第10章 内部排序

1

10.1 概述

1. 什么是排序？

将数据元素/记录的任意序列，排成按关键字有序的序列。

2. 排序的目的是什么？ ——便于查找！

3. 排序算法的好坏如何衡量？

- **时间效率**——排序速度（即排序所花费的全部比较次数）
- **空间效率**——占内存辅助空间的大小
- **稳定性**——若两记录A和B的关键字值相等，但排序后A、B的先后次序保持不变，则称这种排序算法是稳定的。

2

4. 什么叫内部排序？什么叫外部排序？

——若待排序记录都在内存中，称为内部排序；

——若待排序记录一部分在内存，一部分在外存，则称为外部排序。

注：外部排序时，要将数据分批调入内存来排序，中间结果还要及时放入外存，外部排序更复杂。

5. 待排序记录在内存中怎样存储和处理？

① 顺序排序——排序时直接移动记录；

② 链表排序——排序时只修改指针；

③ 地址排序——排序时先移动地址，最后再移动记录。

注：地址排序中可以增设一维数组来专门存放记录的地址。

3

6. 顺序存储（顺序表）的抽象数据类型如何表示？

注：大多数排序算法都是针对顺序表结构的(便于直接移动元素)

```
# define MAXSIZE 20      //设记录数均不超过20个
typedef  int  KeyType ;   //设关键字为整型量（int型）
Typedef  struct {              //定义每个记录（数据元素）的结构
   KeyType     key ;           //关键字
   InfoType    otherinfo;      //其它数据项
}RecordType，node ;            //例如node.key表示其中一个分量
Typedef  struct {                          //定义顺序表L的结构
  RecordType  r [ MAXSIZE +1 ];  //存储顺序表的向量
                                           //r[0]一般作哨兵或缓冲区
  int length ;                             //顺序表的长度
}SqList , L;                  //例如L.r或L.length表示其中一个分量
```

若r数组是表L中的一个分量且为node类型，则r中某元素的key分量表示为：r[i].key

4

7. 内部排序的算法有哪些？

——按排序的规则不同，可分为5类：

- 插入排序
- 交换排序（重点是快速排序）
- 选择排序
- 归并排序
- 基数排序

——按排序算法的时间复杂度不同，可分为3类：

- 简单的排序算法：时间效率低，$O(n^2)$
- 先进的排序算法：时间效率高，$O(n\log_2 n)$
- 基数排序算法：时间效率高，$O(d\times n)$

d = 关键字的位数(长度)

5

10.2 插入排序

插入排序的基本思想是：每步将一个待排序记录，按其关键码大小，插入到前面已经排好序的一组记录的适当位置上，直到记录全部插入为止。

基本策略：通过插入实现排序，保证子序列是有序的。

插入排序有多种具体实现算法：

1) 直接插入排序
2) 折半插入排序
3) 2-路插入排序
4) 表插入排序

（2)–(4) 小改进

5) 希尔排序 → 大改进

6

1）直接插入排序

最简单的排序法！

新元素插入到哪里？在已形成的有序表中线性查找，并在适当位置插入，把原来位置上的元素向后顺移。

例1：关键字序列T=（13，6，3，31，9，27，5，11），请写出直接插入排序的中间过程序列。

【13】, 6, 3, 31, 9, 27, 5, 11
【6, 13】, 3, 31, 9, 27, 5, 11
【3, 6, 13】, 31, 9, 27, 5, 11
【3, 6, 13，31】, 9, 27, 5, 11
【3, 6, 9, 13，31】, 27, 5, 11
【3, 6, 9, 13，27, 31】, 5, 11
【3, 5, 6, 9, 13，27, 31】, 11
【3, 5, 6, 9, 11，13，27, 31】

7

直接插入排序算法的实现：

8

```
void InsertSort ( SqList &L ) { //对顺序表L作直接插入排序
  for ( i = 2;  i <=L.length; ++ i )     //直接在原始无序表L中排序
    if (L.r[i].key < L.r[i-1].key)   //若L.r[i]较小则插入有序子表内
     { L.r[0]= L.r[i];          //先将待插入的元素放入“哨兵”位置
       L.r[i]= L.r[i-1];        //子表元素开始后移
       for ( j=i-2; L.r[0].key < L.r[j].key; --j ) L.r[j+1]= L.r[j];
                                //只要子表元素比哨兵大就不断后移
       L.r[j+1]= L.r[0];     //直到子表元素小于哨兵，将哨兵值送入
                             //当前要插入的位置（包括插入到表首）
     } //if
}// InsertSort
```

L.r

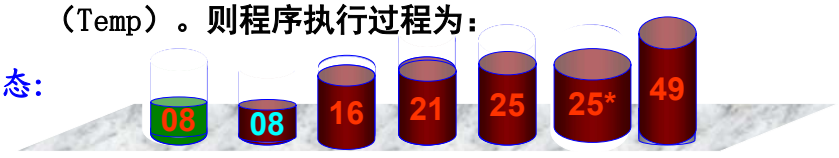
	*	*	…	*	*	*	*	…	*
0	1	2	…	i-2	i-1	i	i+1	…	L.length

例1： 关键字序列T=（21，25，49，25*，16，08），
请写出直接插入排序的具体实现过程。

*表示后一个25

解： 假设该序列已存入一维数组r[7]中，将r[0]作为哨兵（Temp）。则程序执行过程为：

初态：



完成！

时间效率：最好情形：比较n-1次，移动0次。最坏情况下，所有元素比较次数总和为 $(2+3+\ldots+n)\rightarrow O(n^2)$。

此时移动元素的次数也为 $(3+4+\ldots+n+1)=O(n^2)$，故平均时间复杂度为$O(n^2)$。

空间效率：仅占用1个缓冲单元—— $O(1)$

算法的稳定性：因为25*排序后仍然在25的后面——稳定

9

2） 折半插入排序

改进方法：既然子表有序且为顺序存储结构，则插入时采用折半查找可加速。

对应算法见教材（仅用于顺序表）

优点：比较次数减少，全部元素比较次数仅为$O(n\log_2 n)$。

时间效率：虽然比较次数减少，但移动次数并未减少，所以排序效率仍为$O(n^2)$ 。

空间效率：仍为 $O(1)$

稳 定 性：稳定

10

2） 折半插入排序

```
//对顺序表L按非递减有序进行折半插入排序
void BInsertSort(SqList &L) {
   for(i=2; i<=L.length; ++i) {
      L.r[0] = L.r[i]; low = 1; high = i-1;
      while(low <= high) {    //在r[low..high]中折半查找插入的位置
         m = ( low + high ) / 2 ;
         if(L.r[0].key < L.r[m].key) high = m-1;
         else low = m + 1;
      }
      for(j=i-1; j>=high+1; --j) L.r[j+1] = L.r[j]; //记录后移
      L.r[high+1] = L.r[0];          //将r[0]即原r[i]，插入正确位置
   }
}
```

思考：折半插入排序还可以改进吗？
能否减少移动次数？

11

3） 2-路插入排序

对折半插入排序的一种改进，其目的是减少排序过程中的移动次数。

代价：需要增加n个记录的辅助空间。

思路：增开辅助数组d，大小与r相同。

首先将r[1]赋值给d[1]，然后以d[1]内容为中值，将r[i]元素逐个插入到d[1]值之前或之后的有序序列中。

可把d设为循环向量，再设头尾两个指针。

例：待排序列T=(49，38，65，97, 76, 13, 27, 49*，55，04）

排序中途: d=(49，65，76, 97),　　(13, 27, 38)

↑final　　↑first

（参见教材）

12

讨论：若记录是链表结构，用直接插入排序行否？

答：行，而且无需移动元素，时间效率更高！

但是：单链表结构无法实现“折半查找”

4）表插入排序**

基本思想：在顺序存储结构中，给每个记录增开一个指针分量，在排序过程中将指针内容逐个修改为已经整理（排序）过的后继记录地址，形成静态(循环）有序链表。

优点：排序过程中不移动元素，只修改指针，最终形成有序链表。

此方法具有链表排序和地址排序的特点

13

例：关键字序列 T=(21，25，49，25*，16，08），
请写出表插入排序的具体实现过程。

解：假设该序列（结构类型)已存入一维数组r[7]中，将r[0]作为表头结点。则算法执行过程为：

	i	关键字 r[i].key	指针 r[i].link
初态	0	MAXINT	6
i=1	1	21	2
i=2	2	25	4
i=3	3	49	0
i=4	4	25*	3
i=5	5	16	1
i=6	6	08	5

表头结点

指向第1个元素

14

表插入排序的算法

```
int LinkInsertSort (Linklist &L) {
  L.r[0].Key = MaxNum;
  L.r[0]. Link = 1;
  L.r[1].Link = 0;                    //形成初始循环链表
  for ( int i = 2;  i <= L.length; i++ ) {
       int  current = L.r[0]. Link;     //current=当前记录指针
       int pre = 0;                     //pre=当前记录current的前驱指针
      while ( L.r[current]. Key <= L.r[i]. Key) {
         pre = current;          //current指针准备后移，pre跟上;
        current = L.r[current]. Link; }   //找插入点(即p=p->link)
      L.r[i]. Link = current;    //新记录r[i]找到合适序位开始插入
      L.r[pre]. Link = i;        //在pre与current 之间链入
   } //for
} //LinkInsertSort
```

从首结点开始查找

表插入排序算法分析：

① 无需移动记录，只需修改2n次指针值。但由于比较次数没有减少，故时间效率仍为$O(n^2)$ 。

② 空间效率低，因为增开了指针分量（但在运算过程中没有用到更多的辅助单元）。

③ 稳定性：25和25*排序前后次序未变，稳定。

注：此算法得到的只是一个有序链表，查找记录时只能满足顺序查找方式。

改进：可以根据表中指针线索，对所有记录重排，形成真正的有序表（顺序存储），从而能满足折半查找方式。

具体实现见严教材P269。

5）希尔（shell）排序 又称缩小增量排序

基本思想：先将整个待排记录序列分割成若干子序列，分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行一次直接插入排序。

技巧：子序列的构成不是简单地“逐段分割”，而是将相隔某个增量dk的记录组成一个子序列，让增量dk逐趟缩短（例如依次取5, 3, 1），直到dk＝1为止。

优点：关键字值小的元素能很快前移，且若序列基本有序时，再用直接插入排序处理，时间效率会高很多。

17

例：关键字序列T=(49，38，65，97, 76, 13, 27, 49*，55, 04），请写出希尔排序的具体实现过程。

r[i]	0	1	2	3	4	5	6	7	8	9	10
初态:		49	38	65	97	76	13	27	49*	55	04
第1趟 (dk=5)		13	27	49*	55	04	49	38	65	97	76
第2趟 (dk=3)		13	04	49*	38	27	49	55	65	97	76
第3趟 (dk=1)		04	13	27	38	49*	49	55	65	76	97

算法分析：

开始时*dk* 的值较大，子序列中的对象较少，排序速度较快；

随着排序进行，*dk* 值逐渐变小，子序列中对象个数逐渐变多，但此时大多数对象已基本有序，所以排序速度仍然很快。

18

时间效率：$O(n^{1.25})$～$O(1.6n^{1.25})$——由经验公式得到

空间效率：O（1）——因为仅占用1个缓冲单元

算法的稳定性：不稳定——因为49*排序后却到了49的前面

课堂练习：

1. 欲将序列（Q, H, C, Y, P, A, M, S, R, D, F, X）中的关键码按字母升序重排，则初始步长为4的希尔排序一趟的结果是？

答：原始序列：　Q, H, C, Y, P, A, M, S, R, D, F, X

shell一趟后：P, A, C, S, Q, D, F, X, R, H, M, Y

19

2. 以关键字序列（256，301，751，129，937，863，742，694，076，438）为例，写出执行希尔排序（取dk=5,3,1）算法的各趟排序结束时，关键字序列的状态。

原始序列：256，301，751，129，937，863，742，694，076，438

希尔排序

第1趟 256，301，694，076，438，863，742，751，129，937
dk=5

第2趟 076，301，129，256，438，694，742，751，863，937
dk=3

第3趟 076，129，256，301，438，694，742，751，863，937
dk=1

(取dk=5, 3, 1)

20

希尔排序算法（主程序）

```
void   ShellSort(SqList &L，int dlta[ ]，int t){
          //按增量序列dlta[0…t-1]对顺序表L作Shell排序
  for(k=0;  k<t;  ++k)
   ShellInSert(L，dlta[k]);  //增量为dlta[k]的一趟插入排序
} // ShellSort
```

dk值依次装在dlta[t]中

希尔排序算法（其中某一趟的排序操作）

```
void   ShellInsert(SqList &L，int dk) {
        //对顺序表L进行一趟增量为dk的Shell排序，dk为步长因子
for(i=dk+1;  i<=L.length;   ++ i)
   if(r[i].key < r[i-dk].key) {     //开始将r[i] 插入有序增量子表
    r[0]=r[i];                      //暂存在r[0]，此处r[0]仍是哨兵
    for(j=i-dk; j>0&&(r[0].key<r[j].key); j-=dk) r[j+dk]=r[j];
                                    //关键字较大的记录在子表中不断后移
    r[j+dk]=r[0];                   //在本趟结束时将r[i]插入到正确位置
   }//i
} //ShellInsert
```

理解难点：整理动作是二合为一的，r[0]仍是每个dk子集的哨兵，用于子集的彻底排序！

10.3 交换排序

交换排序的基本思想是:

两两比较待排序记录的关键字，如果发生逆序，则交换之，直到所有记录都排好序为止。

交换排序的主要算法有：

1) 冒泡排序$O(n^2)$

2) 快速排序$O(n\log_2 n)$

1) 冒泡排序

基本思路：每趟不断将记录两两比较，并按“前小后大”（或“前大后小”）规则交换。

优点：每趟结束时，不仅能挤出一个最大值到最后面位置，还能同时部分理顺其他元素；一旦下趟没有交换发生，还可以提前结束排序。

前提：顺序存储结构

例：关键字序列 T=(21，25，49，25*，16，08），请写出冒泡排序的具体实现过程。

初态: 21，25，49， 25*，16， 08
第1趟 21，25，25*，16， 08， 49
第2趟 21，25， 16， 08， 25*，49
第3趟 21，16， 08， 25， 25*，49
第4趟 16，08， 21， 25， 25*，49
第5趟 08，16， 21， 25， 25*，49

```
//对顺序表L进行冒泡排序
void BubbleSort(SqList &L){
   flag=1;    //flag用来标记某一趟排序是否发生交换
   m=L.length-1;
   while((m>0) && (flag==1)) {
      flag=0; //flag置为0，如果本趟没有发生交换，则不会执行下一趟
      for(j=1; j<=m; j++)
         if(L.r[j].key>L.r[j+1].key) {
              x=L.r[j]; L.r[j]=L.r[j+1]; L.r[j+1]=x; //交换
              flag=1;
         }
      m--;
   }
}
```

冒泡排序的算法分析

•最好情况：初始排列已经有序，只执行一趟起泡，做 $n-1$ 次关键码比较，不交换/移动对象。

•最坏情形：初始排列逆序，算法要执行$n-1$趟起泡，第i趟$(1 \le i < n)$ 做了$n-i$ 次关键码比较，执行了$n-i$ 次对象交换。

此时的比较总次数KCN和记录移动次数RMN为：

$$KCN = \sum_{i=1}^{n-1}(n-i) = \frac{1}{2}n(n-1)$$

$$RMN = 3\sum_{i=1}^{n-1}(n-i) = \frac{3}{2}n(n-1)$$

因此：

时间效率：O（n^2）—因为要考虑最坏情况

空间效率：O（1） —只在交换时用到一个缓冲单元

稳 定 性：稳定 —25和25*在排序前后的次序未改变

26

冒泡排序的优点：

每一趟整理元素时，不仅可以完全确定一个元素的位置（挤出一个泡到表尾），还可以对前面的元素作一些整理，比一般的排序要快。

冒泡排序的问题：

一趟仅可以确定一个元素。

能否一趟确定多个元素？

快速排序

27

2） 快速排序

基本思想：

从待排序列中任取一个元素（例如取第一个）作为中心，所有比它小的元素一律前放，所有比它大的元素一律后放，形成左右两个子表；同时确定了中心元素的最终位置。

然后再对各子表重新选择中心元素并依此规则调整，直到每个子表的元素只剩一个。此时便为有序序列了。

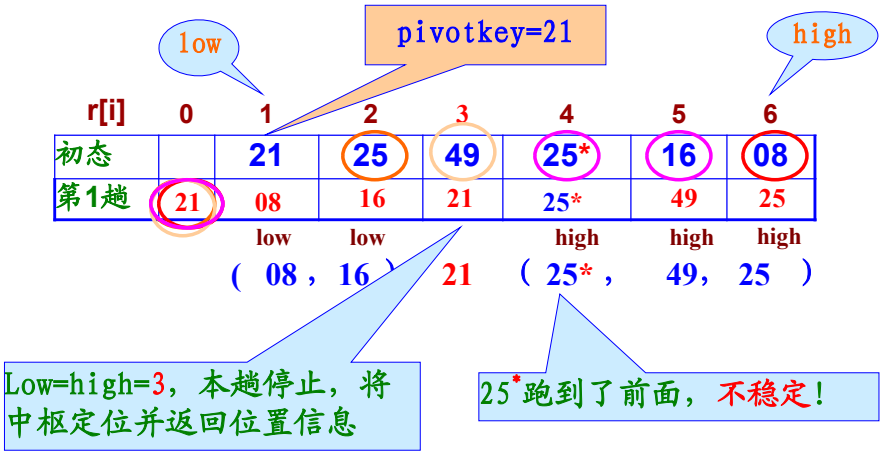
优点：因为每趟可以确定不止一个元素的位置，且呈指数增加，所以排序速度快！

前提：顺序存储结构

28

例1：关键字序列 T=(21，25，49，25*，16，08），如何实现快速排序算法的某一趟过程？

设计技巧：交替/振荡式逼近



29

例2：以关键字序列（256，301，751，129，937，863，742，694，076，438）为例，写出执行快速算法的各趟排序结束时，关键字序列的状态。

原始序列：256，301，751，129，937，863，742，694，076，438

第1趟 076，129，256，751，937，863，742，694，301，438

第2趟 076，129，256，438，301，694，742，751，863，937

第3趟 076，129，256，301，438，694，742，751，863，937

第4趟 076，129，256，301，438，694，742，751，863，937

时间效率：$O(n\log_2 n)$ —每趟确定的元素呈指数增加

空间效率：$O(\log_2 n)$ —递归栈(存每层low，high和pivot)

稳 定 性： 不 稳 定 —跳跃式交换。

30

讨论：如何编程实现？

分析：

①每一趟子表的形成是采用从两头向中间交替式逼近法；

②由于每趟中对各子表的操作都相似，主程序可采用递归算法。

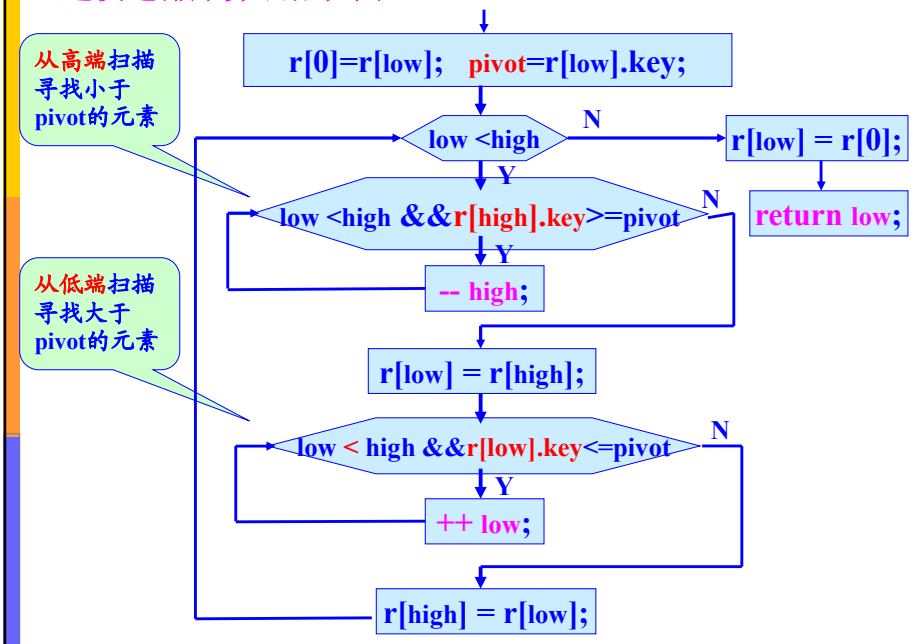
一趟快速排序算法（针对一个子表的Partition操作）

31

```
int Partition(SqList &L,int low,int high){
//交换子表r[low...high]的记录，使支点（枢轴）记录到位，并返回其位置。
//返回时，在支点之前的记录均不大于它，支点之后的记录均不小于它。
r[0]=r[low];  //以子表的首记录作为支点记录，放入r[0]单元
pivotkey=r[low].key;  //取支点的关键码存入pivotkey变量
while(low < high){      //从表的两端交替地向中间扫描
    while(low<high && r[high].key>=pivotkey ) - -high;
    r[low]=r[high];  //比支点小的记录交换到低端；
    while(low<high && r[low].key<=pivotkey)  + +low;
    r[high]=r[low];   //比支点大的记录交换到高端；
      }
r[low]=r[0];      //支点记录到位；
return low;       //返回支点记录所在位置。
}//Partition
```

32

一趟快速排序算法流程图



33

整个快速排序的递归算法：

```
void QSort ( SqList  &L,  int low, int high ) {
                    //对顺序表L中的子序列r[ low…high] 作快速排序
 if ( low < high) {                                  //长度>1
      pivot = Partition ( L, low, high );        //一趟快排，将r[ ]一分为二
      QSort ( L, low, pivot-1);    //在左子区间进行递归快排，直到长度为1
      QSort ( L, pivot+1, high );//在右子区间进行递归快排，直到长度为1
   } //if
} //QSort
```

pivot是局部变量

对顺序表L进行快速排序的操作函数为：

```
void  QuickSort ( SqList  &L) {
      QSort ( L,  1,  L.length );
 }
```

34

讨论2. “快速排序” 比任何排序算法都快吗？

——基本上是，因为每趟可以确定的数据元素是呈指数增加的。

设每个子表的支点都在中间（比较均衡），则：

第1趟比较，可以确定1个元素的位置；
第2趟比较（2个子表），可以再确定2个元素的位置；
第3趟比较（4个子表），可以再确定4个元素的位置；
第4趟比较（8个子表），可以再确定8个元素的位置；
……
只需$\lfloor \log_2 n \rfloor + 1$趟便可排好序。

而且，每趟需要比较和移动的元素也呈指数下降，加上使用了交替逼近技巧，更进一步减少了移动次数，所以速度快。

$$T(n) \approx 2T(n/2) + n, T(1) = 1$$

教材有证明：快速排序的平均排序效率为$O(n\log_2 n)$；
但最坏情况下（例如天然有序）仍为$O(n^2)$，改进措施见严P277。

$$T(n) = T(n-1) + n, T(1) = 1$$

35

讨论3. “快速排序”可以并行化吗？如何并行化？

```
void QSort ( SqList  &L,  int low, int high ) {
  if ( low < high) {
       pivot = Partition ( L, low, high );
       QSort ( L, low, pivot-1);       //*
       QSort ( L, pivot+1, high );     //*
    } //if
  } //QSort
```

应用分治思想，两个//*所标识的递归调用可以并行执行；
但是Partition操作中的交错处理是一个串行处理过程，
其*work*=*O*(*n*), 采用适当的辅助存储结构与算法可以实现并行Partition，使其*span*=*O*(*log n*)。

36

10.4 选择排序

基本思想是：每一趟(第 i 趟)在后面 $n-i+1$ 个待排记录中选取关键字最小的记录作为有序序列中的第 i 个记录。

选择排序有多种具体实现算法：

1) 简单选择排序

2) 锦标赛排序（了解）

3) 堆排序

37

1）简单选择排序

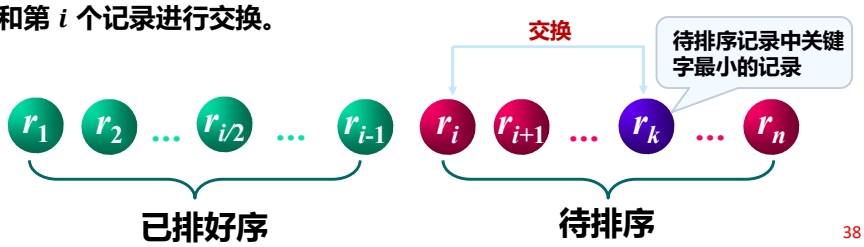
思想：每经过一趟比较就找出一个最小值，与待排序列最前面的位置互换即可。

——先在n个记录中选择最小者放到r[1]位置；然后从剩余n-1个记录中选择最小者放到r[2]位置；…，直到全部有序为止。

优点：实现简单　缺点：每趟只确定一个元素，需要n-1趟

前提：顺序存储结构

第 i 趟在 $n-i+1$ $(i=1,2,\ldots,n-1)$ 个记录中选出关键字最小的记录，并和第 i 个记录进行交换。



38

例：关键字序列T=（21，25，49，25*，16，08），
请给出简单选择排序的具体实现过程。

最小值 08 与 r[1]交换位置

直接选择排序

原始序列： 21，25，49，25*，16，08

第1趟 08，25，49，25*，16，21

第2趟 08，16，49，25*，25，21

第3趟 08，16，21，25*，25，49

第4趟 08，16，21，25*，25，49

第5趟 08，16，21，25*，25，49

时间效率： $O(n^2)$——虽移动次数较少，但比较次数仍多。

空间效率：O（1）——没有附加单元（仅用到1个temp）

算法的稳定性：不稳定——因为排序时，25*到了25的前面。

39

简单选择排序的算法如下：

```
Void SelectSort(SqList  &L ) {  //对顺序表L作简单选择排序
  for (i=1;  i<L.length; ++i){
   j = SelectMinKey(L,i);        //在r[i...L.length]中选择最小记录并定位
  if( i!=j )  r[i] ↔r[j];        //与第i个记录交换
   } //for
} //SelectSort
```

讨论：能否利用（或记忆）首趟的n-1次比较所得信息，从而尽量减少后续比较次数呢？

—— 锦标赛排序和 堆排序

40

2） 锦标赛排序（又称树形选择排序）

基本思想：与体育比赛时的淘汰赛类似。

首先对 n 个记录的关键字进行两两比较，得到 $\lceil n/2 \rceil$ 个优胜者（关键字小者），作为第一步比较的结果保留下来。然后在这 $\lceil n/2 \rceil$ 个较小者之间再进行两两比较，…，如此重复，直到选出最小关键字的记录为止。

优点：减少比较次数，加快排序速度

缺点：空间效率低

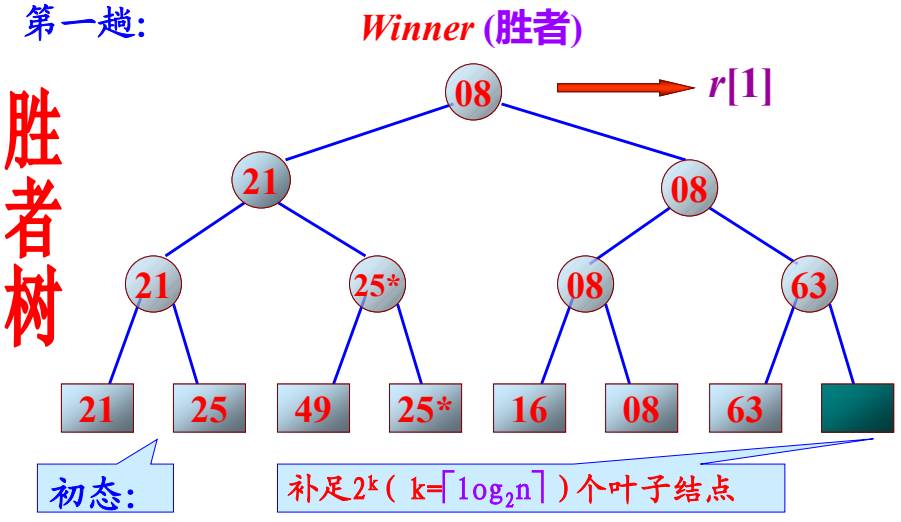
例：关键字序列T=（21，25，49，25*，16，08，63），
请给出锦标赛排序的具体实现过程。

41

注：为便于自动处理，每个记录多开两个特殊分量：

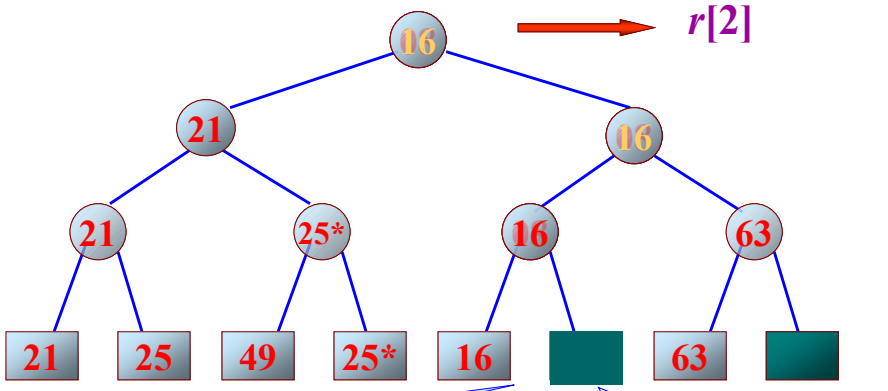
key	otherinfo	Index(结点位置编号)	Tag（是否参加比较）

第一趟：



42

第二趟: *Winner* (胜者)

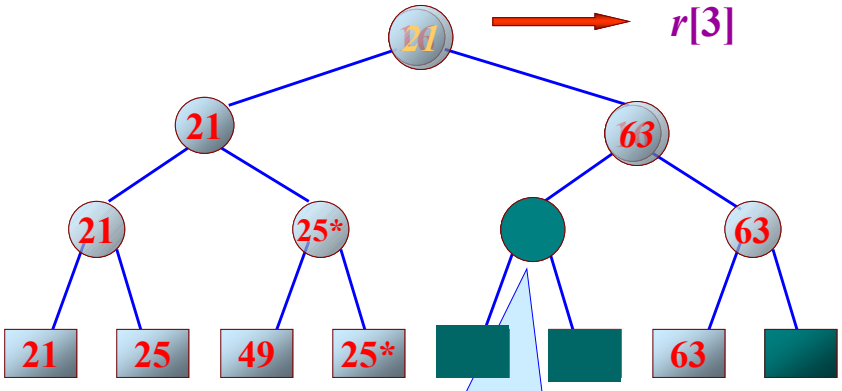


求次小值16时，只需比较2次，即只比较$\lceil \log_2 n \rceil - 1$次

令其Tag = 0, 不参与比较

43

第三趟: *Winner* (胜者)



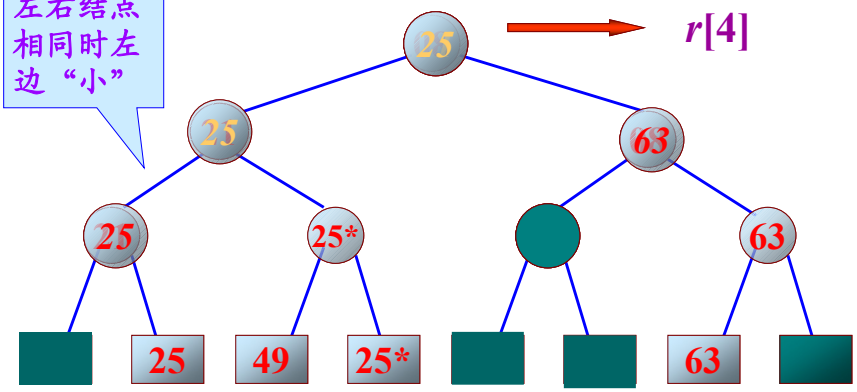
令其Tag = 0, 不参与比较

44

第四趟:

Winner (胜者)

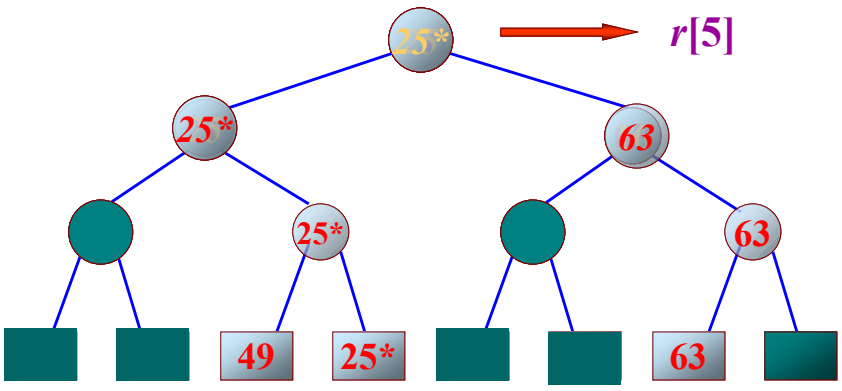
左右结点相同时左边“小”



45

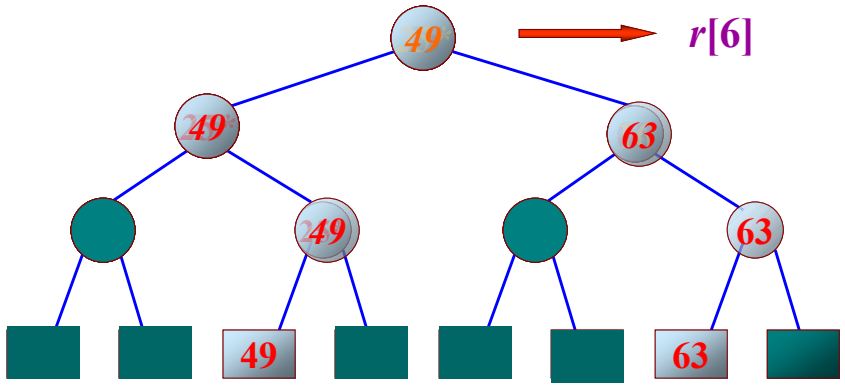
第五趟:

Winner (胜者)



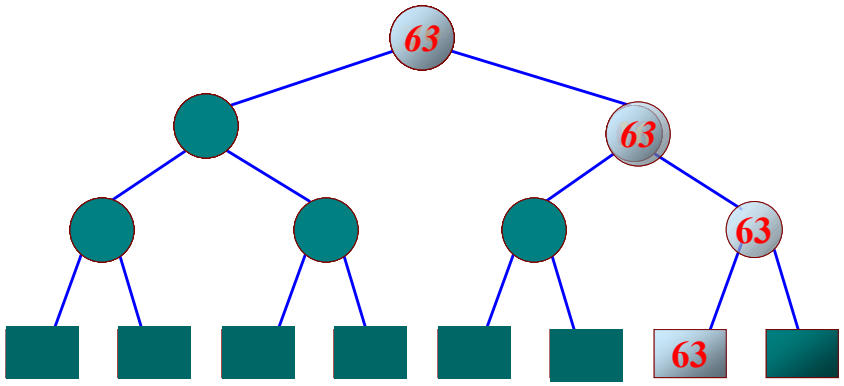
46

第六趟: *Winner* (胜者)



47

第七趟: *Winner* (胜者) → *r*[7]



48

算法分析：

- 锦标赛排序构成的树是完全（满）二叉树，其深度为 $\lceil \log_2 n \rceil + 1$，其中 n 为待排序元素（叶子结点）个数。
- 时间复杂度：$O(n\log_2 n)$ —n个记录各自比较约$\log_2 n$次
- 空间效率： $O(n)$ —胜者树的附加内结点共有n_0-1个！
- 稳定性：稳定 —可事先约定左结点“小”

$n_0 = 2^k$ = 叶子总数

讨论：锦标赛排序巧妙利用了第一次和前若干次扫描结果，加快了排序速度。其它同类方法？

——堆排序

3） 堆排序

1．什么是堆？　2．怎样建堆？　3．怎样堆排序？

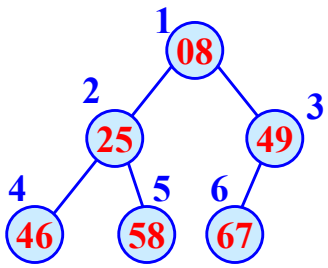
堆的定义：设有n个元素的序列 k_1，k_2，…，k_n，当且仅当满足下述关系之一时，称之为堆。

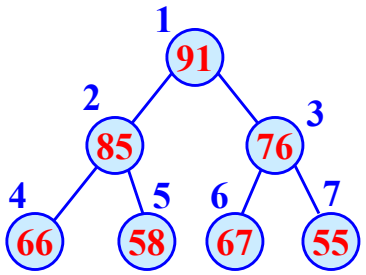
$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \quad \text{或者} \quad \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases} \qquad i=1, 2,\dots n/2$$

解释：如果让满足以上条件的元素序列（k_1，k_2，…，k_n）顺次排成一棵完全二叉树，则此树的特点是：

树中所有结点的值均大于（或小于）其左右孩子，此树的根结点（即堆顶）必最大（或最小）。

例：有序列T1=（08, 25, 49, 46, 58, 67）和序列T2=（91, 85, 76, 66, 58, 67, 55）, 判断它们是否是“堆”？





√ （小根堆）
（小顶堆）
（最小堆）

√ （大根堆）
（大顶堆）
（最大堆）

51

2. 怎样建堆？

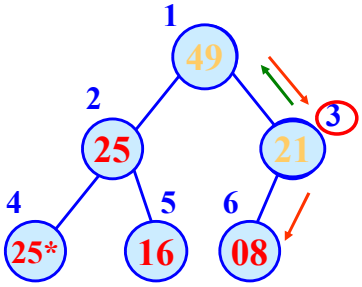
步骤：从最后一个非终端结点开始往前逐步调整，让每个双亲大于（或小于）子女，直到根结点为止。

终端结点（即叶子）没有任何子女，无需单独调整

例：关键字序列T= (21，25，49，25*，16，08），请建大根堆。

解：为便于理解，先将原始序列画成完全二叉树的形式：

这样可以很清晰地从⌊n/2⌋开始调整。



完全二叉树的第一个非终端结点编号必为⌊n/2⌋！！（性质5）

i=3: 49大于08，不必调整;
*i=2: 25大于25*和16，也不必调整;*
i=1: 21小于25和49，要调整!
而且21还应当向下比较！！

52

建堆算法（堆排序算法中的第一步：求序列的最大值）

```
typedef SqList HeapType;
void HeapSort (HeapType &H ) {
                    //H是顺序表，含有H.r[ ]和H.length两个分量
  for (i = H.length / 2; i >0; - - i ) //把r[1…length]建成大根堆
    HeapAdjust(H.r, i, H.length );  //使r[i…length]成为大根堆
   ……
} // HeapSort
```

*HeapAdjust*是针对结点 *i* 的堆调整函数，其含义是：从结点i开始到堆尾为止，自上向下比较，如果子女的值大于双亲结点的值，则互相交换，即把局部调整为大根堆。这一过程称之为筛选。

53

针对结点 *i* 的堆调整函数*HeapAdjust* 表述如下：

——从结点i开始到当前堆尾*m*为止，自上向下比较，如果子女的值大于双亲结点的值，则互相交换，即把局部调整为大根堆。

```
HeapAdjust(&r, i, m ){
current=i; temp=r[i]; child=2*i;  //temp暂存r[i]值，child是其左孩子
while(child<=m){      //检查是否到达当前堆尾，未到尾则整理
  if ( child<m && r[child].key<r[child+1].key )
  child= child+1;     //让child指向两子女中的大者位置
  if ( temp.key>=r[child].key ) breack; //根大则不必调整，函数结束
  else {  r[current]=r[child];        //否则子女中的大者上移
  current= child; child=2* child; } //将根下移到孩子位置并继续向下整理!
 }// while
 r[current]=temp;     //直到自下而上都满足堆定义，再安置入口结点
 } // HeapAdjust
```

*HeapAdjust*堆调整函数，若结点*i*到结点*m*的子树深度为*k*，则总的关键字比较次数≤2(*k* -1)。

54

3. 怎样进行整个序列的堆排序？

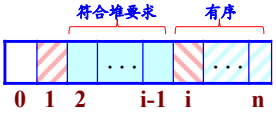
难点：将堆的当前顶点输出后，如何将剩余序列重新调整为堆？

方法：将当前顶点与堆尾记录交换，然后仿建堆动作重新调整为堆，则堆顶为次大关键字，与r[n-1]交换，如此反复直至排序结束。

即：将任务转化为—>

H.r[i…m]中除r[i]外，其他都具有堆特征。

现调整r[i]的值，使H.r[i…m]为堆。



55

基于初始堆进行堆排序的算法步骤：

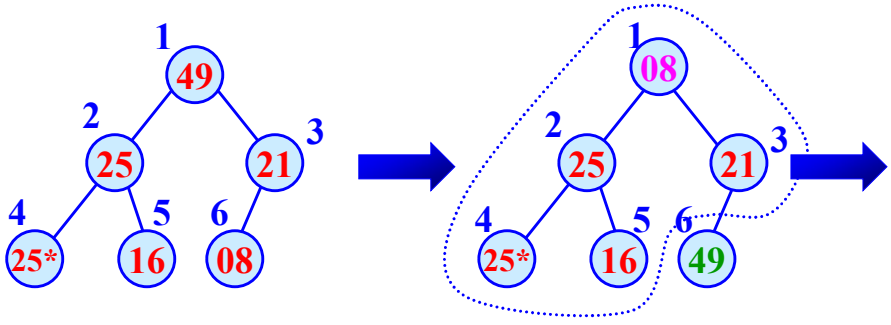
堆的第一个对象*r*[1]具有最大的关键码，将*r*[1]与*r*[*n*]对调，把具有最大关键码的对象交换到最后；

再对前面的*n*-1个对象，使用堆的调整算法，重新建立堆。结果具有次最大关键码的对象又上浮到堆顶，即*r*[1] 位置；

再对调*r*[1]和*r*[*n*-1]，然后对前*n*-2个对象重新调整，…，如此反复，最后得到全部排序好的对象序列。

56

例：对刚才建好的大根堆进行排序：



49	25	21	25*	16	08

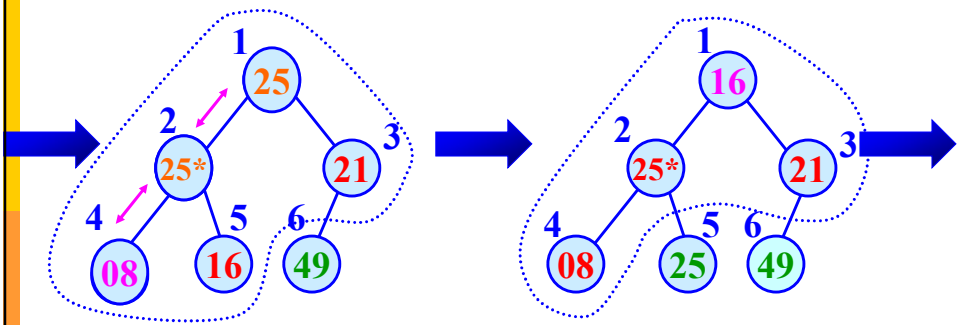
初始最大堆

08	25	21	25*	16	49

交换 1号与 6 号记录

r[1]　r[n]

57



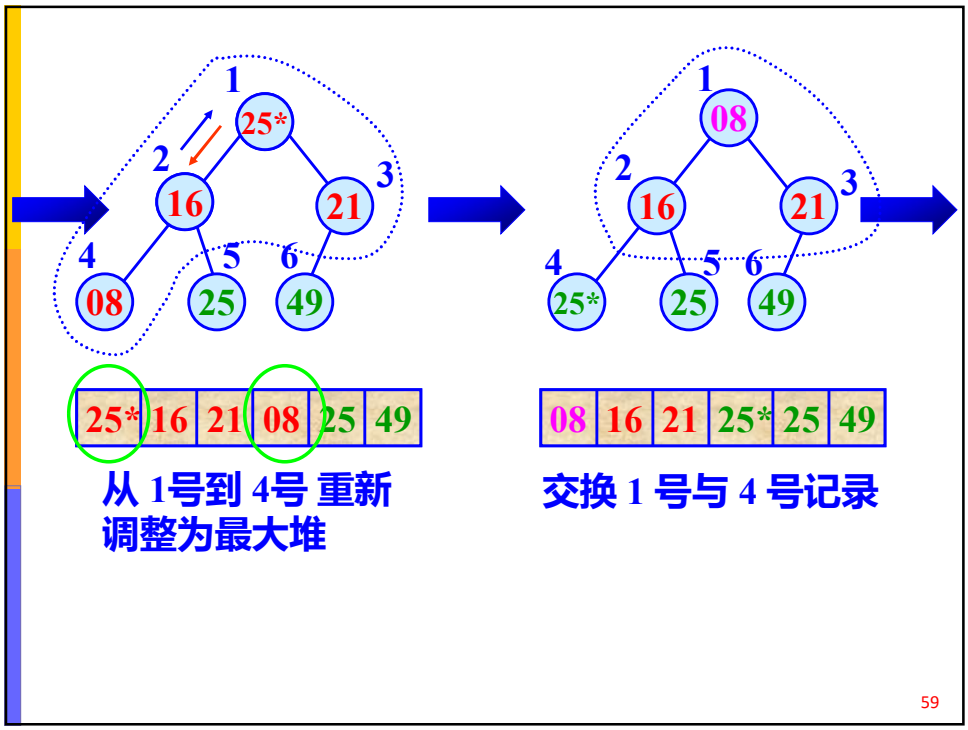
25	25*	21	08	16	49

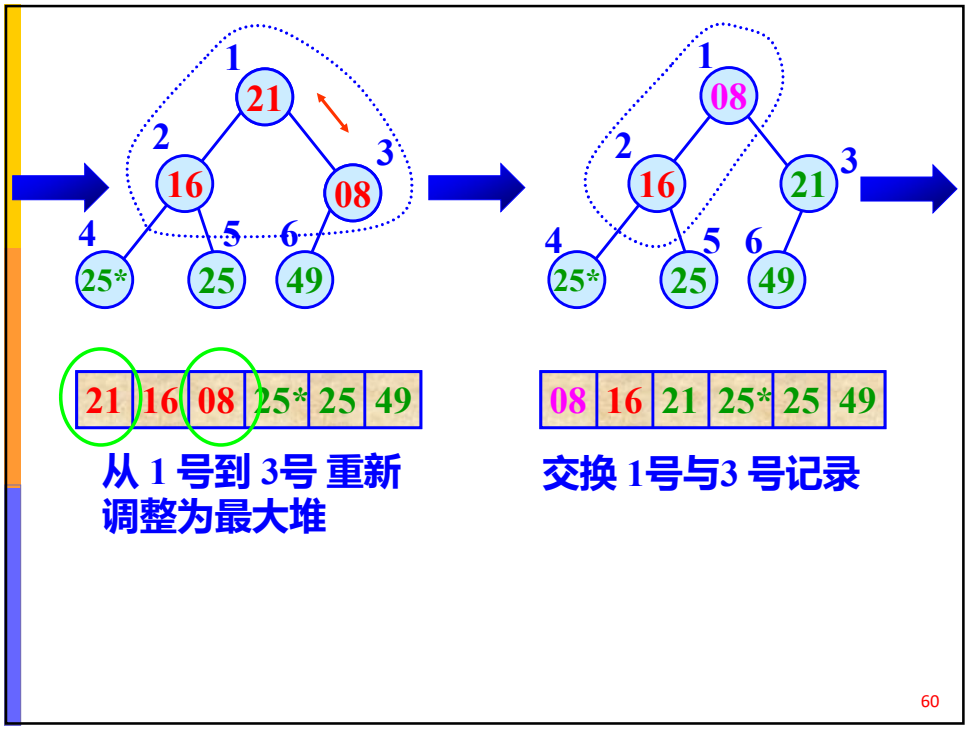
从 1 号到 5 号 重新调整为最大堆

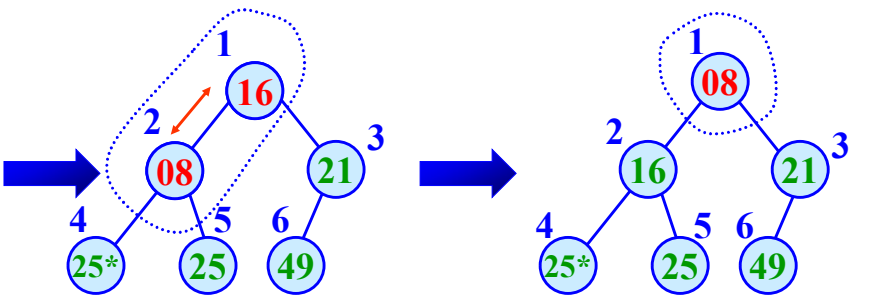
16	25*	21	08	25	49

交换 1号与 5 号记录

58


1
25*
2
16
21
3
4
08
25
5
6
49
25* 16 21 08 25 49
从 1号到 4号 重新
调整为最大堆
1
08
2
16
21
3
4
25*
25
5
6
49
08 16 21 25* 25 49
交换 1 号与 4 号记录
59


1
21
2
16
08
3
4
25*
25
5
6
49
21 16 08 25* 25 49
从 1 号到 3号 重新
调整为最大堆
1
08
2
16
21
3
4
25*
25
5
6
49
08 16 21 25* 25 49
交换 1号与3 号记录
60



16	08	21	25*	25	49

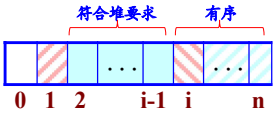
从 1 号到 2 号 重新调整为最大堆

08	16	21	25*	25	49

交换 1 号与2 号记录排序完毕。

61

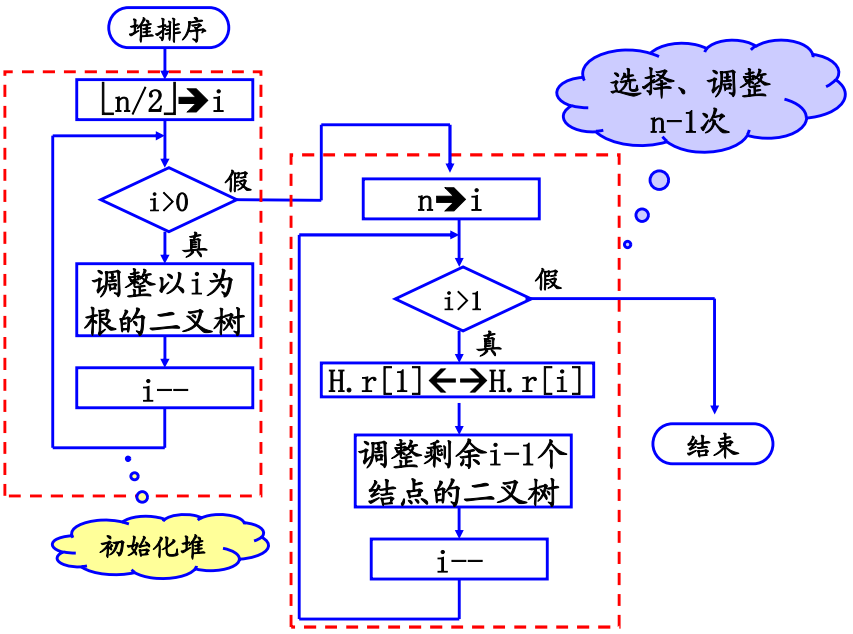
堆排序的算法




```
void HeapSort (HeapType &H ) {
//对顺序表H进行堆排序
   for ( i = H.length / 2;  i >0; - - i )
      HeapAdjust(H,i, H.length );    //for,建立初始堆
   for ( i = H.length; i > 1; - -i) {
      H.r[1] ←→ H.r[i];     //交换，要借用temp
      HeapAdjust( H, 1,i-1 );  //重建最大堆, m=i-1
   }
}
```

重建时，2至i-1号结点已符合堆的要求，故只需从1号结点开始调整。因每次从堆顶开始调整，故每次调用耗时$O(\log_2 n)$.

62



建初始堆算法的时间复杂度

- 对深度为k的堆，调整算法进行关键字比较次数至多为$2(k-1)$次，
- 建立n个元素的初始堆，调用HeapAdjust算法 $\lfloor n/2 \rfloor$次，总共进行的关键字比较次数不超过4n次。

n个结点的完全二叉树深度为$h=\lfloor \log_2 n \rfloor+1$

需调整的层为h-1层至1层，以第i层某结点为根的二叉树深度为h-i+1，第i层结点最多为2^{i-1}个，故调整时比较关键字最多为：

$\sum_{i=h-1}^{1} 2^{i-1}*2*(h-i+1-1)=\sum_{i=h-1}^{1} 2^{i}*(h-i)$，设其为t，

令j=h-i，当i=h-1时j=1；当i=1时 j=h-1，

$$t=\sum_{j=1}^{h-1} 2^{h-j}*j=2^{h-1}*1+2^{h-2}*2+\cdots+2^{2}*(h-2)+2^{1}*(h-1)=2^{h+1}-2h-2$$

$$< 2^{h+1}=2^{\lfloor \log_2 n \rfloor+2}<=4*2^{\log_2 n}=4n$$

堆排序算法分析：

- 时间效率： T(n) = O($n\log_2 n$)。因为整个排序过程中需要调用n-1次*HeapAdjust*( )算法，而此算法耗时为$O(\log_2 n)$;
- 注意： p282 初始建堆的关键字比较次数≤4n, $T(n)=\Theta(n)$。
- 空间效率：O(1)。在for循环中交换记录时用到一个临时变量temp。
- 稳定性： 不稳定。
- 优点：对小文件效果不明显，但对大文件有效。
- 并行性？
- 应用：实现优先级队列（Priority Queues）

小根堆，具有运算：Insert，DeleteMin

65

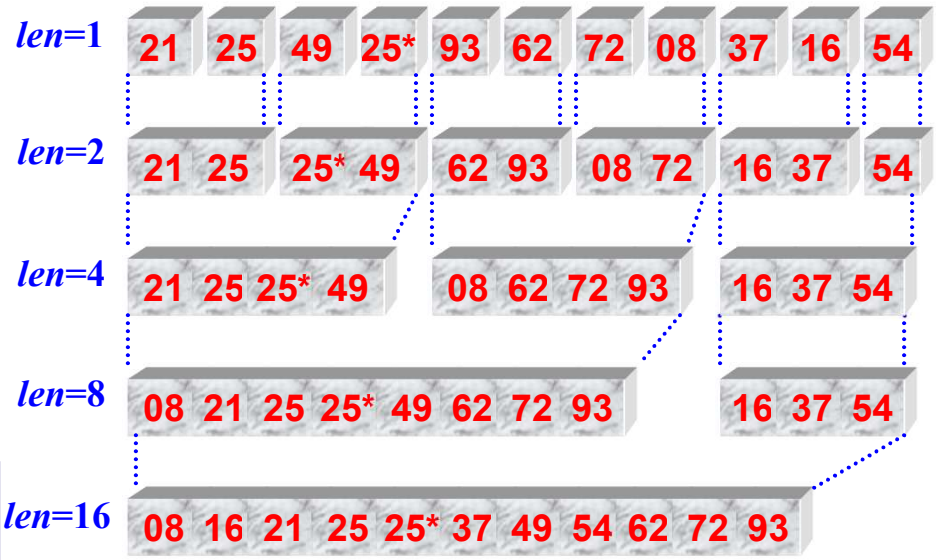
10.5 归并排序

归并排序的基本思想是：将两个（或以上）的有序表组成新的有序表。

更实际的意义：可以把一个长度为 n 的无序序列看成是 n 个长度为 1 的有序子序列 ，首先做两两归并，得到$\lceil n/2 \rceil$个长度为 2 的有序子序列 ；再做两两归并，…，如此重复，直到最后得到一个长度为 n 的有序序列。

例：关键字序列T=（21，25，49，25*，93，62，72，08，37，16，54），请给出归并排序的具体实现过程。

66



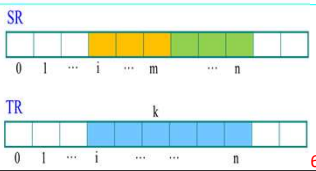
整个归并排序仅需 $\lceil \log_2 n \rceil$ *趟*

67

一趟归并排序算法：（两路有序并为一路）

```
void Merge (SR，&TR，i, m, n) {
    // 将有序的SR[i…m]和SR[m+1…n]归并为有序的TR[i…n]
  for(k=i , j=m+1; i<=m && j<=n; ++k ) {
      if ( SR[i]<= SR[j] )TR[k]=SR[i++];
      else TR[k]=SR[j++]; // 将两个SR记录由小到大并入TR
 } // for
  if (i<=m) TR[k…n]=SR[i…m]; // 将剩余的SR[i…m]复制到TR
  if (j<=n) TR[k…n]=SR[j…n]; // 将剩余的SR[j…n]复制到TR
} // Merge
```

- 时间效率：T(n)=Work = O(n)
- 并行归并(分治法)： Span = O($\log^2 n$)



68

递归形式的两路归并完整排序算法：

主函数调用为
(L.r, L.r, 1, L.length)

```
void MSort (SR[ ],&TR1[ ]，s, t) {
                    // 将无序的SR[s…t]归并排序为TR1[s…t]
if ( s==t )TR1[s]=SR[s];       // 当len=1时返回
   else {
     m=(s+t)/2;    // 将SR [s…t]平分为SR [s…m]和SR [m+1…t]
     MSort (SR，&TR2，s, m);   // 将SR 一分为二, 2分为4…
                    // 递归地将SR [s…m]归并为有序的TR2[s…m]
    MSort (SR，&TR2，m+1, t );
                  // 递归地将SR [m+1…t]归并为有序的TR2[m+1…t]
    Merge(TR2，TR1，s, m, t );
                  // 将TR2 [s…m]和TR2 [m+1…t]归并到TR1 [s…t]
  } //if
} // MSort
```

先由“长”无序变成“短”有序，
再从“短”有序归并为“长”有序。

实现时，辅助数组TR2怎么管理？

69

```
/* Lpos = start of left half, Rpos = start of right half */

void Merge(ElementType A[], ElementType TmpArray[],
           int Lpos, int Rpos, int RightEnd)
{
   int i, LeftEnd, NumElements, TmpPos;

   LeftEnd = Rpos - 1;
   TmpPos = Lpos;
   NumElements = RightEnd - Lpos + 1;

   /* main loop */
   while (Lpos <= LeftEnd && Rpos <= RightEnd)
      if (A[Lpos] <= A[Rpos])
         TmpArray[TmpPos++] = A[Lpos++];
      else
         TmpArray[TmpPos++] = A[Rpos++];

   while (Lpos <= LeftEnd)  /* Copy rest of first half */
      TmpArray[TmpPos++] = A[Lpos++];
   while (Rpos <= RightEnd) /* Copy rest of second half */
      TmpArray[TmpPos++] = A[Rpos++];

   /* Copy TmpArray back */
   for (i = 0; i < NumElements; i++, RightEnd--)
      A[RightEnd] = TmpArray[RightEnd];
}
```

```
void MSort(ElementType A[], ElementType TmpArray[],
           int Left, int Right)
{
   int Center;

   if (Left < Right)
   {
      Center = (Left + Right) / 2;
      MSort(A, TmpArray, Left, Center);
      MSort(A, TmpArray, Center + 1, Right);
      Merge(A, TmpArray, Left, Center + 1, Right);
   }
}

void Mergesort(ElementType A[], int N)
{
   ElementType *TmpArray;

   TmpArray = malloc(N * sizeof(ElementType));
   if (TmpArray != NULL)
   {
      MSort(A, TmpArray, 0, N - 1);
      free(TmpArray);
   }
   else
      FatalError("No space for tmp array!!!");
}
```

70

归并排序算法分析：

- **时间效率：** $T(n) = Work = O(n\log_2 n)$

因为在递归的归并排序算法中，一趟两路归并排序，需要调用*merge*( )函数$\lceil n/(2len)\rceil \approx O(n/len)$次，而每次*merge*( )要执行比较$O(len)$次，另外整个归并过程有$\lceil \log_2 n\rceil$“层”，所以算法总的时间复杂度为$O(n\log_2 n)$。

- **空间效率：** $O(n)$

因为需要一个与原始序列同样大小的辅助序列（TR）。
这正是此算法的缺点。

- **稳定性：稳定**
- **并行归并排序**

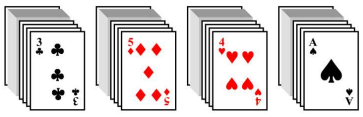
71

10.6 基数排序

❑ 前面的各种排序算法中，需要进行关键字间的比较。

❑ 基数排序是一种借助多关键字排序的思想来实现单关键字排序的算法，不需要进行记录关键字间的比较。

✓例如：整理扑克牌、图书馆卡片的排序。



❑ 多关键字排序：

n个元素的序列$\{R_1, R_2, \ldots, R_n\}$，每个元素$R_i$有d个关键字$(K^0_i, K^1_i, \ldots, K^{d-1}_i)$，则序列对关键字$(K^0_i, K^1_i, \ldots, K^{d-1}_i)$有序是指：对于序列中任意两个记录$R_i$和$R_j$记都满足下列有序关系：

$$(K^0_i, K^1_i, \ldots, K^{d-1}_i) < (K^0_j, K^1_j, \ldots, K^{d-1}_j)$$

其中K^0称为最主位关键字，K^{d-1}称为最次位关键字。

72

- 多关键字排序：$\{R_1, R_2, \ldots, R_n\}$
- R_i有d个关键字$(K^0_i, K^1_i, \ldots, K^{d-1}_i)$

✓最高位优先(MSD)：

先对最主位关键字K^0进行排序，将序列分成若干个子序列，每个子序列中的元素具有相同的K^0值，然后分别就每个子序列对关键字K^1进行排序，按K^1值的不同再分成更小的子序列，依次重复，直至对K^{d-2}进行排序之后得到的每个子序列中的元素都具有相同的$(K^0, K^1, \ldots, K^{d-2})$，而后分别为每个子序列对$K^{d-1}$ 进行排序，最后将所有子序列依次联接在一起成为一个有序序列。

✓最低位优先(LSD)：

先对最次位关键字K^{d-1}进行排序，然后对K^{d-2}进行排序，依次重复，直至对K^0进行排序后便成为一个有序序列。

73

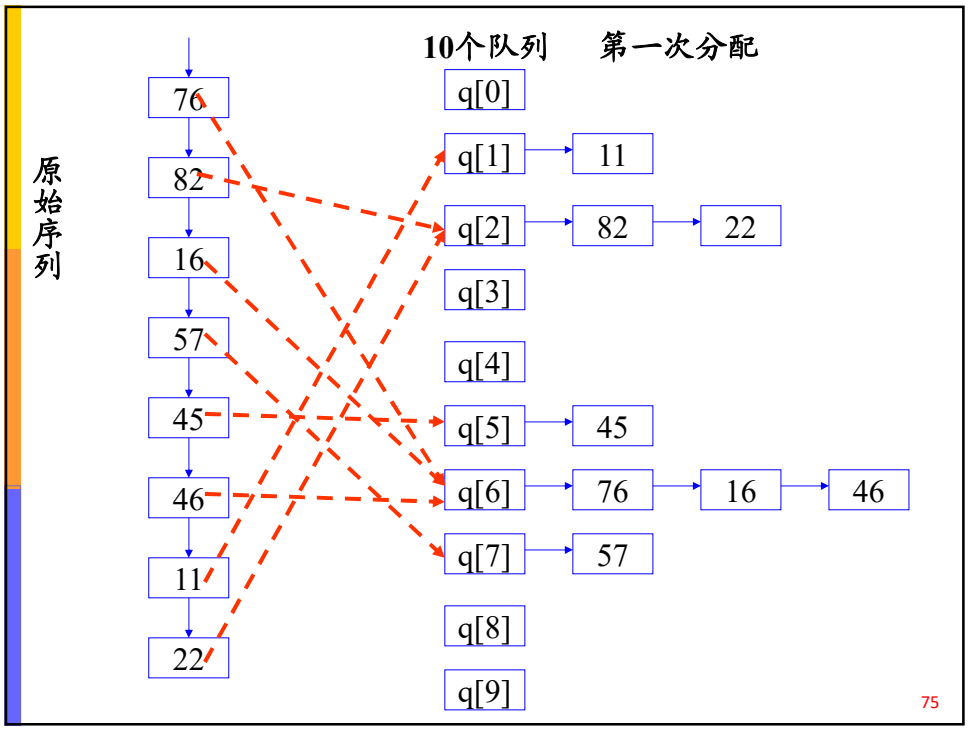
- 链式基数排序：(课外思考：如何实现基于顺序表的基数排序？)

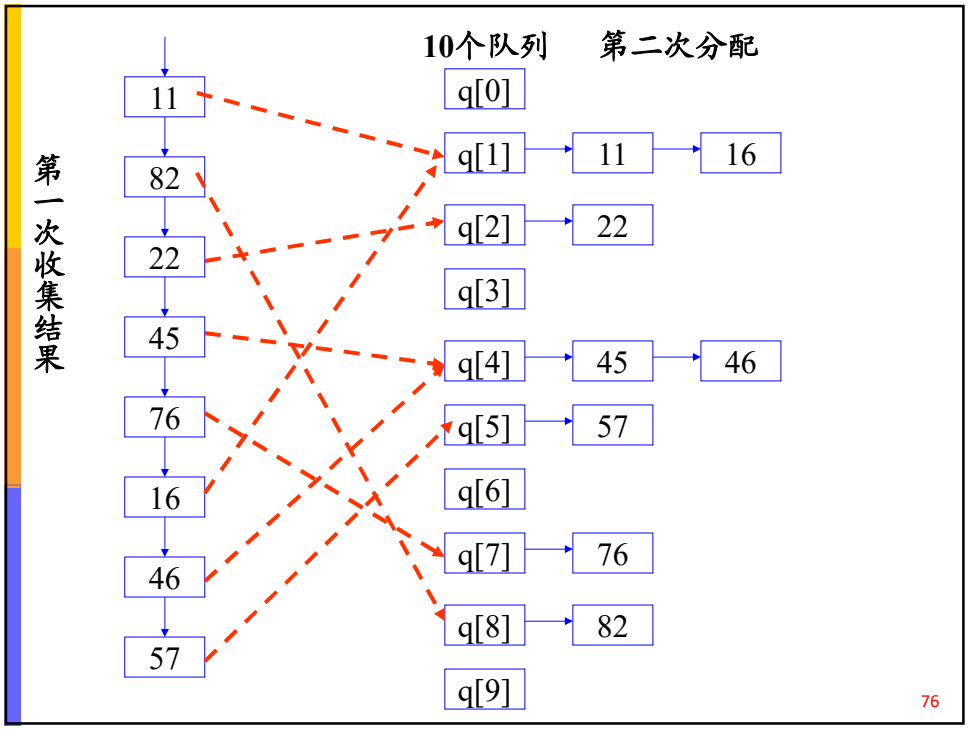
✓ 对于整型或字符型的单关键字，可以看成是由多个数位或多个字符构成的多关键字。

✓ 仅分析关键字自身每位的值，通过分配、收集进行处理。

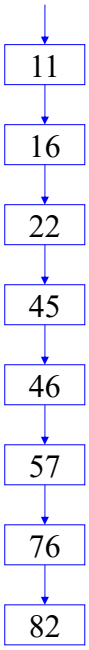
1. 待排序记录以指针相链，构成一个链表（静态链表）；
2. “分配”时，按当前“关键字位”所取值，将记录分配到不同的“链队列”中，每个队列中记录的“关键字位”相同；
3. “收集”时，按当前关键字位取值从小到大将各队列首尾相链成一个链表；
4. 对每个关键字位均重复 2）和 3）两步。

74


10个队列　第一次分配
原始序列
76
82
16
57
45
46
11
22
q[0]
q[1]
11
q[2]
82
22
q[3]
q[4]
q[5]
45
q[6]
76
16
46
q[7]
57
q[8]
q[9]
75


10个队列　第二次分配
第一次收集结果
11
82
22
45
76
16
46
57
q[0]
q[1]
11
16
q[2]
22
q[3]
q[4]
45
46
q[5]
57
q[6]
q[7]
76
q[8]
82
q[9]
76

第二次收集结果



算法分析（请自学教材中的算法）：

（1）设关键字有效位为d位，需要d趟分配、回收。

（2）每趟分配运算时间O(n)

（3）收集：基数为rd，即rd个队列。从rd个队列中收集，运算时间为O(rd)

（4）一趟分配、收集运算时间为O(n+rd)，整个算法的时间复杂度为O(d*(n+rd))

（5）基数排序是稳定的。

（6）辅助空间：每个队列设首尾指针，共2rd个指针；n个记录需要n个指针域。

77

如何编程实现？

1. 用静态链表存储排序n个记录（在r[ ]中增开n个指针分量；）

2. 在*radix*个静态队列中，每个队列都要设置两个指针：

int *f* [*radix*] 指示队头（f[j]初始为空）；

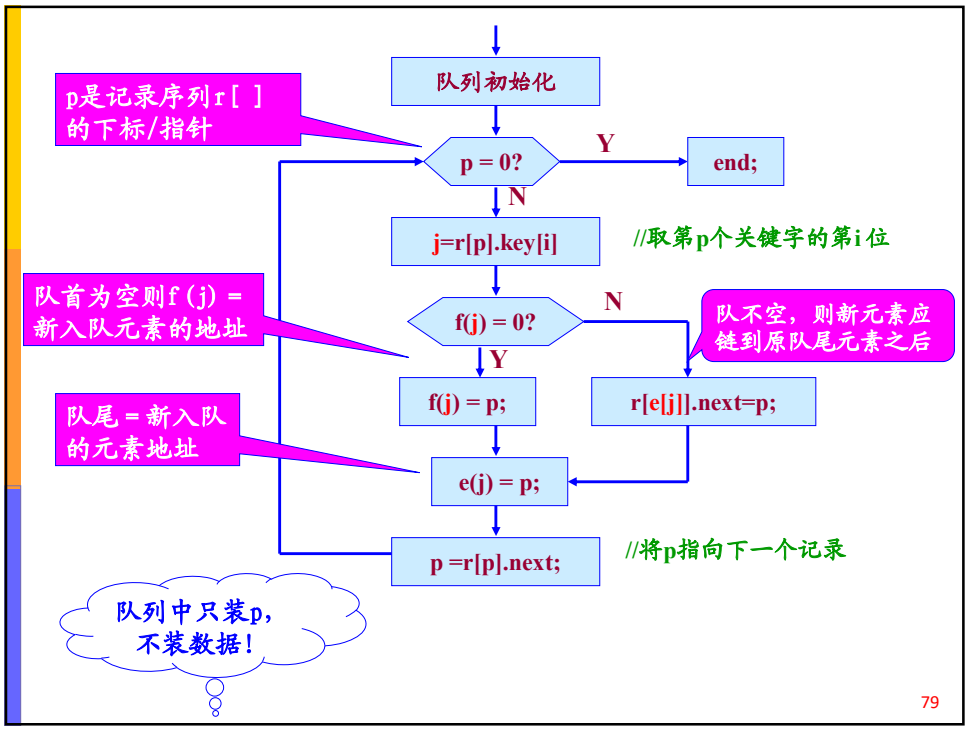
int *e* [*radix*] 指向队尾（e[j]可省略初始化，队尾元素随时赋值）；

分配到同一队列的关键码用指针链接起来。

（注：以上一共增开了n+2 *radix*个附加指针分量）

3. 存入r[ ]中的待排记录可分为3部分：

78



链表基数排序的算法：

```
void RadixSort (SLList &L,  int d,   int radix ) {
   int f[radix], e[radix];
   for ( int i = 1; i < n; i++ )  L.r[i].next=i+1;
    L.r[n].next=0;                  //静态链表初始化，将各记录顺次链接。
    int p= 1;                       //p是链表元素的地址指针
   for ( i = d-1; i >= 0; i-- ) {  //做 d 趟分配/收集, i是key的第i位
                                    //LSD算法规定先从后面开始排序
     for ( int j = 0; j < radix; j++ )  f[j] = 0;     //初态 = 各队列清空
     while ( p!= 0 ) {                  // 开始将n个记录分配到radix个队列
        int k = L.r[p]. key[i];         //取当前记录之key分量的第 i 位
        if ( f[k] == 0) f [k] =p;       //若第k个队列空, 此记录成为队首;
        else L.r[e[k]].next=p;          //若队列不空,链入原队尾元素之后
        e[k] =p;                        //修改队尾指针，该记录成为新的队尾
         p= L.r[p].next;
     } //while 选下一关键字，直到本趟分配完
```

80

```
    j = 0;          // 开始从0号队列（总共radix个队列）开始收集
    while ( f [j] == 0 ) j++;    //若是空队列则跳过
    L.r[0].next=p = f [j];       //建立本趟收集链表的头指针
    int last = e[j];             //建立本趟收集链表的尾指针
    for ( k = j+1; k < radix; k++)  //逐个队列链接（收集）
      if ( f [k] ) {                  //若队列非空，空队列则跳过
          L.r[last].next=f [k]; last = e[k]; //队尾指针链接
      } // for
    L.r[last].next=0;            //本趟收集链表之尾部应为0
  } // for
} // RadixSort
```

注：在此算法中，数组r[n]被反复使用，用来存放原始序列和每趟收集的结果。记录未移动，仅指针改变。

81

各种内部排序方法的比较

排序方法	最好情况	平均时间	最坏情况	辅助存储	稳定性
简单排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定*
快速排序	O(nlgn)	O(nlgn)	$O(n^2)$	O(lgn)	不稳定
堆排序	O(nlgn)	O(nlgn)	O(nlgn)	O(1)	不稳定
归并排序	O(nlgn)	O(nlgn)	O(nlgn)	O(n)	稳定
基数排序	O(d(n+rd))	O(d(n+rd))	O(d(n+rd))	O(rd)	稳定
简单选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定*
直接插入	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
折半插入	O(nlgn)	O(nlgn)	O(nlgn)	O(1)	稳定
冒泡	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定

注：简单排序含所有插入排序（Shell除外）、冒泡与简单选择排序。

折半插入排序只考虑了关键字比较次数。

82

思考： 若初始记录基本有序，则选用哪些排序方法比较适合？若初始记录基本无序，则最好选用哪些排序方法？请解释理由（排序方法各列举两种即可）。

答：基本有序时可选用直接插入、简单选择、堆排序、锦标赛排序、冒泡排序、归并排序、(希尔排序)等方法，其中插入排序和冒泡应该是最快的。因为主要是比较操作，移动元素很少。此时平均时间复杂度为O(n)。

无序的情况下最好选用快速排序、希尔排序、简单选择排序等，这些算法的共同特点是，通过“振荡”让数值相差不大但位置差异很大的元素尽快到位。

83

10.8 外部排序

一. 问题的提出

- 待排序的记录数量很大，不能一次装入内存，则无法直接利用前几节讨论的内部排序方法(否则将频繁地访问外存)；
- 对外存中数据的读/写是以“数据块”为单位进行的；

读/写外存中一个“数据块”的数据所需要的时间为：

$$T_{I/O} = t_{seek} + t_{la} + n \times t_{wm}$$

其中 t_{seek} 为寻查时间(查找该数据块所在磁道)

t_{la} 为等待(延迟)时间

$n \times t_{wm}$ 为传输数据块中 n 个记录的时间

84

二、外部排序的基本过程

由相对独立的两个步骤组成:

- 产生初始归并段/顺串(文件预处理):
  - 把含有n个记录的文件，按内存缓冲区大小分成若干长度为L的子文件/段;
  - 分别调入内存用有效的内排序方法排序后构造若干有序子序列，并送回外存。通常称外存中这些记录的有序子序列为“归并段”/顺串。
- 多路归并:
  - 对初始归并串逐趟合并，直至最后在外存上得到按关键字有序的整个记录序列。

85

例如: 假设有一个含10,000个记录的磁盘文件，而当前所用的计算机一次只能对1,000个记录进行内部排序（设每个物理块可以容纳200个记录，即每一次访问外存可以读/写200个记录，内存缓冲区可以容纳5个物理块），则首先利用内部排序的方法得到10个初始归并段R1~R10，然后进行逐趟归并。

假设进行2-路归并(即两两归并)，则

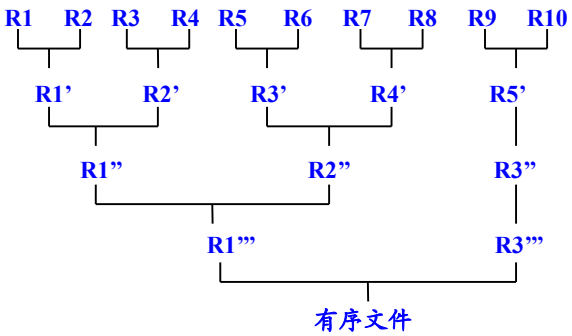
第一趟由10个归并段得到5个归并段;

第二趟由5个归并段得到3个归并段;

第三趟由3个归并段得到2个归并段;

最后一趟归并得到整个记录的有序序列。

86



从上图可见，由10个初始归并段到一个有序文件，共进行了4趟归并，每一趟从m个归并段到$\lceil m/2 \rceil$个归并段。

这种归并方法称为2-路平衡归并，倒过来则形成一棵二叉树，称为二路归并树。

分析上述外排过程中访问外存(对外存进行读/写)的次数:

- 物理块大小为200。
- 则对于10,000个记录，处理一遍需访问外存100次(读和写各50次)。
  - ✓ 求得10个初始归并段需访问外存100次;
  - ✓ 每进行一趟归并需访问外存100次;
  - ✓ 总计访问外存（粗略统计）：100 + 4 × 100 = 500次

外排总的时间还应包括:

内部排序所需时间和逐趟归并时进行内部归并的时间。

外排序总时间:

内排序: $10t_{IS}$

I/O操作: $100t_{IO}$ (排序) $+(100+80+80+100)t_{IO}$(归并)$=460t_{IO}$

内部归并: $(10000+8000+8000+10000)t_{mg}=36000\ t_{mg}$

显然，除去内部排序的因素外，外部排序的时间取决于逐趟归并所需进行的“趟数”。

89

例如 若对上述例子采用5–路归并，则只需进行2趟归并，总的访问外存的次数将压缩到

$$100 + 2 \times 100 = 300 \text{ 次}$$

一般情况下，假设待排记录序列含 m 个初始归并段，外排时采用 k–路归并，则归并趟数为 $s=\lceil \log_k m \rceil$;

随着k的增大归并趟数将减少，因此对外排而言，通常采用多路归并。k 的大小可选，但需综合考虑各种因素。

为降低 s:

（1）增大 k，会增大内部归并排序代价。→ 败者树: 严11.3

（2）减少 m，采用置换–选择排序: 严11.4。

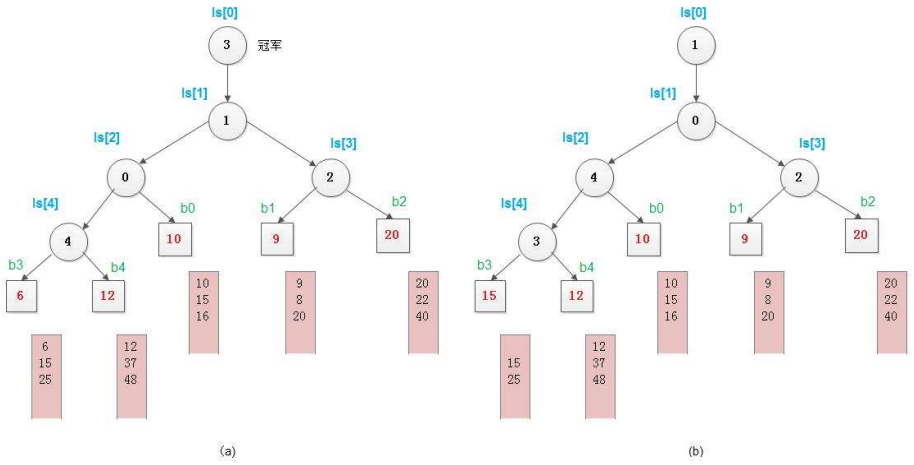
90

败者树/Tree of Loser（了解）

- Loser Trees are complete binary tree for n players where n external nodes and n – 1 internal nodes are present. Each leaf node in the loser tree stores the current comparison records of each run in the merging process. The loser of the match is stored in the internal nodes. But in this overall winner is stored at ls[0].
- 将新进入选择树的结点与其父结点进行比赛：将败者存放在父结点中；而胜者再与上一级的父结点比较。
- 比赛沿着到根结点的路径不断进行，直到ls[1]处。把败者存放在结点ls[1]中，胜者存放在ls[0]中。
- 败者树的特点：记录败者，胜者参加下一轮比赛。
- 败者树的优点：重构代价低。

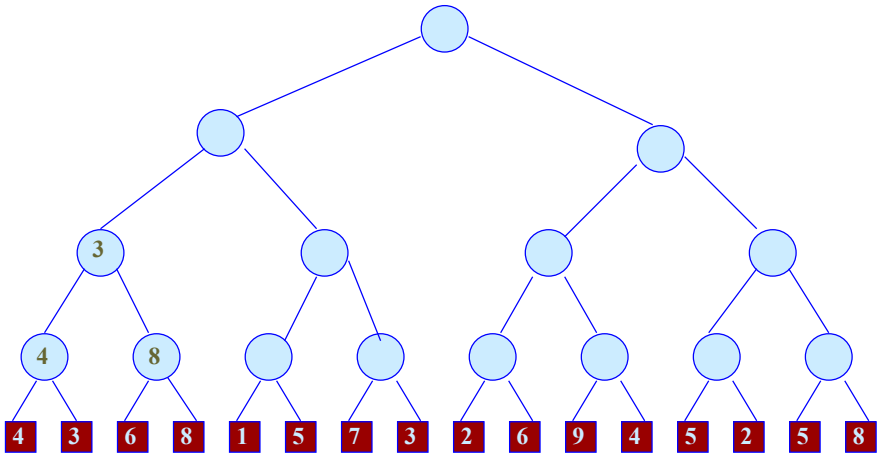
91

败者树/Tree of Loser



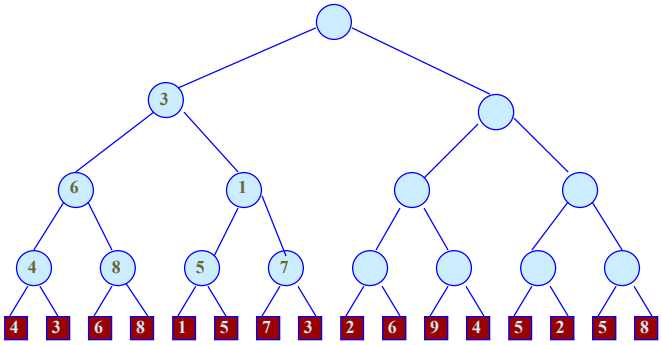
92

Min Loser Tree For 16 Players

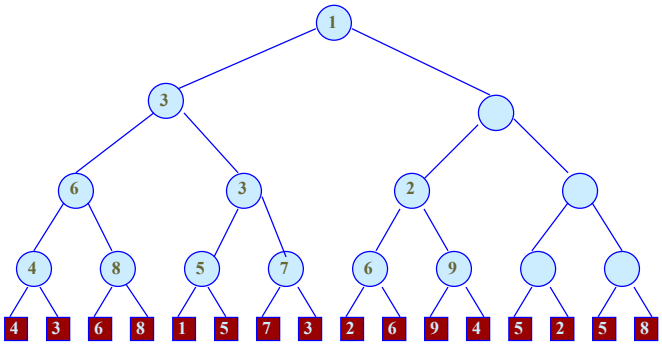


93

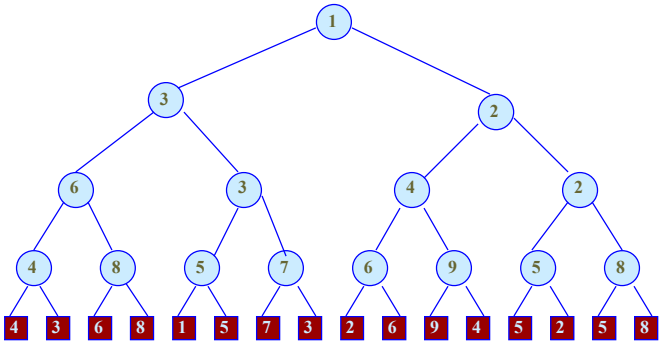
Min Loser Tree For 16 Players



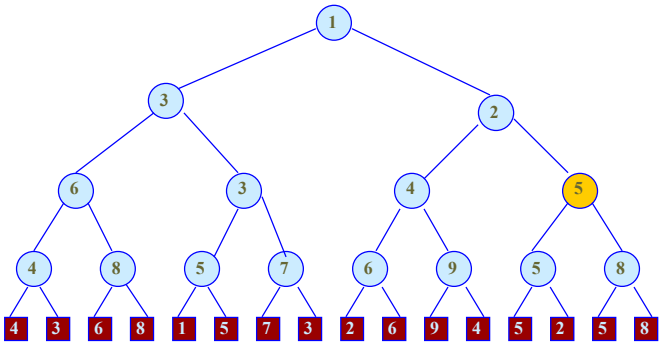
Min Loser Tree For 16 Players



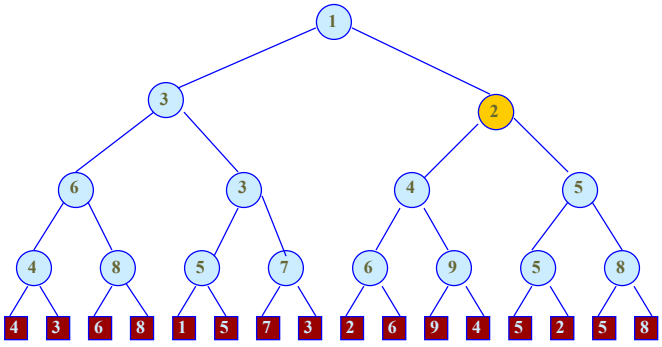
Min Loser Tree For 16 Players



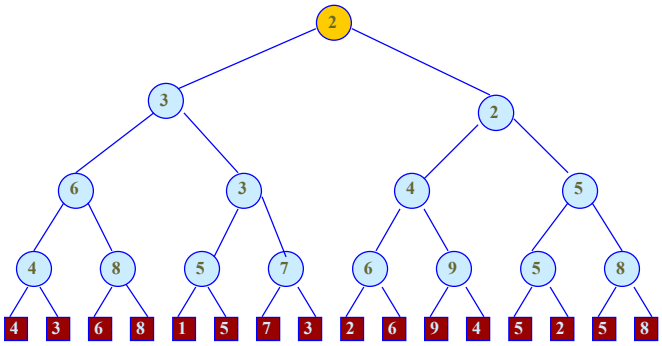
Min Loser Tree For 16 Players

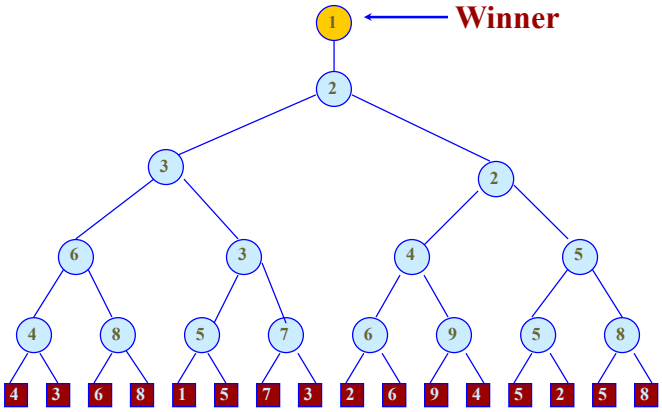


Min Loser Tree For 16 Players



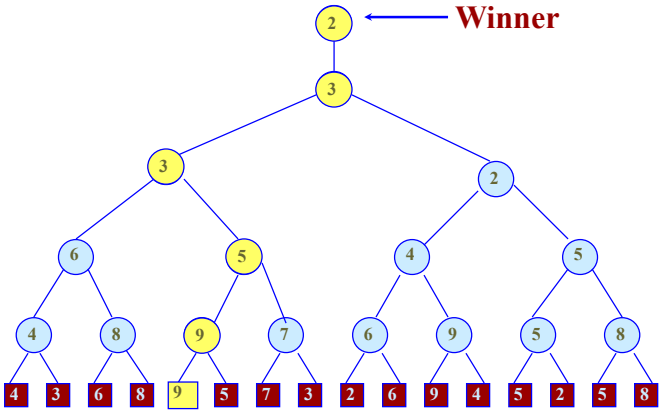
Min Loser Tree For 16 Players





Complexity Of Loser Tree Initialize

- One match at each match node.
- One store of a left child winner.
- Total time is O(n).
- More precisely θ(n).



Replace winner with 9 and replay matches.

Complexity Of Replay

- One match at each level that has a match node.
- O(log n)
- More precisely θ (log n).

置换-选择排序（了解）

k-路归并趟数为$s=\lceil \log_k m \rceil$，$m=\lceil n/l \rceil$， *n*为文件记录数，*l* 为初始归并段中的记录数。

- 置换-选择排序：输入、选择最小/最大关键字、输出交叉或并行进行。 (*l* :↑)
- 假设初始待排序文件为输入文件FI，初始归并段文件为输出文件FO，内存工作区为WA，FO和WA初始为空，并设内存工作区WA的容量可容纳*w*个记录。则置换-选择排序的操作过程为：
  - ✓(1) 从FI输入*w*个记录到缓冲区WA;
  - ✓(2) 从WA中选出其中关键字取最小值的记录，记为MIN记录;
  - ✓(3) 将MIN记录输出到FO中去;
  - ✓(4) 若FI不空，则从FI输入下一个记录到WA中;
  - ✓(5) 从WA中所有比MIN的关键字大的记录中选出最小关键字记录，作为新的MIN;
  - ✓(6) 重复(3)-(5)直到在WA中选不出新的MIN，则得到一个初始归并段，输出归并段结束标志;
  - ✓(7) 重复(2)-(6)，直到WA为空，由此得到全部初始归并段。

104

42.[10分] 对含有n (n>0)个记录的文件进行外部排序，采用置换-选择排序生成初始归并段时需使用一个工作区，工作区中能保存m个记录，请回答问题，

(1) 19个记录51，94，37, 92，14，63，15，99，48，56，23，60，31，17，42，8，90，166，100。 m>=4时，可生成几个初始归并段，各是什么？

(2)对任意m (n>>m>0)生成的第一个初试归并段长度max, min分别是？

第一步
输入 51，94，37，92
输出 37 ,14 放入，51，94，14，92，初始段文件 37
输出 51,63 放入，63，94，14，92，初始段文件 37，51
输出 63,15 放入，15，94，14，92，初始段文件 37，51，63
输出 92,99 放入，15，94，14，99，初始段文件 37，51，63，92
输出 94,48 放入，15，48，14，99，初始段文件 37，51，63，92，94
输出 99 当选不出 MINIMAX 值时，表示一个归并段已经生成初始段文件
37，51，63，92，94，99
第一个初试归并段 37，51，63，92，94，99
第二个初始归并段 14，15，23，31，48，56，60，90，166
第三个初始归并段8,17,43,100
一共 3 个段
（2）对任意 m（n>>m>0）生成的第一个初试归并段长度 max，min 分别是？
max 最大长度是 n，min 最小长度是 m

105

FO	WA	FI
		51, 94, 37, 92, 14, 63, 15, 99, 48, 56, 23, 60, 31, 17, 43, 8, 90, 166, 100
	51, 94, 37, 92	14, 63, 15, 99, 48, 56, 23, 60, 31, 17, 43, 8, 90, 166, 100
37	51, 94, , 92	14, 63, 15, 99, 48, 56, 23, 60, 31, 17, 43, 8, 90, 166, 100
37	51, 94, 14, 92	63, 15, 99, 48, 56, 23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51	, 94, 14, 92	63, 15, 99, 48, 56, 23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51	63, 94, 14, 92	15, 99, 48, 56, 23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63	, 94, 14, 92	15, 99, 48, 56, 23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63	15, 94, 14, 92	99, 48, 56, 23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92	15, 94, 14,	99, 48, 56, 23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92	15, 94, 14, 99	48, 56, 23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94	15, , 14, 99	48, 56, 23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94	15, 48, 14, 99	56, 23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99	15, 48, 14,	56, 23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99	15, 48, 14, 56	23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99, *	15, 48, 14, 56	23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99, *, 14	15, 48, , 56	23, 60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99, *, 14	15, 48, 23, 56	60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99, *, 14, 15	, 48, 23, 56	60, 31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99, *, 14, 15	60, 48, 23, 56	31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99, *, 14, 15, 23,	60, 48, , 56	31, 17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99, *, 14, 15, 23,	60, 48, 31, 56	17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31	60, 48, , 56	17, 43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31	60, 48, 17, 56	43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48	60, , 17, 56	43, 8, 90, 166, 100
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48	60, 43, 17, 56	8, 90, 166, 100
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56	60, 43, 17,	8, 90, 166, 100
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56	60, 43, 17, 8	90, 166, 100
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56, 60	, 43, 17, 8	90, 166, 100
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56, 60	90, 43, 17, 8	166, 100
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56, 60, 90	, 43, 17, 8	166, 100
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56, 60, 90	166, 43, 17, 8	100
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56, 60, 90, 166	, 43, 17, 8	100
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56, 60, 90, 166	100, 43, 17, 8	
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56, 60, 90, 166, *	100, 43, 17, 8	
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56, 60, 90, 166, *, 8	100, 43, 17,	
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56, 60, 90, 166, *, 8, 17	100, 43, ,	
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56, 60, 90, 166, *, 8, 17, 43	100, , ,	
37, 51, 63, 92, 94, 99, *, 14, 15, 23, 31, 48, 56, 60, 90, 166, *, 8, 17, 43, 100	, , ,	

106

42（10分）现有n(n>100000)个数保存在一维数组M中，需要查找M中最小的10个数。(1)设计上述查找算法，使平均情况下的比较次数尽可能少，简单描述算法思想，无需写程序。(2)说明你所设计的算法平均情况下的时间复杂度和空间复杂度。

方法一：最小值（进行10趟选择排序）用数组A[1:n]表示一维数组M

完成10趟选择排序后，A[1:10]中为最小的10个数。

方法二：堆（堆排序思想）

先用 A[1:10] 原地建立大顶堆（注：这里不能用小顶堆），遍历 A[11:n]，每个元素A[i]逐一和堆顶元素A[1]进行比较，其中11≤i≤n，如A[i]大于等于堆顶元素 A[1]，无需任何操作，如果该元素小于堆顶元素 A[1]，那么就删除堆顶元素，将该元素放入堆顶，即令A[1]=A[i]，然后将A[1:10]重新调整为大顶堆。

最后堆 A[1:10] 中留存的元素即为最小的10个数。

时间复杂度为：O(*n*)，空间复杂度为：O(1).

107

最佳归并树

起因

由于初始归并段通常不等长，进行归并时，长度不同的初始归并段归并的顺序不同，读写外存的总次数也不同。

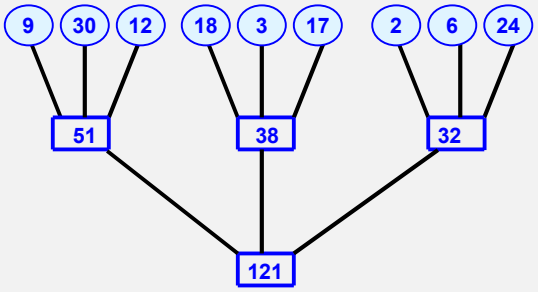
目的

减少读写外存的次数。

最佳归并树

e.g:假定由置换－选择分类法生成了 9 个初始归并段，记录数分别为 9、30、12、18、3、17、2、6、24 。如果进行 3-路归并，请讨论在各种情况下的对外存的读写次数。

A.



从外存读 121 个记录

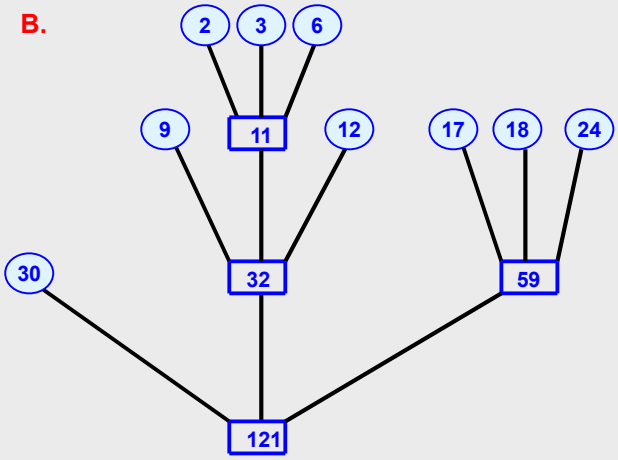
写入外存 121 个记录
从外存读 121 个记录

写入外存 121 个记录

总共读写外存 484 个记录

最佳归并树

B.



从外存读 11 个记录

写入外存 11 个记录
从外存读 91个记录

写入外存 91 个记录
从外存读 121 个记录

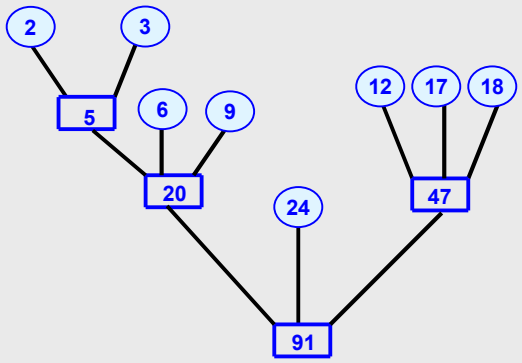
写入外存 121 个记录

总共读写外存 446 个记录

最佳归并树

按照 **HUFFMAN** 树的思想，记录少的段最先合并。不够时增加虚段。如下例所示。

C.



从外存读 **5**个记录

写入外存 **11** 个记录

写入外存 **67** 个记录

从外存读 **91**个记录

写入外存 **91** 个记录

总共读写外存 **326** 个记录

课外思考：严 10.6 10.25 11.2

112