6

查找

查找概念

查找(或检索)是在给定信息集上寻找特定信息元素的过程。

待查找的数据单位(或数据元素)称为记录。记录由若干数据项(或属性)组成，如学生记录:

学号	姓名	性别	年龄	……

其中，“学号”、“姓名”、“性别”、“年龄”等都是记录的数据项。若某个数据项的值能标识(或识别)一个或一组记录，称其为关键字(key)。若一个key能唯一标识一个记录，称此key为主key。如“学号”的值给定就唯一对应一个学生，不可能多个学生的学号相同，故“学号”在学生记录里可作为主key。若一个key能标识一组记录，称此key为次key。如“年龄”值为20时，可能有若干同学的年龄为20岁，故“年龄”可作次key。下面主要讨论对主key的查找。

查找概念

查找定义

设记录表L=(R_1 R_2......R_n)，其中R_i(1≤i≤n)为记录，对给定的某个值k，在表L中确定key=k的记录的过程，称为查找。若表L中存在一个记录R_i的key=k，记为Ri.key=k，则查找成功，返回该记录在表L中的序号i(或Ri 的地址)，否则(查找失败)返回0(或空地址Null)。

查找方法

查找方法有顺序查找、折半查找、分块查找、Hash表查找等等。查找算法的优劣将影响到计算机的使用效率，应根据应用场合选择相应的查找算法。

查找概念

平均查找长度

评价一个算法的好坏，一是时间复杂度T(n)，n为问题的体积，此时为表长；二是空间复杂度D(n)；三是算法的结构等其他特性。

对查找算法，主要分析其T(n)。查找过程是key的比较过程，时间主要耗费在各记录的key与给定k值的比较上。比较次数越多，算法效率越差（即T(n)量级越高），故用“比较次数”刻画算法的T(n)。另外，不能以查找某个记录的时间来作为T(n)，一般以“平均查找长度”来衡量T(n)。

平均查找长度ASL（Average Search Length）：对给定k，查找表L中记录比较次数的期望值(或平均值)，即：

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

P_i为查找R_i的概率。等概率情况下P_i=1/n；C_i为查找R_i时key的比较次数(或查找次数)。

顺序表的查找

所谓顺序表(Sequential Table)，是将表中记录(R_1 R_2……R_n)按其序号存储于一维数组空间，如图所示。其特点是相邻记录的物理位置也是相邻的。

记录Ri的类型描述如下:

```
typedef struct
{ keytype key;  //记录key//
      ……          //记录其他项//
  } Retype;
```

其中，类型keytype是泛指，即keytype可以是int、float、char或其他的结构类型等等。为讨论问题方便，一般取key为整型。

顺序表类型描述如下:

```
#define maxn 1024     //表最大长度//
typedef struct {   Retype data[maxn];  //顺序表空间//
                   int len;   //当前表长，表空时len=0//
                 } sqlist;
```

若说明：sqlist r，则（r.data[1]，……，r.data[r.len]）为记录表(R_1……R_n)， R_i.key为r.data[i].key，r.data[0]称为监视哨，为算法设计方便所设。

顺序查找(Sequential Search)算法及分析

算法思路

设给定值为k，在表(R_1 R_2……R_n)中，从R_n开始，查找key=k的记录。若存在一个记录R_i（l≤i≤n）的key为k，则查找成功，返回记录序号i；否则，查找失败，返回0。

算法描述

```
int sqsearch(sqlist r, keytype k)  //对表r顺序查找的算法//
{   int i;
  r.data[0].key = k;  //k存入监视哨//
    i = r.len;  //取表长//
    while(r.data[i].key != k) i--;  //顺序往前查找//
    return (i);
}
```

算法用了一点技巧：先将k存入监视哨，若对某个i(≠0)有r.data[i].key=k，则查找成功，返回i；若i从n递减到1都无记录的key为k，i再减1为0时，必有r.data[0].key=k，说明查找失败，返回i=0。

顺序查找(Sequential Search)算法及分析

算法分析

设$C_i(1\le i\le n)$为查找第i记录的key比较次数(或查找次数)：

若r.data[n].key = k， $C_n=1$；

若r.data[n-1].key = k， $C_{n-1}=2$；

……

若r.data[i].key = k， $C_i=n-i+1$；

……

若r.data[1].key = k， $C_1=n$

故ASL = O(n)。而查找失败时，查找次数等于n+l，同样为O(n)。

对查找算法，若ASL=O(n)，则效率是很低的，意味着查找某记录几乎要扫描整个表，当表长n很大时，会令人无法忍受。下面关于查找的一些讨论，大多都是围绕降低算法的ASL量级而展开的。

折半查找算法及分析

当记录的key按关系≤或≥有序时，即：

$R_1.key \leq R_2.key \leq \cdots\cdots \leq R_n.key$ （升序）

或 $R_1.key \geq R_2.key \geq \cdots\cdots \geq R_n.key$ （降序）

算法思路

对给定值k，逐步确定待查记录所在区间，每次将搜索空间减少一半(折半)，直到查找成功或失败为止。

设两个指针(或游标)low、high，分别指向当前待查找表的上界(表头)和下界(表尾)。对于表(R_1 R_2……R_n)，初始时low=1、high=n，令：

mid=

指向当前待查找表中间的那个记录。下面举例说明折半查找的过程。

折半查找算法及分析

例1 设记录表的key序列如下:

序号: 1 2 3 4 5 6 7 8 9 10 11 12 (n=12)

1	2	3	4	5	6	7	8	9	10	11	12
03	12	18	20	32	55	60	68	80	86	90	100

low mid low high mid high

mid

现查找k=20的记录。

①mid=$\lfloor (1+12)/2 \rfloor$ =6。因k<r.data[6].key=55，若20存在，一定落在“55”的左半区间(搜索空间折半)。令:

high=mid-1。

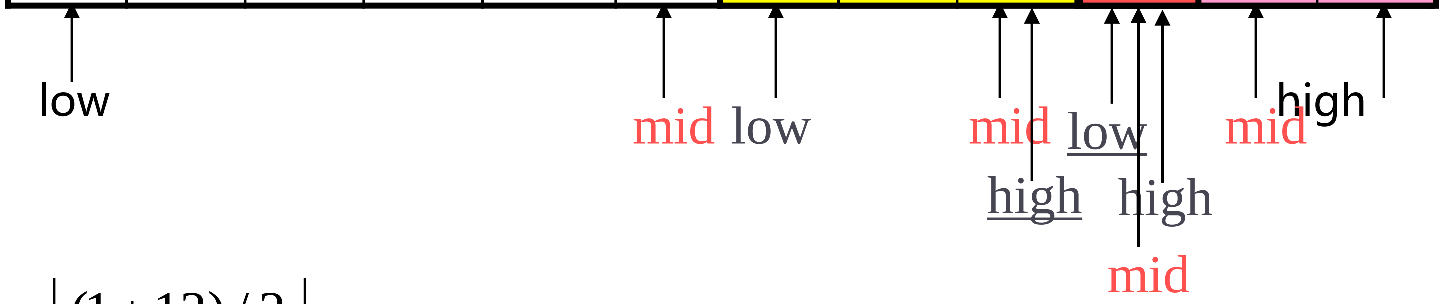
②②mid= $\lfloor (1+5)/2 \rfloor$=3。因k>r.data[3].key=18，若20存在，一定落在“18”的右半区间。令: low=mid+1。

②③mid= $\lfloor (4+5)/2 \rfloor$ =4。因k=r.data[4].key=20，查找成功，返回mid=4。

折半查找算法及分析

再看查找失败的情况，设要查找k=85的记录。

序号：	1	2	3	4	5	6	7	8	9	10	11	12 (n=12)
	03	**12**	**18**	**20**	**32**	**55**	**60**	**68**	**80**	**86**	**90**	**100**



①mid= $\lfloor (1+12)/2 \rfloor$ =6。因k>r.data[6].key=55，若85存在，一定落在“55”的右半区间。令： low=mid+1 。

②mid= $\lfloor (7+12)/2 \rfloor$ =9。因k>r.data[9].key=80，若85存在，一定落在“80”的右半区间。令：low=mid+1。

③mid= $(10+12)/2$ =11。因k<r.data[11].key=90，若85存在，一定落在“90”的左半区间。令：high=mid-1。

④mid= $\lfloor (10+10)/2 \rfloor$ =10。因k<r.data[10].key=86，若85存在，一定落在“86”的左半区间。令：high=mid-1。

此时，下界low=10，而上界high=9，表明搜索空间不存在，故查找失败，返回0。

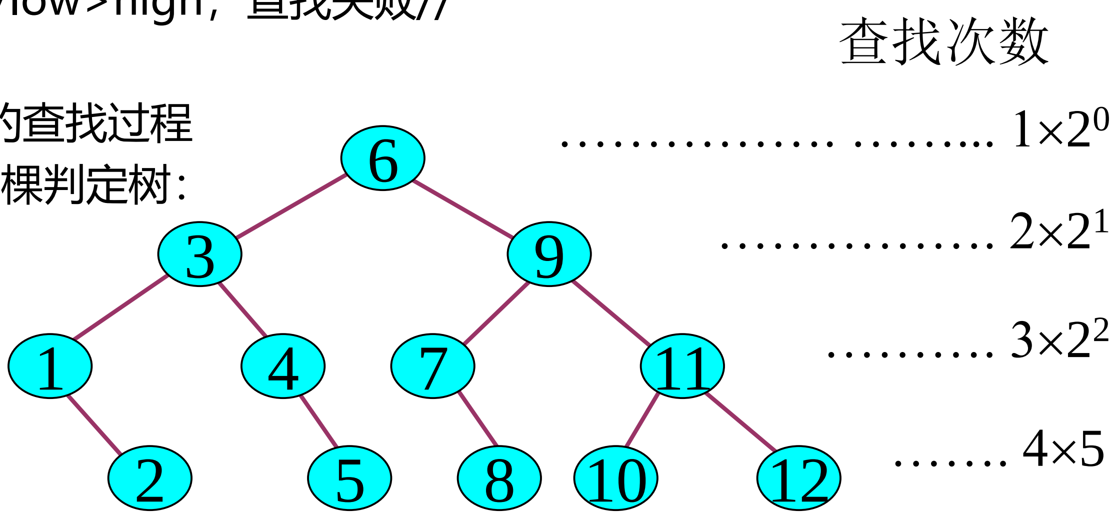
折半查找算法及分析

算法描述

```
int Binsearch(sqlist r, keytype k)    //对有序表r折半查找的算法//
{  int low, high, mid;  low = 1;high = r.len;   //上下界初值//
    while (low <= high)       //表空间存在时//
    {  mid = (low+high) / 2;   //求当前mid//
        if (k == r.data[mid].key)  return (mid);   //查找成功，返回mid//
        if (k < r.data[mid].key) high = mid-1;    //调整上界，向左部查找//
        else low = mid+1; }         //调整下界，向右部查找//
   return(0); }      //low>high，查找失败//
```

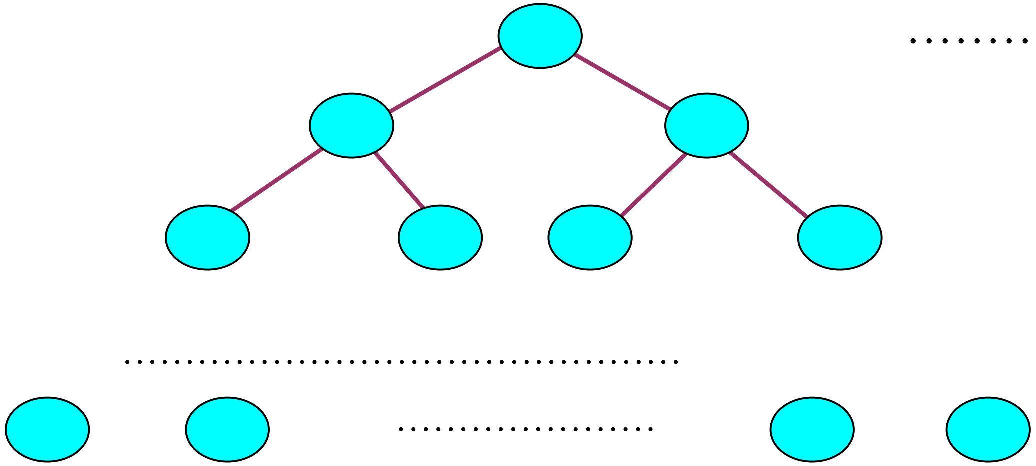
算法分析

对例1中记录表的查找过程可得到如图所示的一棵判定树：



折半查找算法及分析

不失一般性，设表长n=2h-l，h=log$_2$(n+1)。记录数n恰为一棵**h**层的满二叉树的结点数。对照例1，得出表的判定树及各记录的查找次数如图所示。



查找次数

……………… ……... 1×2^0

……………. 2×2^1

………. 3×2^2

………

….. $h\times2^{h-1}$

$$ASL=\sum_{i=1}^{n}P_iC_i=\frac{1}{n}\sum i\cdot2^{i-1}$$

$$\sum_{i=1}^{h}i\cdot2^{i-1}=1\cdot2^0+2\cdot2^1+3\cdot2^2+\ldots\ldots+(h-1)2^{h-2}+h\cdot2^{h-1}$$

$$2S=1\cdot2^1+2\cdot2^2+3\cdot2^3+\ldots\ldots+(h-1)2^{h-1}+h\cdot2^h$$

$$S=2S-S=h\cdot2^h-(2^0+2^1+2^2+\ldots\ldots+2^{h-1})=h\cdot2^h-(2^h-1)$$

$$(n+1)\log_2(n+1)-n$$

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

$n\longrightarrow\infty$ 时，ASL=O(log$_2$(n+1))，大大优于O(n)。

分块查找(索引顺序查找）算法及分析

分块

设记录表长为n，将表的n个记录分成b=　　个块，每块s个记录（最后一块记录数可以少于s个），即:

$$(\underbrace{R_1\cdots\cdots R_s}_{块\ 1}\ \underbrace{R_{s+1}\cdots\cdots R_{2s}}_{块\ 2}\ \cdots\cdots\ \underbrace{\cdots\cdots\ R_n}_{块\ b})$$

且表分块有序，即第i（1≤i≤b-1）块所有记录的key小于第i+1块中记录的key，但块内记录可以无序。

建立索引

每块对应一索引项:

k_{max}	link

其中k_{max}为该块内记录的最大key；link为该块第一记录的序号（或指针）。

分块查找(索引顺序查找)

例2 设表长n=19，取s=5，$b=\lceil 19/5 \rceil = 4$，分块索引结构，如图所示。

算法思路　分块索引查找分两步进行：

(1)由索引表确定待查找记录所在的块；

(2)在块内顺序查找。

如查找k=19的记录，因19>18，不会落在第1块；又19<50，若19存在，必在第2块内。取第2块起址(6)，查找到key为19的记录号为9。

查找失败情况，一是给定k值超出索引表范围；二是若k落在某块内，但该块中无key=k的记录。

索引表是按照k_{max}有序的，可对其折半查找。而块内按顺序方法查找。

K_{max}	序号
18	1
50	6
84	11
108	16

(索引表)

序号	R.key
1	18
2	10
3	9
4	8
5	16
6	20
7	38
8	42
9	19
10	50
11	84
12	72
13	56
14	55
15	76
16	100
17	90
18	88
19	108

Hash表的查找

Hash表的含义

Hash表，又称散列表。在前面讨论的顺序、折半、分块查找和树表的查找中，其ASL的量级在O(n)～O(log2n)之间。不论ASL在哪个量级，都与记录长度n有关。随着n的扩大，算法的效率会越来越低。ASL与n有关是因为记录在存储器中的存放是随机的，或者说记录的key与记录的存放地址无关，因而查找只能建立在key的“比较”基础上。

理想的查找方法是：对给定的k，不经任何比较便能获取所需的记录，其查找的时间复杂度为常数级O(C)。这就要求在建立记录表的时候，确定记录的key与其存储地址之间的关系f，即使key与记录的存放地址H相对应：

$$key \xrightarrow{\quad f \quad} H\text{:}\ \boxed{\text{记 录}}$$

或者说，记录按key存放。

Hash表的查找

之后，当要查找key=k的记录时，通过关系f就可得到相应记录的地址而获取记录，从而免去了key的比较过程。这个关系f就是所谓的Hash函数（或称散列函数、杂凑函数），记为H(key)。它实际上是一个地址映象函数，其自变量为记录的key，函数值为记录的存储地址（或称Hash地址）。

另外，不同的key可能得到同一个Hash地址，即当$key_1 \neq key_2$时，可能有$H(key_1)=H(key_2)$，此时称key_1和key_2为**同义词**。这种现象称为“**冲突**”或“碰撞”，因为一个数据单位只可存放一条记录。

一般，选取Hash函数只能做到使冲突尽可能少，却不能完全避免。这就要求在出现冲突之后，寻求适当的方法来解决冲突记录的存放问题。

Hash表的查找

例3 设记录的key集合为C语言的一些保留字，即k={case、char、float、for、int、while、struct、typedef、union、goto、viod、return、switch、if、break、continue、else}，构造关于k的Hash表时，可按不同的方法选取H(key)。

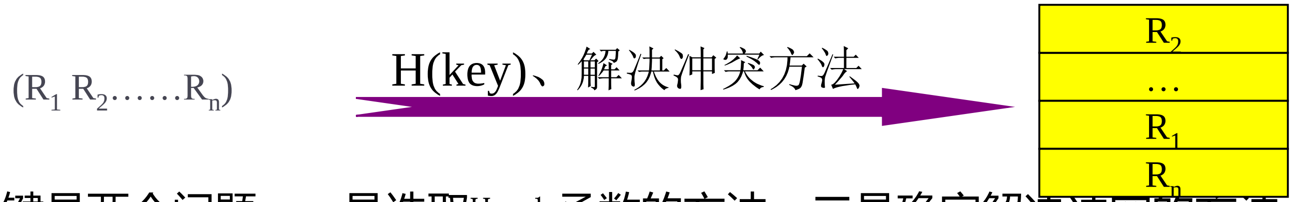
（1）令H_1(key)=key[0]-‘a’，其中key[0]为key的第一个字符。显然这样选取的Hash函数冲突现象频繁。如：H1(float)=H1(for)=‘f’-‘a’=5。解决冲突的方法之一是为“for”寻求另一个有效位置。

（2）令H_2(key)=(key[0]+key[i-1]-2*‘a’)/2。其中key[i-1]为key的最后一个字符。如：H_2(float)=(‘f’+‘t’-2*‘a’)/2=12，H_2(for)=(‘f’+‘r’-2*‘a’)/2=11，从而消除了一些冲突的发生，但仍无法完全避免，如：H_2(case)=H_2(continue)。

Hash表的查找

综上所述，对Hash表的含义描述如下：

根据选取的Hash函数H(key)和处理冲突的方法，将一组记录(R_1 R_2……R_n)映象到记录的存储空间，所得到的记录表称为Hash表，如图：



关于Hash表的讨论关键是两个问题，一是选取Hash函数的方法；二是确定解决冲突的方法。

选取（或构造）Hash函数的方法很多，原则是尽可能将记录均匀分布，以减少冲突现象的发生。以下介绍几种常用的构造方法。

直接地址法

平方取中法

叠加法

保留除数法

随机函数法

直接地址法

此方法是取key的某个线性函数为Hash函数，即令：

$$H(key)=a \cdot key+b$$

其中a、b为常数，此时称H(key)为直接Hash函数或自身函数。

例4 设某地区1～100岁的人口统计表如表：

记录	R_1	R_2	……	R_{25}	……	R_{100}
岁数	1	2	……	25	……	100
人数	3000	2500	……	20000	……	10

给定的存储空间为b+l～b+l00号单元（b为起始地址），每个单元可以存放一条记录Ri(l≤i≤l00)。取“岁数”为key，令：

$$H(key)=key+b$$

则按此函数构造的Hash表如下图所示：

保留除数法

又称质数除余法，设Hash表空间长度为m，选取一个不大于m的最大质数p，令： H(key)=key%p

例如：m=8 16 32 64 128 256 512 1024 ……

p=7 13 31 61 127 251 503 1019 ……

为何选取p为不大于m的最大质数呢？举例说明。

例8 设记录的key集合k={28，35，63，77，105……}，若选取p=21=3*7（包括质数因子7），有：

key：28 35 63 77 105 ……

H(key)=key%21： 7 14 0 14 0 ……

使得包含质数因子7的key都可能被映象到相同的单元，冲突现象严重。若取p=l9（质数），同样对上面给定的key集合k，有：

key：28 35 63 77 105

H(key)=key%19： 9 16 6 1 10

H(key)的随机度就好多了。

处理冲突的方法

选取随机度好的Hash函数可使冲突减少，一般来讲不能完全避免冲突。因此，如何处理冲突是Hash表不可缺少的另一方面。

设Hash表地址空间为0～m-l（表长为m）：

			…		R(k)		…		
H(key):	0	1		j-1	j	j+1			m-1

冲突是指：表中某地址j∈[0，m-1]中已存放有记录，而另一个记录的H(key)值也为j。处理冲突的方法一般为：在地址j的前面或后面找一个空闲单元存放冲突的记录，或将相冲突的诸记录拉成链表等等。

在处理冲突的过程中，可能发生一连串的冲突现象，即可能得到一个地址序列H_1、H_2……H_n，H_i∈[0，m-l]。H_1是冲突时选取的下一地址，而H_1中可能已有记录，又设法得到下一地址H_2……直到某个H_n不发生冲突为止。这种现象称为“**聚积**”，它严重影响了Hash表的查找效率。

冲突现象的发生有时并不完全是由于Hash函数的随机性不好引起的，聚积的发生也会加重冲突。还有一个因素是表的装填因子α，α=n/m，其中m为表长，n为表中记录个数。一般α在0.7～0.8之间，使表保持一定的空闲余量，以减少冲突和聚积现象。

开放地址法

当发生冲突时，在H(key)的前后找一个空闲单元来存放冲突的记 录，即在H(key)的基础上获取下一地址：

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

其中m为表长，%运算是保证H_i落在[0，m-l]区间；d_i为地址增量。d_i的取法有多种：

（1）d_i=1，2，3，……(m-1)——称为线性探查法；

（2）$d_i=1^2$，-1^2，2^2，-2^2，……——称为二次探查法。

式（1）、(2)表示：第1次发生冲突时，地址增量d_1取1或1^2；再冲突时，d_2取2或$-l^2$，……，依此类推。

开放地址法

例9 设记录的key集合**k={23，34，14，38，46，16，68，15，07，31，26}**，记录数**n**=11。令装填因子**α**=0.75，取表长**m**=⌈**n/α**⌉=15。用"保留余数法"选取Hash函数（**p**=13）：

H(key)=key%13

采用"线性探查法"解决冲突。依据以上条件，依次取k中各值构造的Hash表HT，如下图所示（表HT初始为空）。

- k={23，34，14，38，46，16，68，15，07，31，26}
- H(key)=key%13；H_i=(H(key)+d_i)%15；d_i=1，2，3，……(m-

HT:

H(key)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
HT	26	14	15	16	68	31	∧	46	34	07	23	∧	38	∧	∧

H(68)=68%13=3(冲突), 取H_1=(3+1)%15=4(空), 故68存入4单元。

H(07)=7%13=7(冲突), 取H_1=(7+1)%15=8(冲突), 取H_2=(7+2)%15=9(空),故07存入9单元。

若采用二次探测法：d_i=1^2，-1^2，2^2，-2^2，……，表为：

HT:

H(key)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
HT	**26**	**14**	**15**	**16**	**68**	**31**	**07**	**46**	**34**	∧	**23**	∧	**38**	∧	∧

其中，H(07)=7%13=7(冲突)， 取H_1=(7+ 1^2)%15=8(冲突)，取 H_2=(7 -1^2)%15=6(空)，故07存入6单元。

链地址法

发生冲突时，将各冲突记录链在一起，即同义词的记录存于同一链表。

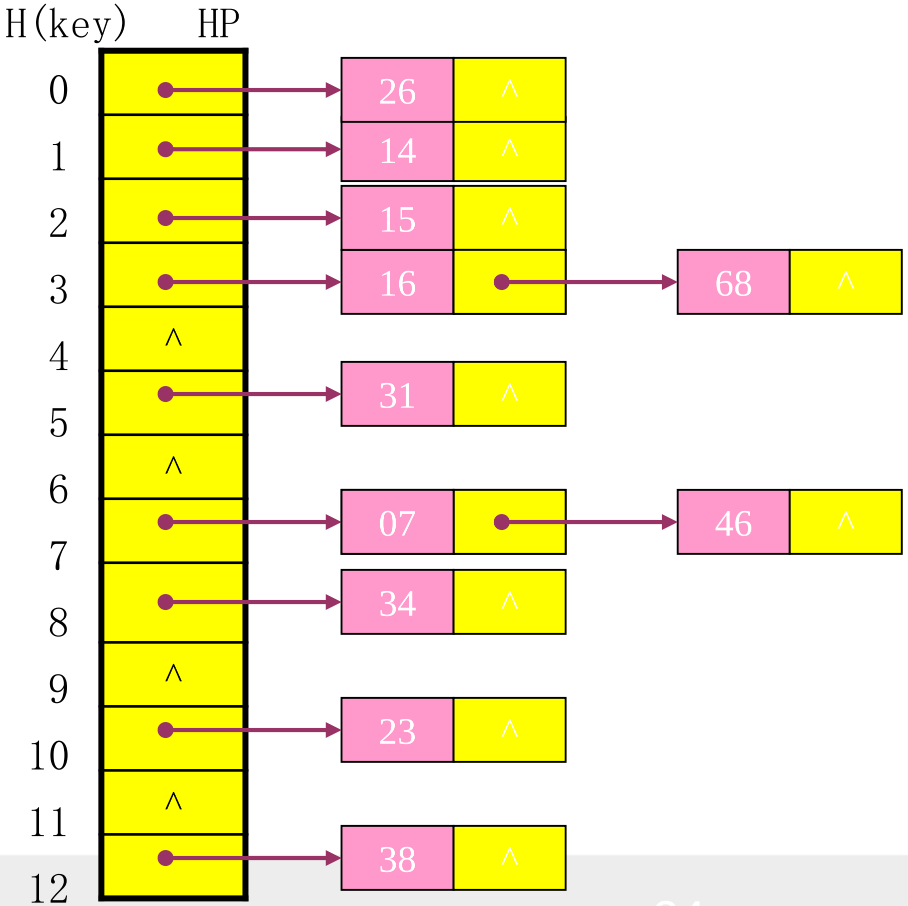
设H(key)取值范围（值域）为[0，m-l]，建立头指针向量HP[m]，HP[i]（0≤i≤m-l）初值为空。凡H(key)=i的记录都链入头指针为HP[i]的链表。

例10 设H(key)=key%13，

其值域为[0,12], 建立指针向量HP[l2]。 对例9中：

k={ 23，34，14，38，46，16，68，15，07，31，26 }

依次取其中各值，用链地址法解决冲突时的Hash表如**图**：



链地址法解决冲突的优点：无聚积现象；删除表中记录容易实现。而开放地址法的Hash表作删除时，不能将记录所在单元置空，只能作删除标记。

Hash表的查找及分析

Hash表的查找特点是：怎么构造的表就怎么查找，即造表与查找过程统一。

算法思路：对给定k，根据造表时选取的H(key)求H(k)。若H(k)单元=^，则查找失败，否则k与该单元存放的key比较，若相等，则查找成功；若不等，则根据设定的处理冲突方法，找下一地址H_i，直到查找到或等于空为止。

线性探查法解决冲突时Hash表的查找及插入

```
#define m 64  //设定表长m//
typedef struct
{ keytype key;  //记录关键字//
                ……
}Hretype;
Hretype HT[m];  //Hash表存储空间//
int Lhashsearch(Hretype HT[m],keytype k)  //线性探查法解决冲突时的查找//
{           int j,d,i=0;
            j=d=H(k);   //求Hash地址并赋给j和d//
            while((i<m)&&(HT[j].key!=NULL)&&(HT[j].key!=k))
            { i++;    j=(d+i)%m; } //冲突时形成下一地址//
            if(i==m) return(-1);  //表溢出时返回-1//
            else return(j);
}   //HT[j].key==k,查找成功;HT[j].key==NULL,查找失败//
void LHinsert(Hretype HT[m], Hretype R)  //记录R插入Hash表的算法//
{
            int j=LHashsearch(HT,R.key);  //查找R，确定其位置//
            if((j= = -1)||(HT[j].key==R.key)) ERROR();  //表溢出或记录已存//
            else HT[j]=R;                  //插入HT[j]单元//
}
```

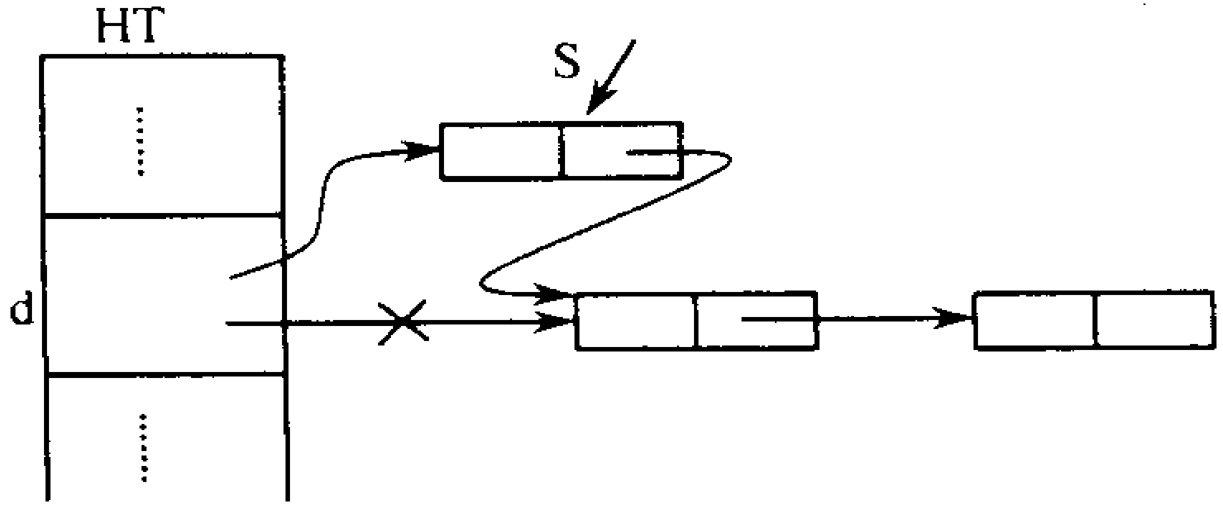
链地址法解决冲突时Hash表的查找及插入

```
typedef struct node  //记录对应结点//
        {       keytype key;
                ……
                struct node *next;
        }Renode;
        Renode *LinkHsearch(Renode *HT[m],keytype k) //链地址法解决冲突时的查找//
        {
                Renode *p;
                int d=H(k);  //求Hash地址d//
                p=HT[d];  //取链表头结点指针//
                while(p&&(p->key!=k))
                p=p->next;  //冲突时取下一同义词结点//
                return(p);
}       //查找成功时p->key==k，否则p=∧//
```

链地址法解决冲突时Hash表的查找及插入

```
void LinkHinsert(Renode *HT[m], Renode *S)//将指针S所指记录插入表HT的算法（如图所示）
{
        int d;
        Renode *p;
        p=LinkHsearch(HT,S->key); //查找S结点//
        if(p) ERROR();  //记录已存在//
        else
        {
        d=H(S->key);
        S->next=HT[d];
        HT[d]=S;
        }
}
```

插入图示：



7

排序

概述

排序(Sort)是将无序的记录序列（或称文件）调整成有序的序列。

对文件（File）进行排序有重要的意义。如果文件按key有序，可对其折半查找，使查找效率提高；在数据库（Data Base）和知识库（Knowledge Base）等系统中，一般要建立若干索引文件，就牵涉到排序问题；在一些计算机的应用系统中，要按不同的数据段作出若干统计，也涉及到排序。排序效率的高低，直接影响到计算机的工作效率。

排序定义

设含有n个记录的文件**f**=(R_1 R_2……R_n)，相应记录关键字（key）集合k={k_1 k_2……k_n}。若对1、2……n的一种排列：

$$P_{(1)}\ P_{(2)}\cdots\cdots P_{(n)}\quad (1\le P_{(i)}\le n,\ i\ne j\text{时},\ P_{(i)}\ne P_{(j)})$$

有： $k_{P(1)} \le k_{P(2)} \le \cdots\cdots \le k_{P(n)}$ ——递增关系

或 $k_{P(1)} \ge k_{P(2)} \ge \cdots\cdots \ge k_{P(n)}$ ——递减关系

则使**f** 按key线性有序：（$R_{P(1)}$ $R_{P(2)}$……$R_{P(n)}$），称这种运算为排序(或分类)。

关系符"≤"或"≥"并不一定是数学意义上的"小于等于"或"大于等于"，而是一种次序关系。但为讨论问题方便，取整型数作为key，故"≤"或"≥"可看作通常意义上的符号。

排序定义

稳定排序和非稳定排序

设文件f=（R_1……R_i……R_j……R_n）中记录R_i、R_j（i≠j，i、j=1……n）的key相等，即K_i=K_j。若在排序前R_i领先于R_j，排序后R_i仍领先于R_j，则称这种排序是稳定的，其含义是它没有破坏原本已有序的次序。反之，若排序后R_i与R_j的次序有可能颠倒，则这种排序是非稳定的，即它有可能破坏了原本已有序记录的次序。

内排序和外排序

若待排文件f在计算机的内存储器中，且排序过程也在内存中进行，称这种排序为内排序。内排序速度快，但由于内存容量一般很小，文件的长度（记录个数）n受到一定限制。若排序中的文件存入外存储器，排序过程借助于内外存数据交换（或归并）来完成，则称这种排序为外排序。我们重点讨论内排序的一些方法、算法以及时间复杂度的分析。

内排序方法

截止目前，各种内排序方法可归纳为以下五类：

（1）插入排序

（2）交换排序

（3）选择排序

（4）归并排序

（5）基数排序。

插入排序（Insert Sort）

直接插入排序

折半插入排序

链表插入排序

Shell（希尔）排序

……

直接插入排序

设待排文件f=（R_1 R_2……R_n）相应的key集合为k={k_1 k_2……k_n}，文件f对应的结构可以是前面所述的三种文件结构之一。

排序方法

先将文件中的（R_1）看成只含一个记录的有序子文件，然后从R_2起，逐个将R_2至R_n按key插入到当前有序子文件中，最后得到一个有序的文件。插入的过程上是一个key的比较过程，即每插入一个记录时，将其key与当前有序子表中的key进行比较，找到待插入记录的位置后，将其插入即可。另外，假定排序后的文件按递增次序排列（以下同）。

直接插入排序

例1 设文件记录的key集合k={50，36，66，76，95，12，25，36}（考虑到对记录次key排序的情况，允许多个key相同。如此例中有2个key为36，后一个表示成36，以示区别），按直接插入排序方法对k的排序过程如下：k={50，36，66，76，95，12，25，36}

[50] 36

[36, 50] 66

[36, 50, 66] 76

[36, 50, 66, 76] 95

[36, 50, 66, 76, 95] 12

[12, 36, 50, 66, 76, 95] 25

[12, 25, 36, 50, 66, 76, 95] 36

[12, 25, 36, 36, 50, 66, 76, 95]

直接插入排序

一般, 插入k_i（2≤i≤n），当$k_j \le k_i < k_{j+1} \ldots\ldots k_{i-1}$时，先将子表$[k_{j+1} \ldots\ldots k_{i-1}]$从$k_{i-1}$起顺序向后移动一个记录位置，然后$k_i$插入到第j+1位置，即:

$$[k_1 \ \ldots \ k_j \quad k_{j+1} \ \ldots \ k_{i-1}] \quad k_i$$

$$[k_1 \ldots k_j \ \ k_i \quad k_{j+1} \ \ldots \ k_{i-1}]$$

文件结构说明:

```
#define maxsize 1024   ‖文件最大长度‖
typedef  int keytype;   ‖设key为整型‖
typedef struct   ‖记录类型‖
{ keytype key;  ‖记录关键字‖
    ……
    ……     ‖记录其它域‖
}Retype;
typedef struct   ‖文件或表的类型‖
{ Retype   R[maxsize+1];    ‖文件存储空间‖
   int len;    ‖当前记录数‖
} sqfile;
```

若说明sqfile F；则：（F.R[1]，F.R[2]……F.R[F.len]）为待排文件，它是一种顺序结构；文件长度n=F.len；F.R[0]为工作单元，起到“监视哨”作用；记录的关键字ki写作F.R[i].key。

直接插入排序算法描述

```
void Insort (sqfile F)  ∥对顺序文件F直接插入排序的算法∥
{  int i, j;
   for (i=2; i<=F.len; i++)   ∥插入n - 1个记录∥
   {  F.R[0] = F.R[i];  ∥待插入记录先存于监视哨∥
      j = i-1;
      while (F.R[0].key < F.R[j].key)   ∥key比较∥
     {  F.R[j+1] = F.R[j];    ∥记录顺序后移∥
         j--;
      }
      F.R[j+1] = F.R[0];  ∥原R[i]插入j+1位置∥
   }
}
```

排序的时间复杂度取耗时最高的量级，故直接插入排序的**$T(n)=O(n^2)$。**

折半插入排序

排序算法的$T(n)=O(n^2)$,是内排序时耗最高的时间复杂度。随着文件记录数n的增大，效率降低较快。下面的一些插入排序的方法，都是围绕着改善算法的时间复杂度而展开的。另外，直接插入排序是稳定排序。

为减少插入排序过程中key的比较次数，可采用折半插入排序。

排序方法

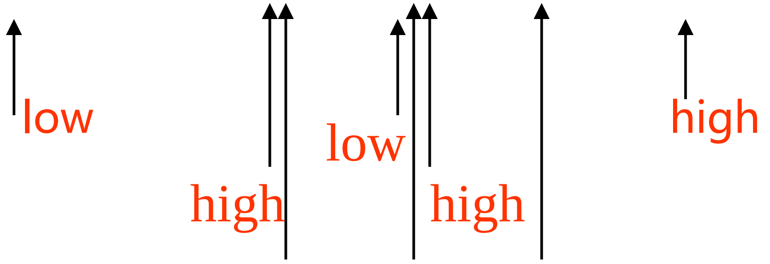
同直接插入排序，先将（R[1]）看成一个子文件，然后依次插入R[2]……R[n]。但在插入R[i]时，子表[R[1]……R[i-1]]已是有序的，查找R[i]在子表中的位置可按折半查找方法进行，从而降低key的比较次数。

折半插入排序

例2 设当前子表key序列及插入的k_i=28如下：

序号： 1 2 3 4 5 6

[15 20 25 3 0 35 40] **28**



令：

$$mid = \lfloor (low + high)/2 \rfloor$$

28>R[3].key=25，low=3+1=4；

28<R[5].key=35， high=5－1=4；

28<R[4].key=30，high=4－1=3。
此时，low>high，所以“28”应插在low=4所指示的位置 。

折半插入排序算法描述

```
void Binsort (sqfile F)   ‖对文件F折半插入排序的算法‖
{   int i, j, low, high, mid;
    for (i=2; i<=F.len; i++)   ‖插入n - 1个记录‖
    {  F.R[0] = F.R[i];    ‖待插入记录存入监视哨‖
       low = 1; high = i-1;
       while (low  <= high)    ‖查找R[i]的位置‖
       {  mid = (low+high) / 2;
          if (F.R[0].key >= F.R[mid].key)   low = mid+1;    ‖调整下界‖
          else  high = mid-1;    ‖调整上界‖
       }
       for (j=i-1; j>=low; j--)
       F.R[j+1] = F.R[j];    ‖记录顺移‖
       F.R[low] = F.R[0];     ‖原R[i]插入low位置‖
    }
}
```

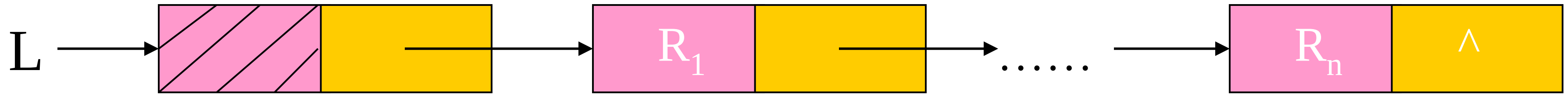
插入R[i]（2≤i≤n）时，子表记录数为i - 1。同折半查找算法的讨论，查找R[i]的key比较次数为 ，故总的key比较次数C为：

$$C = \sum_{i=2}^{n} \log_2 i < (n-1)\log_2 n = O(n\log_2 n)$$

显然优于O(n^2)。但折半插入排序的记录移动次数并未减少，仍为O(n^2) 。故算法的时间复杂度T(n)仍为**O(n^2)** 。

链表插入排序

设待排序文件f=（R_1 R_2……R_n），对应的存储结构为单链表结构，如图：

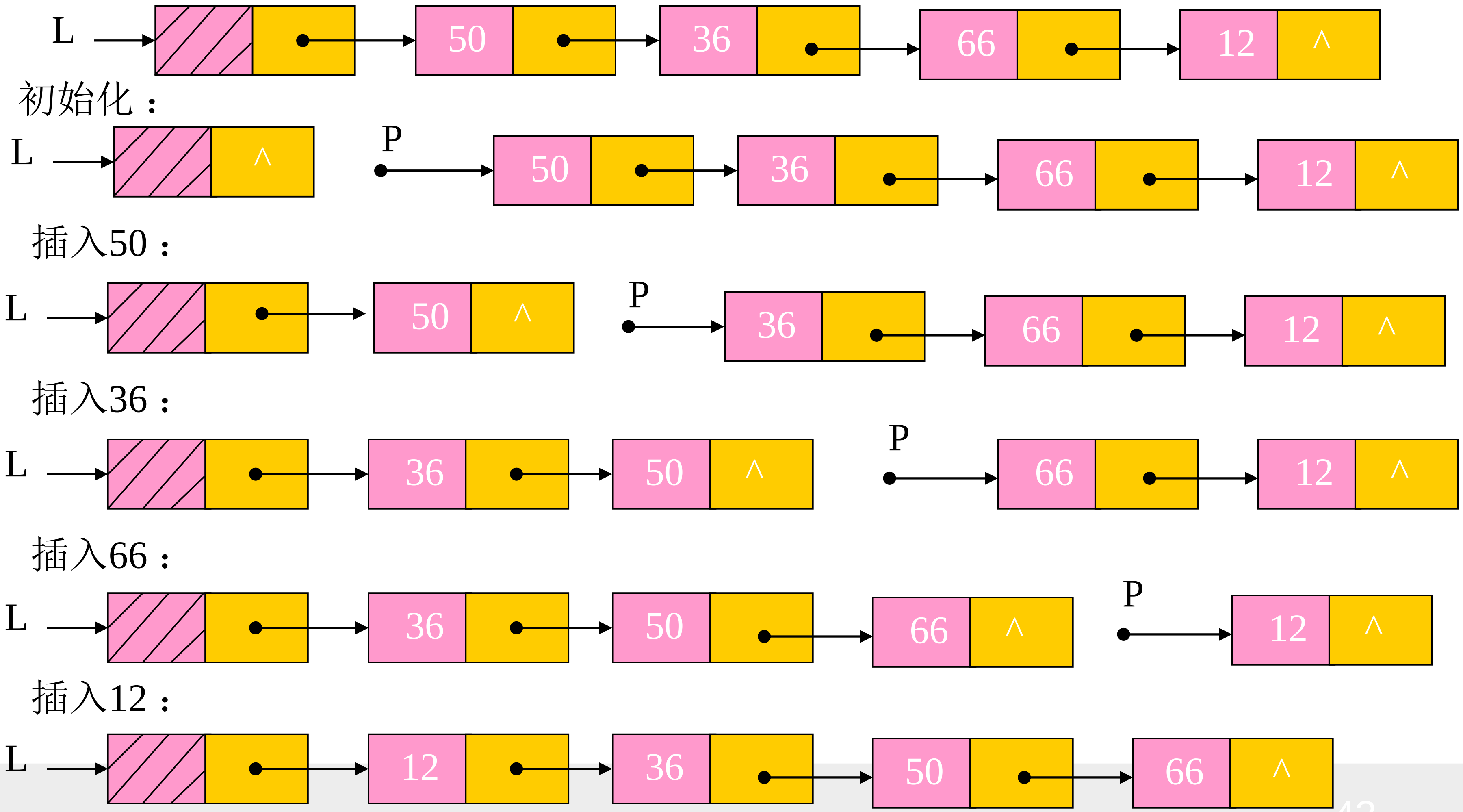


排序方法

链表插入排序实际上是一个对链表遍历的过程。先将表置为空表，然后依次扫描链表中每个结点，设其指针为p，搜索到p结点在当前子表的适当位置，将其插入。

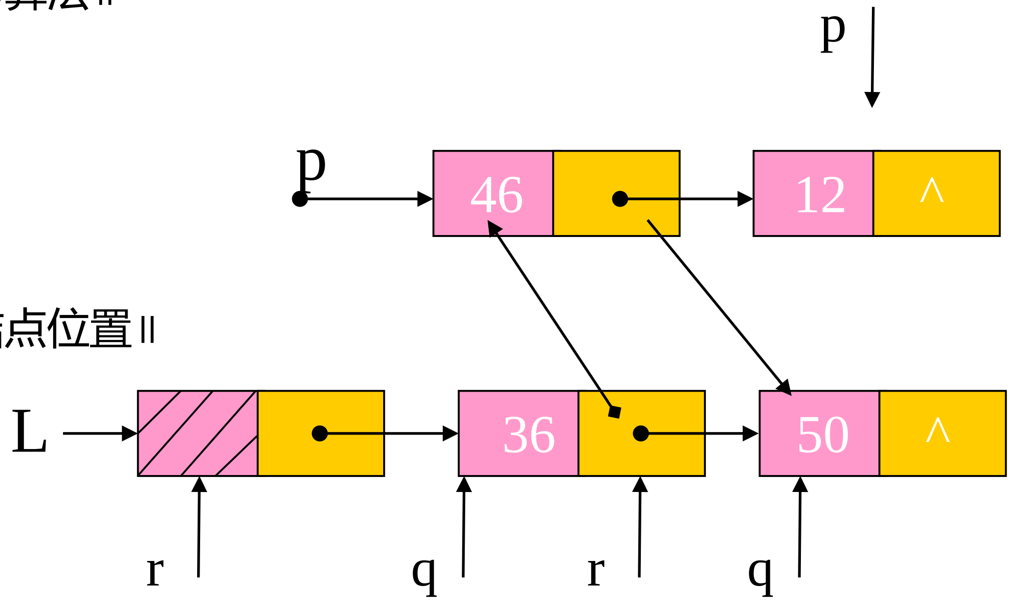
链表插入排序

例3 设含4个记录的链表如图：



链表插入排序算法描述

```
typedef struct node_t   //存放记录的结点//
{  keytype key;    //记录关键字//
     ……             //记录其它域//
   struct node _t *next;   //链指针//
} linkode_t,*linklist_t;
void  Linsertsort (linklist_t L)   //链表插入排序算法//
{  linklist_t p, q, r, u;
   p = L->next;          //p为待插入结点指针//
   L->next=NULL;  //置子表为空//
   while ( p )  //若待排序结点存在//
   {  r = L; q = L->next;  //r为q的前驱指针//
      while (q && q->key<=p->key)  //找p结点位置//
      {  r = q; q = q->next; }
      u = p->next;
      p->next = q; r->next = p; //插入//
      p=u;
    }
  }
```



交换排序

"起泡"排序（Bubble Sort）

"快速"排序（Quick Sort）

起泡排序

设待排文件f=（R_1 R_2……R_n），相应key集合k={k_1 k_2……k_n}。

排序方法

从k_1起，两两key比较，逆序时交换，即：

k_1~k_2，若k_1>k_2，则$R_1 \Leftrightarrow R_2$；

k_2~k_3，若k_2>k_3，则$R_2 \Leftrightarrow R_3$；

……

k_{n-1}~k_n，若k_{n-1}>k_n，则$R_{n-1} \Leftrightarrow R_n$。

经过一趟比较之后，最大key的记录沉底，类似水泡。接着对前n - 1个记录重复上述过程，直到排序完毕。

起泡排序的时间复杂度**T(n)=O(n^2)**。

注意：在某趟排序的比较中，若发现两两比较无一记录交换，则说明文件已经有序，不必进行到最后两个记录的比较。

起泡排序

例5 设记录key集合k={50，36，66，76，95，12，25，36}，排序过程如下：

K	第1趟	第2趟	第3趟	第4趟	第5趟	第6趟
50	36	36	36	36	12	12
36	50	50	50	12	25	25
66	66	66	12	25	36	3
76	76	12	25	<u>36</u>	<u>3</u>	
95	12	25	<u>36</u>	5		
12	25	<u>36</u>	6			
25	<u>36</u>	7				
<u>36</u>	9					

从此例可以看出，起泡排序属于稳定排序。

起泡排序算法描述

```
void Bubsort (sqfile F)  ∥对顺序文件起泡排序的算法∥
{   int i, flag;  ∥flag为记录交换的标记∥
     Retype temp;
     for (i=F.len; i>=2; i--)   ∥最多n - 1趟排序∥
     {   flag = 0;
          for (j=1; j<=i-1; j++)  ∥一趟的起泡排序∥
          if (F.R[j].key > F.R[j+1])  ∥两两比较∥
          {  temp = F.R[j];   ∥R[j] ⇔ R[j+1]∥
              F.R[j] = F.R[j+1];
              F.R[j+1] = temp;
              flag=1;
           }
           if (flag == 0)  break;  ∥无记录交换时排序完毕∥
       }
  }
```

快速排序

快速排序是对起泡排序的一种改进，目的是提高排序的效率。

排序方法

经过key的一趟比较后，确定某个记录在排序后的最终位置。

设待排文件的key集合**$k=\{k_1\ k_2 \ldots\ldots k_i \ldots\ldots k_j \ldots\ldots k_{n-1}\ k_n\}$**，对k中的**$k_1$**，称作枢轴（Pirot）或基准。

（1）**逆序比较**：$k_1 \sim k_n$，若$k_1 \le k_n$，则k_1不可能在k_n位置，$k_1 \sim k_{n-1}$，……直到有个k_j，使得$k_1 > k_j$，则k_1有可能落在k_j位置，将$k_j \Rightarrow k_1$位置，即key比基准（k_1）小的记录向左移。

（2）**正序比较**：$k_1 \sim k_2$，若$k_1 > k_2$，则k_1不可能在k_2位置，$k_1 \sim k_3$，……直到有个k_i，使得$k_1 < k_i$，则k_1有可能落在k_i位置，将$k_i \Rightarrow k_j$位置（原k_j已送走），即key比基准大的记录向右移。

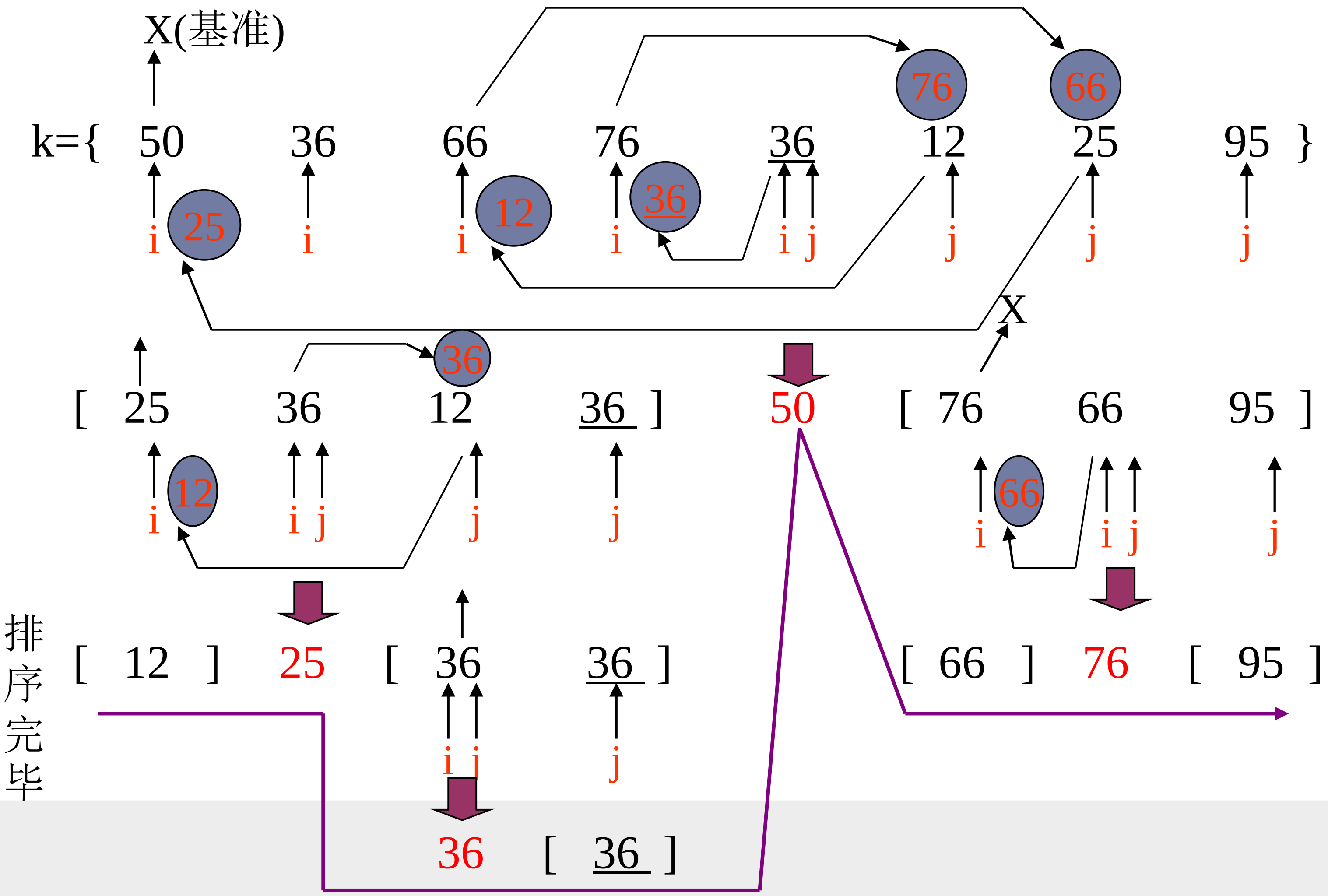
反复逆序、正序比较，当**i=j**时，**i**位置就是基准**k_1**的最终落脚点（因为比基准小的key统统在其“左部”，比基准大的统统在其“右部”，作为基准的key自然落在排序后的最终位置上），并且**k_1**将原文件分成了两部分：

k’	k_1	k”
左部		右部

对**k’**和**k”**，套用上述排序过程(可递归)，直到每个子表只含有一项时，排序完毕。

快速排序

例6 设记录的key集合**k={50，36，66，76，36，12，25，95}**，每次以集合中第一个key为基准的快速排序过程如下：



快速排序算法描述

```
typedef struct   //栈元素类型//
 {  int low, high;    //当前子表的上下界//
 } stacktype;
int qkpass(sqfile F, int low, int  high)  //对子表(R[low]……R[high])一趟快排算法//
{   int i = low,  j = high;
     keytype  x = F.R[low].key;   //存入基准key//
     F.R[0] = F.R[low];              //存入基准记录//
     while (i < j)
    {  while (i<j && x<=F.R[j].key)  j--;   //逆序比较//
       if (i < j)  F.R[i] = F.R[j];   //比x小的key左移//
       while (i<j && x>=F.R[i].key)  i++;  //正序比较//
       if (i < j)  F.R[j] = F.R[i];   //比x大的key右移//
     }
     F.R[i] = F.R[0];   //基准记录存入第i位置//
     return (i);   //返回基准位置//
 }
```

快速排序算法描述

```
void  qksort (sqfile F)  //对文件F快速排序的算法（非递归）//
{   int i, low, high;
     stacktype  u;      //栈元素//
     S = CreateStack();  //置栈空//
     u.low = 1;  u.high = F.len;  PushStack(S, u); //上下界初值进栈//
     while ( ! EmptyStack (S) )
    {  u = PopStack(S);     //退栈//
        low = u.low;  high = u.high;  //当前表的上下界//
        while (low < high)
        {  i  = qkpass(F, low, high);   //对当前子表的一趟快排//
            if (i+1 < high)
           { u.low = i+1; u.high = high;  PushStack (S, u); }  //i位置的右部上下界进栈//
            high = i - 1;  //排当前左部//
        }
    }
 }
```

海量视频 贴身学习

超多干货 实时更新

THANKS

— 谢谢 —