# 《王道计算机》

## 数据结构

### 树与二叉树

#### 树的基本概念

树中的结点数等于所有结点的度+1

度为m的树第i层最多有mi-1个结点

高度为h的m叉树至多有 个结点

有n个结点的m叉树的最小高度为

总分支数=1\*n1+2\*n2+…+m\*nm度为m的结点引出m条分支

总结点数=总分支数+1

#### 二叉树的概念

满二叉树：高度h，含2h-1个结点

标号i，若有双亲，则为

##### 完全二叉树性质

若，则i为分支结点，否则为叶子结点

若有度为1的点，仅一个且有左孩子

按层编号，一旦某i结点为叶子结点或仅有左孩子，则编号大于i的结点均为叶子结点

N为奇数，每个分支结点都有左右孩子，n为偶数，则编号最大的分支结点n/2只有左孩子  
即n为奇时，不含度为1的点

##### 二叉树性质

非空时，第k层最多有2k-1个结点

高度为h，最多有2h-1个结点

按层编号后：

* + i>1时，双亲，若i为偶，为左孩子，i奇则为右孩子
  + 时，结点i的左孩子为2i，否则无左孩
  + 时，结点i的右孩为2i+1，否则无
  + I所在层次（深度）为

具有n个结点（n>0）的完全二叉树高度为  
或

##### 习题知识点

设高度为h的二叉树上只有度为0,2则结点数至少为2h-1

设二叉树右2n个结点，则度为1的结点有奇数个  
N=2n=N1+2N2+1  
N1=2(N-N2)-1



高度为h的完全二叉树最少有2h-1个结点

第n层有叶结点的完全二叉树，高度为n或n+1层  
比如，第六层有8个叶结点，则最少有25-1+8个结点  
最多有：7层，(25-8)\*2+26-1个结点

完全二叉树,n个结点

* + n为奇数不含度为1的点，即N1=0

N0=N2+1=(n+1)/2

N2=(n-1)/2

n=2N0-1=2N2+1

* + n为偶数，含一个度为1的点，即N1=1

N0=n/2=N2+1

N2=n/2-1

n=2N0=2(N2+1)

* + N0已知，则nmax=2N0 此时N1=1  
    nmin=2N0-1 此时N1=0

n个结点的二叉树采用二叉链存储，则空指针数量为n+1  
空=2\*结点数-非空=n+1

高度为h的满m叉树

* + 第i层结点个数：mi-1
  + 编号为i的结点，若其双亲结点存在，则为
  + 结点i的第一个子女编号：j=(i-1)\*m+2
  + 结点i的第k个子女编号：(i-1)\*m+k+1
  + 结点i的第m个子女编号：i\*m+1
  + 编号i的结点有右兄弟的条件：
    - 当结点i不是双亲的第m个子女才有右兄弟，设其双亲结点编号为j:  
      j的第m个孩子  
      (j-1)\*m+m+1=j\*m+1=  
      所以，的时候才有右兄弟  
      或者(i-1)%m!=0（满足不为第m个孩子）

#### 二叉树的遍历和线索二叉树

可唯一确定二叉树

非线索二叉树中，如有n个结点，则有n+1个空指针

二叉树中序遍历的最后一个点一定是从根结点开始沿右子女指针链走到底，可能是叶子结点，也可能是分支结点

若a有左孩子b，右孩子c，则无论先中后序序列，b都在c前面

先序和后序序列正好相反的话，则该二叉树只有一个叶结点

中序：n在m前的条件：n在m左方，可以不同层

后序：n在m前的条件：n是m的子孙或者同层的左方

先序中序后序序列中，叶子结点的先后顺序相同

先序遍历序号要借助栈。先序和中序的关系相当于以先序序列为入栈顺序，中序序列为出栈顺序

N个结点的线索二叉树有n+1个线索（存疑）

一棵左子树为空的二叉树在先序线索话后，其中空的链域个数为2个

* + 根的左子树为空且无前驱结点
  + 先序的最后一个元素为叶子结点，无后继结点

线索二叉树是利用二叉树的n+1个空指针来存放结点的前驱和后继

并非每个结点都可以通过线索找到前驱和后继

后序线索树的遍历仍不能有效求解

先序序列确定，n个结点，则有个不同的二叉树

#### 树、森林

等价关系

|  |  |  |
| --- | --- | --- |
| 树 | 森林 | 二叉树 |
| 先根遍历 | 先序遍历 | 先序遍历 |
| 后根遍历 | 中序遍历 | 中序遍历 |

若树中的任两个叶子结点都不存在相同的双亲，则树中的叶子数才有可能与其对应的二叉树中的叶子数相等

设F是一个森林，B是F变来的二叉树，若F中有n个非终端结点，则B中右指针为空的结点个数有n+1个



将森林F转化为对应的二叉树T，F中叶结点的个数等于T左孩子指针为空的结点个数

#### 树与二叉树的应用

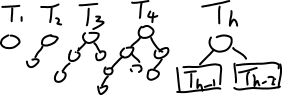
中序遍历二叉排序树，可得到有序数列

Huffman树，N个结点构造，过程中新建了N-1个结点（双分支结点），因此Huffman树中系欸但总数为2N-1

Huffman树中不含度为1的结点

具有n个结点的二叉排序树，最理想深度

平衡二叉树最少结点数递推公式：  
N0=0,N1=1,N2=2,Nh=1+ Nh-1+Nh-2  
h为二叉树高度，Nh为构造此高度的平衡二叉树所需最少结点数；所有非叶结点的平衡因子均为1



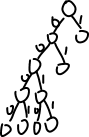
Huffman树权值最小的两个结点一定是兄弟结点

Huffman树中任一非叶结点的权值一定不小于下一层任一结点的权值

Huffman树不一定是一棵完全二叉树

没有一个编码是另一个编码的前缀，则称这样的编码为前缀编码

设Huffman编码长度不超过4，若已对两个字符编码为1和01，则还可以对（4）个字符编码  
长度4,则高度最高为5，已有0和01，说明第2，3层各有一个叶子，为使从第3层起编码更多字符，余下的二叉树应为满二叉树,4个叶子



度为m的Huffman树，只有度为0和m的结点，叶子结点有N0个，度为m的结点有Nm个，总数N，N=N0+Nm。

* + 有N个结点的Huffman树有N-1条分支
  + m\* Nm=N-1= Nm+ N0-1
  + (m-1)\* Nm= N0-1
  + Nm=( N0-1)/(m-1)

合并排序，长度分别为m,n,最坏情况下需比较m+n-1次

N个顶点，无向完全图N(N-1)/2条边，有向完全图N(N-1)

### 图

#### 图的基本概念

若n个顶点的图，小于n-1条边，则一定不是连通图

无向图：连通，连通图，连通分量，极大连通子图，极小连通子图

有向图：强连通图，强连通分量

无向图，n顶点e条边

有向图：

如果一个图有n个顶点，并且边数大于n-1，则一定有环

路径长度：路径上边的数目

简单路径：顶点不重复出现

简单回路：除第一个和最后一个顶点，其余顶点不重复出现

有向树：有一个顶点入度为0，其余顶点的入度均为1的有向图

路径：由顶点和相邻顶点序偶成的边形成的序列

稀疏图，应用邻接表而非邻接矩阵

无向连通图边数顶点数-1

无向连通图至少有一个顶点的度为1 错  
完全图，或n边n顶环图

强连通图，任何顶点到其他顶点都有路径，并不要求直接有弧

N个顶点，强连通图边至少为n个

在有n个顶点的有向图，每个顶点的度最大可达2n-2

具有n个顶点的图是一个环，则有n棵生成树

一个具有n个顶点,e条边的无向图是一个森林，则该森林必有n-e棵树

邻接矩阵：

* + 有向图：第i行表示第i个顶点的出度；第i列表示入度
  + 无向图：第i行/列表示第i个顶点的度

#### 图的存储及基本操作

邻接矩阵是图的顺序存储结构，邻接表是图的链接存储结构

##### 邻接矩阵

设G的邻接矩阵为A，An的元素An[i][j]等于由顶点i到顶点j的长度为n的路径数目

邻接矩阵空间复杂度为O(|v|2)

##### 邻接表

邻接表空间复杂度

* + 无向：O(|v|+2|e|)
  + 有向：O(|v|+|e|)

##### 习题知识点

若一个有向图的邻接矩阵，对角线以下元素为0，则一定存在拓扑排序

一个图的邻接矩阵表示唯一，邻接表表示不唯一

有向图的邻接表存储结构中，顶点v在边表中出现的次数时顶点v的入度

N个顶点的无向图邻接表最多由n(n-1)个边表结点

假设由n个顶点e条边，有向图用邻接表表示，则删除与某个顶点相关的所有边的时间复杂度为O(n+e)

求有向图结点的度，需要遍历整个邻接表

#### 图的遍历

##### BFS：

空间：O(|v|)，

时间：邻接表：O(|v|+|e|)；邻接矩阵O(|v|2)

##### DFS:

空间：O(|v|)

时间：邻接表：O(|v|+|e|)；邻接矩阵O(|v|2)

对于同样一个图，基于邻接矩阵的遍历所得到的DFS序列和BFS序列时唯一的，基于邻接表的DFS/BFS序列不唯一

对于无向图，调用BFS/DFS次数为该图连通分量的个数

##### 习题知识点

各边权值相等时，BFS可解决单源最短路径；权值不等时，BFS解决不了

利用深度优先遍历可以判断G中是否存在回路：

* + 对于无向图：DFS中遇到回边必存在环
  + 对于有向图，这条回边可能是指向DFS森林另一棵生成树上顶点的弧；但如果从有向图的某个顶点v出发进行深度优先遍历，若在DFS(V)结束前出现一条从顶点u到v的子孙，则有向图必存在包含顶点v和u的环

BFS生成树的高度DFS树的高度

一个无向图G是一棵树的条件：G须是无回路的连通图或者有n-1条边的连通图

当图G中各边权值不同时，G的最小生成树就是唯一的

MST（最小生成树）权值唯一

MST的边数=顶点数-1

Prim:

* + 时间：O(|v|2)，不依赖|e|，适用于边稠密的图

Kruskal

* + 时间：O(|e|log|e|)，使用边稀疏而点多的图

最短路径：

* + 单源：Dijkstra，时间O(|v|2)
  + 互相：Floyd-Warshall，时间O(|v|3)

Floyd:

* + 递推产生一个n阶方阵A(-1), A(0),…A(k),…A(n-1),其中A(k)[i][j]表示从顶点Vi到顶点Vj的路径长度，k表示绕行第k个顶点的运算步骤  
    A(-1)[i][j]=arcs[i][j]  
    A(k)[i][j]=Min{ A(k-1)[i][j], A(k-1)[i][k]+ A(k-1)[k][j]} k=0,1,2…n-1
  + A(0)[i][j]是从顶点Vi到顶点Vj，中间顶点是V0的最短路径长度
  + A(k)[i][j]是从Vi到Vj，中间顶点的序号不大于k的最短路径的长度

Flody算法允许图中带负权值的边，但不允许有包含负权值的边组成的环路

Floyd同样适用于带权无向图

DAG:有向无环图

拓扑排序时间复杂度O(|v|+|e|)

对于一般图，如果他的邻接矩阵是三角矩阵，则存在拓扑排序，反之不一定对

若有向图具有有序的拓扑排序序列，则邻接矩阵为三角

时间Vk的最早发生时间Ve(k)指从开始顶点V到Vk的最长路径长度

* + Ve(源点)=0
  + Ve(k)=Max{Ve(j)+weight(Vj,Vk)}
  + 计算Ve(k)时，从前往后

时间Vk的最迟发生时间Vl(k)：在不推迟整个工程下，即保证它所指的事件Vi在Ve(i)时刻能够发生，该事件必须发生

* + Vl(汇点)=Ve(汇点)
  + Vl(j)=Min{Vl(k)-weight(vj,Vk)}
  + 计算Vl(k)时，从后往前计算

活动ai的最早开始时间e(i)：指该活动的起点所表示的时间最早发生时间，如果边<Vk,Vj>表示活动ai,则e(i)=Ve(k)

活动a­i最迟开始时间l(i)：指该活动的终点所表示的事件最迟发生事件与该活动所需事件之差。如果<Vk,Vj>表示活动ai，则l(i)=Vl(j)-weight(Vk,Vj)

一个活动ai,l(i)与e(i)的差额d(i)=l(i)-e(i)，称l(i)-e(i)=0的活动ai是关键活动

所有权值最小的边一定会出现在MST中 x

Prim从不同顶点开始得到的MST不一定相同

最短路径一定是简单路径

Dijkstra不适合求带负权值的最短路径

Floyd求亮点最短路径时，当最短路径发生改变，Pathk-1就不是Pathk的子集

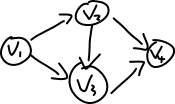
判断有向图是否有环

* + DFS
  + 拓扑排序
  + 求关键路径
  + 求最短路径不行

拓扑排序，时间复杂度：邻接表O(n+e)，邻接矩阵O(n2)

顶点数目>1的强连通分量中必然存在回路

拓扑有序序列唯一，则图中每个顶点的入度和出度最多为1 ×



如果有向无环图拓扑排序唯一则可以唯一确定该图 ×

求关键路径以拓扑排序为基础

### 查找

#### 顺序查找和折半查找

##### 顺序查找

一般顺序表查找

* + ASL成功=
  + ASL不成功=n+1

有序顺序表查找

* + ASL不成功=

若有序序列有n个元素，则对应折半查找判定树有n个圆形分支结点和n+1个方形的叶结点

* + 方形：查找不成功
  + 圆形：一个记录

##### 折半查找

ASL成功=

时间：O(log2n)

存储要求时顺序存储，有序，链式不行

##### 分块查找：

块内无序，块间有序

第k个块的最大关键字小于第k+1个块的所有记录的关键字

索引表：表中每个元素含有各块的最大元素和各块第一个元素的地址，表有序

将长度n的查找表均分为b块，每块s个记录，块内块间都用顺序查找，则

若，ASL取最小值

若对索引表采用折半查找则

##### 习题知识点

顺序查找：成功时有序无序ASL一样，失败时ASL有序<ASL无序

折半查找对应的判定树时平衡二叉树AVL

折半查找失败时，比较次数最多为树的高度  
或者  
判定树中方形结点时虚构，不计入比较次数

折半ASL和最大查找长度：O()  
二叉排序树最大O(n)，最短O()

折半，高度最低  
因为是AVL，分支高度差。最高，所以最低

K分查找法：高度

#### B树和B+树

##### B树

一棵m阶B树，或为空，或为满足如下特性的m叉树：

* + 每个结点至多有m棵子树，即每个结点最多含m-1个关键字
  + 若根结点不是终端结点，则至少有两棵子树
  + 除根结点外的所有非叶结点至少有棵子树，即至少有个关键字
  + 非叶结点结构

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | P0 | K1 | P1 | K2 | P2 | … | Kn | Pn |

* + - Ki为关键字，K1<K2<…<Kn
    - Pi为指向字根结点的指针且Pi-1所指的子树中所有的结点均小于Ki
    - Pi所指的子树中所有的结点均大于Ki
    - n()为节点中关键字的个数
  + 所有叶结点都在同一层，无任何信息（像折半查找中失败的点，实际上叶子结点不存在，指向这些结点的指针为空）
  + 所有结点平衡因子=0

B树的高度不包括最后一层叶子结点

B树n个关键字，高度h，m阶

##### B+树

一棵m阶B+树满足如下特性：

* + 每个分支结点最多有m棵子树
  + 非叶根结点至少有2棵子树，其他分支结点至少有棵子树
  + 结点的子树个数与关键字个数相等
  + 所有叶结点包含全部关键字及指向相应记录的指针，叶结点中关键字按大小排列，相邻叶结点相互链接
  + 所有分支结点（可看成是索引的索引）中仅包含它的各个子节点（即下一级的索引块）中关键字的最大值及指针

m阶B树和B+树的区别

* + B+中，具有n个关键字的结点只含有n棵子树  
    B中，n个关键字，有n+1个子树
  + B+:每个非根结点关键字个数：  
     根结点：  
    B: 根：
  + B+:叶结点包含信息，非叶结点仅起索引作用。非叶结点的每个索引项：对应子树的最大关键字和指向该子树的指针，不含有该关键字对应记录的存储地址
  + B+：叶结点包含了全部关键字，B树中，叶结点包含的关键字和其他结点包含的关键字是不重复的

B+两种查找方式

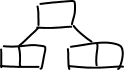
* + 从最小关键字开始的顺序查找
  + 从根结点开始，多路查找

具有n个关键字的m阶B树，应有n+1个叶结点

##### 习题知识点

m叉树，高h，结点数=

5阶B树，并非存在至少有一个有4个关键字的结点，各个结点只有两个关键字也可以



有15个关键字的4阶B树，含关键字的结点个数最多为15个，此时每个结点含关键字应最少才可以:

含有n个非叶结点的m阶B树至少有个关键字

B,B+树都能有效支持随机查找

B,B+都是平衡的多叉树

B,B+都可以用于文件的索引结构

#### 散列表

##### 基本散列函数

直接定址法：H(key)=a\*key+b；不会产生冲突，适合关键字基本连续分布的情况

除留余数法：p为不大于散列表长度m，但最接近或等于的质数：H(key)=key%p

##### 处理冲突的方法

开放定址法：Hi=(H(key)+di)%m；m表示散列表长，d表示增量序列

* + 线性探测法：di=1,2,3..m-1;会“聚集”(“堆积”)
  + 平方探测法：di=12,-12, 22,- 22…k2,-k2,其中km/2，m必须使一个可以表示成4k+3的质数：可以避免“堆积”，但不能探测到散列表上所有的元素
  + 再散列法：di=Hash2(Key)；最多经过m-1次探测就会遍历表中所有位置，回到H0，不易产生聚集
  + 伪随机序列法：di=伪随机数序列。
  + 注意：开放定制法，不能随便删除表中已有元素。可以做删除标记，进行逻辑删除

拉链法：把同义词存储在一个线性链表中

散列表查找效率取决于

装填因子:，定义为一个表的装满程度  
散列表的平均查找长度依赖于，不直接依赖于n或m

##### 题目中知识点

折半查找法只能在顺序存储结构上

“堆积”问题是由于同义词之间或非同义词之间发生冲突引起的，同时解决冲突的方法选择不当

同义词冲突不等于聚集

链地址法（拉链法）不会引起聚集现象

产生冲突的概率与装填因子的大小成正比

堆积现象，对存储效率，散列函数和装填因子都不会有影响，平均查找长度会因为堆积现象而增大

在计算查找失败时的平均查找长度时，既不是根据表中元素个数，也不是根据表长来决定的，而是根据散列函数（Mod后面的数字，如Mod7）来计算平均查找长度。王道p258第5题

#### 字符串模式匹配KMP

next求值的算法

1. void get\_next (char T[ ], int next[ ]){
2. int i, j = 0;
3. next[1] = 0;
4. while(i <= T[0]){ //T[0]用于保存字符串的长度
5. if( j == 0 || T[i] == T[j]) {
6. ++i; ++j; next[i] = j;
7. }else
8. j = next[j];
9. }
10. }

KMP算法

1. int KMP ( char S[ ] , char T[ ] , int next[ ] , int pos){
2. //利用模式串T的next函数求T在主串S中
3. //第pos个字符之后的位置的KMP算法
4. //T非空，1<=pos<=strlen(S)
5. i=pos;
6. j=1;
7. while(i<=S[0]&&j<=T[0]){
8. If(j==0||S[i]==T[j]){
9. ++i; ++j;
10. }else
11. j=next[j]
12. }
13. if(j>T[0]) return i-T[0];
14. else return 0;
15. }

### 排序

#### 排序

拓扑排序不属于内部排序

不是所有的内部排序都要经过比较操作，基数排序就不是

算法的稳定性与算法优劣无关

链表不适用于折半排序

对同一线性表使用不同的排序方法，得到的排序结果可能不同

对任意序列进行基于比较的排序，求最少的比较次数，应考虑在最坏的情况下，对任意n个关键字排序的比较次数至少为，比如7个关键字，至少比较13次

#### 插入排序

##### 直接插入排序

比较和移动次数取决于待排序表的初始状态

适用于顺序存储和链式存储

边比较边移动元素

##### 折半插入排序

将比较和移动元素分离出来，即先折半查找元素的待插入的位置，然后再统一移动

折半插入排序仅仅减少了比较元素的次数，该比较次数与待排序表的初始状态无关，仅取决于表中的元素个数n

元素的移动次数没有改变，依赖于待排序表的初始状态

##### 希尔排序

先将待排序表分割成若干形如L[i,i+d,i+2d,…,i+kd]的特殊子表，分别进行直接插入排序，当整个表都已经基本有序时，再对全体记录进行一次直接插入排序

只适用于当线性表为顺序存储的情况

希尔排序，总的比较次数和移动次数比直接插入排序小得多

##### 习题知识点

不考虑与哨兵比较，直接插入排序最坏的情况下，要做n(n-1)/2次比较

在待排序序列基本有序的前提下，直接插入排序效率最高

直接插入排序比较次数和初始状态有关，折半插入排序比较次数和初始状态无关，之和元素个数有关

堆排序，冒泡排序，快排，简单选择排序每一回合都会有元素放在最终位置，希尔排序，直接插入排序没有这个属性

直接插入排序有局部有序属性，即如果从后向前插，前几个元素是有序排列的

#### 交换排序

##### 冒泡排序

最好时，只需要比较n-1次，移动0次；

最差时，初始状态为逆序，需要进行n-1趟排序，第i趟需要进行n-i次比较，每次比较都必须移动元素三次来交换元素位置

冒泡排序也会局部有序，并且是全局有序（比直接插入厉害点）

##### 快速排序

需要借助递归工作栈，容量应与递归调用的最大深度一致。最好：，最坏：要进行n-1次调用即O(n)

时间效率和划分是否对称有关，如果划分两个区域分别包含n-1和0个元素，就最差；能均分就最好

快排不产生有序子序列，即没有局部有序性

##### 习题知识点

插入排序，第i趟之后前i+1个元素应该时有序的

快速排序在要排序的数据已基本有序的情况下最不利

冒泡和选择排序经过i趟，将有i个元素处于最终位置（最左或最右）

可能时执行第一趟快排之后所得到的序列的判断：如果存在一个元素，其左/其右全都大于/小于自身，即可

快排递归次数与每次划分后得到的分区的处理顺序无关

#### 选择排序

##### 简单选择排序

简单选择排序中元素移动的操作次数很少，不会超过3(n-1)，最好的情况下移动0次，此时对应表已经有序；但是元素之间的比较的次数与序列初始状态无关，始终是n(n-1)/2次

##### 堆排序

构造初始堆，就是一个反复筛选的过程。N个结点的二叉树，最后一个结点是第个结点的孩子。对第个结点为根的子树进行筛选（大根堆，若根结点的关键字小于左右子女中的较大者，则交换），之后依次对各结点（-1~1）为根的子树进行筛选。交换的时候可能破坏下一级的堆，需要采用上述方法对下一级构造。

向下调整的时间与树高有关，为O(h)。在元素个数为n的序列上建堆，其时间复杂度为O(n)，即在线性时间内，就可以将一个无序数组建成一个堆

堆排序的时候，堆顶元素就是最大值（大顶堆），输出堆顶元素后，将堆底元素送入堆顶，再向下调整，再输出堆顶元素

删除堆顶操作，将堆底元素放到堆顶，然后向下调整

插入操作，先插到堆的末尾，再向上调整

时间效率：建堆时间为O(n)，之后有n-1次向下调整操作，每次调整的时间复杂度是O(h)，故时间复杂度为O(nlogn)

##### 习题知识点

小根堆，关键字最大的结点一定存储在这个堆所对应的完全二叉树的叶子结点中，二叉树中最后一个非叶子结点，所以关键字最大的结点存储范围+1~n

插入排序，快排，冒泡排序，排序过程中的比较次数与序列出书状态有关，选择排序比较次数始终是n(n-1)/2

插入排序，快速排序和归并排序，只有将所有元素排序完成后，才能得到前k小的元素序列。

冒泡排序，堆排序和简单选择排序，每一趟都会确定一个最小/大的元素。

堆排序，n个元素，建立初始堆，时间不超过4n，取得第k小的元素之前的排序序列所化的时间为klogn，总时间为4n+klogn，冒泡和简单选择排序完成此功能所花的时间为kn，当k5时，堆排序最优

#### 归并排序和基数排序

##### 归并排序

整个归并排序进行趟

每一趟时间复杂度为O(n)

对于n个元素进行k-路归并排序，排序的趟数m满足从而又考虑到m为整数，所以

##### 基数排序

最高位优先MSD，最低位优先LSD

假设线性表由结点序列a0­,a1,…an-1组成，每个结点aj的关键字由d元组（）组成，其中

一趟排序需要辅助空间为r（r个队列），所以空间复杂度O(r)

基数排序与序列的初始状态无关

基数排序需要进行d趟分配和收集，一趟分配需要O(n),一趟收集O(r),所以基数排序时间复杂度O(d(n+r))

##### 习题知识点

基数排序不需要进行关键字的比较

归并排序的比较次数的数量级与序列的初始状态无关

选择排序的比较次数与序列初始状态无关

将两个各含有n个元素的有序表合并成一个有序表，最少比较n次，最多比较2n-1

#### 各种内部排序算法的比较

##### 时间复杂度

简单选择排序，直接插入排序和冒泡排序平均复杂度O(n2)，直接插入排序和冒泡排序最好的时候可以达到O(n)，且简单选择排序与序列的初始状态无关。

快速排序基于分治的思想，最坏会达到O(n2)但实际应用中常常优于其他算法。

归并同样基于分治思想，其分割子序列与初始序列的排列无关

| 算法种类 | 时间复杂度 | | | 空间复杂度 | 稳定？ |
| --- | --- | --- | --- | --- | --- |
| 最好情况 | 平均情况 | 最坏情况 |
| 直接插入排序 | O(n) | O(n2) | O(n2) | O(1) | √ |
| 冒泡排序 | O(n) | O(n2) | O(n2) | O(1) | √ |
| 简单选择排序 | O(n2) | O(n2) | O(n2) | O(1) |  |
| 希尔排序 |  |  |  | O(1) |  |
| 快速排序 | O(nlog2n) | O(nlog2n) | O(n2) | O(log2n) |  |
| 堆排序 | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(1) |  |
| 2-路归并排序 | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(n) | √ |
| 基数排序 | O(d(n+r)) | O(d(n+r)) | O(d(n+r)) | O(r) | √ |

##### 排序算法小结

若n较小（n），则可以采用直接插入排序或简单选择排序。由于直接插入排序所需的记录移动操作比简单选择排序多，因此，当记录本身信息量较大时，用简单选择排序比较好

若文件的初始状态已经基本有序，则选用直接插入排序或者冒泡排序

若n比较大，则应采用时间复杂度为O(nlog2n)的算法：快排，堆排或归并排序。

* + 快排是基于比较的内部算法中最好的，当待排序的关键字是随机分布时，快排的平均时间更短。
  + 堆排所需要的辅助空间小于快排，并且不会出现快排可能出现的最坏的情况
  + 归并算法是稳定的，快排和堆排都不稳定。通常将它和直接插入排序结合起来一块用：先利用直接插入排序求得较长的有序子文件，然后再两两归并，因为两者都稳定，所以这个方法也稳定

任何基于“比较”的排序算法，至少需要O(nlog2n)

若n很大，记录的关键字位数较少且可以分解时，采用基数排序比较好

当记录本身信息量较大时，为避免耗费大量时间移动记录，可用链表作为存储结构

##### 习题知识点

交换类的排序，其趟数和原始序列有关，所以冒泡排序与初始序列有关；直接插排，趟数固定n-1，简单选择，趟数固定n-1；基数，趟数固定d

简单选择排序，快排，堆排每一趟都至少可以确定一个元素最终位置

堆是用于排序的，所以对其中的元素进行查找的时候，是无序的，效率不高

基数排序元素的移动次数与关键字的初始排列次数无关

#### 外部排序

##### 外部排序的方法

外部排序通常采用归并排序的方法，包括两个相对独立的阶段

* + 首先，根据内存缓冲区大小，将外存上含n个记录的文件分成若干个长度为h的文件，依次读入内存并利用有效的内部排序方法对他们进行排序，并将排序后得到的有序子文件重新写回外存，称这些有序子文件为 归并段 或 顺串
  + 然后，对这些归并段进行逐趟归并，使归并段（有序的子文件）逐渐由小到大，直至得到整个有序文件

外部排序的时间=内部排序所需的时+外存信息读写的时间+内部归并所需的时间  
TES=r\*TIS+d\*TIO+S\*(n-1)\*Tmg  
其中r是初始归并段个数，TIS是对每一个初始归并断进行内部排序的时间，d是访问外存块的次数，TIO是每一个块的存取时间，S是归并趟数，n是每趟参加二路归并的记录个数，Tmg是每作一次内部归并，取得一个关键字最小记录的时间。

要提高外排序的速度，应着力减少d，即I/O次数

增大归并路数，可减少归并趟数，从而减少总的磁盘I/O次数

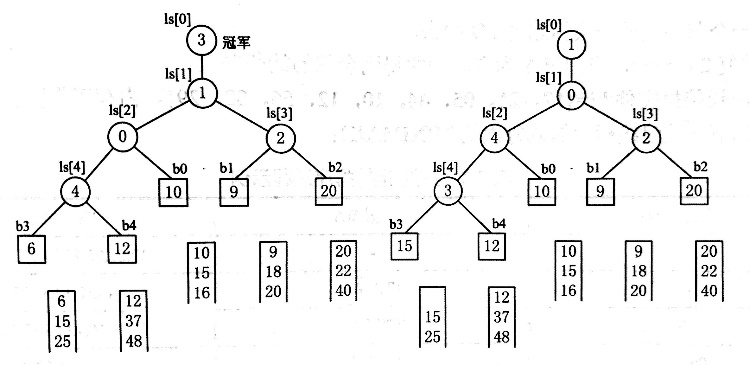
一般地，对r各初始归并段，作m路平衡归并，归并树可用严格m叉树（即只有度为m和度为0的结点的m叉树）

* + 第一趟可将r个初始归并段归并为个归并段
  + 以后每趟归并将l个归并段归并成个归并段
  + 直到最后形成一个大的归并段
  + 树的高度=归并的趟数S=
  + 只要增大归并路数，或减少初始归并段个数r，都能减少归并趟数S，以减少读写磁盘数d，达到提高外排速度的目的

##### 多路平衡归并与败者树

归并趟数S=，从而增加归并路数可以减少归并趟数S，进而减少访问外存的次数。但这样会增加内部排序的时间。

* + 内部排序，m个元素选择关键字最小的元素需要比较m-1次。
  + 每趟归并n个元素需要作(n-1)\*(m-1)次比较
  + S趟归并总共需要比较次数为
  + 其中的在初始归并段个数r与记录个数n一定时是常数。但随着m的增长而增长，则内部归时间也随着m增长。这将抵消增大m而减少外存访问次数所得到的效益，因此，不能使用普通的内部归并排序算法

为了使内部排序不受m增大的影响，引入败者树

* + 因为m归并的败者树深度为，因此m个记录中选择最小关键字，最多需要次比较，所以总比较次数为：  
    可见，使用败者树后，内部归并的比较次数与m无关了。
  + 因此，只要内存空间允许，增大归并路数m将有效减少归并树的高度，从而减少I/O次数d，提高效率

归并路数m并不是越大越好，归并路数m增加，相应地需要增加输入缓冲区个数，如果可供使用的内存空间不变，势必要减少每个输入缓冲区的容量，使得内外存交换数据的次数增大。当m过大时，虽然归并趟数会减少，但读写外存的次数仍然会增加

##### 置换-选择排序（生成初始归并段）

减少初始归并段个数r也可以减少归并趟数S。

设初始待排文件FI，初始归并段文件为FO，内存工作区为WA，内存工作区可容纳m个记录

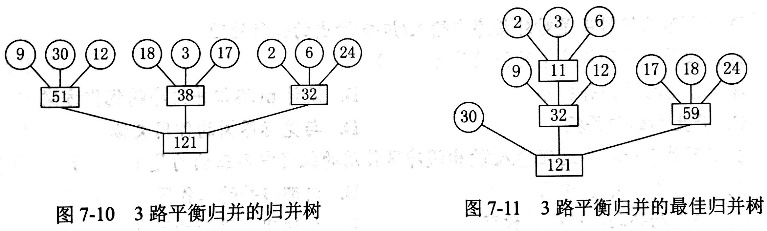
1. 从待排文件FI输入w个记录到工作区WA
2. 从内存工作区WA中选出其中关键字取最小的记录，即为MINIMAX（以后再选出关键字比它大的记录归入本归并段，比他小的归入下一归并段）
3. 将MINIMAX记录输出到FO中取
4. 若FI未读完，则从FI输入下一个记录到WA
5. 从WA中所有关键字比MINIMAX记录的关键字大的记录中选出最小的关键字记录，作为新的MINIMAX
6. 重复3-5，直到WA中选不出新的MINIMAX记录为止，由此得到一个初始归并段，输出一个归并段d额结束标志到FO中去
7. 重复2-6，直到WA为空。由此得到全部初始归并段

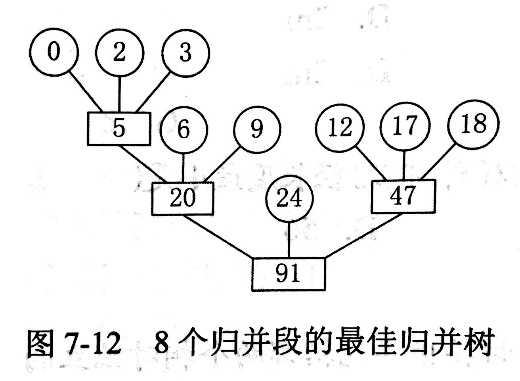
以上算法中，选择MINIMAX记录的过程利用败者树实现

##### 最佳归并树

图示：

* + 叶结点表示参加归并的一个初始归并段
  + 叶结点上的权值表示该初始归并段中的记录数
  + 根结点表示最终生成的归并段
  + 叶结点到根结点的路径长度表示在归并过程中的归并趟数
  + 各非叶结点代表归并成新的归并段
  + 归并树的带权路径长度WPL即为归并过程中的总读记录数。
  + 总的I/O次数=2Xwpl

初始归并段为满m叉树时，利用Huffman树的思想创造最佳归并树。如三叉最佳归并树

如果初始归并段不足构成一棵严格m叉树时，需添加长度为0的虚段  


* + 判定添加虚段的数目：
  + 设度为0的结点有n0(=n)个，度为m的结点有nm个，则对严格m叉树有n0=(m-1)nm+1，由此可以得nm=(n0-1)/(m-1)
  + 如果(n0-1)%(m-1)=0，则说明这n0个结点（初始归并段）正好可以构造m叉归并树。此时，内节点有nm个
  + 如果(n0-1)%(m-1)=u0，则说明对于这n0个叶结点，其中有u个多余，再加上m-u-1个空归并段，就可以建立归并树

##### 习题知识点

置换-选择排序时外排中生成初始归并段的算法

用置换-选择排序得到的初始归并段的长度不是等长的，其长度平均是传统等长初始归并段的2倍，从而使初始归并段数减少到原来的近二分之一

最佳归并树的作用在外排中的作用是设计m路归并排序的优化方案，而不是完成m路归并排序

外排中输入/输出缓冲区作用：

* + 暂存输入/输出记录
  + 内部归并的工作区
  + 产生初始归并段的工作区
  + 就是排序的内存工作区

在作m路平衡排序的过程中，为实现输入/内部排序/输出的并行处理，需要设置2m个输入缓冲区和2个输出缓冲区，以便执行内部排序时，能同时输入/输出操作。若仅设置m个输入缓冲区，则仅能进行串行操作