# 《王道计算机》

## 数据结构

### 树与二叉树

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

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

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）

可唯一确定二叉树

非线索二叉树中共，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个系欸但，则有个不同的二叉树

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

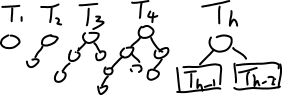
中序遍历二叉排序树，可得到有序数列

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



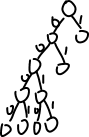
Huffman树权值最小的两个结点一定是兄弟结点

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

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

### 查找

#### 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

##### 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