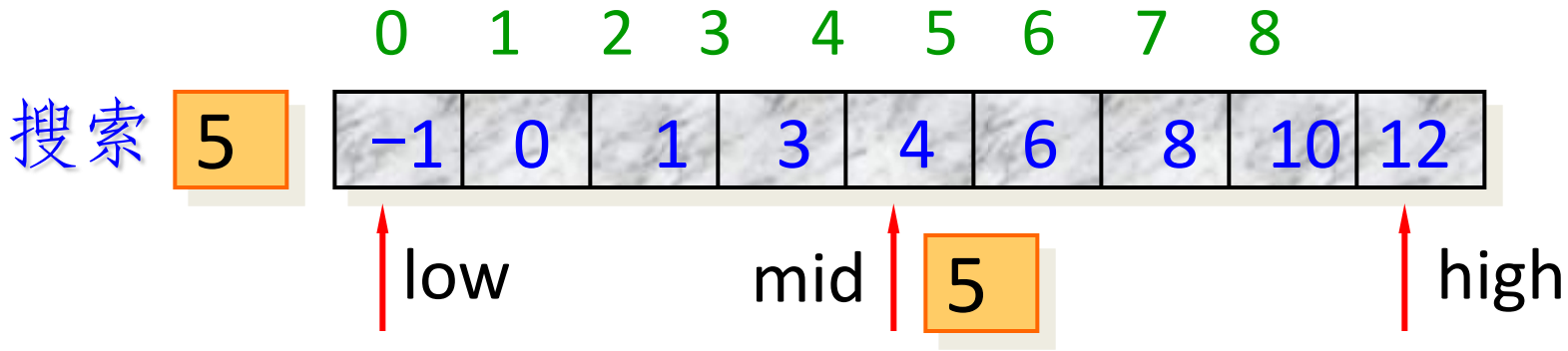
(2) mid=6,由于Elem[mid]>Key, 因此, high=mid-1=5;

(3)mid=5, Elem[mid] =Key,因此查找成功.





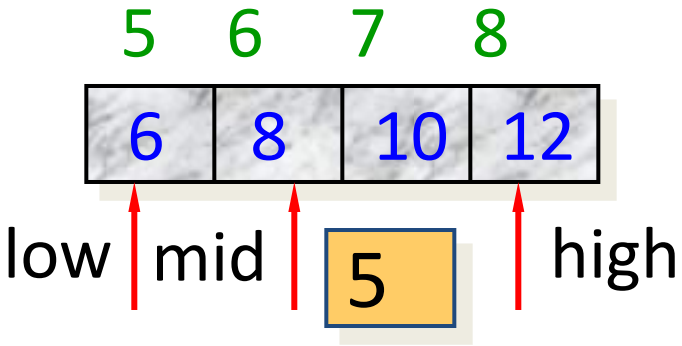
查找成功的例子

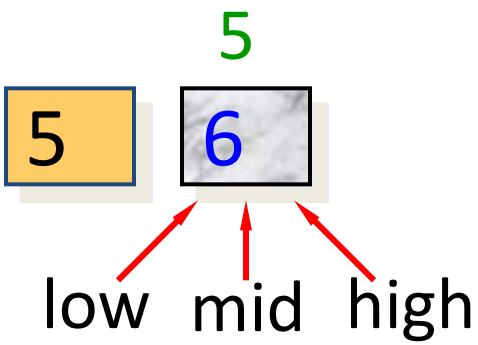


(1) low=0,high=8,mid=4;由于Elem[mid]<Key, 因此， low = mid+1=5;

(2) mid=6,由于Elem[mid]>Key, 因此, high=mid-1=5;

(3)mid=5,由于Elem[mid] >Key,因此, high=mid-1=4. 由于此时high<low,因此查找失败.





查找失败的例子

折半查找算法实现

```
int Search_Bin(TB SL, TYPE key)
{ int low = 1, high = SL.length,mid;
  while(low<=high)
  { mid=(low+high)/2;
     if(key==SL.elem[mid].key)return mid;
     if(key<SL.elem[mid].key)high=mid-1;
     if(key>SL.elem[mid].key)low=mid+1;
  }
  return 0;
}
```

折半查找算法总结

- 平均查找长度：

$$ASL_{succ} = \frac{1}{n}(1\times 2^0 + 2\times 2^1 + \cdots + k\times 2^{k-1})$$

查找成功:

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

时间复杂度：$O(\log_2 n)$

- 优点：查找效率高；
- 缺点：查找结构有限制；插入、删除操作困难。
- 适用场合：记录按关键字值有序排列，查找结构为顺序表结构，数据不经常变动（静态查找）。

1.3 二叉排序树查找

若要对动态查找表进行高效率的查找操作（包含可能的数据删除或插入操作），可采用二叉排序树作为查找表的组织形式。

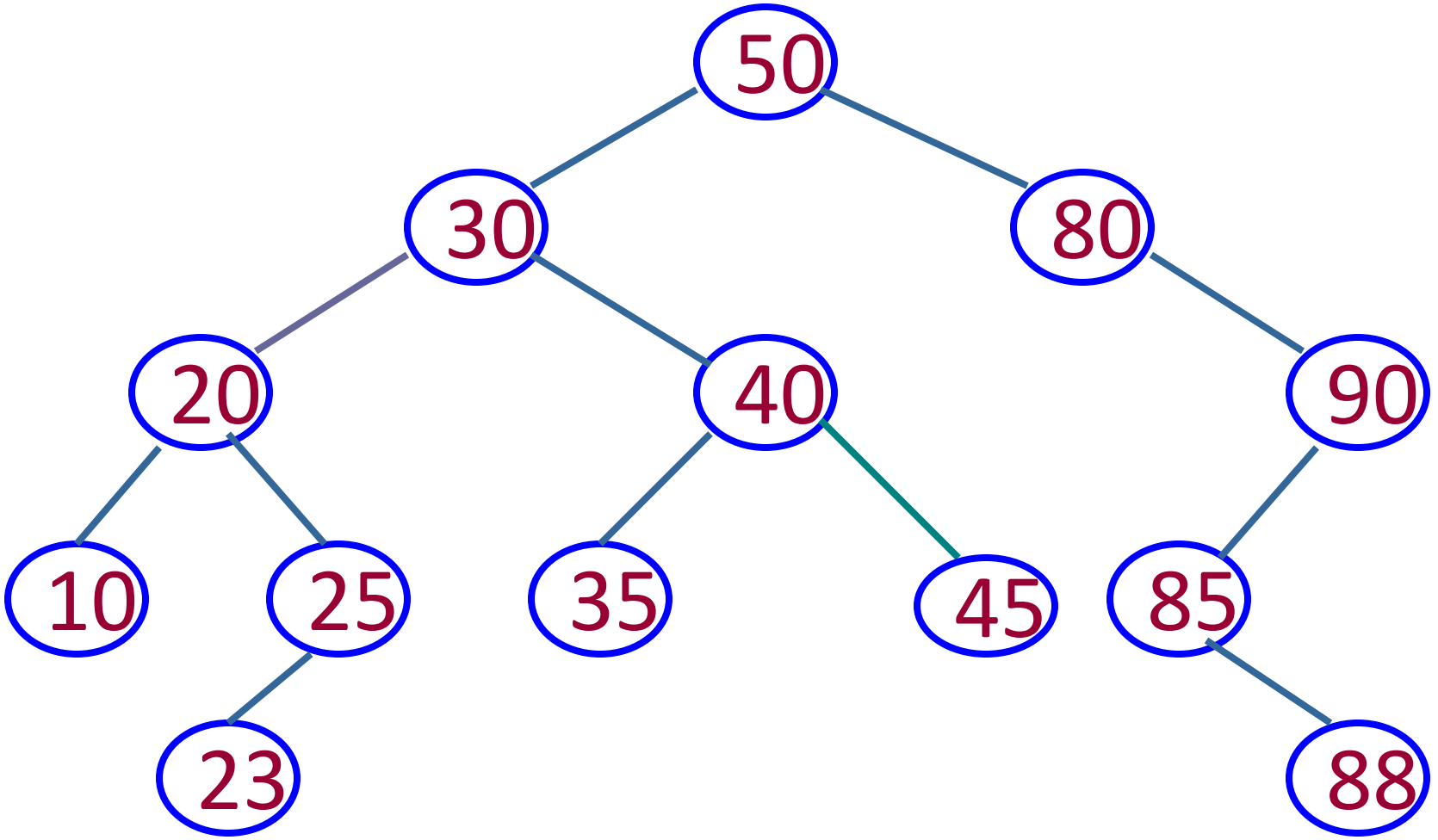
二叉排序树的概念

二叉排序树（或二叉查找树）或者是一棵空树；或者是具有如下特性的二叉树：

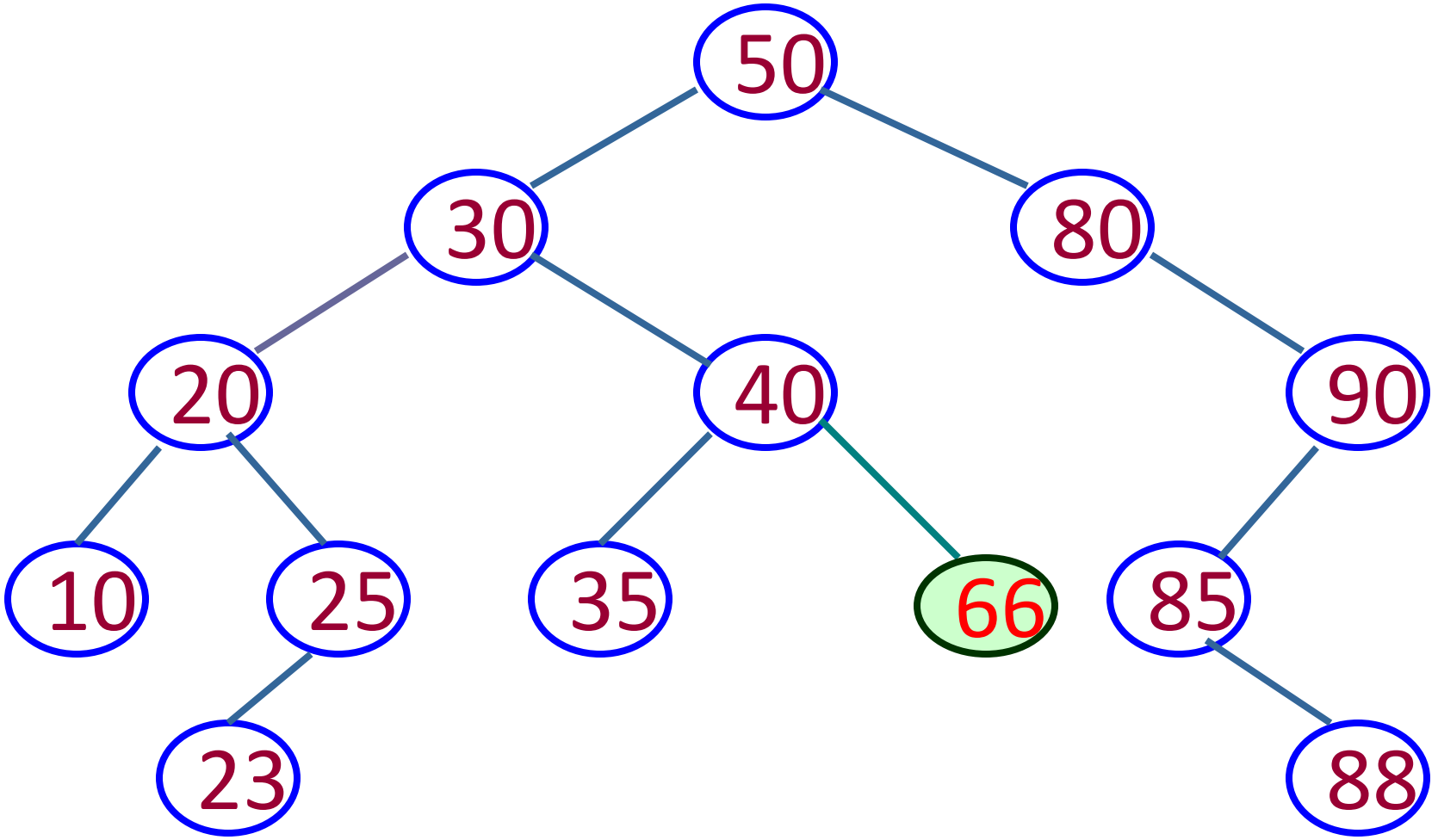
（1）若它的左子树不空，则左子树上所有结点的值**均小于**根结点的值；

（2）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；

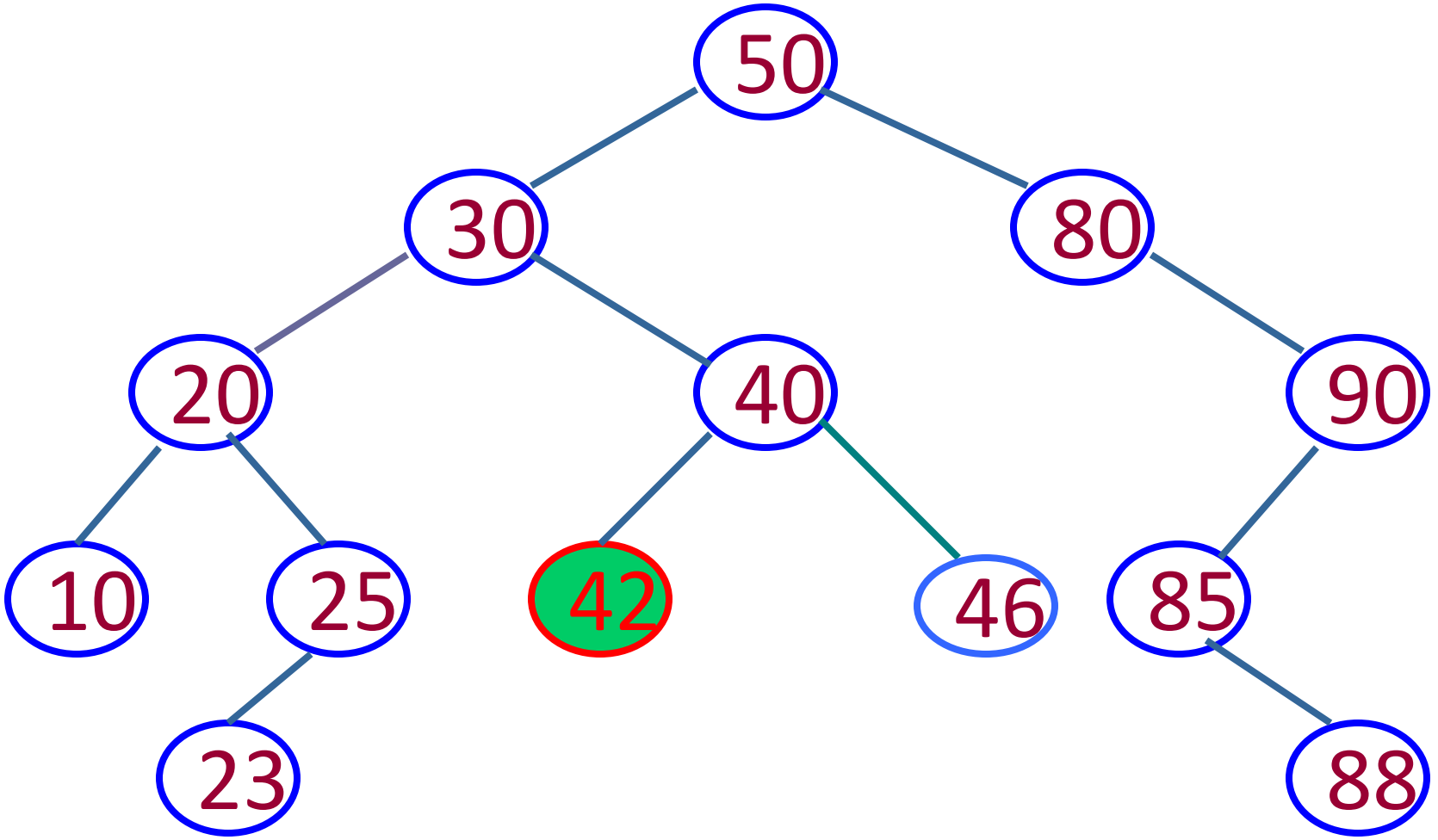
（3）它的左、右子树也都分别**是二叉排序树**。



是二叉排序树



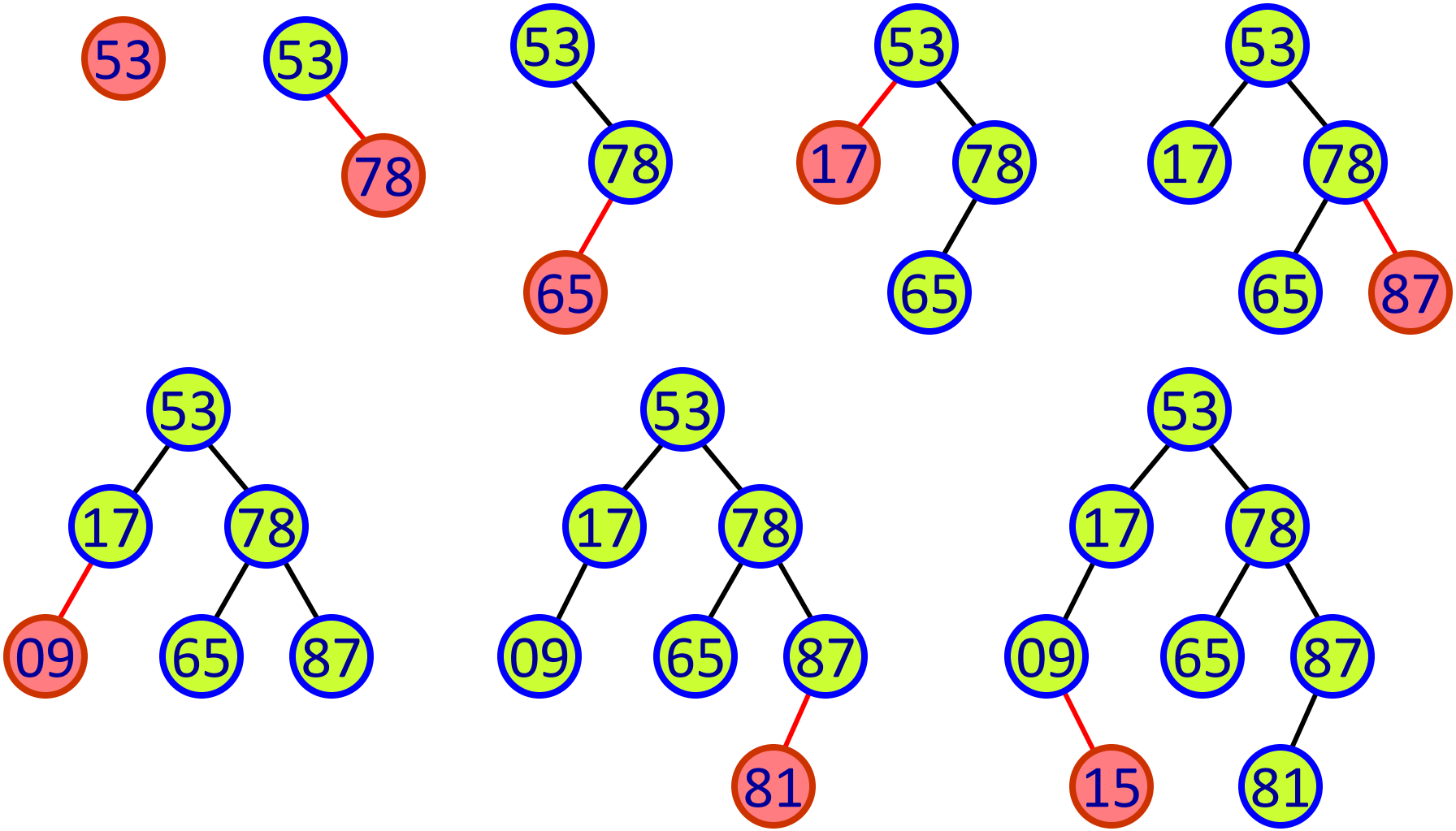
不是二叉排序树



也**不**是二叉排序树

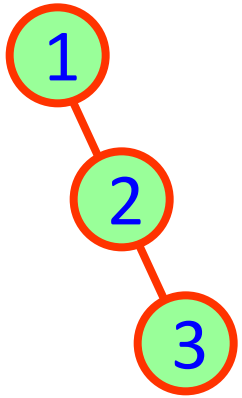
根据输入序列构造二叉排序树

输入数据 { 53, 78, 65, 17, 87, 09, 81, 15 }

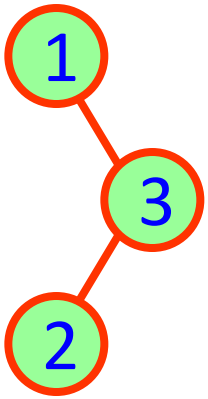


注意：同样n个数据{1, 2, …, n}，输入顺序不同，建立的二叉排序树形态也不同。例如，

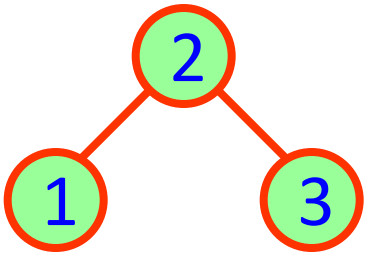
{1,2,3}



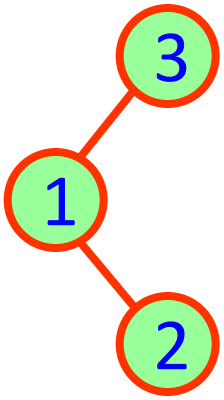
{1,3,2}

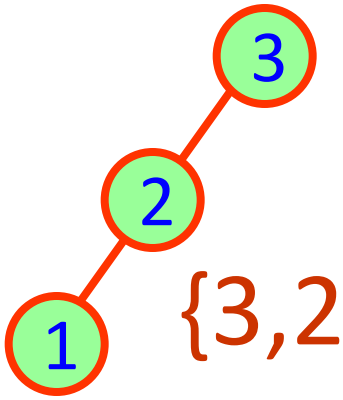


{2,1,3}



{3,1,2}





{3,2,1}

二叉排序树与中序遍历

如果对一棵二叉排序树进行**中序遍历**，其遍历序列是将树中的各结点关键字按**从小到大的顺序排列**起来。

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;

从根结点出发，沿着左分支或右分支逐层向下直至关键字等于给定值的结点。——查找成功

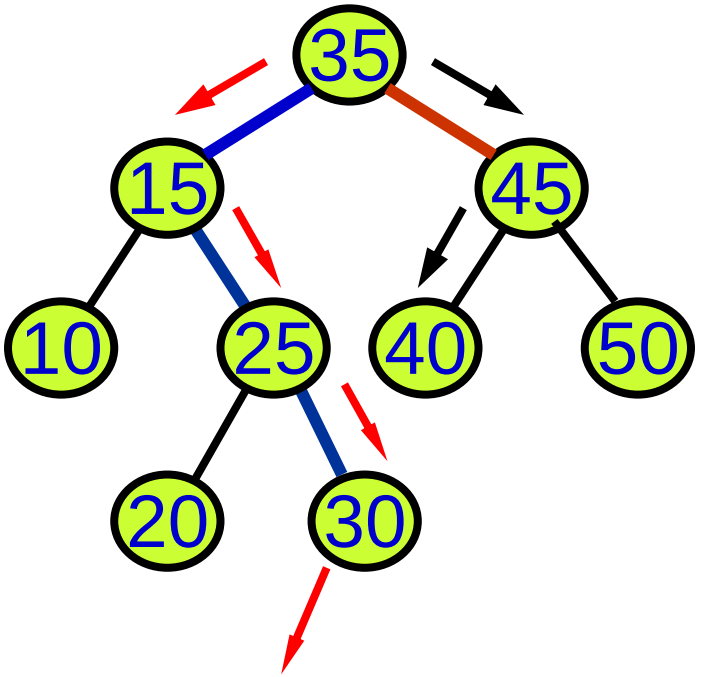
或者

从根结点出发，沿着左分支或右分支逐层向下直至指针指向空树为止。——查找不成功

在查找过程中，生成了一条查找路径。

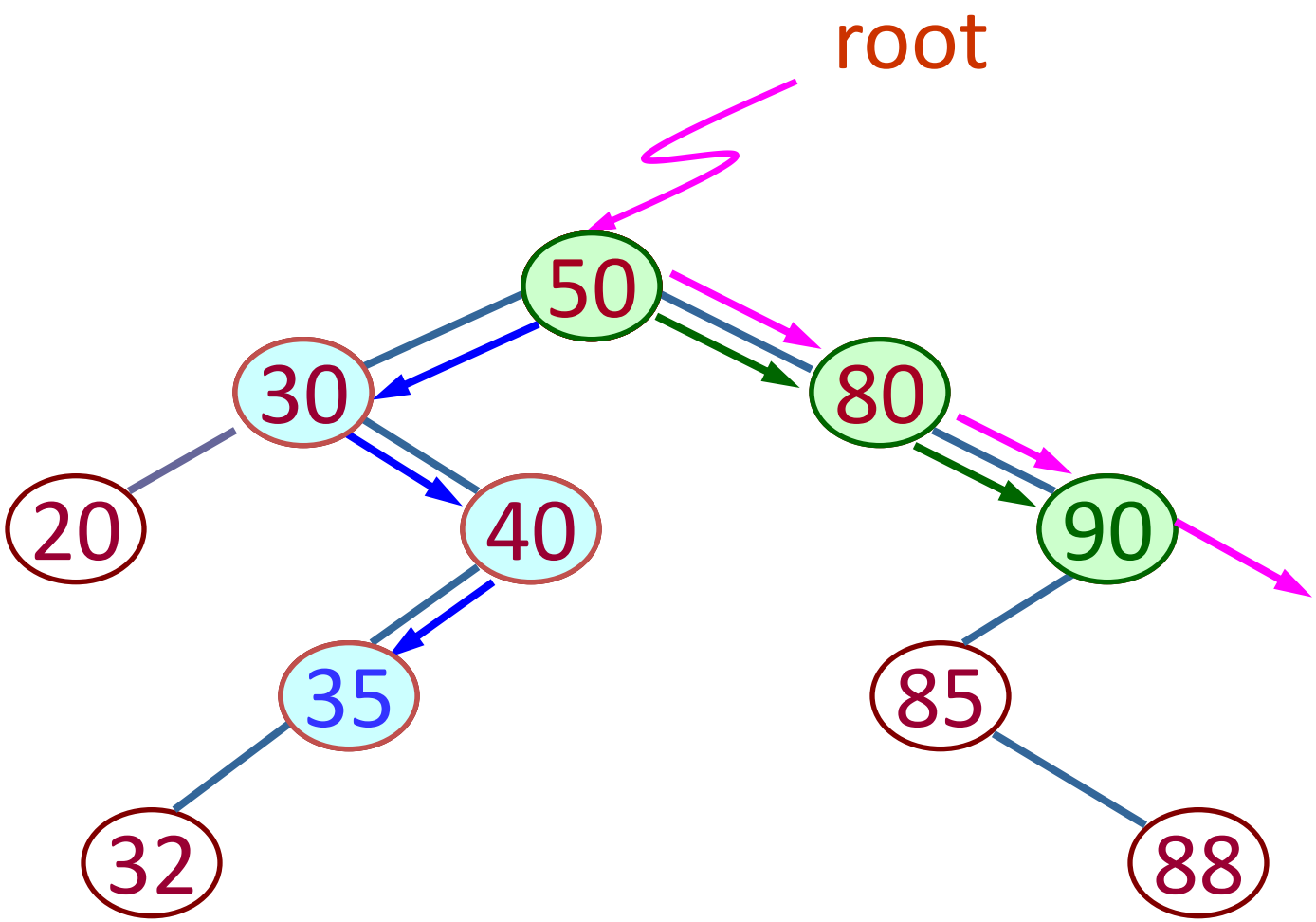
查找28

查找失败



查找40

查找成功



查找关键字

＝50，35，90，95

二叉排序树结点的删除

二叉排序树**结点的删除**，相当于删去有序序列上的一个记录，**应保证**删除结点后，**二叉排序树的特性不变**。

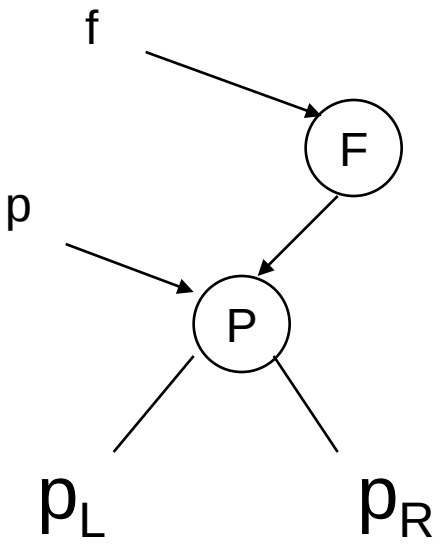
删除结点可有三种情况：

1.若删除结点为**叶子结点**，即其左子树和右子树均为空树。由于叶子结点的存在若不破坏整株树的结构，则只需**修改其父结点的指针**即可。

2.若删除结点只有左子树或只有右子树，此时只要令左子树或右子树的根结点直接代替被

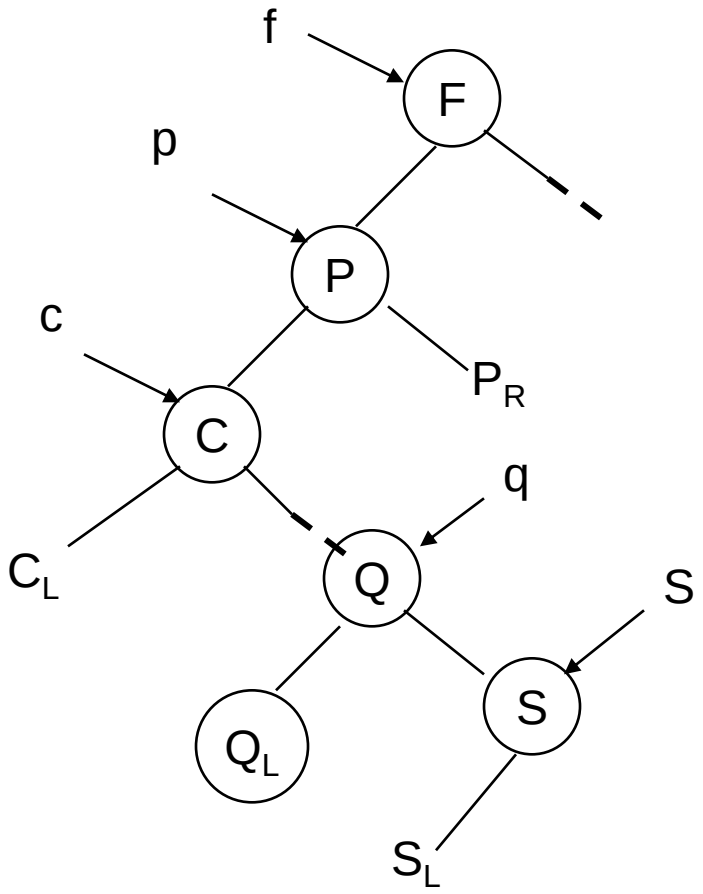
删除结点即可，也不破坏二叉排序树的特性。

3.若删除结点的左、右子树均不空，不能简单处理，可有两种情况。例：

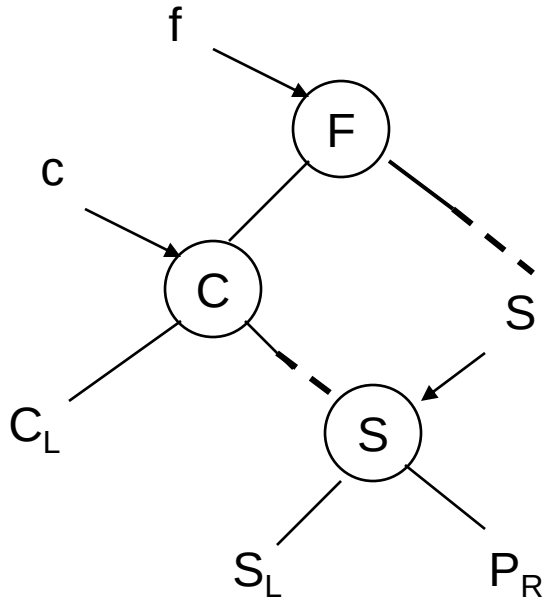


(a)

方法1: 将删除结点的右子树,作为删除结点的左子树的最右结点的右子树.

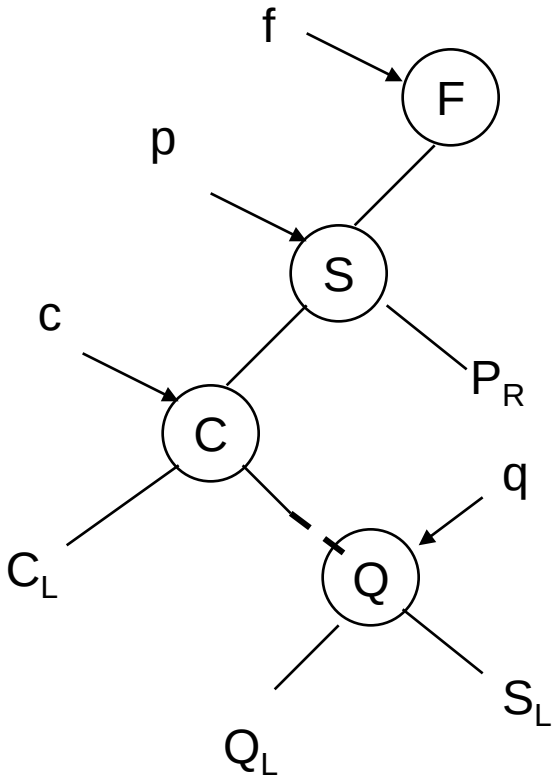


(b)



(c)

方法2: 用删除结点的左子树的最右结点代替删除结点,同时用删除结点的左子树的最右结点的左子树(如果有)代替该结点.(左子树最大结点取代,也可以右子树最小结点取代)



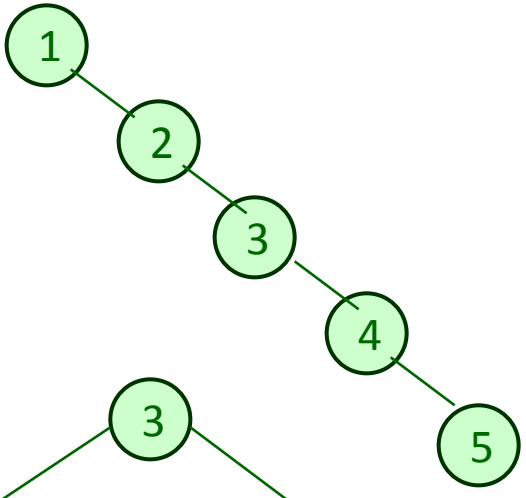
(d)

二叉排序树查找性能分析

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。但是，由值相同的 *n*个关键字，构造得到的不同形态的各棵二叉排序树的平均查找长度的值可能不同，甚至可能差别很大。

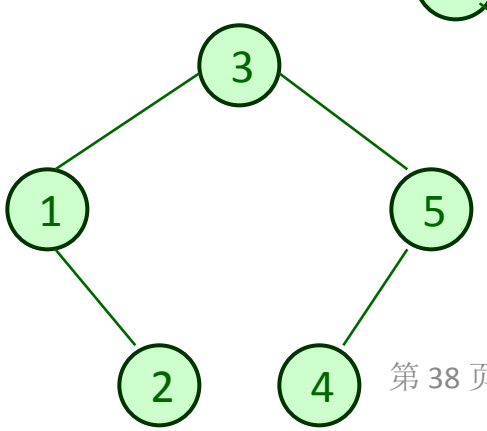
由关键字序列 {1，2，3，4，5}
构造而得的二叉排序树:

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

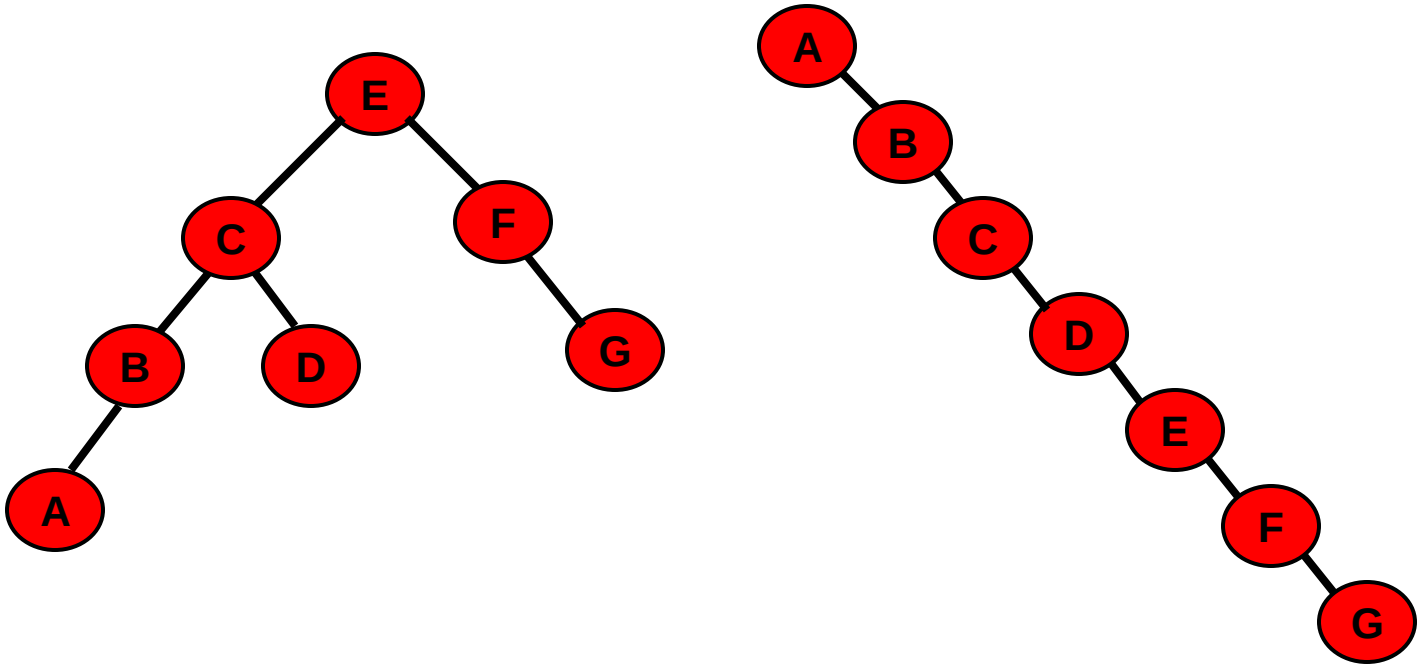


二叉排序树总结

- 就平均查找性能而言，二叉排序树查找和折半查找相差不大;
- 就维护表的有序性而言，二叉排序树更为有效（二叉排序树的结点插入、删除方便，无需移动大量结点）。因此，**适用于动态查找环境**。

平衡二叉树

• 起因：提高查找速度，避免最坏情况出现。如右图情况的出现。



• 平衡因子（平衡度）：结点的平衡度是结点的左子树的高度－右子树的高度。

• 平衡二叉树：每个结点的平衡因子都为 ＋1、－1、0 的二叉树。或者说每个结点的左右子树的高度最多差一的二叉树。

平衡二叉排序树——AV树

定义：

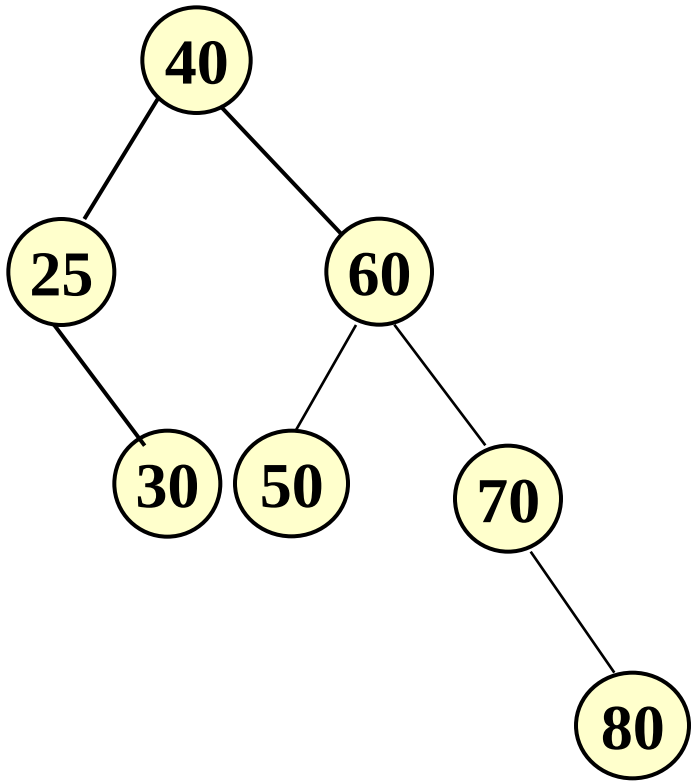
一棵平衡二叉排序树或者是空树，或者是具有下列性质的二叉排序树：

1、左子树与右子树的高度之差的绝对值小于等于1；

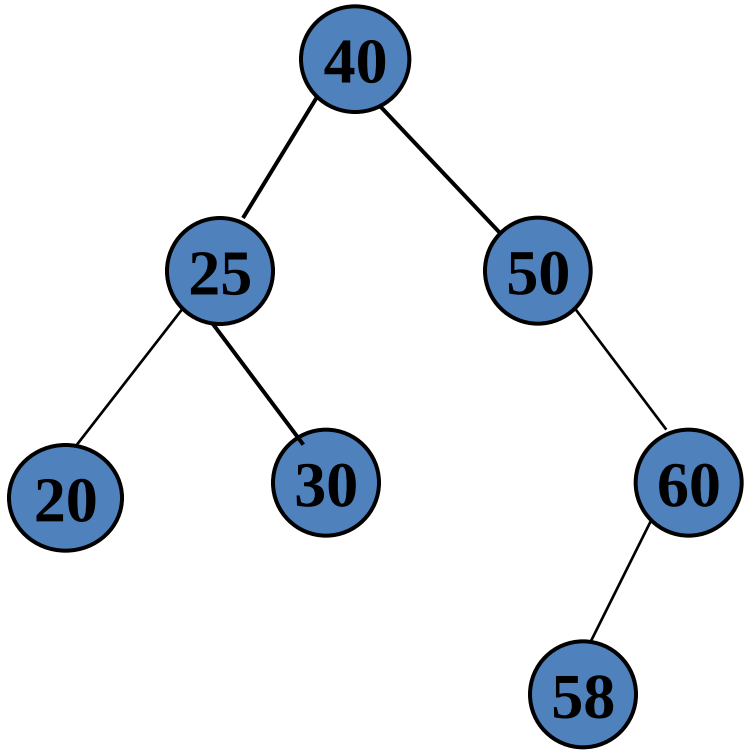
2、左子树和右子树也是平衡二叉排序树。

平衡二叉排序树的平均查找长度为O（$\log_2 n$）。

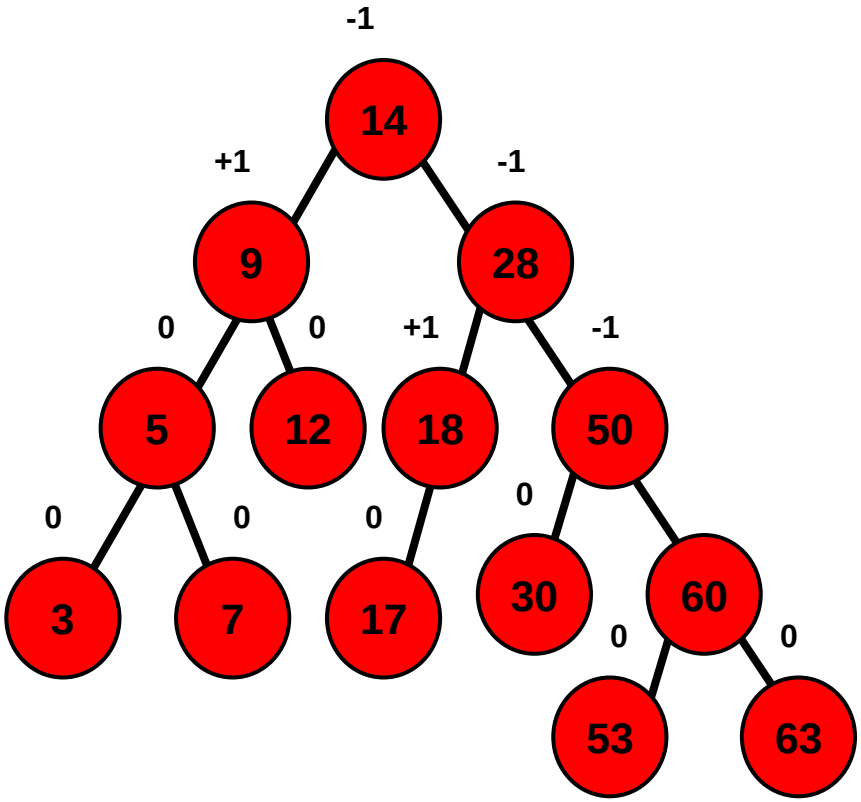
平衡因子：结点的左子树深度与右子树深度之差：-1，0，1。



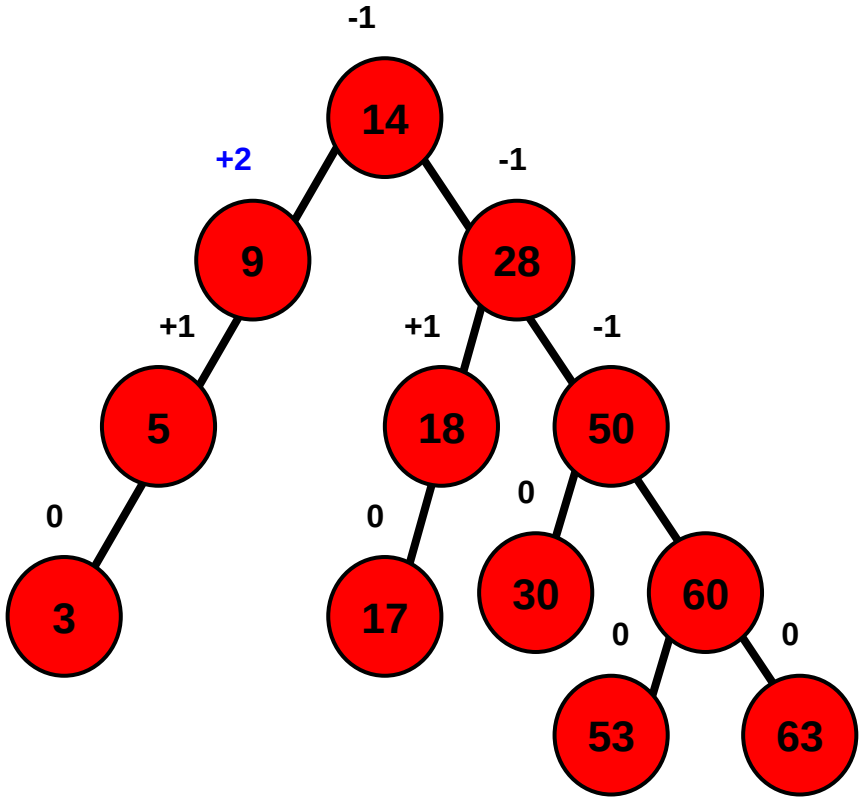
是平衡树



不是平衡树



是平衡树不是丰满树

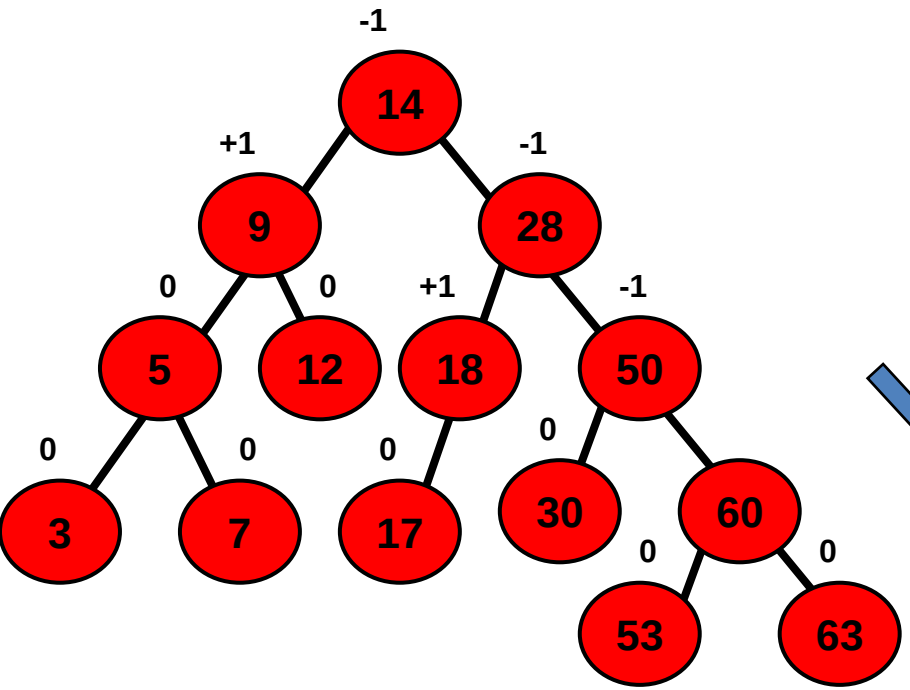


不是平衡树

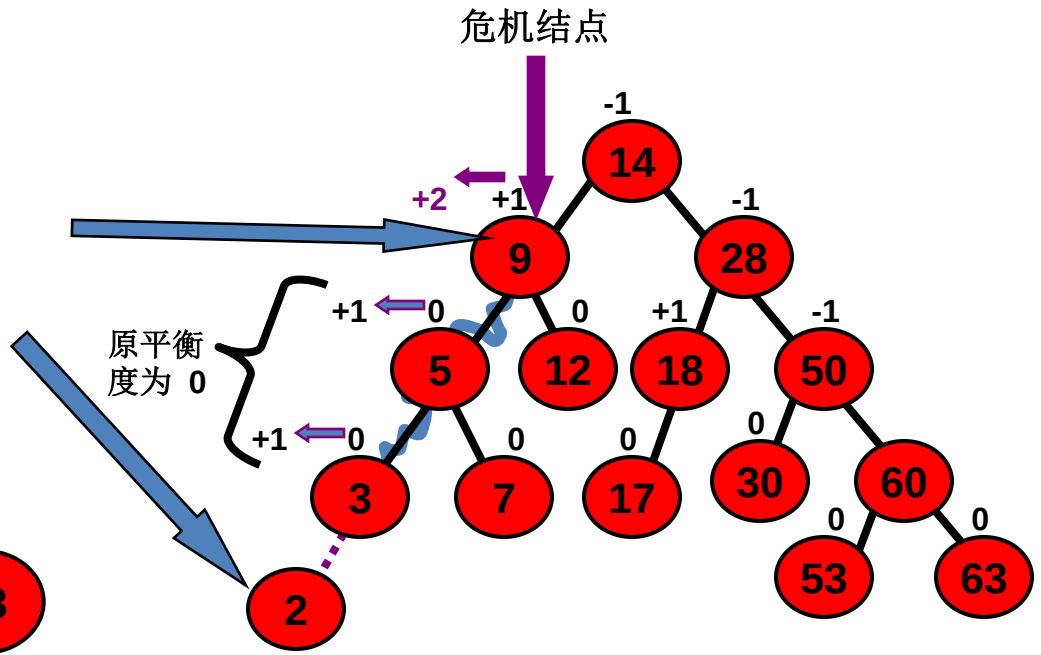
•以插入为例：

在左图所示的平衡树中插入数据为 2 的结点。

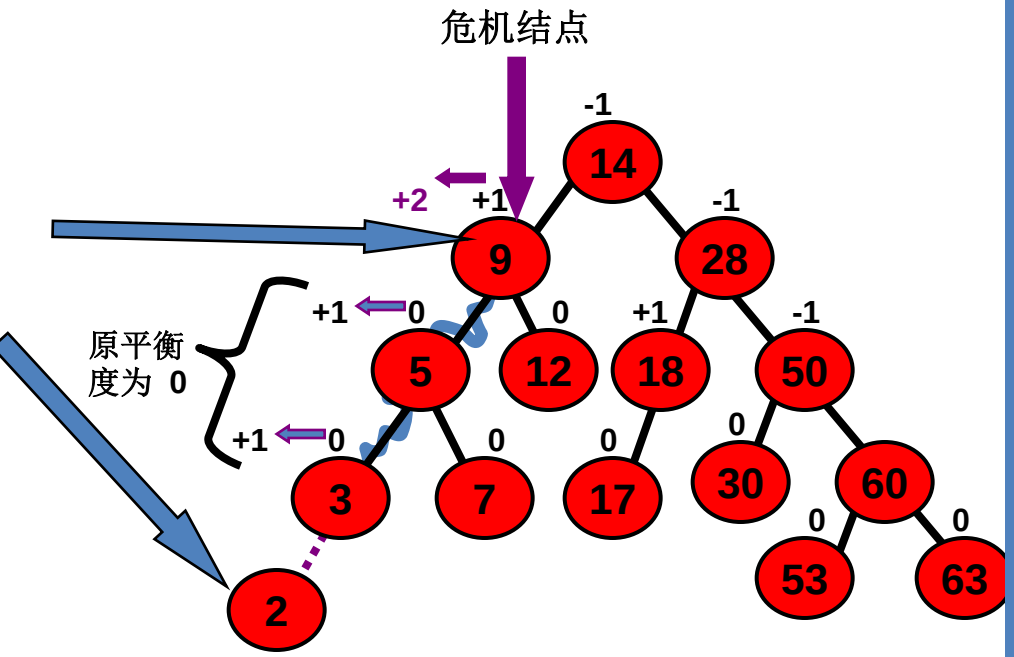
插入之后仍应保持平衡分类二叉树的性质不变。



平衡树



如何用最简单、最有效的办法保持平衡分类二叉树的性质不变？



如何用最简单、最有效的办法保持平衡分类二叉树的性质不变？

解决方案：

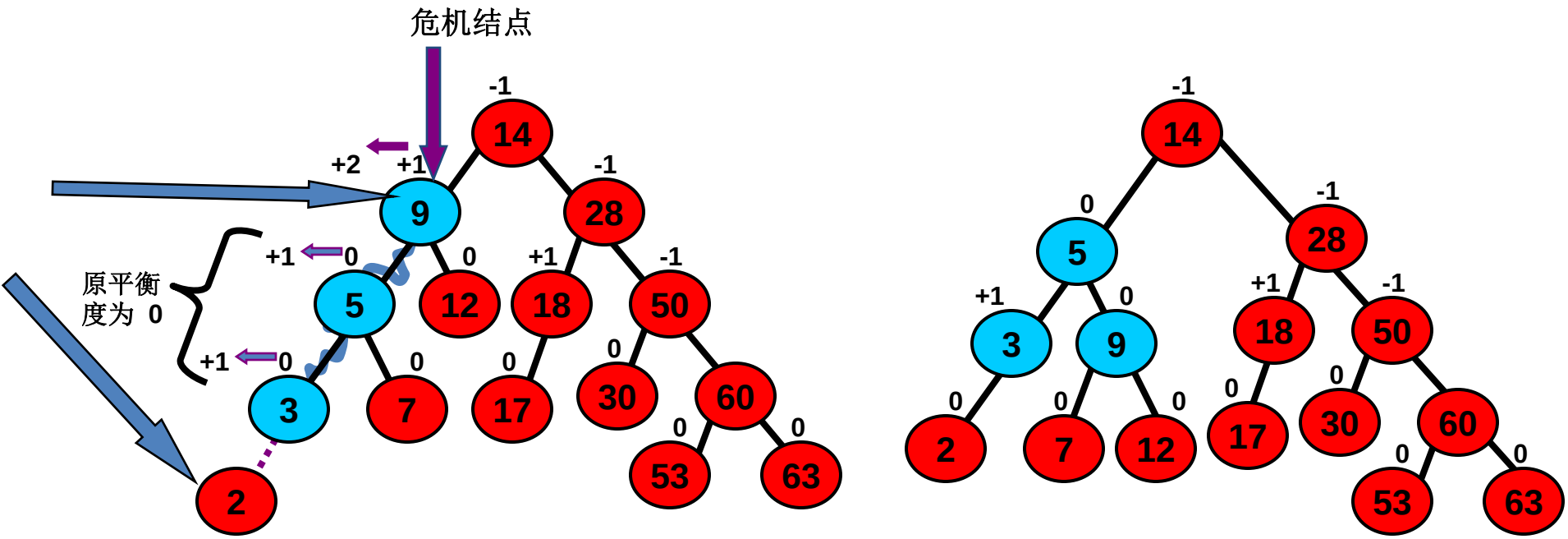
不涉及到危机结点的父亲结点，即以危机结点为根的子树的高度应保持不变（图为 3 ）。

新结点插入后，找到第一个平衡度不为 0 的结点（如左图结点 9）即可。新插入的结点和

第一个平衡度不为 0 的结点（也可能是危机结点）之间的结点，其平衡度皆为 0。

在调整中，仅调整那些在平衡度变化的路径上的结点（如：3 5 9），其它结点不予调整。

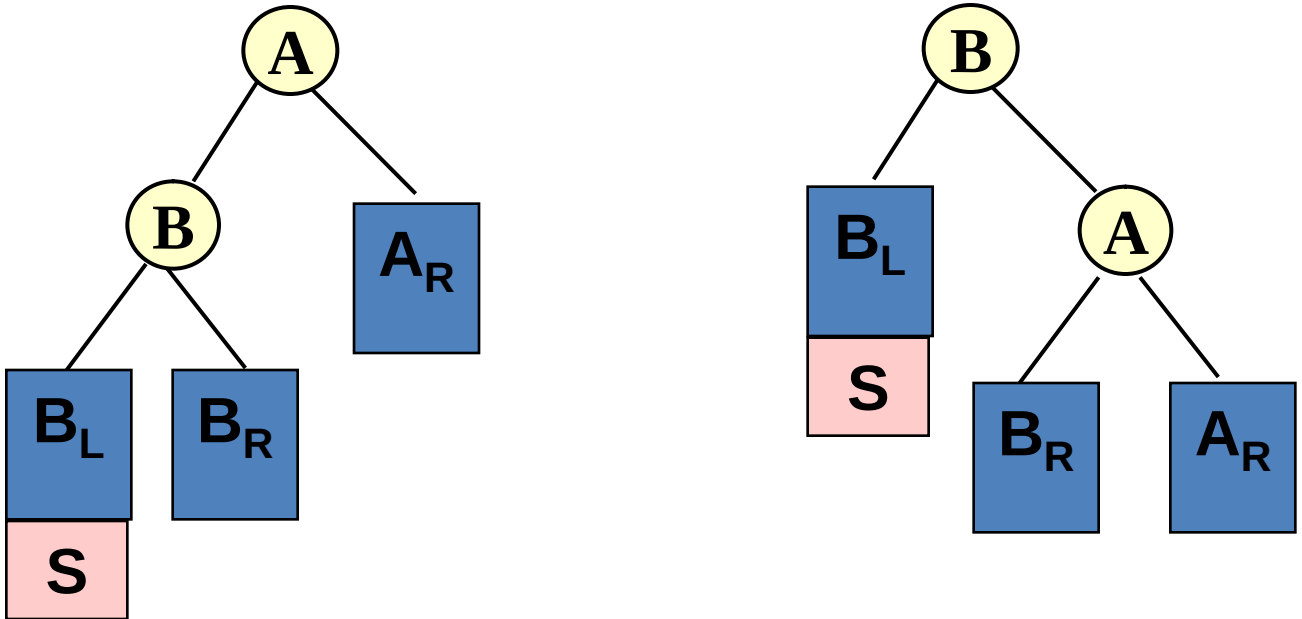
仍应保持二叉排序树的性质不变。



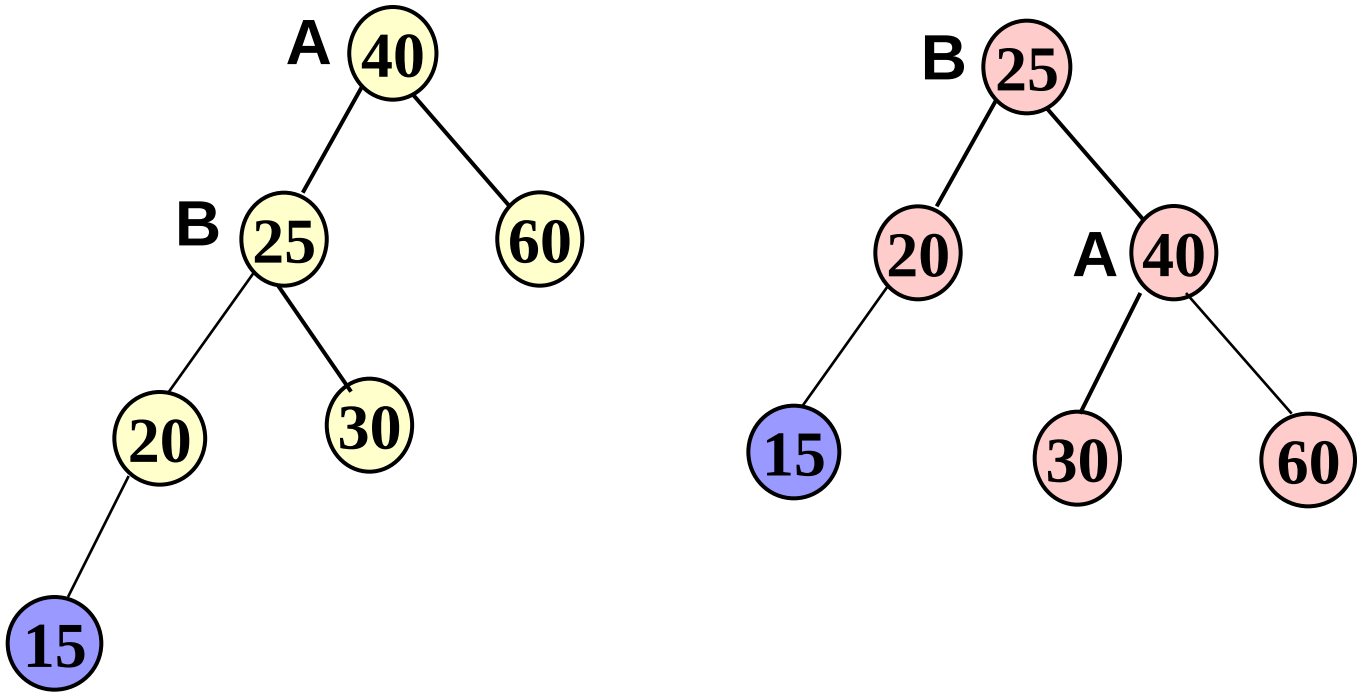
关键：将导致出现危机结点的情况全部分析清楚，就可以使得平衡分类二叉树的性质保持不变！！

不平衡二叉排序树的调整——LL型

最低层失衡结点为A，在结点A的左子树的左子树上插入新结点S后，导致失衡，由A和B的平衡因子可知，B_L、B_R和A_R深度相同，为恢复平衡并保持二叉排序树的特性，可将A改为B的右子，B原来的右子B_R改为A的左子，这相当于以B为轴，对A做了一次顺时针旋转。



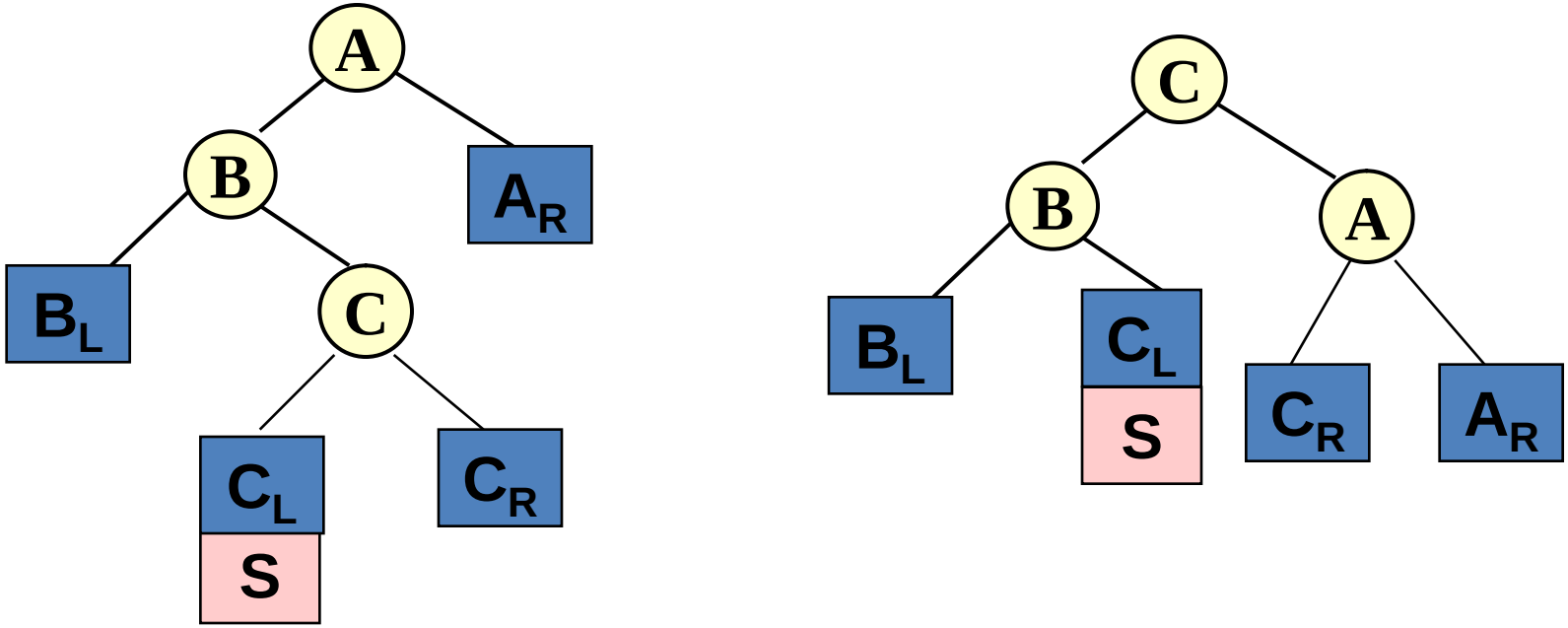
不平衡二叉排序树的调整——LL型



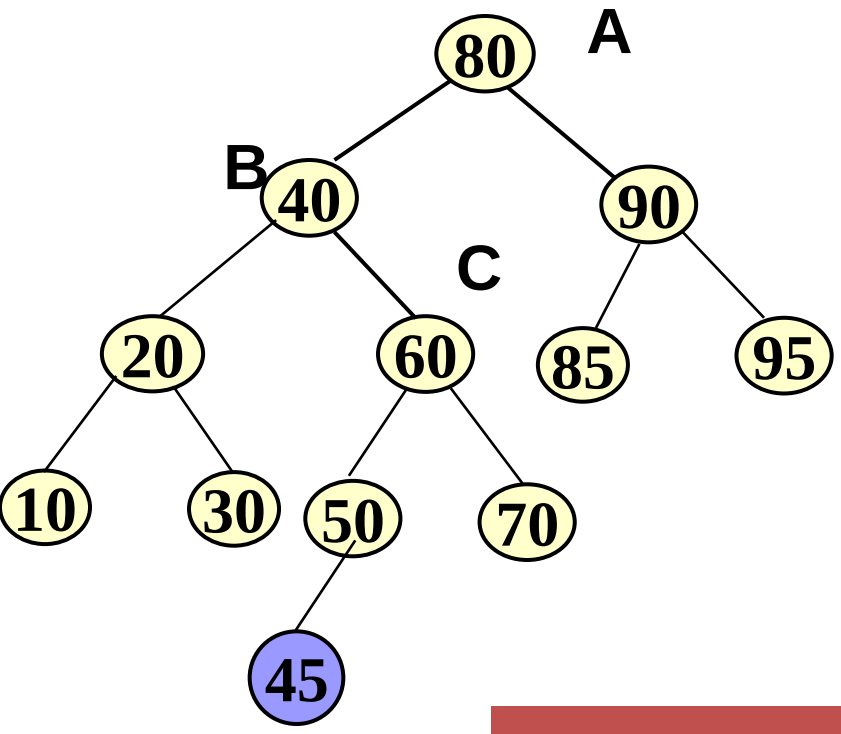
A->lchild=B->rchild
B->rchild=A

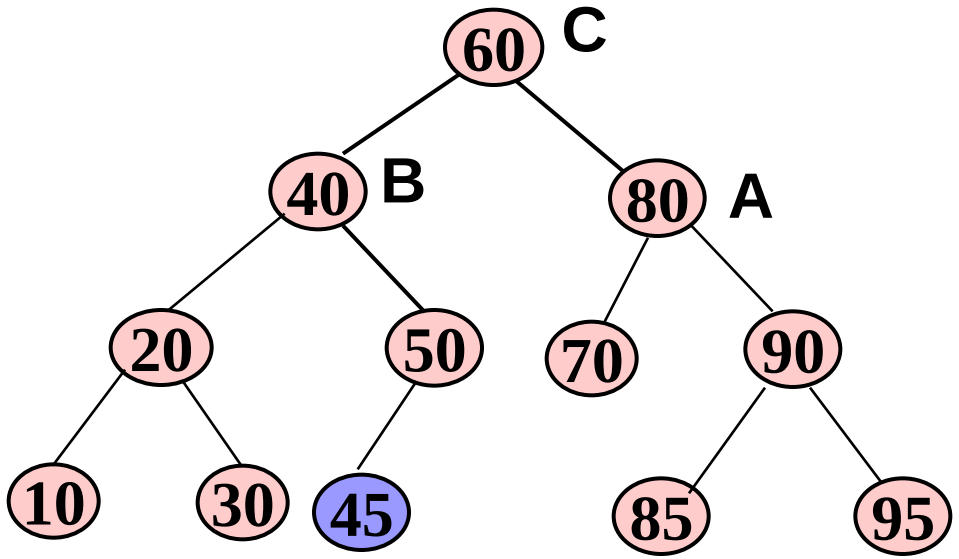
不平衡二叉排序树的调整——LR型

最低层失衡结点为A，在结点A的左子树的右子树上插入新结点S后，导致失衡，假设在C_L下插入S，由A、B、C的平衡因子可知，C_L与C_R深度相同，B_L与A_R深度相同，且B_L、A_R的深度比C_L、C_R的深度大1；为恢复平衡并保持二叉排序树的特性，可将B改为C的左子，而C原来的左子C_L改为B的右子，然后将A改为C的右子，C原来的右子C_R改为A的左子；这相当于以B为轴，对A做了一次顺时针旋转。



不平衡二叉排序树的调整——LR型

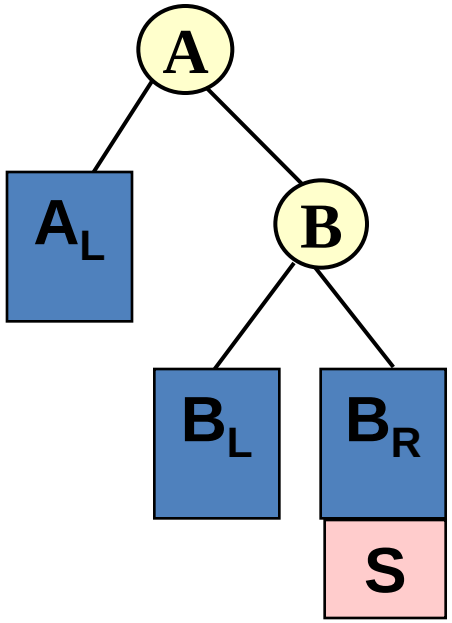


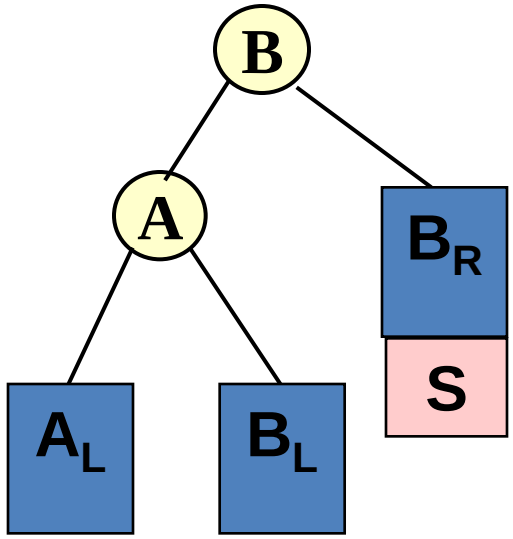


A->lchild=c->rchild
B->rchild=c->lchild
c->rchild=A
c->lchild=B

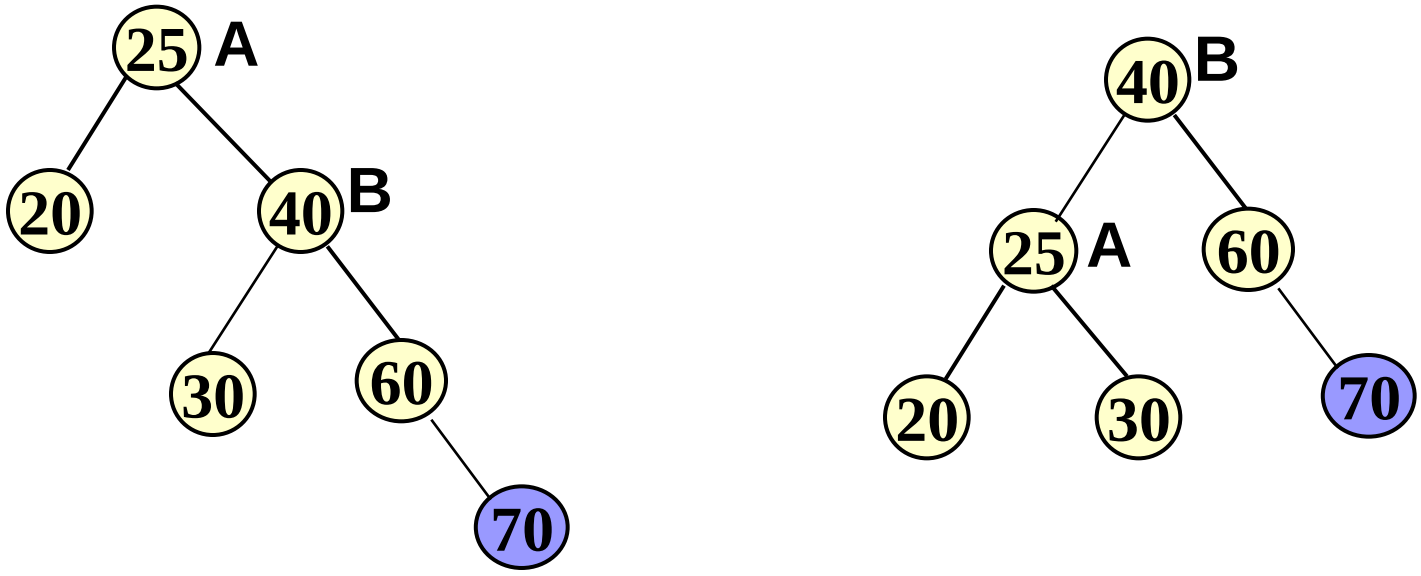
不平衡二叉排序树的调整——RR型

最低层失衡结点为A，在结点A的右子树的右子树上插入新结点S后，导致失衡，由A和B的平衡因子可知，B_L、B_R和A_L深度相同，为恢复平衡并保持二叉排序树的特性，可将A改为B的左子，B原来的左子B_L改为A的右子，这相当于以B为轴，对A做了一次逆时针旋转。





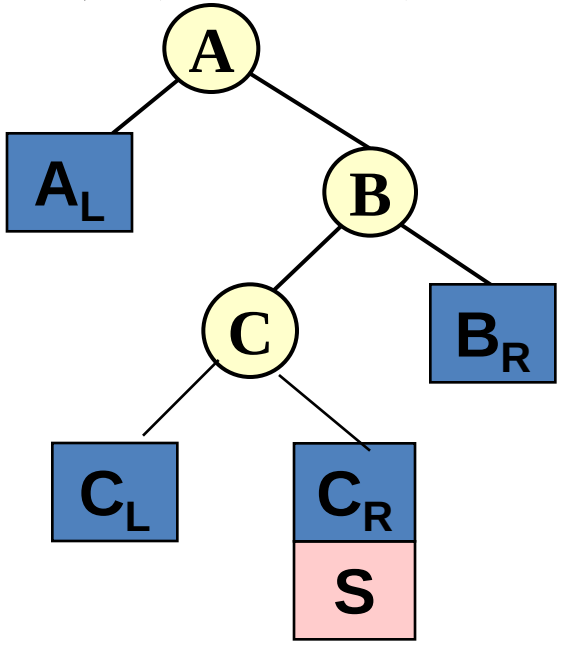
不平衡二叉排序树的调整——RR型

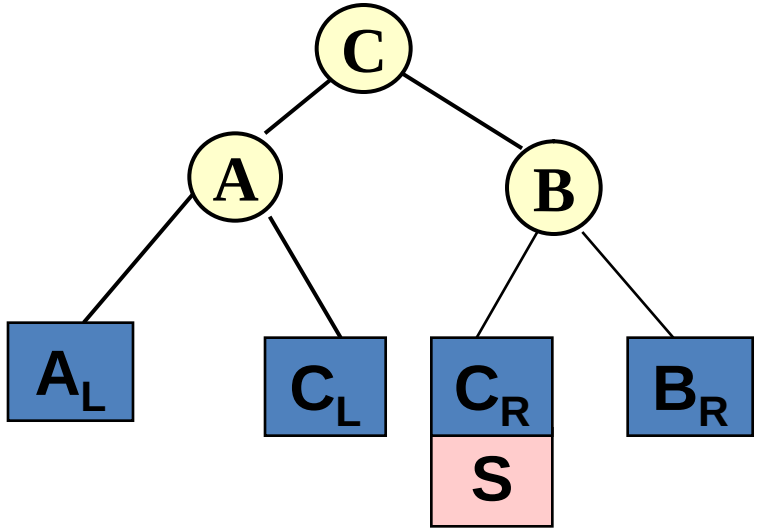


A->rchild=B->lchild
B->lchild=A

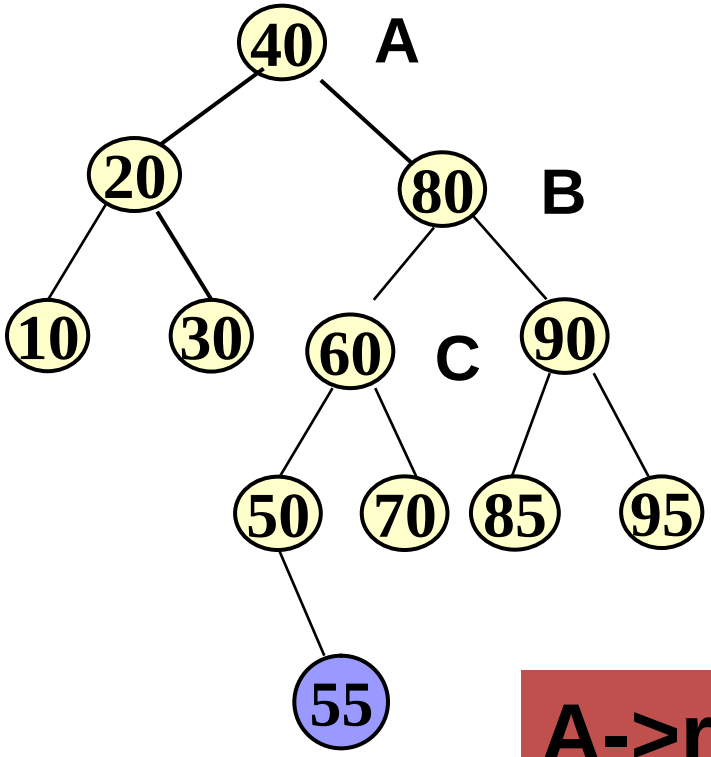
不平衡二叉排序树的调整——RL型

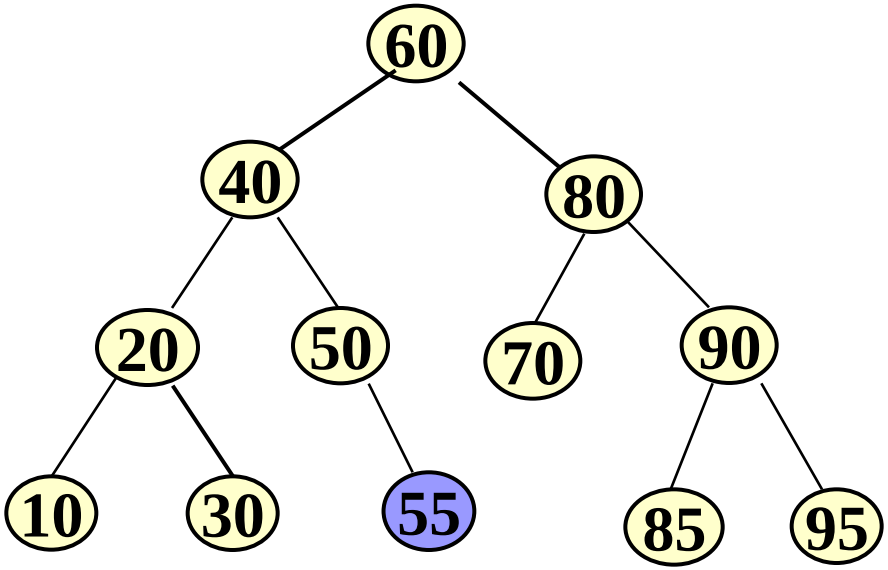
最低层失衡结点为A，在结点A的右子树的左子树上插入新结点S后，导致失衡，假设在C_R下插入S，由A、B、C的平衡因子可知，C_L与C_R深度相同，A_L与B_R深度相同，且A_L、B_R的深度比C_L、C_R的深度大1；为恢复平衡并保持二叉排序树的特性，可将B改为C的右子，而C原来的右子C_R改为B的左子，然后将A改为C的左子，C原来的左子C_L改为A的右子；这相当于以B为轴，对A做了一次顺时针旋转。





不平衡二叉排序树的调整——RL型





A->rchild=c->lchild
B->lchild=c->rchild
c->lchild=A
c->rchild=B

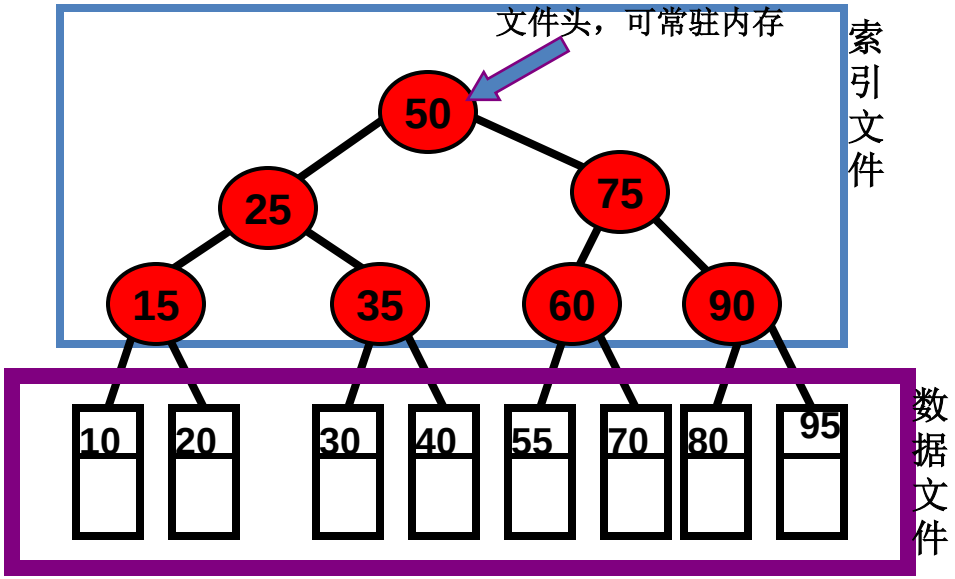
B_ 树

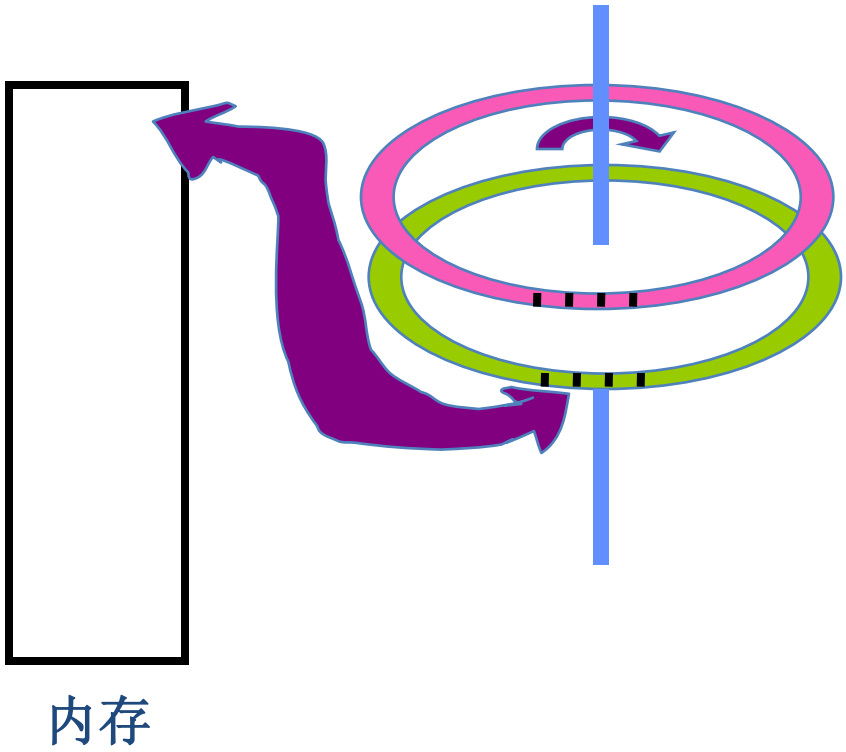
1、引入

大量数据存放在外存中，通常存放在硬盘中。由于是海量数据，不可能一次调入内存。因此，要多次访问外存。但硬盘的驱动受机械运动的制约，速度慢。所以，主要矛盾变为减少访外次数。

在 **1970** 年由 **R bayer** 和 **E macreight** 提出用**B_** 树作为索引组织文件。提高访问速度、减少时间。

用二叉树组织文件，当文件的记录个数为 **100，000**时，要找到给定关键字的记录，需访问外存**17**次（**log100,000**）,太长了！





文件访问示意图：索引文件、数据文件存在盘上

m 阶 B_ 树的 定义

定义：一棵m阶的B-树，或者为空树，或为满足下列特性的m叉树：

⑴树中每个结点至多有m棵子树；

⑵若根结点不是叶子结点，则至少有两棵子树；

⑶除根结点之外的所有非终端结点至少有⌈m/2⌉棵子树；

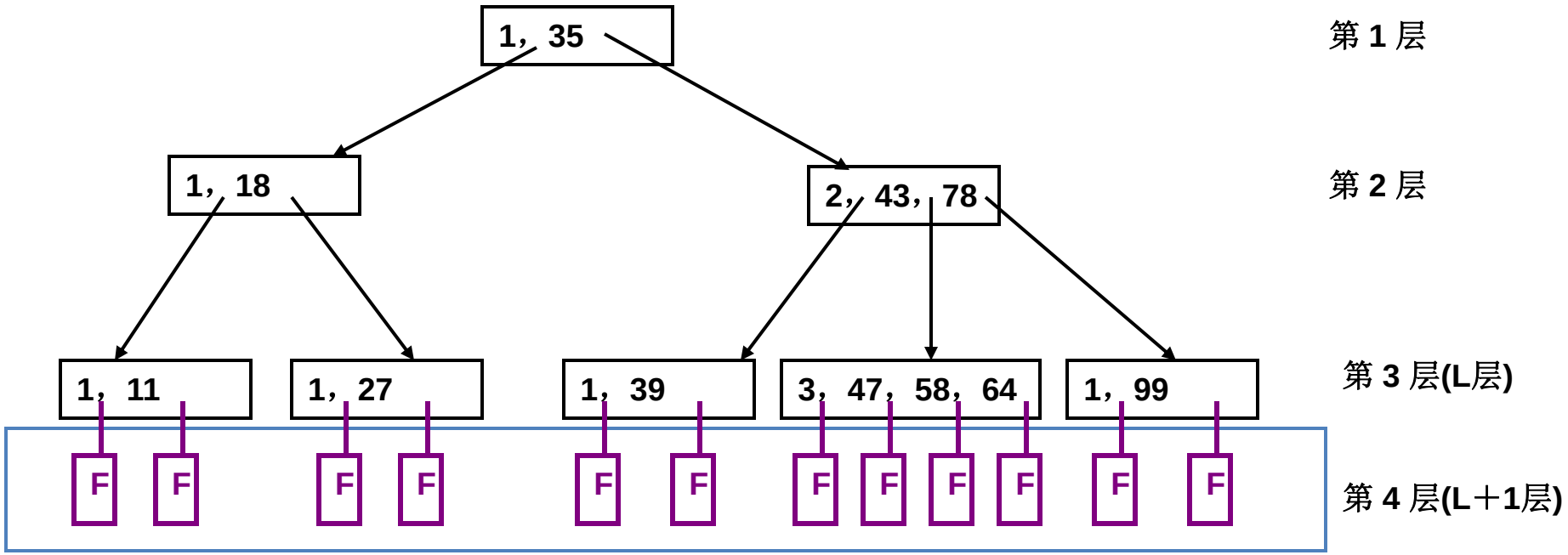
⑷所有的非终端结点中包含以下信息数据：（n，A0，K1，A1，K2，…，Kn，An）

其中：Ki（i=1,2,…,n）为关键码，且Ki<Ki+1，Ai为指向子树根结点的指针(i=0,1,…,n)，且指针Ai-1所指子树中所有结点的关键码均小于Ki (i=1,2,…,n)，An所指子树中所有结点的关键码均大于Kn，⌈m/2⌉–1≤n≤m –1 ，n为关键码的个数。

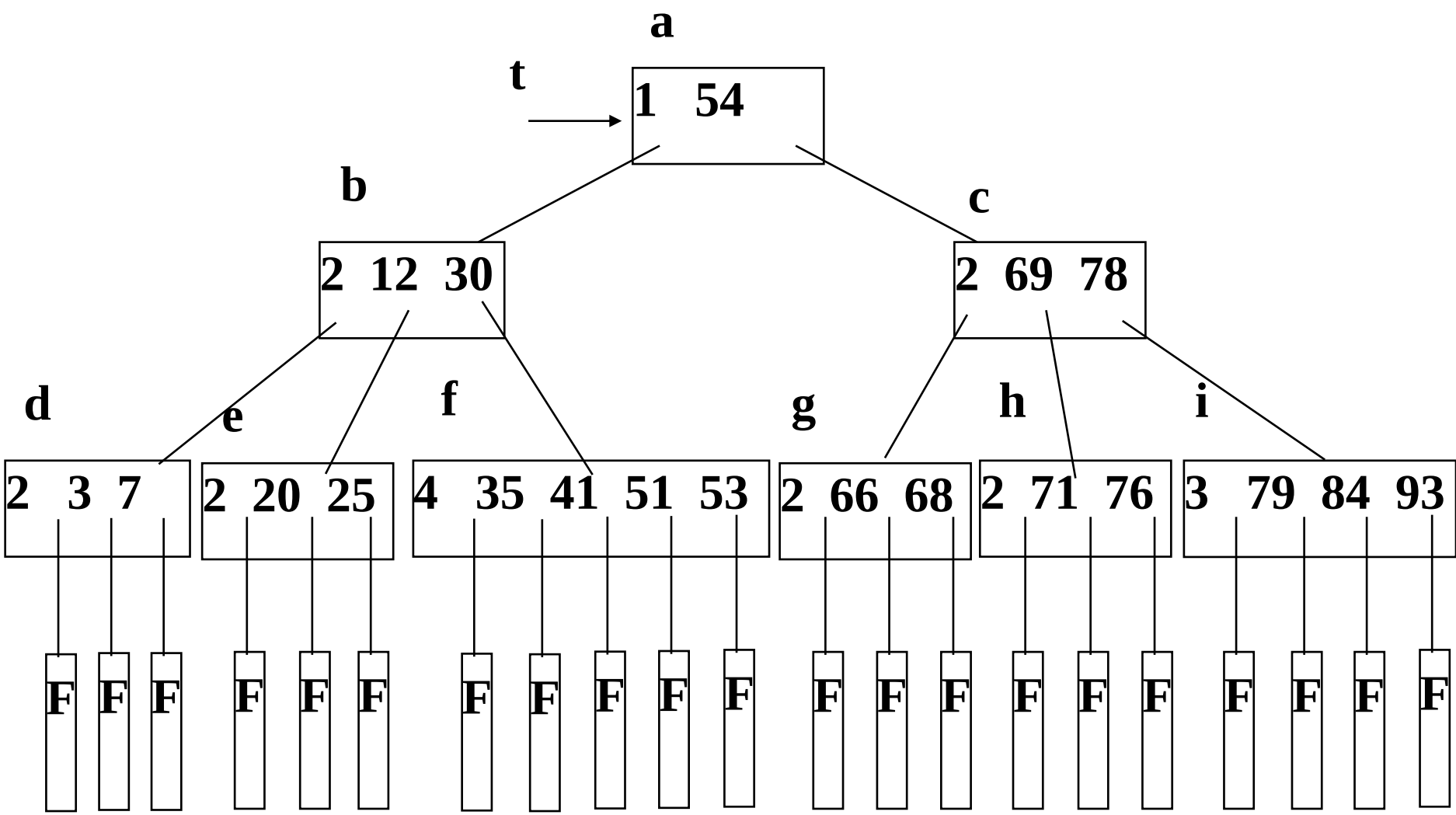
⑸所有的叶子结点都出现在同一层次上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

例如：m = 4 阶 B_ 树。除根结点和叶子结点之外，每个结点的儿子个数至少为⌈m/2⌉= 2 个；结点的关键字个数至少为 1 。该 B_ 树的深度为 4。

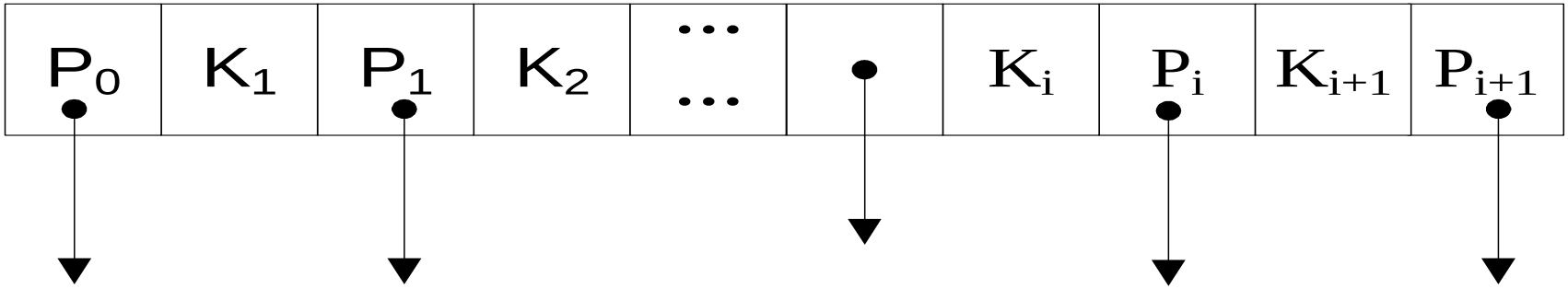
叶子结点都在第 4 层上。



一棵5阶的B-树，其深度为4



B树(续)



B树的查找

1、首先找到根结点,与根结点各键值进行比较，如果 $K_x<K_1$则查找P_0指向的结点；如果 $K_x>K_{i+1}$则查找P_{i+1}指向的结点；如果$K_i<K_x<K_{i+1}$则查找P_i指向的结点。

2、如果顺着某个指针往下找时遇到树叶，则说明K_x不在B树中，即查找失败。

3、在下一个结点查找时，重复1

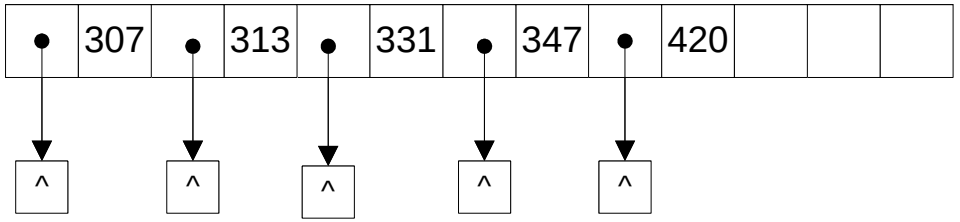
B树(续)

B树的插入

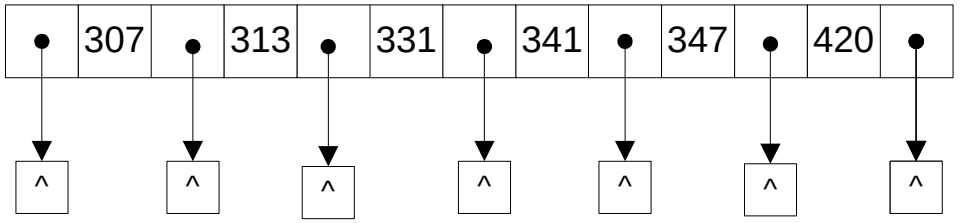
- 在往m阶B树中插入一个关键字（key）时，不是在树中添加一个树叶，而是首先在最底层的某个非终端结点中添加一个关键字，若插入后该结点的关键字数目不超过m-1,则插入完毕；否则要进行分裂结点操作，并且这个分裂过程可能会一直波及到根结点。
- 设某个结点已有m-1个关键字，那么，在插入一个关键字后，该结点将分裂成两个结点：左边一个结点包含前 $\lceil m/2 \rceil - 1$ 个关键字，右边一个结点包含后m-[m/2]个关键字，而第[m/2]个关键字被插入到上一级（双亲）结点中。

阶为7的B树

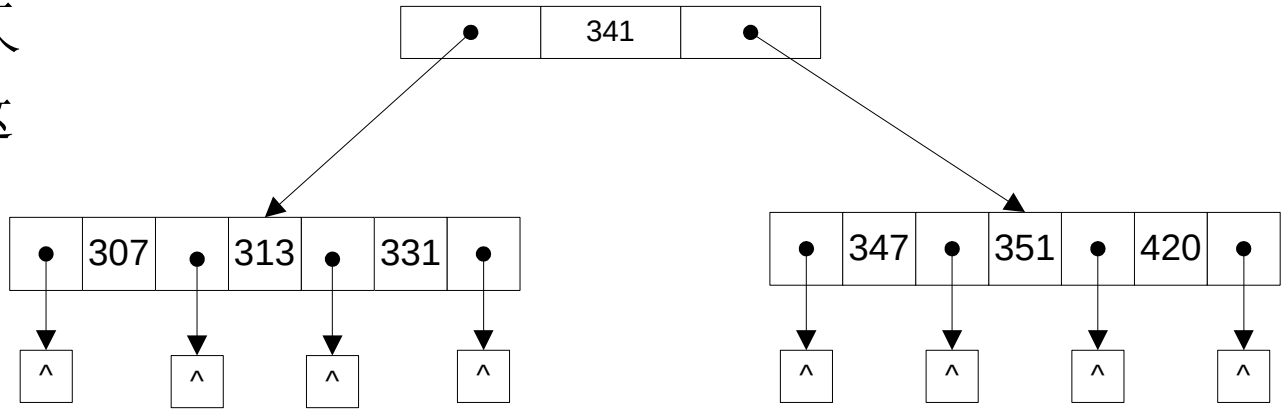
一般来说,要把一个新项目插入到阶为m的B树中处,当所有叶都在l级上,则把新的键插入到l-1,如果该结点现在含有m个键,则把它分为两个结点,并且把键$K_{[m/2]}$插入到原来结点的父亲处(于是,父亲处结点的指针p为序列P,$K_{[m/2]}$,P`所替代).如果需要分裂根结点,根结点是没有父亲的,则只需建立包含单个键$K_{[m/2]}$的一个新根结点和两个指针,这样树就长高了一级.



插入341



插入351



B树(续)

B树的删除

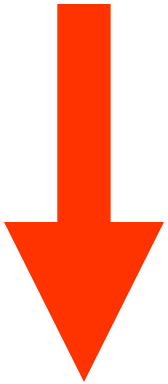
- 1、如果要删除的关键字不在最下面一层的非终端结点中，则可先把待删关键字与它在B树中的后继对换位置，然后删除关键字
- 2、如果要删除的关键字在最下面一层的非终端结点中，则把它从所在的结点中删除时，可能会导致此结点中关键字少于$\lceil m/2 \rceil - 1$(即它的孩子结点少于[m/2]个)因此，需要进行结点的合并操作。
- 结点的合并操作可以根据以下两种情况分别进行。

- (1)若被删出关键字所在结点中正好有 $\lceil m/2 \rceil - 1$ 个关键字而与该结点相邻的左兄弟（或右兄弟）结点中的关键字多于$\lceil m/2 \rceil - 1$ 个则可将其兄弟结点中最大（或最小）的关键字移到双亲结点中，而将双亲结点中该上移关键字的后面一个（或前面一个）关键字下移到被删除关键字所在的结点中。
- (2)若被删除关键字所在的结点和其相邻的兄弟结点中的关键字都是$\lceil m/2 \rceil - 1$ 个，则可将删除了关键字的结点和它的双亲结点的一个关键字合并到它的兄弟结点中。如果因此使双亲结点中的关键字少于 $\lceil m/2 \rceil - 1$ 个，则需要继续进行合并。

1.4 哈希表技术

- 以上讨论的表示查找表的各种结构的共同特点：记录在表中的**位置和它的关键字**之间**不存在**一个**确定的关系**，
- 查找的过程为给定值依次和关键字集合中各个关键字进行比较，
- 查找的**效率**取决于和给定值进行**比较的关键字个数**。
- 用这类方法表示的查找表，其**平均查找长度都不为零**。

- 问题：对于频繁使用的查找，我们希望平均查找长度ASL接近于0



只有一个办法

①预先知道所查关键字在表中的位置

②记录在表中位置和其关键字之间存在一种确定的关系

- 例如：为每年招收的 1000 名新生建立一张查找表，其关键字为学号，其值的范围为 xx000 ~ xx999 (前两位为年份)。

- 若以下标为000 ~ 999 的顺序表表示学号
- 查找过程可以简单进行：
  - 取给定值（学号）的后三位，不需要经过比较便可直接从顺序表中找到待查关键字。

- 哈希查找的基本思想：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在表中的存储位置之间建立一个函数关系，以 f(key) 作为关键字为 key 的记录在表中的位置，通常称这个函数 f(key) 为**哈希函数**。
- 哈希地址：由哈希函数求出的记录存储位置称为哈希地址，表示成：$addr(a_i)=f(k_i)$
  - ai是表中的一个元素(记录)
  - $addr(a_i)$是a_i的存储地址
  - k_i是a_i的关键字

- 哈希表：应用哈希函数，由记录的关键字确定记录在表中的地址，并将记录放入此地址，这样构成的表叫哈希表，也叫散列表。
- 哈希查找：又叫散列查找，利用哈希函数进行查找的过程叫~

例：对于如下 9 个关键字

{Zhao, Qian, Sun, Li, Wu, Chen, Han, Ye, Dai}

设哈希函数

f(key) = ⌊(ASC(关键字第一个字母) -ASC('A')+1)/2⌋

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	Chen	Dai		Han		Li		Qian	Sun		Wu	Ye	Zhao

问题：若添加关键字 Zhou，怎么办？

能否找到另一个哈希函数？

- 哈希函数是一个映象，即 将关键字的集合映射到某个地址集合上，它的设置很灵活，只要这个地址集合的 大小不超出允许范围即可；
- 由于哈希函数是一个压缩映象，因此，在一般情况下，很容易产生“**冲突**”现象，
  - 即： key1<> key2，
  - 而 f(key1) = f(key2)
- 很难找到一个不产生冲突的哈希函数。一般情况下，只能选择恰当的哈希函数，使冲突尽可能少地产生。
- 在构造哈希表时，除了需要选择一个“好”(尽可能少产生冲突)的哈希函数之外；还需要找到一种处理冲突 的方法。

构造哈希函数的几种方法

对数字的关键字可有下列构造方法

1. 直接定址法
2. 数字分析法
3. 平方取中法
4. 折叠法
5. 除留余数法
6. 随机数法

若是非数字关键字，则需先对其进行数字化处理

(1) 直接定址法

- 构造：取关键字或关键字的某个线性函数作哈希函数

  H(key) = key

  或者　　H(key) = a. key + b

- 特点
  - 直接定址法所得地址集合与关键字集合大小相等，不会发生冲突
  - 实际中能用这种哈希函数的情况很少

(2) 数字分析法

- 构造：对关键字进行分析，取关键字的若干位或其组合作哈希地址
- 特点：适于关键字位数比哈希地址位数大，且可能出现的关键字事先知道的情况

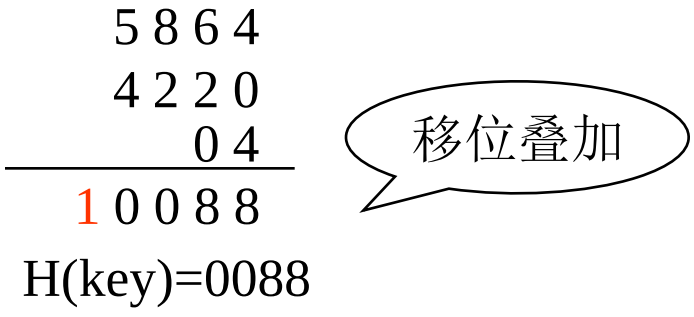
(3) 平方取中法

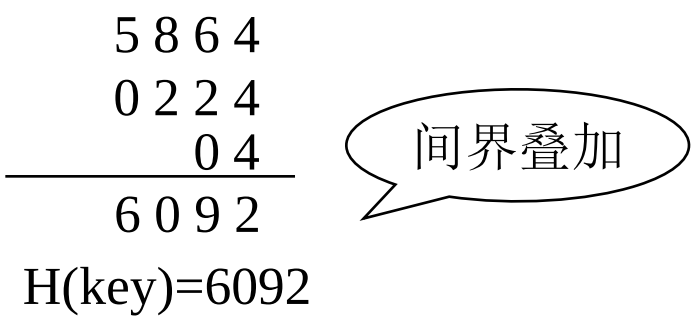
- 构造：以关键字的平方值的中间几位作为存储地址。求“关键字的平方值”的目的是“扩大差别”，同时平方值的中间各位又能受到整个关键字中各位的影响。
- 特点：关键字中的每一位都有某些数字重复出现频度很高的现象。

(4) 折叠法

- 构造：将关键字分割成位数相同的几部分，然后取这几部分的叠加和（舍去进位）做哈希地址
- 种类
  - 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
- 特点：适于关键字位数很多，且每一位上数字分布大致均匀情况

例　关键字为：0442205864，哈希地址位数为4





(5) 除留余数法

- 构造：取关键字被某个不大于哈希表表长m的数p除后所得余数作哈希地址，即

  H(key)=key % p

  p<=m 且P 一般应为接近 m 的素数或是不含20以下的质因子

- 特点
  - 简单、常用: 可与上述几种方法结合使用
  - p的选取很重要: p选的不好，容易产生同义词

对P要加一定的限制

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

假定hash表的长度为12

若取 p=9, 则他们对应的哈希函数值将为:

3, 3, 0, 6, 6, 3

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。

(6) 随机数法

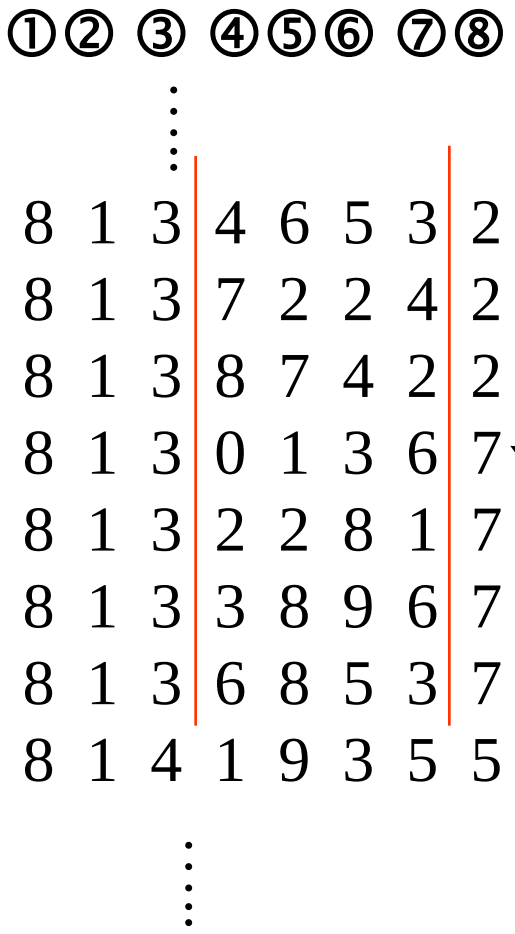
- 构造：取关键字的随机函数值作哈希地址

  H(key)=random(key)

- 适于关键字长度不等的情况

实际造表时，采用何种构造哈希函数的方法取决于建表的关键字集合的情况(包括关键字的范围和形态)，**总的原则是使产生冲突的可能性降到尽可能地小。**

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



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

处理冲突的方法

- 处理冲突：为产生冲突的地址寻找下一个哈希地址。
- 主要有二种方法
  1. 开放定址法
  2. 链地址法

(1) 开放定址法

- 思想：为产生冲突的关键字寻找一个新的地址 $H_i(key)$， 求得一个地址序列：

$H_0, H_1, H_2, \ldots, H_s \quad 1 \le s \le m-1$

其中：$H_0 = H(key)$

$H_1 = (H(key) + d_1) \% m$

$H_2 = (H(key) + d_2) \% m$

……

$H_i = (H(key) + d_i) \% m$

$i=1, 2, \ldots, s$

- 对增量 d_i 有三种取法
  - 线性探测再散列：d_i=1,2,3,……m-1
  - 二次探测再散列：d_i=1^2,-1^2,2^2,-2^2,3^2,…,±k^2
  - 伪随机探测再散列：d_i=伪随机数序列
- 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个 H_i 值能覆盖哈希表中所有地址。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 % 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

0	1	2	3	4	5	6	7	8	9	10
			38	38	60	17	29	38		

线性探测

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

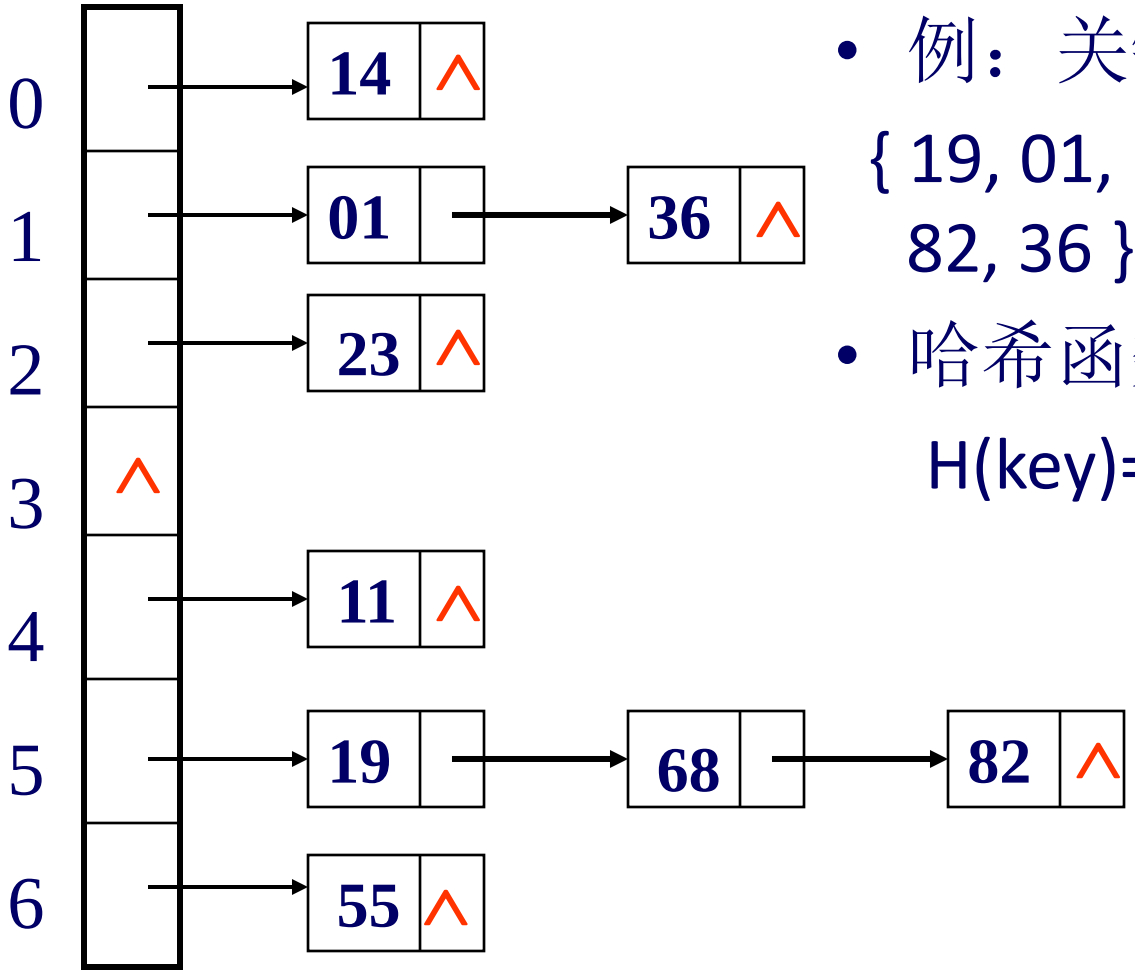
二次探测

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

随机探测

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

(2) 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：关键字集合

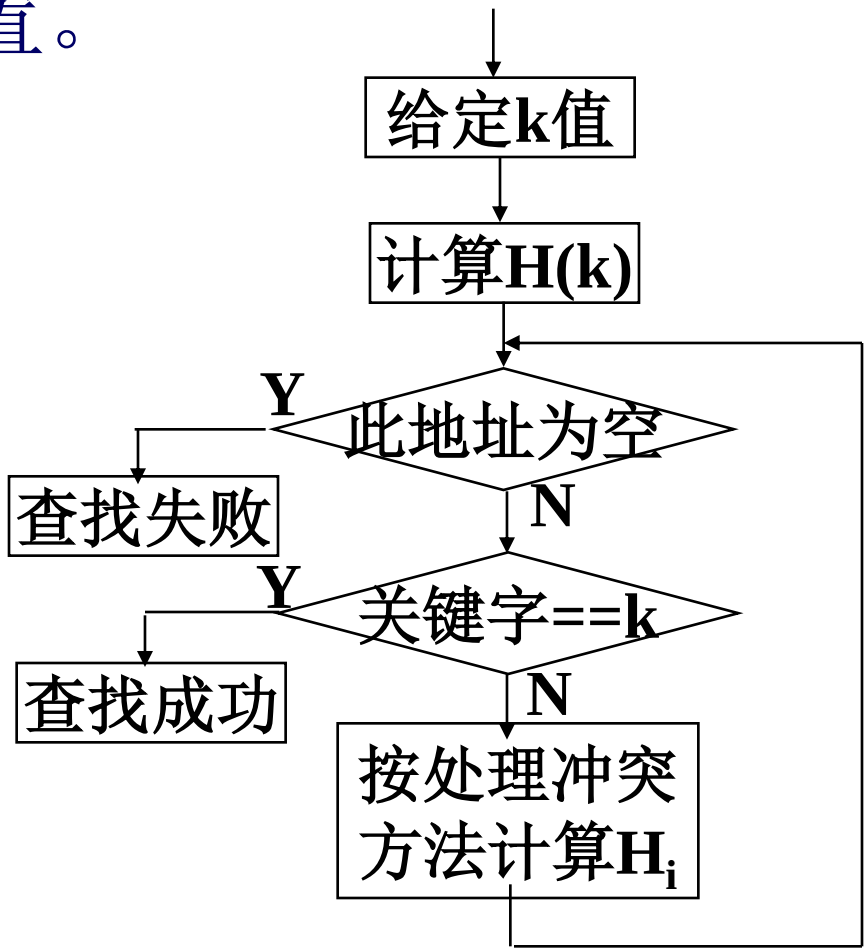
{ 19, 01, 23, 14, 55, 68,11, 82, 36 }

- 哈希函数为：

H(key)=key % 7

哈希表的查找及其性能分析

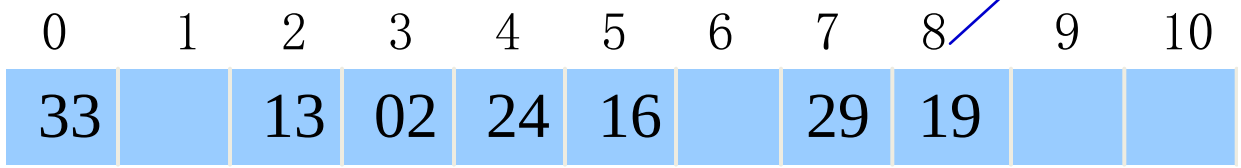
- 查找过程：和造表过程一致，对于给定值，由哈希函数和解决冲突的方法定位记录的存储位置。



例：给出哈希表HT，哈希函数 H(key)=key % 11,解决冲突方法：开放地址法中线性探测再散列

Hi(key)=(H(KEY)+d_i)% 11 (d_1=1, d_2=2, d_3=3,...),试查找关键字19 、02

查找关键字19



用哈希函数求19对应的哈希地址：

H(19)=19 % 11=8

将HT[8]与19比较

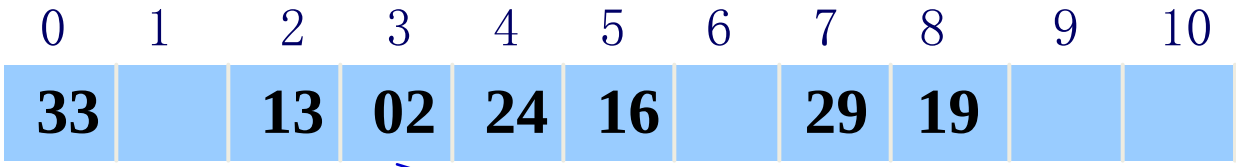
相等

查找成功

查找关键字02

用哈希函数求02对应的哈希地址：
H(02)=02 % 11=2

用解决冲突方法为02求下一个“地址”　（取d_1=1）
H_1 (02)=(H(02)+ d_1)% 11=3



将HT[2]与02比较
不相等

将HT[3]与02比较
相等
查找成功

- 查找算法实现：开放定址哈希表

```
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
```

```
bool 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 &&
                              (K!= H.elem[p].key))
        collision(p, ++c);  // 求得下一探查地址 p
    if (K==H.elem[p].key) return 1;
                // 查找成功，返回待查数据元素位置 p
    else return  0;  // 查找不成功
} // SearchHash
```

```
bool InsertHash (HashTable &H, Elemtype e)
{  Int c = 0;
   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
   else    if ( 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
}
```

- 哈希表查找分析
  - 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—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: 关键字集合

  { 7, 15, 20, 31, 48, 53,64, 76, 82, 99 }

线性探测处理冲突时， ASL = **2.4**

平方探测处理冲突时， ASL = **2.0**

链地址法处理冲突时， ASL = **1.7**

2 Sort
(排序)

什么是排序？

排序是计算机内经常进行的一种操作，其目的是将一组“无序”的记录序列调整为“有序”的记录序列。

例如：将下列关键字序列

52, 49, 80, 36, 14, 58, 61, 23, 97, 75

调整为

14, 23, 36, 49, 52, 58, 61 ,75, 80, 97

一般情况下，

假设含n个记录的序列为$\{R_1, R_2, \ldots, R_n\}$

其相应的关键字序列为$\{K_1, K_2, \ldots, K_n\}$

这些关键字相互之间可以进行比较，即在它们之间存在着这样一个关系：

$$K_{p1} \leqslant K_{p2} \leqslant \ldots \leqslant K_{pn}$$

按此固有关系将上式记录序列重新排列为

$$\{R_{p1}, R_{p2}, \ldots, R_{p3}\}$$

的操作称作排序。

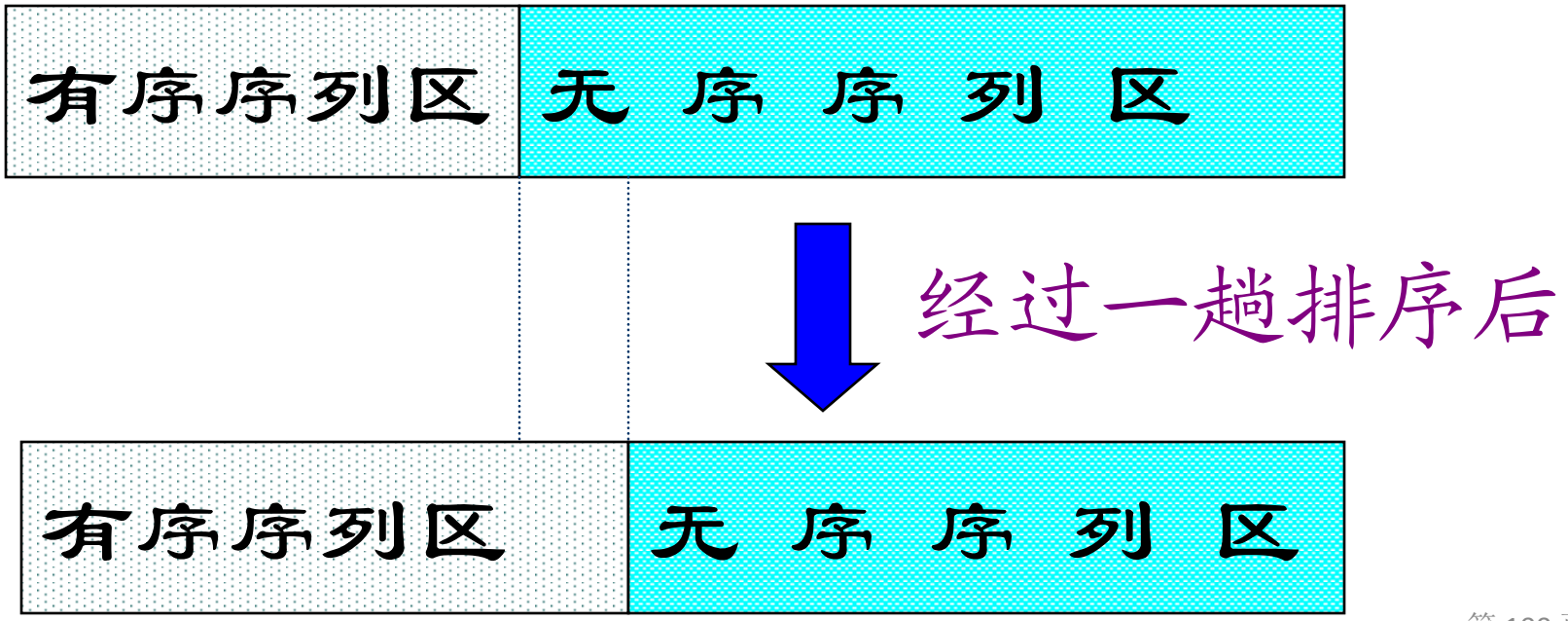
内部排序和外部排序

若整个排序过程不需要访问外存便能完成，则称此类排序问题为内部排序；

反之，若参加排序的记录数量很大，整个序列的排序过程不可能在内存中完成，则称此类排序问题为外部排序。

内部排序的方法

内部排序的过程是一个逐步扩大有序记录序列长度（同时也缩小无序序列长度）的过程。



排序方法的稳定性能

1. 稳定的排序方法指的是，对于两个关键字相等的记录，它们在序列中的相对位置，在排序之前和经过排序之后，没有改变。否则，称这种排序方法为不稳定的。

排序之前：{ · · · · · $R_i(K)$ · · · · · $R_j(K)$ · · · · · }

排序之后：{ · · · · · $R_i(K)$ $R_j(K)$ · · ·.. · · · · }

例如：排序前 (56, 34, 47^*, 23, 66, 18, 82, 47)

若采用某种排序方法排序后得到结果:

(18, 23, 34, 47, 47^*, 56, 66, 82)

则可确定该排序方法是不稳定的。

2. 稳定的排序方法对所有实例均具有同样效果；不稳定的排序方法则不能保证所有实例均具有同样效果，但至少能举出一个排序实例体现其不稳定（或者说，只要有一个反例就可证明算法的不稳定）。

例如：对{4*, 3, 4, 2}进行快速排序，得到{2, 3, 4, 4*}，说明快速排序是不稳定的排序方法。

基于不同的“扩大”有序序列长度的方法，内部排序方法大致可分下列几种类型：

交换类　　选择类　　插入类

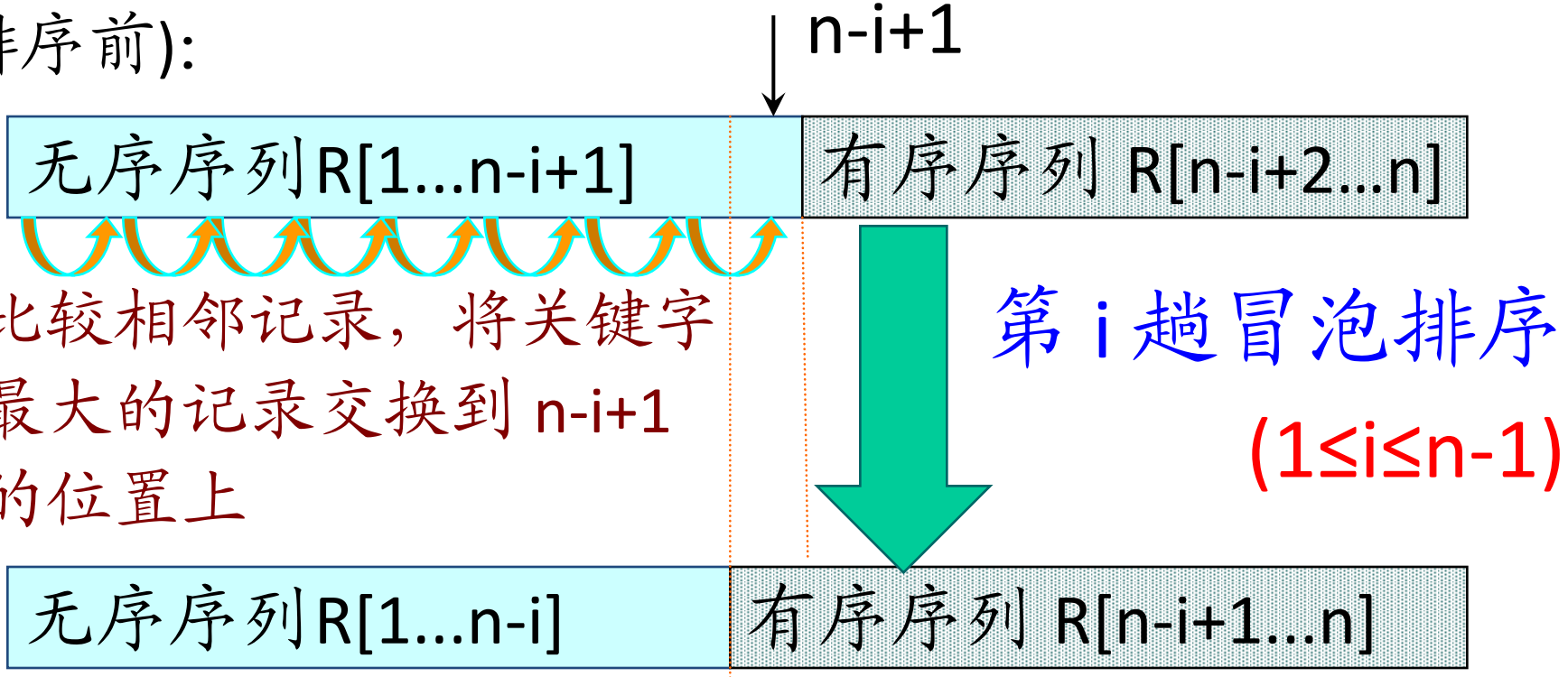
归并类　　其它方法

1. 交换类

通过“交换”无序序列中的记录从而得到其中关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。

冒泡排序方法

假设在排序过程中，记录序列R[1...n]的状态为(第i趟排序前):



- 第 i 遍冒泡排序的基本思想：从第一个记录开始，两两比较待排序记录集（[1...n-i+1]）相邻记录关键字，若关键字逆序则交换两记录位置。经此1趟排序后，待排序记录集中关键字最大的记录将移到第n-i+1位置上。

冒泡排序例子

冒泡排序方法实例分析

初始序列：45* 23 32 39 50 88 05 66 45

	1	2	3	4	5	6	7	8	9	交换次数	swap
第1趟：	23	32	39	45*	50	05	66	45	[88]	6	1
第2趟：	23	32	39	45*	05	50	45	[66	88]	2	1
第3趟：	23	32	39	05	45*	45	[50	66	88]	2	1
第4趟：	23	32	05	39	45*	[45	50	66	88]	1	1
第5趟：	23	05	32	39	[45*	45	50	66	88]	1	1
第6趟：	05	23	32	[39	45*	45	50	66	88]	1	1
第7趟：	05	23	[32	39	45*	45	50	66	88]	0	0

最后结果：05 23 32 39 45* 45 50 66 88

- 冒泡排序的结束条件为：已经排序n－1趟或者最近一趟没有进行“记录交换”(swap=0)。

```
void BubbleSort(Elem R[ ], int n) {
    int i = 1, j , swap;        //swap为交换标志
    do //一趟排序
       { swap = 0;        //每趟排序前交换标志置为0
       for (j = 1;  j < n-i;  j++)
           if (R[j].key > R[j+1].key)
           {Exchange(R[j], R[j+1]);    //交换两记录的存储位置
             swap = 1;     //交换过记录，交换标志swap置为1
           }
       i ++;  // 缩小无序区间
    }while (i<n & swap==1) //若已排序n-1遍或无交换，结束
} // BubbleSort
```

冒泡排序法特点

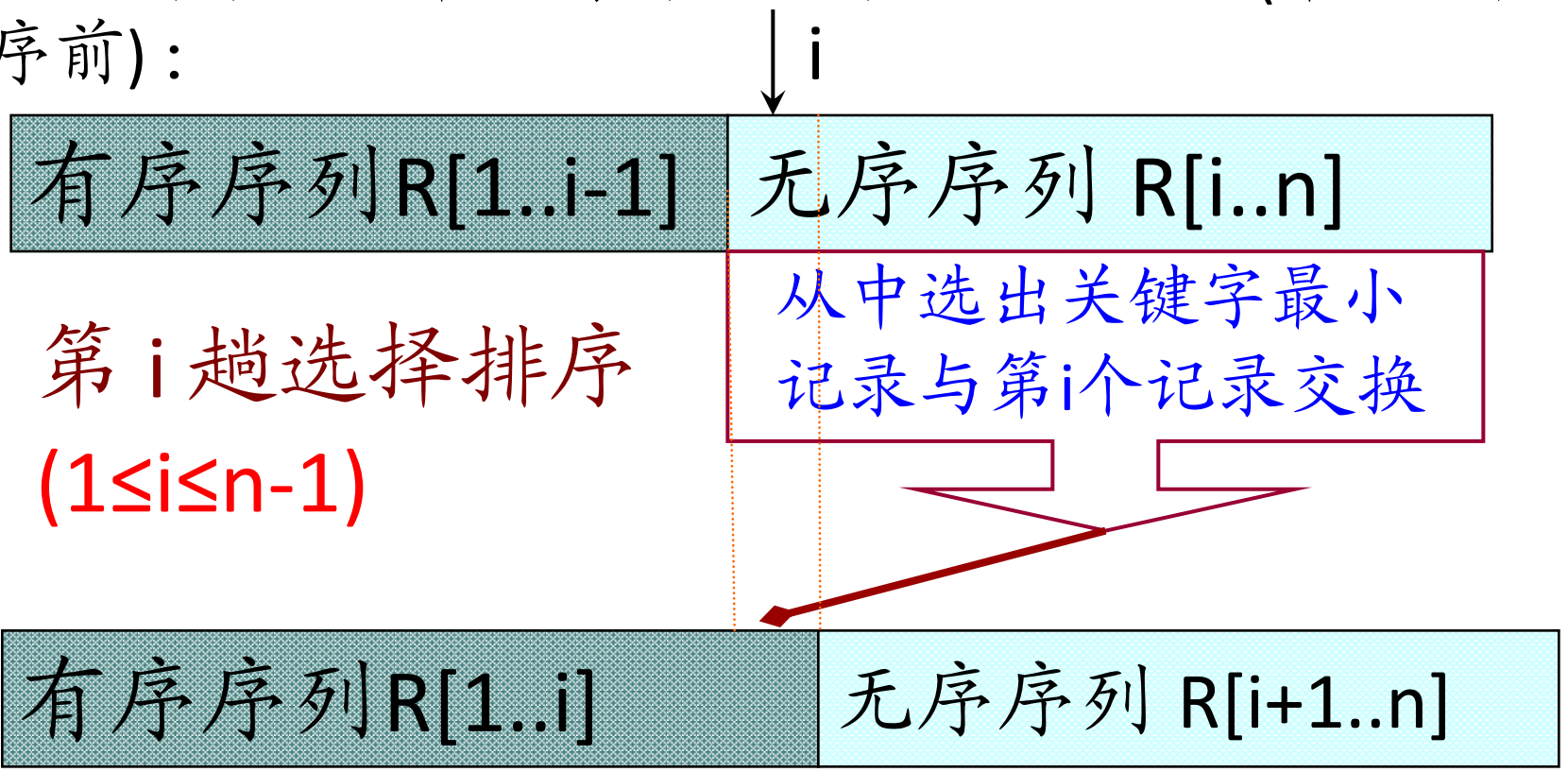
- 冒泡排序最好的情况是“待排序记录集递增有序”，此时只需要进行1趟排序操作，进行n－1次比较，不交换记录;
- 冒泡排序最差的情况是“待排序记录集递减有序”，需要进行n-1趟排序，总比较次数为n（n－1）/2次，总交换记录次数为3n（n-1）/2次;
- 冒泡排序只需要用到少量的辅助空间(与n无关)
- 冒泡排序是稳定的排序方法,时间复杂度为$O(n^2)$
- 冒泡排序的效率较低，适用于小型文件的排序。

2. 选择类

从记录的无序子序列中“选择”关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。

选择排序方法

假设排序过程中，待排记录序列的状态为(第i趟排序前)：



- 第 i 遍选择排序的基本思想：通过比较关键字从无序序列 R[i...n]中找出关键字最小的记录，将它与无序序列中的第1个记录（即R[i]）交换位置。

选择排序

选择排序法示例

初始序列：45* 34 27 18 72 45 40 66

		交换次数	比较次数
第1趟：	[18] 34 27 45* 72 45 40 66	1	7
第2趟：	[18 27] 34 45* 72 45 40 66	1	6
第3趟：	[18 27 34] 45* 72 45 40 66	1	5
第4趟：	[18 27 34 40] 72 45 45* 66	1	4
第5趟：	[18 27 34 40 45] 72 45* 66	1	3
第6趟：	[18 27 34 40 45 45*] 72 66	1	2
第7趟：	[18 27 34 40 45 45* 66] 72	1	1

排序结果：18 27 34 40 45 45* 66 72

选择排序的算法描述如下：

```
void SelectSort (Elem R[], int n ) {
  // 对记录序列R[1..n]作简单选择排序
  for (i=1; i<n; ++i) {
      // 找到第 i 个最小的记录，并交换到位
      j = SelectMinKey(R, i);
            // 在 R[i..n] 中选择关键字最小的记录
      if (i!=j)  R[i]←→R[j]; // 与第 i 个记录交换
  }
} // SelectSort
```

选择排序法特点

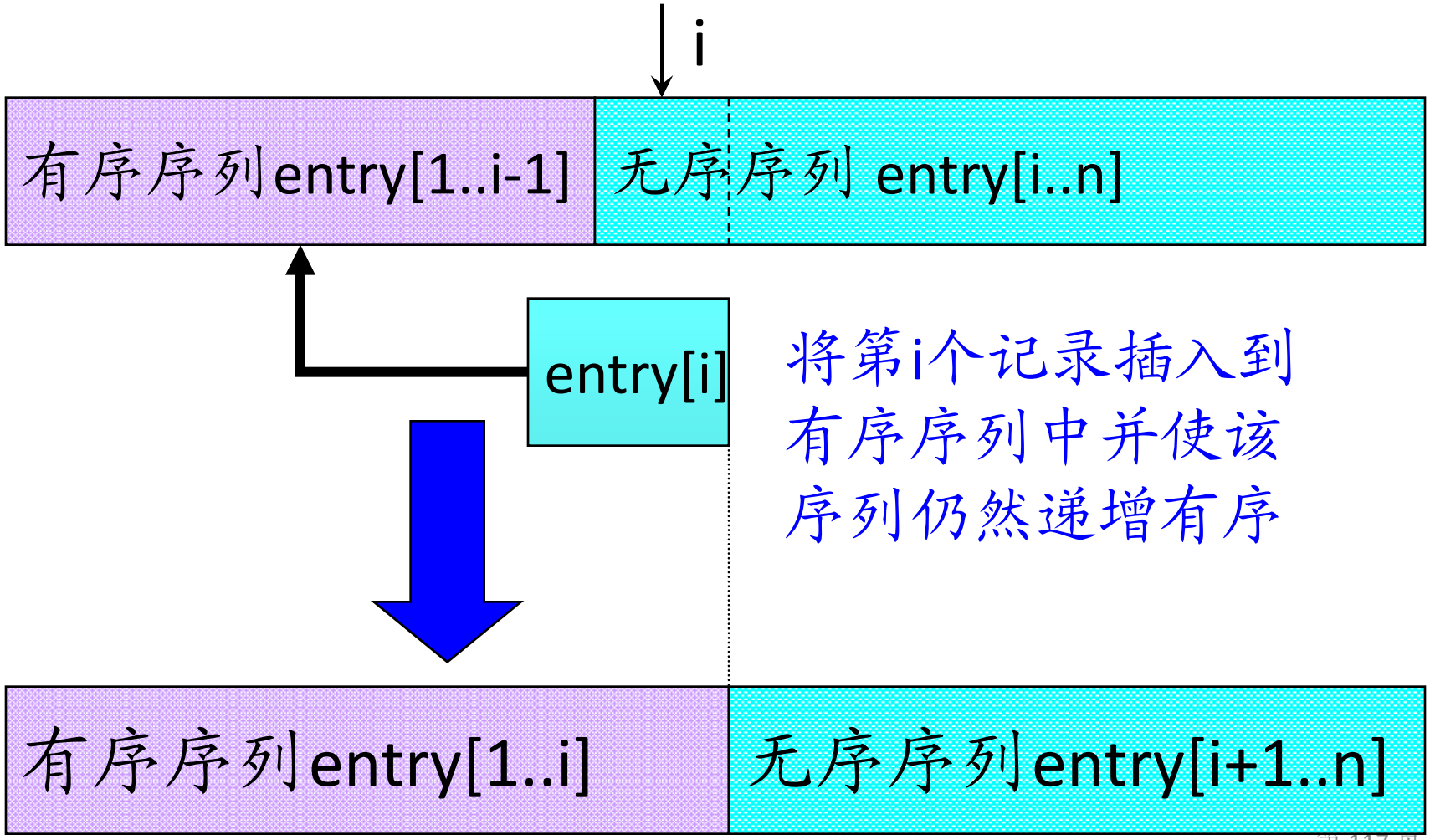
- 选择排序在任意情况下都要进行n-1趟排序操作，进行n（n－1）/2 次比较;
- 在最好的情况下（待排序记录集递增有序），不需交换记录；在最差的情况（待排序记录集递减有序），需要进行n-1次交换;
- 选择排序只需用到少量的辅助空间，与n无关;
- 选择排序是不稳定的排序方法，时间复杂度为$O(n^2)$;
- 选择排序的效率较低，适用于小型文件的排序。

3. 插入类

将无序子序列中的一个或几个记录“插入”到有序序列中，从而增加记录的有序子序列的长度。

插入排序

第i趟插入排序的基本思想：



插入排序

- 实现“一趟插入排序”可分三步进行:

1. 在entry[1..i-1]中查找entry[i]的插入位置j，使得

   entry[1..j].key ≤ entry[i].key < entry[j+1..i-1].key;

2. 将entry[j+1..i-1]中的所有记录均后移一个位置;

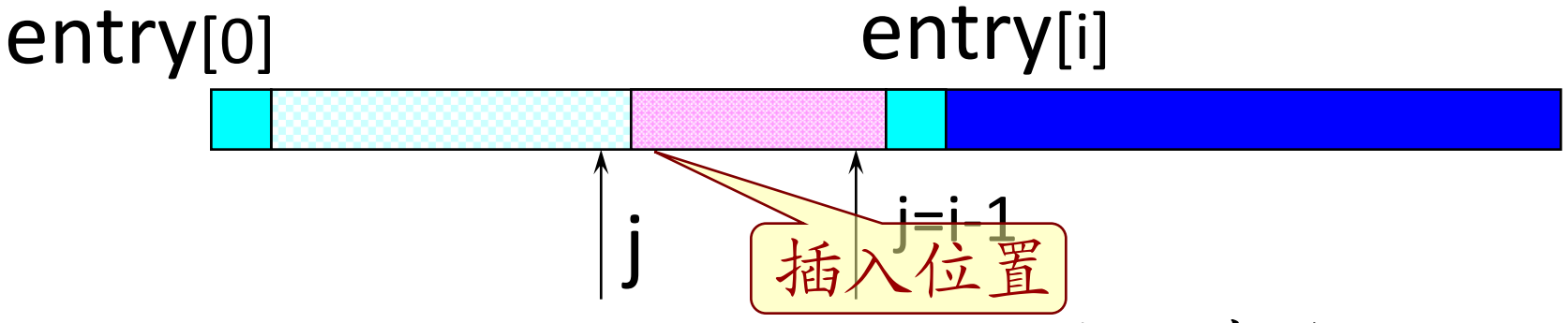
3. 将entry[i] 插入(复制)到entry[j+1]的位置上。

- 不同的实现方法导致不同的算法实现
  - 直接插入排序（基于顺序查找）
  - 折半插入排序（基于折半查找）

一、直接插入排序

利用“顺序查找”实现“在entry[1..i-1]中查找entry[i]的插入位置”。算法的实现要点：

- 从entry[i-1]起从后向前进行顺序查找第一个比之小的元素，监视哨设置在entry[0];



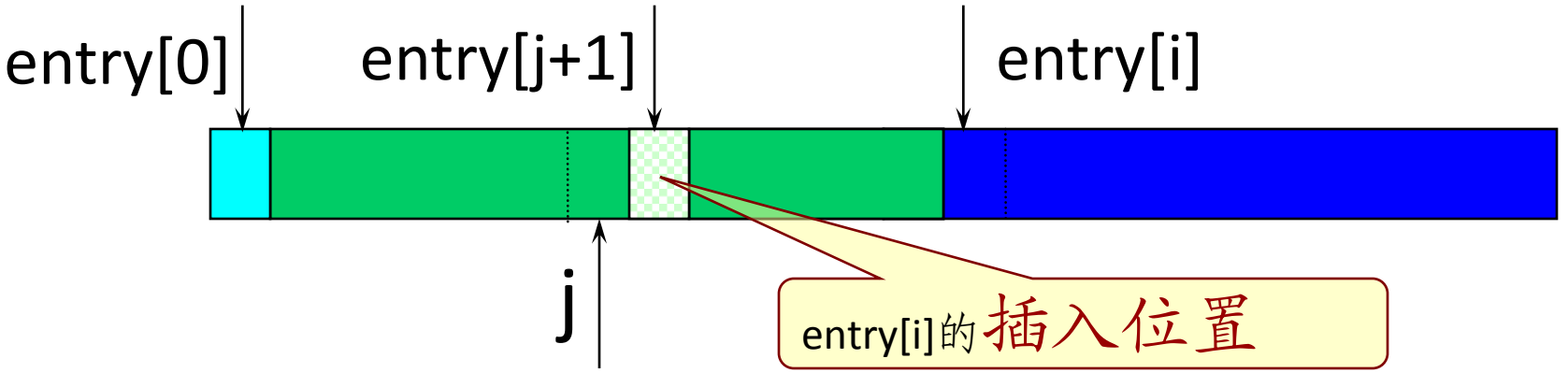

```
entry[0] = entry[i];          // 设置“哨兵”
for (j=i-1; entry[i].key< entry[j].key; --j);
```

循环结束表明entry[i]的插入位置为 *j +1*

- 对于在查找过程中找到的那些关键字大于entry[i].key的记录，在查找的同时实现记录向后移动一个单元;

```
for (j=i-1; entry[0].key< entry[j].key; --j);
```

entry[j+1] = entry[j]



上述循环结束后可以直接进行“插入”操作:

entry[j+1] = entry[0]

- 令 i = 2，3，…, n,

  实现整个序列的排序。

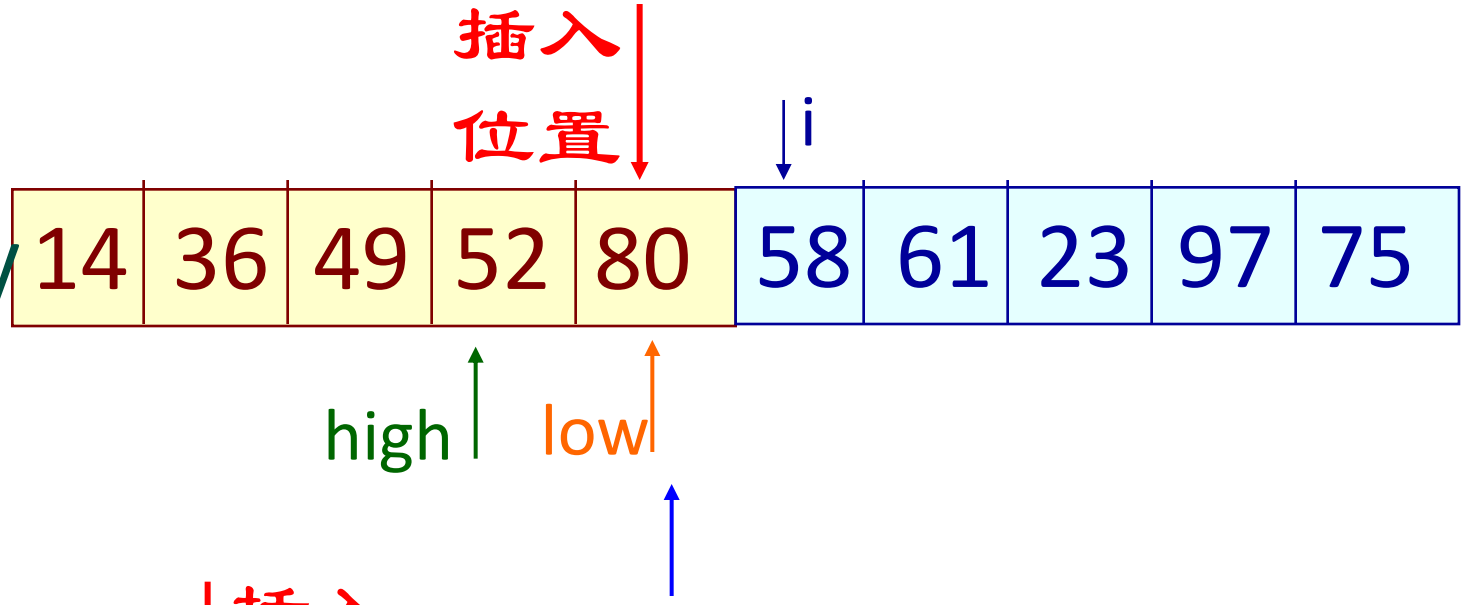
```
for ( i=2;  i<=n;  ++i )
  if (entry[i].key< entry[i-1].key)
   {  在 entry[1..i-1]中查找entry[i]的插入位置;
      插入entry[i] ;
   }
```

二、折半插入排序

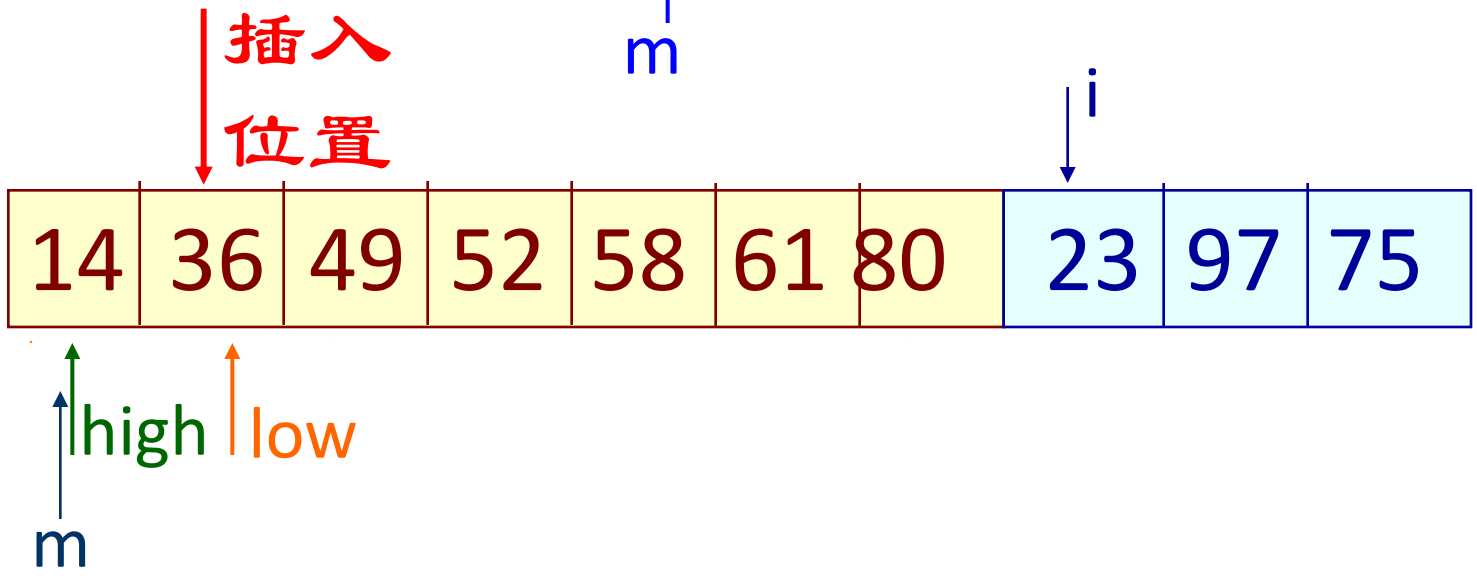
因为 entry[1…i-1] 是一个按关键字有序的有序序列，则可以利用折半查找实现“在entry[1..i-1]中查找entry[i]的插入位置”，如此实现的插入排序为折半插入排序。

说明：折半查找在low<=high时结束，插入位置在low所指向的单元。

例如:


插入
位置
i
entry
14
36
49
52
80
58
61
23
97
75
high
low
m

再如:


插入
位置
i
entry
14
36
49
52
58
61
80
23
97
75
high
low
m

插入排序法示例

初始序列：34 66 27 18 72 40 45

第1趟：[34 66] 27 18 72 40 45

第2趟：[27 34 66] 18 72 40 45

第3趟：[18 27 34 66] 72 40 45

第4趟：[18 27 34 66 72] 40 45

第5趟：[18 27 34 40 66 72] 45

第6趟：[18 27 34 40 45 66 72]

排序结果：18 27 34 40 45 66 72

注意：第一趟排序前的无序序列！

插入排序法特点

- 插入排序在任意情况下都要进行n-1趟排序操作；
- 插入排序只需用到少量的辅助空间，与n无关；
- 插入排序是稳定的排序方法，时间复杂度为 $O(n^2)$；
- 插入排序的效率较低，适用于小型文件的排序。

4. 归并类

通过“归并”两个或两个以上的记录有序子序列，逐步增加记录有序序列的长度。

4. 归并类

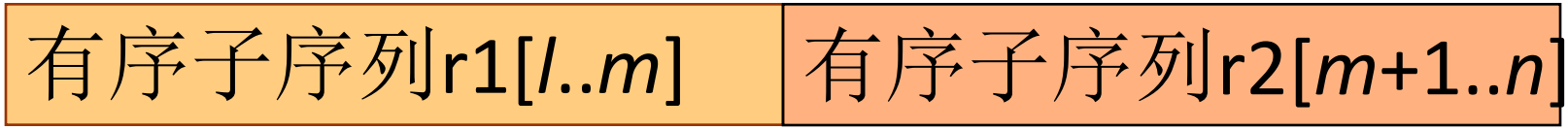
归并排序(Merge sort)

We chop(分割) the list into two sublists of sizes as nearly equal as possible and then sort them separately(分别地). Afterward, we carefully merge(归并) the two sorted sublists into a single sorted list.

归并排序的过程基于下列基本思想进行:

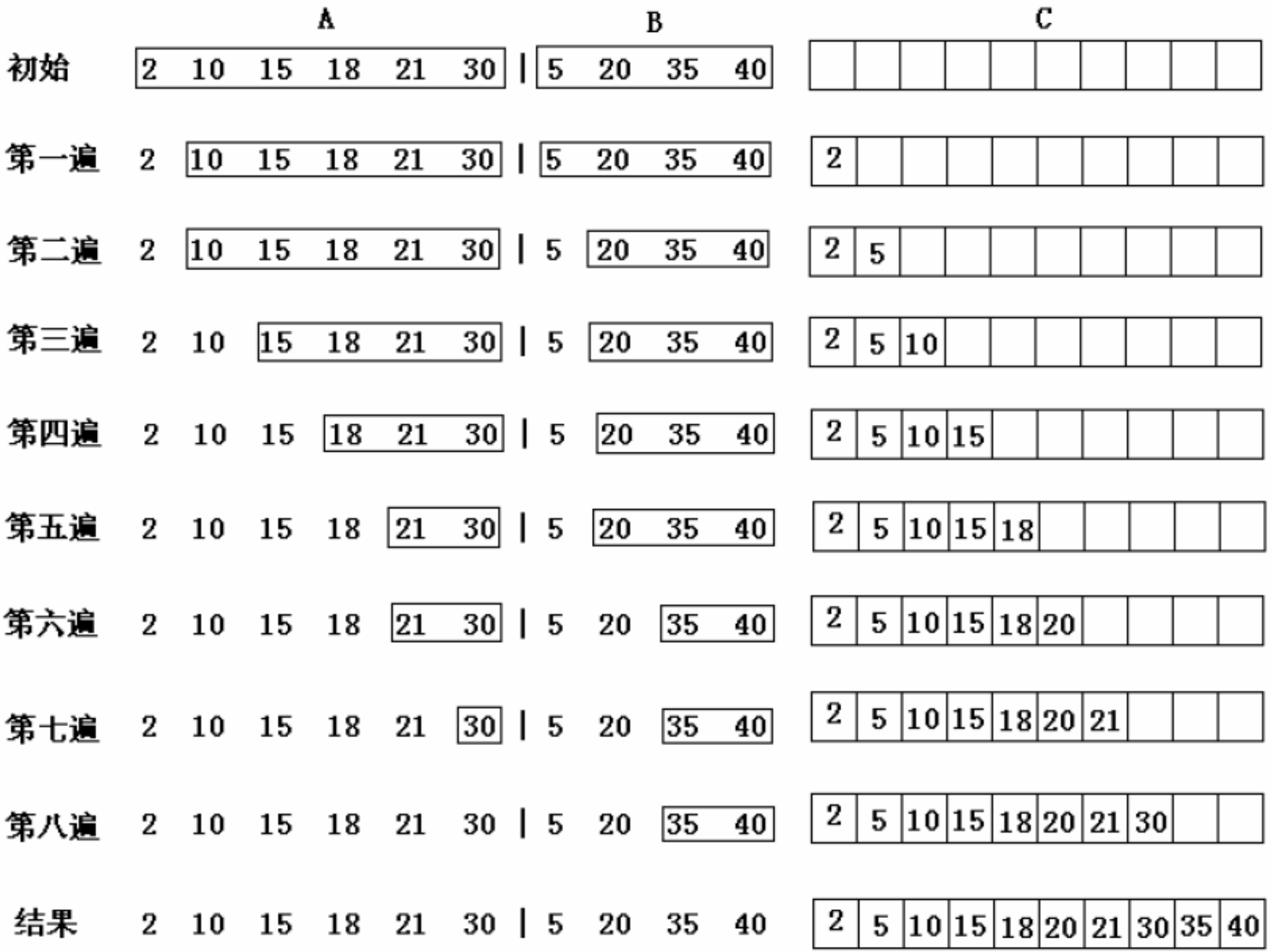
1. 将待排序记录集分割为两个或两个以上子序列;
2. 对这些子序列分别排序（可递归使用归并排序）
3. 将这些有序子序列 “归并” 为一个有序序列。

在内部排序中，通常采用的是2路归并排序。即：将两个位置相邻的记录有序子序列r1[*l*..*m*]和r2[*m*+1..*n*]



归并为一个记录的有序序列R[*l*..*n*] 。

这个操作对顺序表而言，是轻而易举的，其算法如下页所示。



两个有序子序列归并过程示意图

2路归并排序算法的基本思想

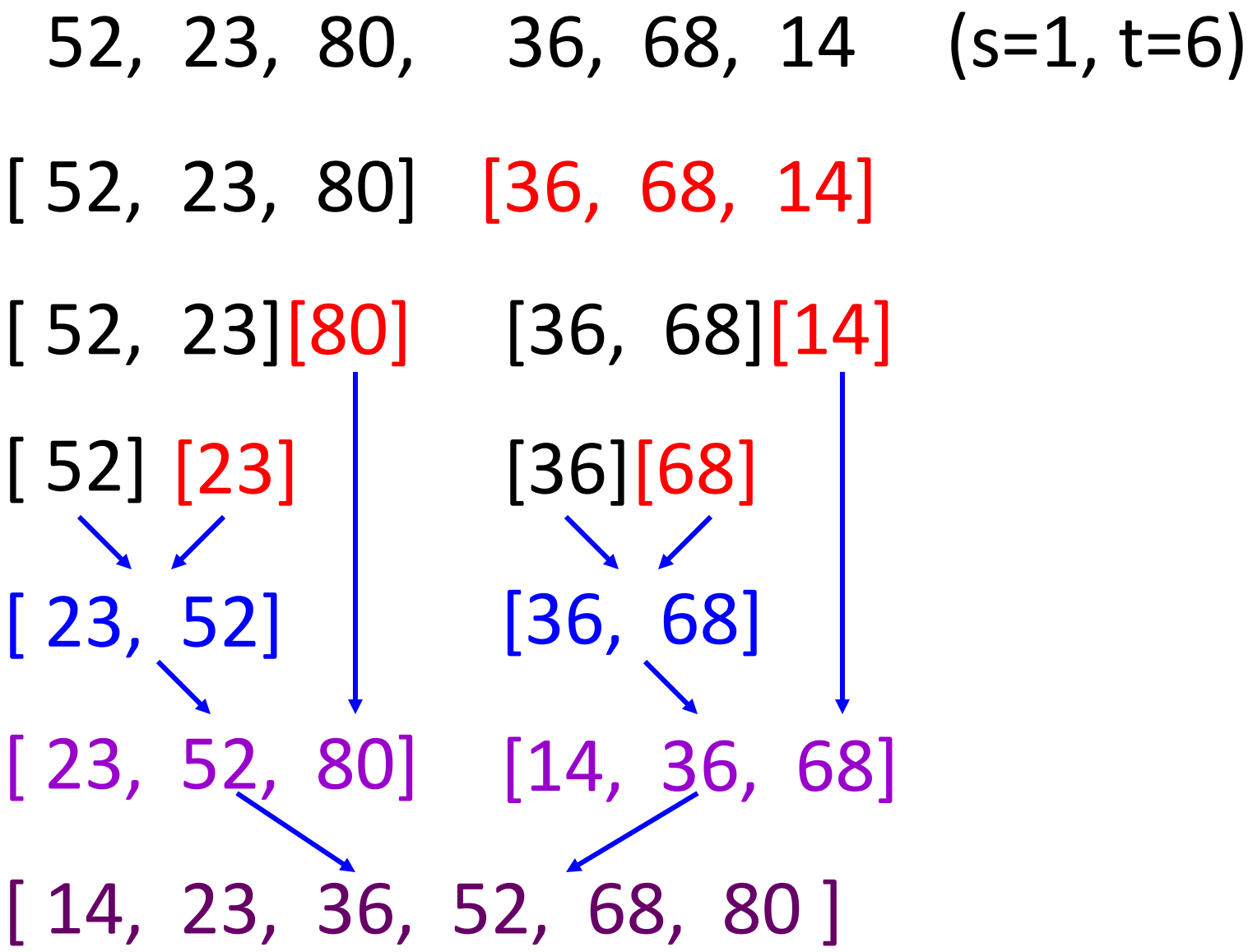
如果将无序序列 R[s..t]分割成两部分:

$R[s..\lfloor (s+t)/2 \rfloor]$ 和 $R[\lfloor (s+t)/2 \rfloor+1..t]$

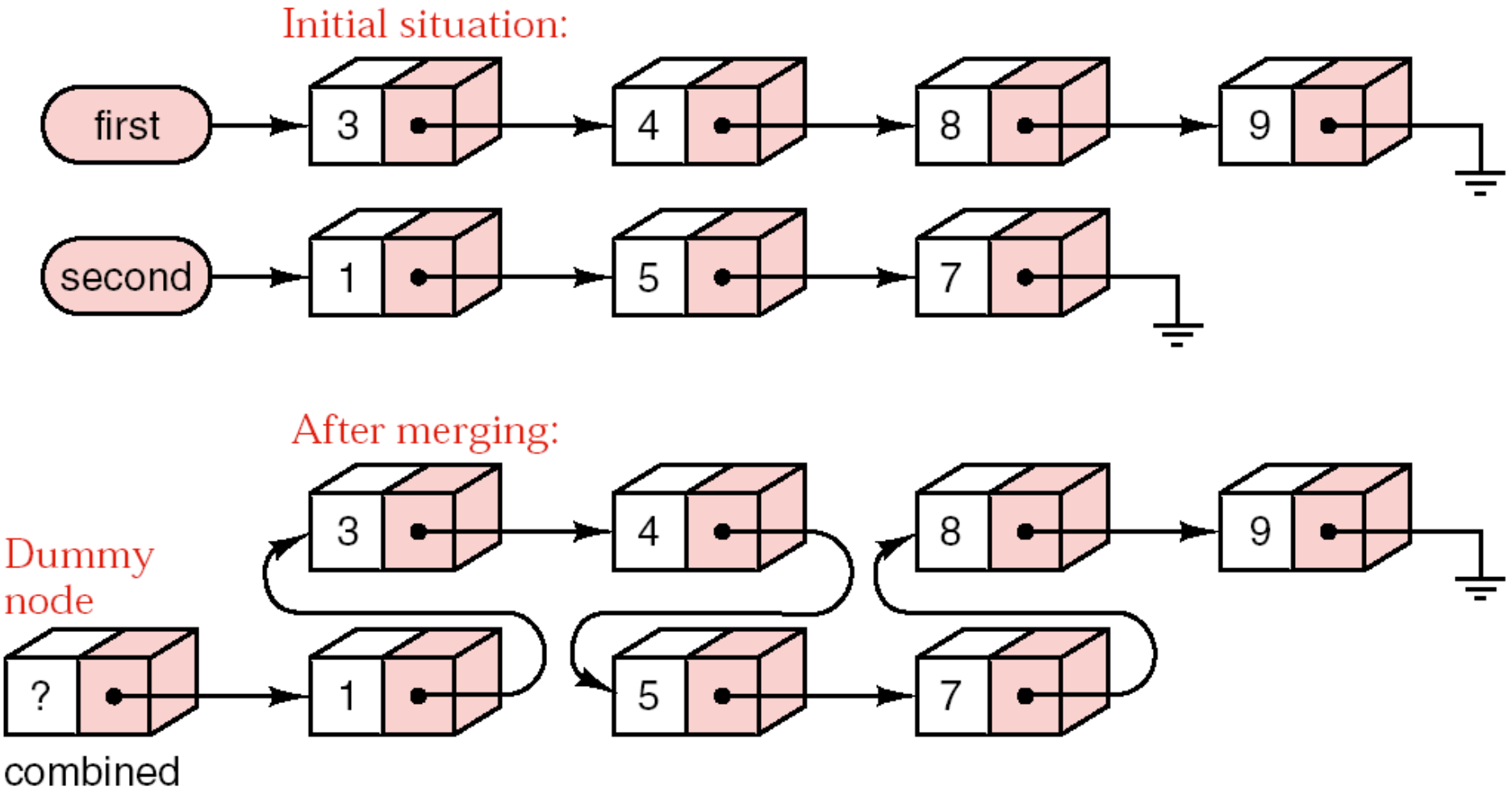
并分别按关键字排序得到两有序子序列，则利用上述归并算法很容易将它们归并成一个有序记录序列。

因此，关键在于如何对这两部分进行排序。方法：可递归使用2路归并排序。

例如:

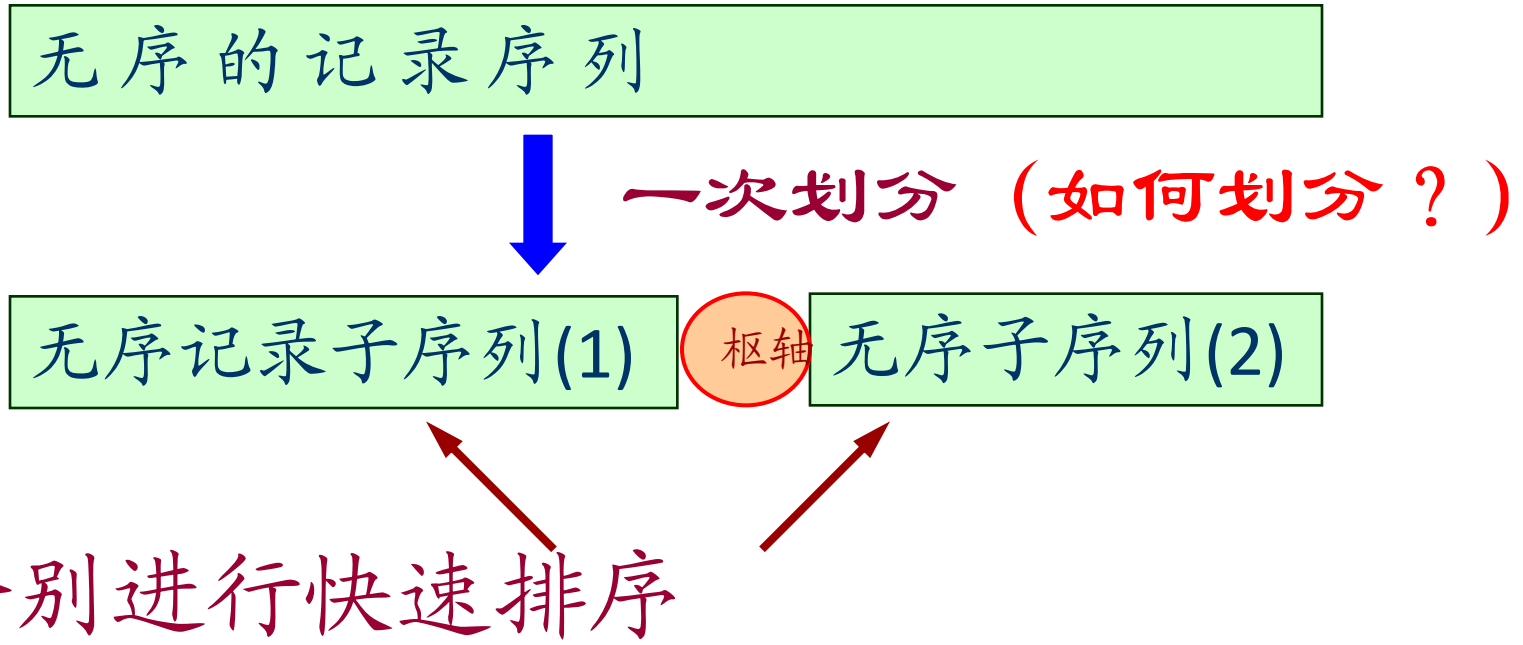

52, 23, 80, 36, 68, 14 (s=1, t=6)
[52, 23, 80] [36, 68, 14]
[52, 23][80] [36, 68][14]
[52] [23] [36][68]
[23, 52] [36, 68]
[23, 52, 80] [14, 36, 68]
[14, 23, 36, 52, 68, 80]

归并两有序子序列示意图



5. 其它方法 （快速排序法）

思想：首先对无序的记录序列进行“一次划分”，之后分别对分割所得两个子序列“递归”进行快速排序。



快速排序法

一趟快速排序（一次划分）的过程

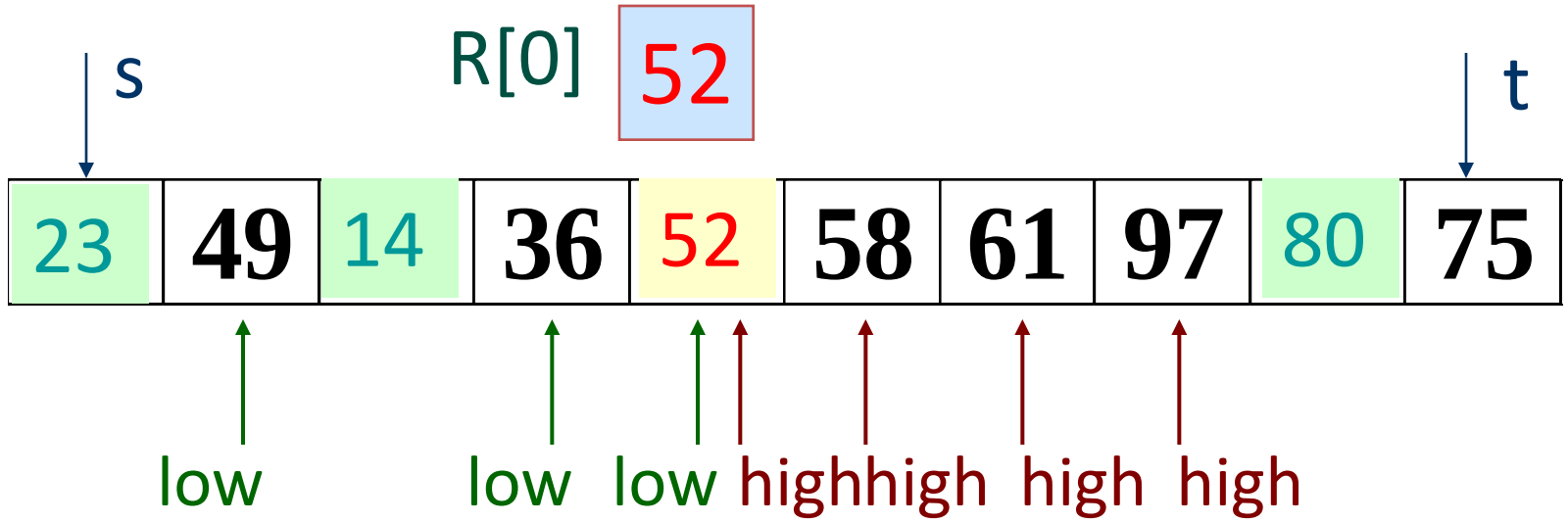
目标：找一个记录，以它的关键字作为“枢轴”，凡其关键字小于枢轴的记录均移动至该记录之前，反之，凡关键字大于枢轴的记录均移动至该记录之后。

致使一趟排序之后，记录的无序序列R[s..t]将分割成两部分：R[s..i-1]和R[i+1..t]，且

$$R[j].key \le R[i].key \le R[k].key$$

$(s \le j \le i-1)$　枢轴　$(i+1 \le k \le t)$。

例如



设 R[s]=52 为枢轴

将 R[high].key 和 枢轴的关键字进行比较，要求R[high].key ≥ 枢轴的关键字

将 R[low].key 和 枢轴的关键字进行比较，要求R[low].key ≤ 枢轴的关键字

可见，经过“一次划分”，将关键字序列

52, 49, 80, 36, 14, 58, 61, 97, 23, 75

调整为：23, 49, 14, 36, (52) 58, 61, 97, 80, 75

在调整过程中，设立了两个指针：low和high，它们的初值分别为：s 和 t，

之后逐渐减小 high，增加 low，并保证 R[high].key≥52，和 R[low].key≤52，否则进行记录的“转移”；直到high = low，将枢轴元素置于该位置。

快速排序法示例

初始序列： 34 66 27 18 72 40 45

第1趟(枢轴为34)： (18 27) 34 (66 72 40 45)

第2趟(枢轴为18)： 18 (27) 34 (66 72 40 45)

第3趟(枢轴为66)： 18 27 34 (45 40) 66 (72)

第4趟(枢轴为45)： 18 27 34 (40) 45 66 72

排序结果： 18 27 34 40 45 66 72

说明：

1.横线标示下一趟快速排序的区间；

2.每趟快速排序均选取待排序区间的首元素为枢轴。

希尔排序

希尔排序(Shell Sort，又称缩小增量法)是一种分组插入排序方法。

1 排序思想

① 先取一个正整数d_1(d_1<n)作为第一个增量，将全部n个记录分成d_1组，把所有相隔d_1的记录放在一组中，即对于每个k(k=1, 2, … d_1)， R[k], R[d_1+k], R[2d_1+k] , … 分在同一组中，在各组内进行直接插入排序。这样一次分组和排序过程称为一趟希尔排序；

② 取新的增量d_2<d_1，重复①的分组和排序操作；直至所取的增量d_i=1为止，即所有记录放进一个组中排序为止。

2 排序示例

设有10个待排序的记录，关键字分别为9, 13, 8, 2, 5, 13, 7, 1, 15, 11，增量序列是5, 3, 1，希尔排序的过程如图10-5所示。

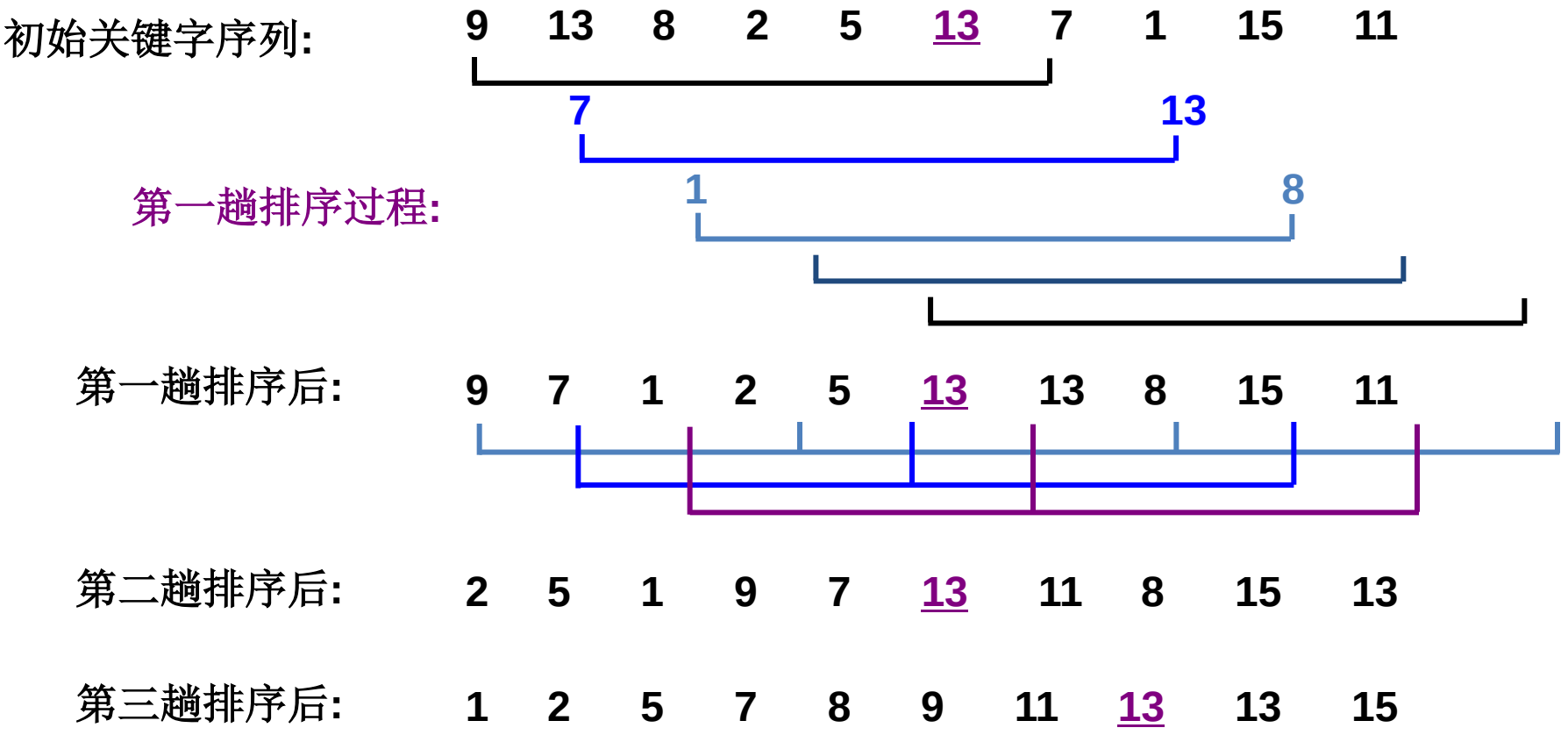


图10-5 希尔排序过程

3 算法实现

先给出一趟希尔排序的算法，类似直接插入排序。

```
void shell_pass(Sqlist *L, int d)
   /*  对顺序表L进行一趟希尔排序, 增量为d  */
  {  int j, k ;
     for (j=d+1; j<=L->length; j++)
        {  L->R[0]=L->R[j] ;          /*  设置监视哨兵  */
           k=j-d ;
           while (k>0&&LT(L->R[0].key, L->R[k].key) )
              {   L->R[k+d]=L->R[k] ; k=k-d ;   }
           L->R[k+j]=L->R[0] ;
        }
  }
```

然后在根据增量数组dk进行希尔排序。

```
void shell_sort(Sqlist *L, int dk[], int t)
  /*   按增量序列dk[0 … t-1],对顺序表L进行希尔排序   */
  {  int m ;
     for (m=0; m<=t; m++)
        shll_pass(L, dk[m]) ;
  }
```

希尔排序的分析比较复杂，涉及一些数学上的问题，其时间是所取的“增量”序列的函数。

希尔排序特点

子序列的构成不是简单的“逐段分割”，而是将相隔某个增量的记录组成一个子序列。

希尔排序可提高排序速度，原因是：

- 分组后n值减小，n^2更小，而$T(n)=O(n^2)$,所以T(n)从总体上看是减小了；
- 关键字较小的记录跳跃式前移，在进行最后一趟增量为1的插入排序时，序列已基本有序。

增量序列取法

- 无除1以外的公因子；
- 最后一个增量值必须为1。

10.4.3 堆排序

1 堆的定义

是n个元素的序列$H=\{k_1, k_2, \ldots k_n\}$，满足：

$$\begin{cases} k_i \le k_{2i} & \text{当} 2i \le n \text{时} \\ k_i \le k_{2i+1} & \text{当} 2i+1 \le n \text{时} \end{cases}$$

或

$$\begin{cases} k_i \ge k_{2i} & \text{当} 2i \le n \text{时} \\ k_i \ge k_{2i+1} & \text{当} 2i+1 \le n \text{时} \end{cases}$$

其中： $i=1,2, \ldots, \lfloor n/2 \rfloor$

由堆的定义知，堆是一棵以k_1为根的完全二叉树。若对该二叉树的结点进行编号(从上到下，从左到右)，得到的序列就是将二叉树的结点以顺序结构存放，堆的结构正好和该序列结构完全一致。

2 堆的性质

① 堆是一棵采用顺序存储结构的完全二叉树， k_1是根结点；

② 堆的根结点是关键字序列中的最小(或最大)值，分别称为小(或大)根堆；

③ 从根结点到每一叶子结点路径上的元素组成的序列都是按元素值(或关键字值)非递减(或非递增)的；

④堆中的任一子树也是堆。

利用堆顶记录的关键字值最小(或最大)的性质，从当前待排序的记录中依次选取关键字最小(或最大)的记录，就可以实现对数据记录的排序，这种排序方法称为堆排序。

3 堆排序思想

① 对一组待排序的记录，按堆的定义建立堆；

② 将堆顶记录和最后一个记录交换位置，则前n-1个记录是无序的，而最后一个记录是有序的；

③ 堆顶记录被交换后，前n-1个记录不再是堆，需将前n-1个待排序记录重新组织成为一个堆，然后将堆顶记录和倒数第二个记录交换位置，即将整个序列中次小关键字值的记录调整(排除)出无序区；

④ 重复上述步骤，直到全部记录排好序为止。

结论：排序过程中，若采用小根堆，排序后得到的是非递减序列；若采用大根堆，排序后得到的是非递增序列。

堆排序的关键

① 如何由一个无序序列建成一个堆？

② 如何在输出堆顶元素之后，调整剩余元素，使之成为一个新的堆？

4 堆的调整——筛选

(1) 堆的调整思想

输出堆顶元素之后，以堆中最后一个元素替代之；然后将根结点值与左、右子树的根结点值进行比较，并与其中小者进行交换；重复上述操作，直到是叶子结点或其关键字值小于等于左、右子树的关键字的值，将得到新的堆。称这个从堆顶至叶子的调整过程为“筛选”，如图10-10所示。

注意：筛选过程中，根结点的左、右子树都是堆，因此，筛选是从根结点到某个叶子结点的一次调整过程。

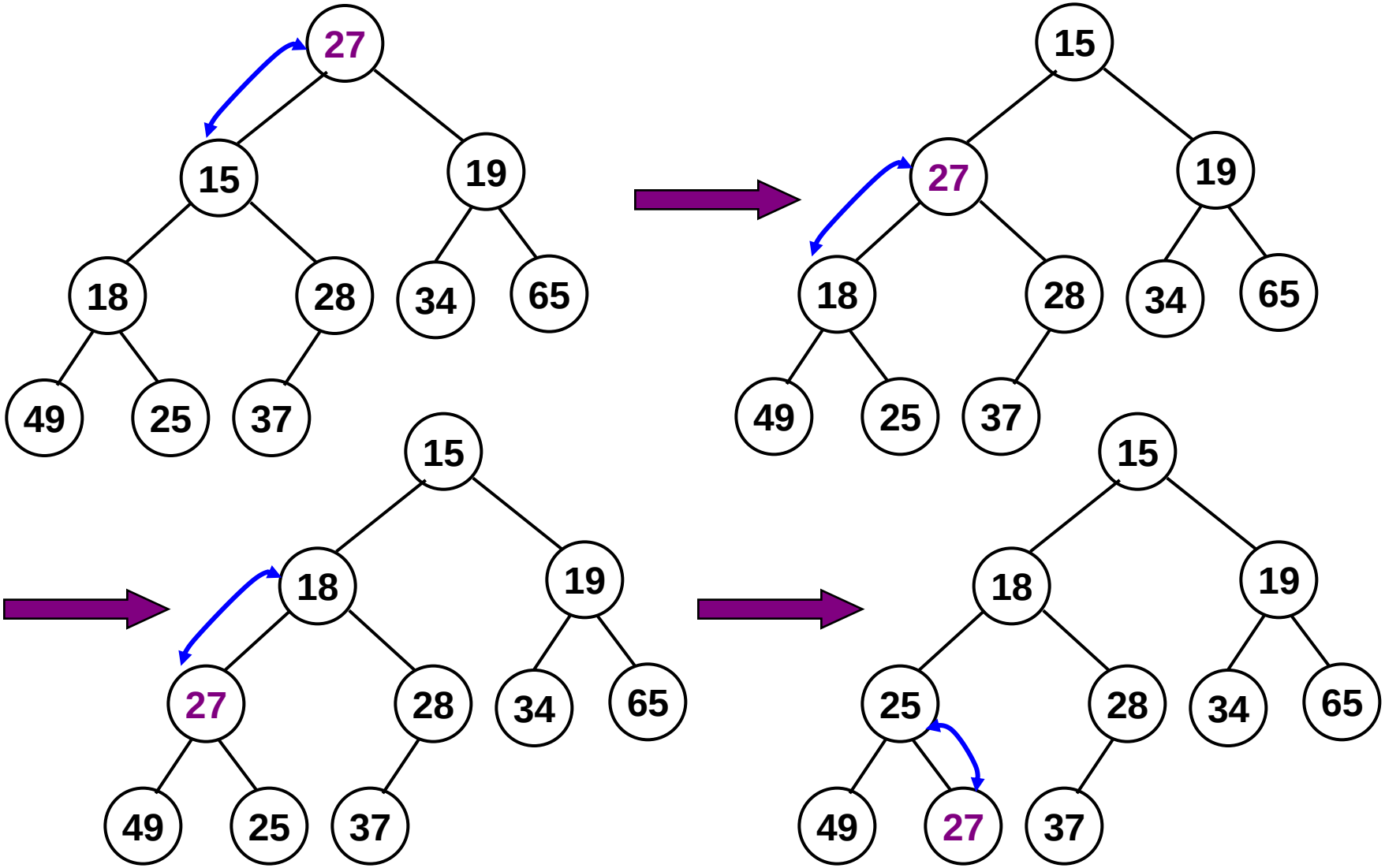


图10-10　堆的筛选过程

(2) 堆调整算法实现

```
void Heap_adjust(Sqlist *H, int s, int m)
   /*  H->R[s…m]中记录关键字除H->R[s].key均满足堆定义  */
   /*  调整H->R[s]的位置使之成为小根堆  */
   {  int j=s, k=2*j ;      /*  计算H->R[j]的左孩子的位置  */
      H->R[0]=H->R[j] ;        /*  临时保存H->R[j]  */
      for (k=2*j; k<=m; k=2*k)
         {  if ((k<m)&&(LT(H->R[k+1].key, H->R[k].key))
               k++ ;       /*  选择左、右孩子中关键字的最小者  */
            if  ( LT(H->R[k].key, H->R[0].key) )
               {   H->R[j]=H->R[k] ; j=k ; k=2*j  }
            else  break ;
         }
```

```
    H->R[j]=H->R[0] ;
  }
```

5 堆的建立

利用筛选算法，可以将任意无序的记录序列建成一个堆，设R[1],R[2], …,R[n]是待排序的记录序列。

将二叉树的每棵子树都筛选成为堆。只有根结点的树是堆。第⌊n/2⌋个结点之后的所有结点都没有子树，即以第⌊n/2⌋个结点之后的结点为根的子树都是堆。因此，以这些结点为左、右孩子的结点，其左、右子树都是堆，则进行一次筛选就可以成为堆。同理，只要将这些结点的直接父结点进行一次筛选就可以成为堆…。

只需从第⌊n/2⌋个记录到第1个记录依次进行筛选就可以建立堆。

可用下列语句实现：

```
for (j=n/2; j>=1; j--)
    Heap_adjust(R, j , n) ;
```

6 堆排序算法实现

堆的根结点是关键字最小的记录，输出根结点后，是以序列的最后一个记录作为根结点，而原来堆的左、右子树都是堆，则进行一次筛选就可以成为堆。

```
void  Heap_Sort(Sqlist *H)
  {  int j ;
     for (j=H->length/2; j>0; j--)
        Heap_adjust(H, j , H->length) ;  /*  初始建堆  */
```

```
    for (j=H->length/2; j>=1; j--)
      {  H->R[0]=H->R[1] ; H->R[1]=H->R[j] ;
         H->R[j]=H->R[0] ;   /*  堆顶与最后一个交换  */
         Heap_adjust(H, 1, j-1) ;
      }
  }
```

7 算法分析

主要过程：初始建堆和重新调整成堆。设记录数为n，所对应的完全二叉树深度为h 。

◆ 初始建堆：每个非叶子结点都要从上到下做“筛选”。第i层结点数≤2i-1，结点下移的最大深度是h-i，而每下移一层要比较2次，则比较次数C1(n)为：

$$C_1(n) \le 2\sum_{i=1}^{h-1}(2^{i-1}\times(h-i)) \le 4(2^h-h-1)$$

∵ $h=\lfloor \log_2 n \rfloor+1$，　∴ $C_1(n) \le 4(n-\log_2 n-1)$

◆ 筛选调整：每次筛选要将根结点“下沉”到一个合适位置。第i次筛选时：堆中元素个数为n-i+1；堆的深度是$\lfloor \log_2(n-i+1) \rfloor+1$，则进行n-1次“筛选”的比较次数$C_2(n)$为：

$$C_2(n) \le \sum_{i=1}^{n-1}(2\times \log_2(n-i+1))$$

∴ $C_2(n)<2n\log_2 n$

∴ 堆排序的比较次数的数量级为：$T(n)=O(n\log_2 n)$；而附加空间就是交换时所用的临时空间，故空间复杂度为：$S(n)=O(1)$。

各种排序方法的综合比较

排序方法	平均时间	辅助空间	稳定排序方法
冒泡排序	$O(n^2)$	$O(1)$	是
插入排序	$O(n^2)$	$O(1)$	是
选择排序	$O(n^2)$	$O(1)$	否
快速排序	$O(n\log n)$	$O(\log n)$	否
归并排序	$O(n\log n)$	$O(n)$	是