# 数据结构

## 《王道计算机》

### 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

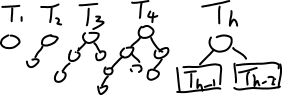
中序遍历二叉排序树，可得到有序数列

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



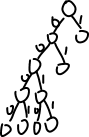
Huffman树权值最小的两个结点一定是兄弟结点

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

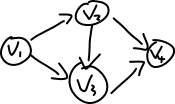
* + DFS
  + 拓扑排序
  + 求关键路径
  + 求最短路径不行

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

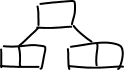
* + 从最小关键字开始的顺序查找
  + 从根结点开始，多路查找

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

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初始状态有关，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

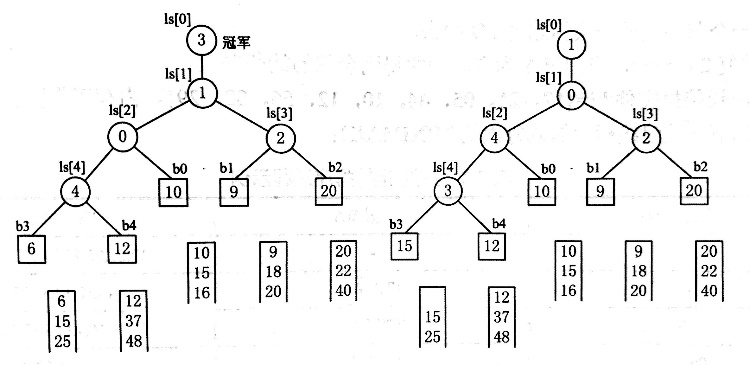
一般地，对r各初始归并段，作m路平衡归并，归并树可用严格m叉树（即只有度为m和度为0的结点的m叉树）

* + 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

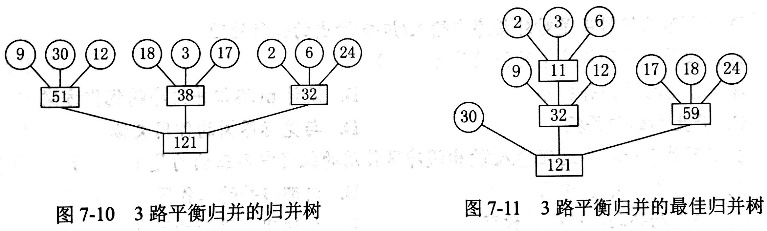
1. 从待排文件FI输入w个记录到工作区WA
2. 从内存工作区WA中选出其中关键字取最小的记录，即为MINIMAX（以后再选出关键字比它大的记录归入本归并段，比他小的归入下一归并段）
3. 将MINIMAX记录输出到FO中取
4. 若FI未读完，则从FI输入下一个记录到WA
5. 从WA中所有关键字比MINIMAX记录的关键字大的记录中选出最小的关键字记录，作为新的MINIMAX
6. 重复3-5，直到WA中选不出新的MINIMAX记录为止，由此得到一个初始归并段，输出一个归并段d额结束标志到FO中去
7. 重复2-6，直到WA为空。由此得到全部初始归并段

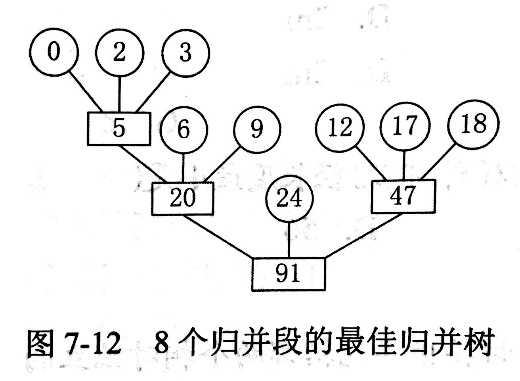
以上算法中，选择MINIMAX记录的过程利用败者树实现

##### 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

# 计算机原理

## 《王道计算机组成原理》

### 重要的名词解释

时钟周期、节拍、T周期：计算机操作最小时间单位，主频的倒数

工作脉冲：控制器的最小时间单位，起定时触发作用，一个时钟周期有一个工作脉冲

指令周期：由多个CPU周期组成

CPU周期，机器周期：包含若干时钟周期

存取周期：存储器进行两次独立的存储器操作所需最小时间间隔

CPU时钟周期，节拍脉冲，T周期，时钟周期

时钟周期总线时钟周期

机器周期由存取周期决定

总线时钟周期

总线周期

总线时钟周期

总线宽度：总线位宽，总线上能同时传输的数据位数，通常为总线根数

总线带宽：字节/秒

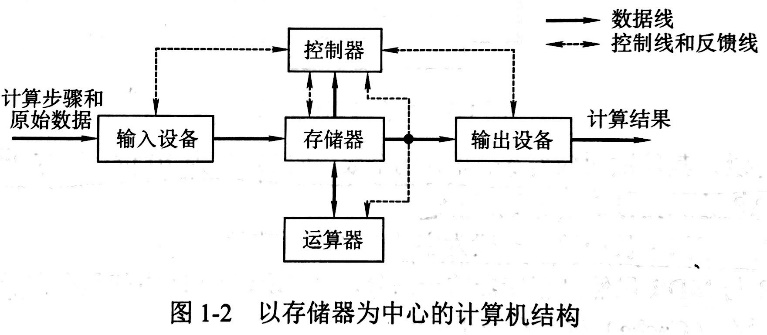
猝发传输：在一个总线周期内传输存储地址连续的多个数据字的总线传输方式

### 计算机系统概述

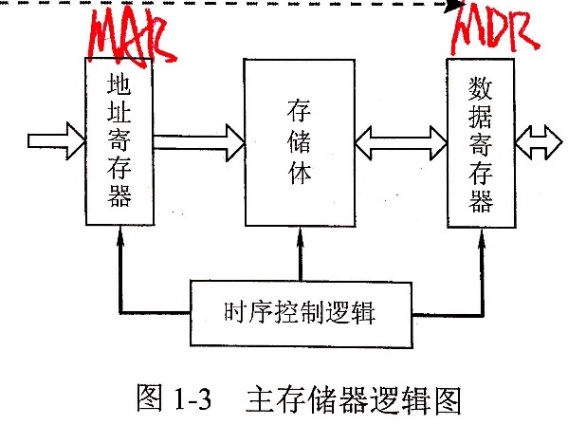
#### 计算机发展历程

机器语言是计算机唯一可以直接执行的语言

##### 计算机层次结构

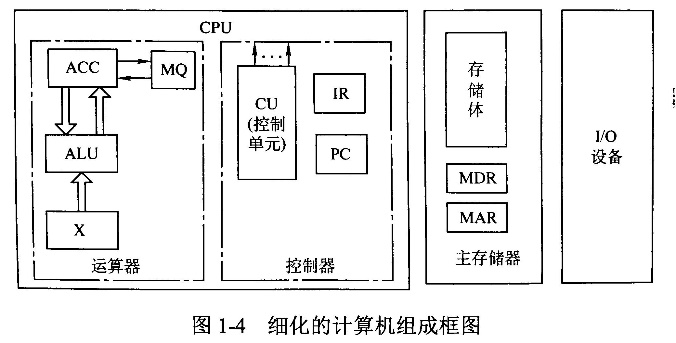
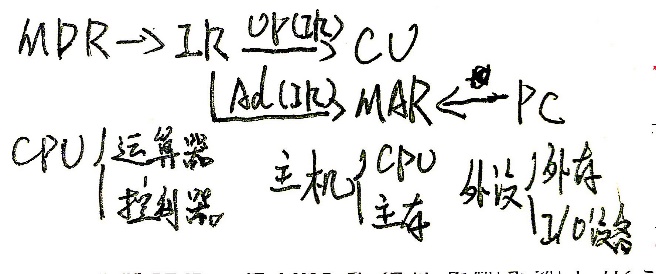
早期计算机以运算器为中心，现代计算机以存储器为中心，使I/O操作尽量绕过CPU

存储器

* + 主存储器：简称主存，也成内存储器，CPU可直接访问
  + 辅助存储器：辅存，也成外存储器，信息须调入内存然后才能被CPU访问
  + **存储单元**可存储**存储字**，一个存储单元有一个存储字，存储字的位数称为**存储字长**，存储字长一般是字节(8bit)或者字节的偶数倍
  + 主存按存储单元的地址进行存取，即**按地址存取方式**
  + **相联存储器**使按内容访问的
  + 主存基本组成
  1. 存储体
  2. **MAR**（地址寄存器）：存放访问地址，经过地址译码后找打所选的存储单元。用于**寻址**，其**位数**对应着**存储单元的个数**，如MAR为10位，则有210=1024个存储单元。**MAR长度**与**PC的长度**相等
  3. **MDR**（数据寄存器）：使主存和其他部件的中介机构，用于暂存要从存储器中读或者写的信息。**MDR的位数**和**存储字长**相等，一般为字节的二次幂整倍
  4. 时序控制逻辑用于产生存储器操作所需的各种时序信号
  + MAR，MDR虽然使存储器一部分，但在现代却是存在于CPU中的，Cacah(高速缓存)也是
  + 运算器核心使ALU（算数逻辑单元）
    - 包含若干通用寄存器
      * 累加器（ACC）（必有）
      * 乘商寄存器（MQ）（必有）
      * 操作数寄存器（X）（必有）
      * 变址寄存器（IX）
      * 基址寄存器（BR）
    - 程序状态寄存器（PSW），保留各类运算指令或测试指令的结果的各类状态信息，以表征系统运行状态

控制器

* + 组成
    - 程序计数器（PC）：存放当前**欲**执行指令的地址，可以自动+1形成下一条指令的地址，与主存的**MAR**有**直接通路**
    - 指令寄存器（IR）：存放当前指令，内容来自主存的**MDR**。指令中的操作码OP(IR)送至CU，用以分析指令并发出各种**微操作命令**序列；而地址码Ad(IR)送到MAR来取操作数
    - 控制单元（CU）：分析指令并发出各种**微操作命令**序列

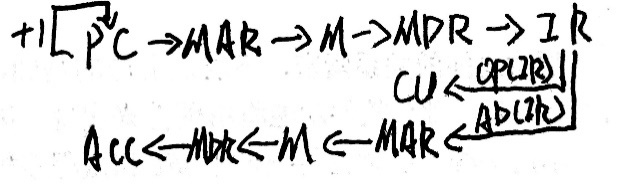
运算器和控制器集成到CPU上，CPU和主存构成主机

##### 软件分类

* + 系统软件：操作系统（os）,数据库管理系统（DBMS），语言处理程序，分布式软件系统，网络软件系统，标准库程序，服务型程序，编译程序
  + 数据库系统（DBS）包含DBMS，但不是系统软件

##### 工作过程

取数指令：将指令地址码知识的存储单元中的操作数取出后送至运算器的ACC中。信息流

* + 取指令：PCMARMMDRIR
  + 分析指令：OP(IR)CU
  + 执行指令：Ad(ID)MARMMDRACC
  + 然后(PC)+1PC
  + 

注意

* + PCMAR
  + (PC)MAR
  + (PC)+1PC

##### 习题知识点

冯诺依曼机的最根本特征是采用”存储程序”原理,基本工作方式:控制流驱动方式;基本特点是按地址访问并顺序执行指令

指令和数据都存放在存储器中,CPU可以根据指令周期的不同阶段来区分:取指阶段:指令；执行阶段：数据。

CPU只有在确定取出的是指令后才会将操作码送去译码

IR,PC,MAR,MDR

* + IR:存放当前欲执行的指令
  + PC:存放下一条指令的地址
  + MAR存放欲访问的存储单元地址
  + MDR存放从存储单元取来的数据

地址译码器在主存中

地址寄存器（MAR）虽然一般属于主存，但是现代计算机中绝大多数CPU内集成了地址寄存器，但不在运算器和控制器中

MAR位数：地址码长度；MDR位数：存储字长

运算器中有：

* + ACC,MQ,X,IX,BR,PSW,ALU,数据总线

速度：寄存器>Cache>内存

编译程序和解释程序的作用都是将高级语言程序转化成机器语言程序

编译程序编译时间长，运行速度快

解释程序方法简单，运行速度慢

解释程序将源程序翻译成机器语言，翻译一条，执行一条

寄存器对汇编语言不透明，可直接操作

软件和硬件在逻辑功能上是等效的，不是等价的

相联存储器既可以按地址寻址也可以按内容寻址

#### 计算机的性能指标

##### 主要性能指标

机器字长

* + 是指计算机进行一次整数运算（定点整数运算）所能处理的二进制数据的位数。与CPU的**寄存器位数**，**加法器**有关。一般等于内部寄存器的大小，字长越长，范围越大，精度越高。通常位字节（Byte,8位）的整数倍。不同计算机，字长可不同

数据通路带宽

* + 数据总线一次能够并行传送信息的位数。这里所说的数据通路带宽是指**外部数据总线**的宽度，它与**CPU内部的数据总线**宽度（**内部寄存器**的大小）有可能不同

主存容量

* + 主存储器的容量，字节或者用字数x字长（512Kx16位）表示。**MAR**的位数反映了存储单元的个数，MAR的位数反映了可寻址范围的最大值（而不一定是实际存储器的存储容量）
  + 例如，MAR16位，表示此存储体内有216=65536个存储单元（可称作64K内存，1K=1024），若MDR为32位，表示存储容量位64Kx32位

运算速度

* + 主频和CPU时钟周期
    - CPU时钟周期：通常为节拍脉冲或T周期，主频倒数，是CPU中最小的时间单位，每个动作至少需要一个时钟周期。MHz
    - 主频（CPU时钟频率）：机器内部主时钟的频率
  + CPI：执行一条指令所需时钟周期数
  + CPU执行时间时间，指运行一条指令所花费的时间  
    CPU执行时间=CPU时钟周期数/主频=（指令条数xCPI）/主频  
    上式表明，CPU性能取决于
    - 主频（时钟频率）
    - CPI
    - 指令条数
  + MIPS=指令条数/（执行时间\*106）=主频/CPI  
    每秒执行多少百万条指令
  + MFLOPS=浮点数操作次数/（执行时间\*106）  
    每秒执行多少百万次浮点运算
  + GFLOPS=浮点数操作次数/（执行时间\*109）  
    每秒执行多少十亿次浮点运算
  + TFLOPS=浮点数操作次数/（执行时间\*1012）  
    每秒执行多少万亿次浮点运算

##### 习题知识点

能缩短程序执行时间：

* + 提高CPU时钟频率
  + 优化数据通路结构
  + 对程序进行编译优化

数据通路的功能是实现CPU内部的运算器和寄存器以及寄存器之间的数据交换

寄存器由触发器构成

计算机的机器字长是指数据运算的基本单位长度

机器字长，指令字长和存储字长，三者在数值上可以相等也可以不等

数据字长是数据总线一次能够并行传送信息的位数，他可以不等于MDR的位数

计算机的位数，即机器字长，也就是计算机一次能够处理的二进制数的长度。如32位微机

操作系统的位数是操作系统可寻址的位数，与机器字长是不一样的。

一般可以通过寄存器判断机器字长

CPU的寄存器

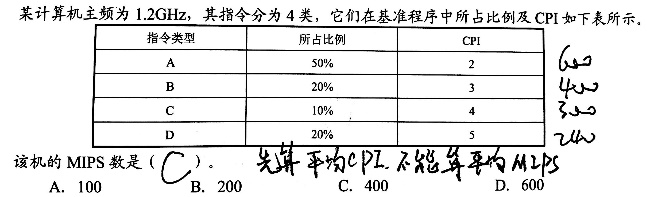
* + 对用户透明：IR,MAR,MDR
  + 不透明：PC，PSW，通用寄存器

**时钟周期**即CPU频率的倒数，是最基本的时间单位。**CPU周期**又称**机器周期**，由多个时钟周期组成

CPI影响因子

* + 系统结构
  + 指令集
  + 计算机组织

决定计算机计算精度的主要技术是计算机的字长

在计算多个指令的综合MIPS的时候，应先算平均CPI，不能分别求出MIPS再求均值

* + CPI=2\*0.5+3\*0.2+4\*0.1+5\*0.2=3  
    主频1.2GHz=1200MHz  
    故MIPS=1200/3=400

CPU提速50%并不是总时间减少50%，而是T=T0/1.5

如果主存储器容量位64K\*32位，且指令字长，机器字长，存储字长三者相等则

* + 216=64K，地址总线宽度为16位
  + 32位表示数据总线宽度
  + MAR=PC=16位
  + MDR=32位
  + 因为三者相等，所以IR,ACC,MQ,X均为32位

#### 易混清单

主频高的CPU不一定快

机器语言和汇编语言与机器指令对应，而高级语言不与指令直接对应

**机器字长**：计算机能直接处理的二进制数据的位数，机器字长一般等于内部寄存器的大小，决定了计算机的运算精度  
等于计算机位数

**操作系统位数**是操作系统可寻址位数，与机器字长不同，与MDR长度相等

**指令字长**：一个指令字中包含二进制代码的位数

**存储字长**：一个存储单元存储二进制代码的长度。

指令字长一般都取存储字长的整数倍，如果指令字长等于存储字长的两倍，就需要2次访存来取出一条指令，因此，取指周期为机器周期的2倍，如果指令字长等于存储字长，则取指周期等于机器周期

**数据字长**：数据总线一次能够并行传送信息的位数

64位机器既可以使用64位操作系统，也可以使用32位操作系统；32位机器不可以使用64位操作系统

### 数据的表示和运算

#### 数制和编码

##### 进位计数制及其相互转换

一个r进制数(KnKn-1…K0K-1…K-m)的数值可以表示为：  
Knrn+Kn-1rn-1+…+ K0r0 +K-1r-1+…+K-mr-m=

在计算机中，整数可以连续的表示，但小数并不是每一个十进制小数都可以准确地用二进制表示。任意的一个二进制小数都可以用十进制小数表示

##### 真值和机器数

带+-号的数称为**真值**。真值是机器数所代表的实际值

##### BCD码

二进制编码的十进制数（BCD），采用4位二进制数来表示以为十进制数中的0~9，故有6种冗余状态

8421码

* + 有权码，最常用。权值从高到低依次为8421
  + 如果两个8421码相加，之和，即(9)10，则不需要修正
  + 如果即(10)10,则需要+(0110)2/(6)10修正（从1010到1111这6个为无效吗），并向高位进位

余3码：

* + 无权码，在8421基础上加(0011)2

2421码

* + 有权码，2421
  + 特点是5的4位二进制数种最高位为1，<5的最高位为0。如510=(1011)2而不是(0101)2

##### 字符与字符串

主存字由2个或4个字节组成的时候，在同一个主存字中：

* + 小端模式： 按先存储低位字节、后存储高位字节（即从低位字节到高位字节顺序）存放字符串的内容
  + 大端模式：按先存储高位字节、后存储低位字节（即从高位字节到低位字节顺序）存放字符串的内容

##### 校验码

数据校验码的码距：通常某种编码都有许多码字组成，任意两个合法码之间最少变化的二进制位数

* + 码距2的数据校验码，开始有**检**错能力
  + 码距越大，检、纠错能力越强
  + 检错能力总是纠错能力

L-1=D+C,

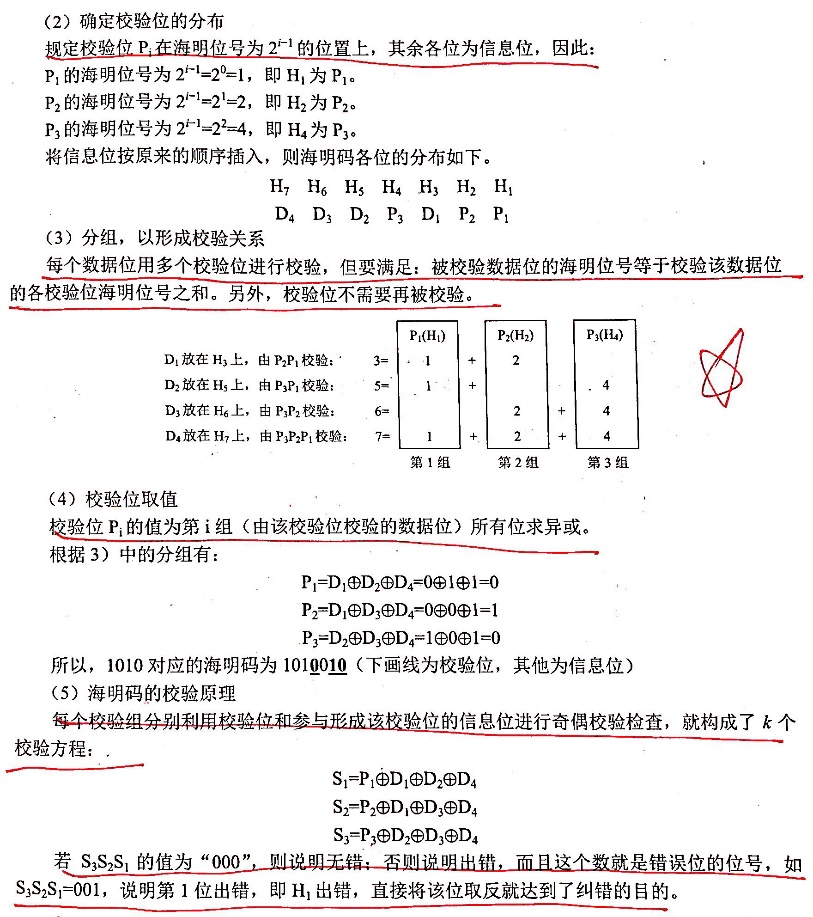
* + 编码最小码距越大，其检测错误的位数D越大，纠正错误的位数C越大
  + 纠错能力检错能力

奇偶校验码

* + 奇校验：整个校验码（有效信息位和校验位）中1的个数为奇数
  + 偶校验
  + 只能发现数据代码中奇数位出错情况，不能纠正错误

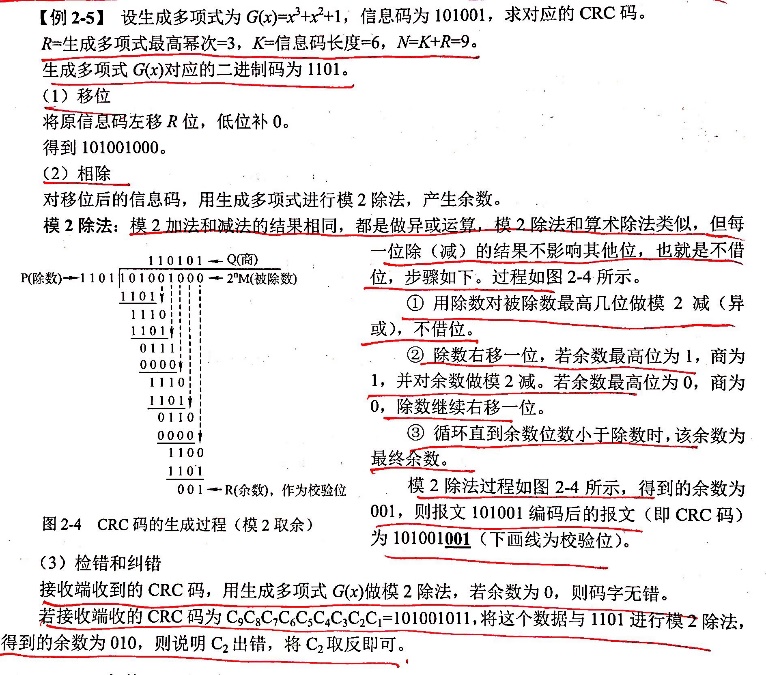
海明校验码

* + 是多重奇偶校验码
  + 求海明码的步骤
  + n=4,k=3，求1010的海明码：
  + 确定海明码位数
    - 为达到检测和纠正1位错：设n为有效信息的位数，k为校验位的位数，则n,k须满足：(若要检测两位错，则需再增加1位校验位，即k+1位)  
      海明码位数为成立，则nk有效。  
      设信息位D4D3D2D1(1010),校验位P3P2P1,对应的海明码为H7H6H5H4H3H2H1



循环冗余校验(CRC)码

* + 再K位信息位信息码后再拼接R位的校验码,整个编码的长度为N,这种编码又称为(N,K)
  + 在发送端,先将K位信息位左移R位,与生成的多项式G(x)做模2运算，生成R位校验码附在信息位后面形成（K+R）位的新的二进制码
  + G(x)的最高次幂为R,共R+1位。  
    如x3+x2+1对应的为1101



##### 习题知识点

(137.5)10=(10001001.1)2=(010001001.100)2=(211.4)8

ASCII码由7位二进制代码表示，从0000000到1111111,共128种编码。在计算机中仍以一个字节存入，最左位为0

字节编制的计算机，小端模式，01234567H，首先存放67H

增加奇偶校验位的位数，并不会提高正确性

CRC码是通过除法运算来建立数据和校验位之间的约定关系的

奇偶校验的ASCII码，校验位存放在最左端那一位

海明码

* + 发现1位，纠正1位：
  + 发现2位，纠正2位：

CRC中，接收端检测出错误后，可以：请求重发，删除数据，通过余数自动纠正

检测纠错1位，8位信息位，需要4个校验位

汉明码求一长串异或时，是不是可以相当于

* + 偶数个1则结果0
  + 奇数个1则结果1

全校验就是偶校验，比如0 10111最左位全校验，右边有偶数个1，所以全校验为0

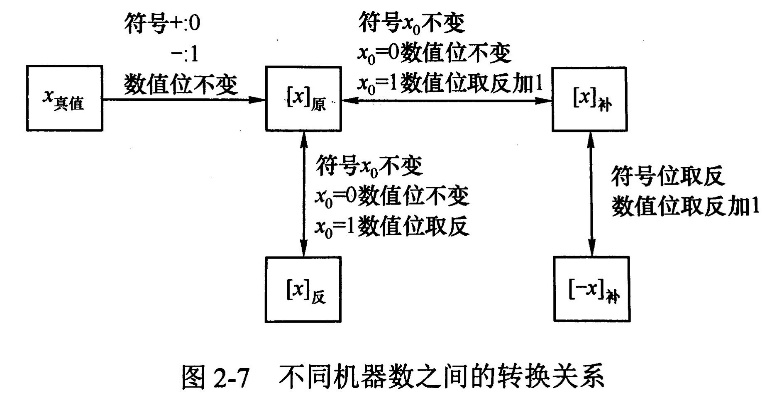
#### 定点数的表示与运算

##### 定点数的表示

设字长n+1位，其中一位符号位

* + 原码
    - 小数：
    - 整数：
    - x1=+0.1101，x2=-0.1101 字长8位  
      [x1]原码=0.1101000，[x2]原码=1.1101000
    - x1=+1110，x2=-1110 字长8位  
      [x1]原码=0,0001110，[x2]原码=1,0001110
  + 补码
    - 小数： [0]补码=0.0000…  
       [-1]补码=1.000…
    - 整数： [-]补码=1000…
    - x1=+0.1001，x2=-0.0110 字长8位  
      [x1]补码=0.1001000，[x2]补码 =1.1010000
    - x1=+1010，x2=-1101 字长8位  
      [x1]补码=0,0001010，[x2]补码 =1,1110011
  + 反码
    - 小数：  
       [+0]反码=0.00…，[-0]反码=1.111…
    - 整数： [-()]反码=1000….
    - x1=+0.0110，x2=-0.0110 字长8位  
      [x1]反码=0.0110000，[x2]反码=1.1001111
    - x1=+1011，x2=-1011 字长8位  
      [x1]反码=0,0001011，[x2]反码=1,1110100
  + 移码：
    - [0]移=1000..(n个0)
    - [-2n]移=0000
    - [2n-1]移码=1111…
    - x1=+10101，x2=-10101 字长8位  
      [x1] 移=1,0010101，[x2]移=01101011
    - 移码特性：
      * 移码就是在真值x上加一个常数（偏置值），通常取2n,相当于x在数轴上向正方向偏移了若干单位
      * [x]移=2n+x ()
      * 移码中0表示的唯一
      * 一个真值的移码和**补码**仅差一个符号位，符号位求反即是移码
      * 移码保持了数据原有的大小顺序，移码大真值就大，移码小真值就小

补码比原码，小数多个-1，整数多个-2n



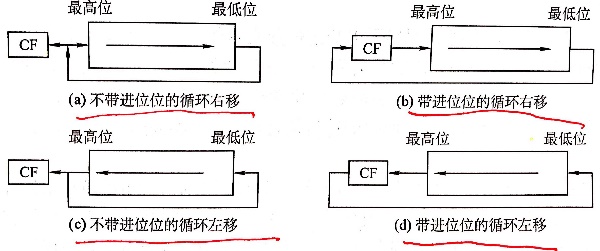
##### 定点数的运算

定点数的移位操作：

* + 算数移位：对有符号数的移位称为算数移位
    - 不论是正数还是负数，移位后符号位均不变，且移位后都相当于对真值补0。所以对于正数移位都是补0，负数则有区别
    - 对于原码，左移移位若不产生溢出，相当于\*2，右移一位，若不考虑舍去的末尾尾数，相当于/2

|  |  |  |
| --- | --- | --- |
|  | 码制 | 添补代码 |
| 正数 | 原码、补码、反码 | 0 |
| 负数 | 原码 | 0 |
| 补码 | 左移添0 |
| 右移添1 |
| 反码 | 1 |

* + 逻辑移位：操作对象时逻辑代码，可以视为无符号数
    - 逻辑左移，高位移丢，低位添0；
    - 逻辑右移，低位移丢，高位添0
  + 循环移位：
    - 带进位标志位CF的循环移位（大循环）
    - 不带进位标志位的循环移位（小循环）



原码定点数的加/减法

* + 加法：判符号位，
    - 相同：绝对值相加，符号位不变
    - 不同，绝对值大的减绝对值小的，符号位和绝对值大的相同
  + 减法：先将减数符号取反，两者按照加法运算
  + 左边出现溢出时，将溢出位丢掉

补码定点数加/减法（设机器字长n+1）

* + 符号位与数值位按同样规则一起参与运算，符号位运算产生的进位要丢掉，结果的符号位由运算得出
  + [A+B]补=[A]补+[B]补
  + [A-B]补=[A]补+[-B]补
  + 溢出位丢掉

符号扩展

* + 正数：原符号位移动到新符号位，其余补0
  + 负数：
    - 原码，同正数
    - 补码：原符号位移动到新符号位，新表示形式的所有附加位，整数补1，小数补0
    - 反码：附加为都用1填充

溢出概念和判别方法

* + 上溢：大于机器所能表示的最大整数
  + 下溢：小于机器所能表示的最小负数
  + 定点小数的表示范围 |x|<1
  + 仅两个符号相同的数相加，或两个符号相异的数相减才可能会溢出
  + 补码定点数加/减法溢出判断3种方法：
    - 采用一位符号位
      * 减法运算在机器中也是加法，所以只要参加操作的两个符号相同，结果又与员操作数不同，就是溢出
      * 设A的符号为As,B的符号位位Bs，运算结果符号位Ss  
        若V=0，无溢出；V=1，有溢出
    - 采用双符号位
      * 双符号位法也称模4补码。两个符号位相同，表示未溢出；不同，表示溢出
      * SS1SS2=00：表示结果为正数，无溢出
      * SS1SS2=01：表示结果正溢出
      * SS1SS2=10：表示结果负溢出
      * SS1SS2=11：表示结果为负数，无溢出
      * V=SS1SS2：V=0，无溢；V=1：溢出
    - 一位符号位根据进位来判断
      * 如果符号位的进位CS与最高数位的进位C1相同，没有溢出；否则溢出
      * V=CSC1

定点数的乘法运算（暂时略过）

定点数的除法运算（暂时略过）

##### 强制类型转换

强制类型转换的结果保持位值不变，知识改变了解释这些位的方便

当大字长变量向小字长变量强制类型转换时，系统把多余的高位字长部分直接截掉，低位直接赋值，这也是一种保持位值的方法

短字长到长字长变量的转换，不仅是把相应的位相等，高位部分还会拓展为原数字的符号位

##### 习题知识点

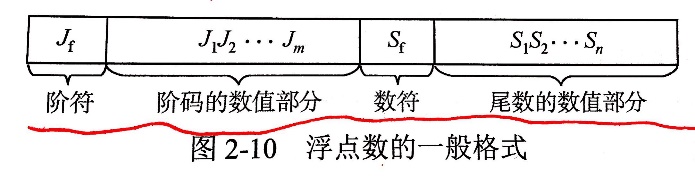
C语言的数据在内存中为补码形式

32位机器：char,1字节；short,2字节；int,4字节；long,4字节；float,4字节；double,8字节

数据转换

* + 有符号和无符号之间的转换。Signed转型化为等长的unsigned型数据，符号位成为数据的一部分，即负数转化为无符号数数值将发生变化。由unsigned转化为signed时，最高位作为符号位，也可能发生数值变化
  + 数据的截取和保留。当一个浮点数转化为整数时，浮点数的小数部分全部舍去，并按整数形式存储。但浮点数的整数部分不能超过整型数允许的最大范围，否则溢出
  + 数据转换中的精度丢失。四舍五入会丢失一些精度，截取小数也会丢失一些精度。数据由long型转化为float或double型，有可能在存储时不能准确的表示该长整数的有效数字，精度也会收到影响
  + 数据转换结果的不确定性。当较长的整数转化为较短的整数时，要将高位截取。如，long型转化称short型，只将低16位送过去，这样就会产生很大的误差。付i单数降格时，如double型转化位float型，当数值超过了float型的表示范围时，所得到的结果时不确定的

边界对齐

1. struct A{
2. int a;
3. char b;
4. short c;
5. };
6. struct B{
7. char b;
8. int a;
9. short c;
10. };
    * sizeof(strcut A)值为8字节；sizeof(struct B)的值却是12字节

16位无符号整数最大216-1=65535

对于相同位数（设为N位，不考虑符号位）的二进制补码和十进制小数，二进制小数能表示的数的个数/十进制小数能表示的个数位(0.2)N

补码表示时，符号位相同，数值位越大，码值越大（用移码理解，移码相当于，补码右移）

不带进位位循环左移，若有溢出(原数值位最高位1)，则CF变为1

[-|x|]补码=[-x]补码，当且仅当x=0或为正

三种溢出判别方法，均须有溢出判别电路，可用“异或”们来实现

8421码是十进制数的编码，不能说就是二进制数

正数的原码、反码、补码都相同

模4补码具有模2补码的全部优点且更容易检查加减运算中的溢出问题

每个模4补码存储时只需要一个符号位，只在把两个模4补码的数送往ALU完成加减运算时，才把每个数的符号位的值同时送到ALU的双符号位中，即只在ALU中采用双符号位

8位定点补码范围：-128~127

补码一位乘法中，最多需要n次位移，n+1次加法运算

原码乘法移位和加法运算最多均为n次

在原码一位乘法中，符号位不参加运算，单独处理，同号正，异号负

N位（不包含符号位）补码一位乘运算过程中一共右移位N次，加上原先的N位，一共是2N位数值位，再加上符号位，即2N+1位

原码不恢复余数除法即加减交替法，只在最终余数为负时（最后一步不够减），才需要恢复余数

补码不恢复余数除法中，异号相除时，够减商0，不够减商1

计算机中，通常用来表示主存地址的是无符号数

由三个1，五个0组成的8位二进制补码，能表示的最小整数是：[10000011]补码=[1,1111101]原码=(-125)10记住补码表示时，符号位相同，数值位越大，码值越大

[X]补=0.1011，[Y]补=1.1110，[-Y]补=0.0010  
[X+Y]补=10.1001=0.1001  
[X-Y]补=[X]补+[- Y]补=0.1101

变形补码就是模4补码00.0000,双符号位

产生进位不一定溢出

#### 浮点数的表示与运算

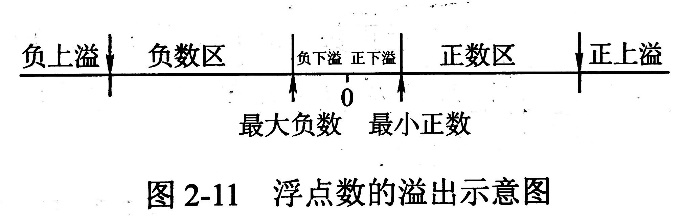
##### 浮点数的表示

浮点数被表示为,r是浮点数阶码的底（隐含），与尾数的基数相同，通常r=2。E和M都是带符号的定点数，E称为阶码，M称为尾数。可见浮点数由**阶码**和**尾数**两部分组成

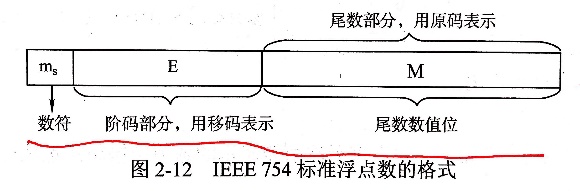
* + 阶码是整数，阶符Jf和阶码的位数m合起来反映浮点数的表示范围及小数点的实际位置；
  + 数符Sf代表浮点数的符号；
  + 尾数的位数n反映浮点数的精度

规格化浮点数

* + 规定尾数的最高数位必须是一个有效值
  + 所谓规格化就是通过调整一个非规格化浮点数的尾数和阶码的大小，使非零的浮点数尾数的最高数位上保证是一个有效值
  + **左规**：将尾数算数左移一位，阶码-1(基数为2时)，左规可能需要进行多次
  + **右规**：当浮点数运算的结果出现溢出(双符号位01,10)时，将尾数算数右移一位，阶码+1(基数为2时)。右规只需进行一次
  + 规格化浮点数的尾数M应该满足：，如果r=2，则有
  + 原码规格化后：
    - 正数为0.1xxxx形式  
      最大值表示为0.11..1；最小值0.100..0  
      尾数的表示范围
    - 负数为1.1xxxx形式  
      最大值表示为1.10..0；最小值1.11..1  
      尾数的表示范围
  + 补码规格化后：
    - 正数为0.1xx形式  
      最大值:0.11..1；最小值:0.100..0  
      尾数的表示范围
    - 负数为1.0xxx形式  
      最大值:1.01..1；最小值:1.00..0  
      尾数的表示范围
  + 基为2：
    - 原码尾数最高位一定是1
    - 补码尾数最高位与符号位相反
  + 基为4：
    - 原码尾数最高两位不全为0
    - 补码(正)尾数最高两位不全为0，(负)不全为1
  + 基为8：
    - 原码尾数最高三位不全为0
    - 补码(正)尾数最高三位不全为0，(负)不全为1

运算结果大于最大正数：正上溢；小于绝对值最大负数为负上溢，正上溢和负上溢统称上溢。产生上溢，计算机中断处理  
运算结果在0至最小正数之间称正下溢；在0至绝对值最小负数之间称负下溢。下溢计算机当作0处理

IEEE754标准

* + 

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 类型 | 数符 | 阶码 | 尾数数值 | 总位数 | 偏置值 | |
| 十六进制 | 十进制 |
| 短浮点数(单精度,float) | 1 | 8 | 23 | 32 | 7FH | 127 |
| 长浮点数(双精度double) | 1 | 11 | 52 | 64 | 3FFH | 1023 |
| 临时浮点数 | 1 | 15 | 64 | 80 | 3FFFH | 16383 |

* + IEEE754标准的浮点数（除临时浮点数外），是尾数用采取隐藏位策略的原码表示，且阶码用移码表示的浮点数
  + 对于规格化的二进制浮点数，数值的最高位总是1，IEEE754将这一位**隐藏**，因此实际上尾数数值是24位(短浮点数)。隐含的1是一位整数。在浮点格式中表示出来的23位尾数是纯小数。例如(12)10=(1100)2,规格化后1.1\*23,其中整数部分的1不存储在23位尾数中
  + 短浮点数和长浮点数都有一个隐含位，临时浮点数没有
  + 存储浮点数阶码部分之前，偏置值要先加到阶码真值上。如1.1\*23阶码值为3，故在短浮点数中，移码表示的阶码为127+3=130(82H);长浮点数中为1023+3=1026(402H)
  + IEEE754标准中，规格化的**短浮点数的真值**为：  
     (-1)sx1.Mx2E-127
  + 规格化的**长浮点数真值**为：  
     (-1)sx1.Mx2E-1023
  + S=0表示正数；  
    短浮点数E(1~254)  
    长浮点数E(1~2046)

|  |  |  |
| --- | --- | --- |
| 格式 | 最小值 | 最大值 |
| 单精度 | E=1,M=0, 1.0x21-127=2-126 | E=254,M=.11… , 1.11…1x2254-127=2127x(2-2-23) |
| 双精度 | E=1,M=0, 1.0x21-1023=2-1022 | E=2046,M=.11… , 1.11…1x22046-1023=21023x(2-2-52) |

定点数和浮点数区别

* + 若定点数和浮点数的字长相同，浮点表示法所表示的数值范围将远远大于定点表示法
  + 若定点数和浮点数的字长相同，浮点数精度低
  + 浮点数包括阶码和尾数，两个运算都要做，并且要求结果规格化，所以运算复杂
  + 定点运算中，运算结果超过数的表示范围，就溢出；浮点运算中，运算结果超出尾数表示范围不一定溢出，只有规格化后阶码超出所能表示的范围时，才发生溢出

偏置值为127（而不是128），是空出8位全1来表示无穷大（若偏置值选128，则无法区分无穷大）。阶码值E(1~254)，空出全0表示非规格化数

##### 浮点数的加减运算

浮点数运算，阶码和尾数分开进行。浮点数的加/减运算一律采用补码

1. 对阶
   * 目的让两个操作数小数点对齐，即让**阶码**相等。
   * 先求阶差，然后**小阶向大阶**看齐，将阶码小的尾数右移一位(基2)，阶+1，直到阶码相等
   * 尾数右移时，舍弃掉有效位会差生误差，影响精度
2. 尾数求和
3. 规格化
   * 以双符号位为例，尾数>0时，补码规格化形式为  
      [S]补码=00.1xx  
     尾数<0时，补码规格化形式为  
      [S]补码=11.0xx
   * **左规**：当尾数出现00.0xx或11.1xx时，需左规，即尾数左移1位，和的阶码-1，直到尾数为00.1xx或11.0xx
   * **右规**：当尾数求和结果溢出(10.xx或01.xx)时，需右规，即尾数右移一位，和的阶码+1
   * 左规一次相当于乘2，右规一次相当于除2；  
     [-1/2]补=1.100不是规格化数，需左规一次，[-1]补=1.00才是规格化数
4. 舍入
   * 在对阶和右规过程中，可能会出现将尾数低位丢失，引起误差，影响精度
   * **0舍1入法**：类似于十进制数中的四舍五入，即尾数右移时，被移去的最高数值位为0，则舍去；被移去的最高数值位为1，则在尾数的末位+1，这样做可能会使尾数又溢出，此时需再做一次右规
   * **恒置1法**：尾数右移时，不论丢掉的最高数值位是1还是0，都使右移后的尾数末位恒置1。这种方法同样有使尾数变大变小两种可能
5. 溢出判断
   * 当尾数之和/差出现01.xx或10.xx时，并不代表溢出，只有将次数右规后，再根据阶码来判断浮点数运算结果是否溢出
   * 浮点数的溢出与否是由**阶码的符号**决定的。以双符号位补码为例，
     + 当阶码符号位出现01时，即阶码大于最大阶码，表示上溢，进入中断处理
     + 当阶码符号位出现10时，即阶码小于最小阶码，表示下溢，按机器0处理
6. 强制类型转换
   * Char-int-long-double;float-double从前往后时范围和精度都从小到大，转换过程没有损失
   * Char为8位ASCII码整数，转化为int，在前面补0
   * Int和unsigned int可以互相转化，但彼此都可能因溢出而造成数据丢失，如8位int和unsigned int  
     a=-1,(unsigned int)a=255;  
     (unsigned int)a=128,(int)a=-128
   * Int和float转换，如果float是小数，转化为int可能会发生精度损失（小数仅保留整数部分）和溢出，从int转float，虽然不会发生溢出，但int可以保留32位，float保留24位，可能有数据舍入，double则不会出现  
     int<224-1时，转float就没问题（存疑）

##### 习题知识点

IEEE754单精度表示-8.25  
(8.25)10=(1000.01)2=(1.00001)2\*23E=3+127=130=(10000010)2 M=(00001)2  
1,10000010,000010000000…=C104 0000H  
此时不用进行规格化M=(00001)2

定义浮点数格式：7位阶码，1位数符，8位尾数。阶码用移码，尾数用补码，则浮点数能表示的范围：  
7位移码，除掉1个符号位，可表示范围  
   
8位不带符号的补码，可表示范围  
   
所以，浮点数能表示的范围

对阶操作不存在阶码减小，都是小阶对大阶

IEEE754短精度，阶码的偏置值127  
如阶码为(10001100)2，则快捷计算  
 (0001100)2+1=(1101)2=13  
普通计算：  
 (10001100)2=128+8+4-127=13

IEEE754长精度则是阶码的后10位（共11位）+1

和非规格化的浮点数相比，采用规格化的浮点数主要是为了增加数据的表示精度

设浮点数共12位，其中阶码含1位阶符共4位，2为底，补码；尾数含1位数符共8位，补码表示，规格化。则该浮点数能表示的最大正数为：  
阶码MAX：23-1=7，尾数：0.1111111，即1-2-7  
 27\*(1-2-7)=27-1

已知X=-0.875\*21，Y=0.625\*22，设浮点数格式为阶符1位，阶码2位，数符1位，尾数3位，通过补码求出Z=X-Y的二进制浮点数规格化结果  
 补码：(-0.875)10=-(0.111)2=(1.001)2  
 X=001,1001 Y=010,0101  
 X对齐：010,1100  
 尾数部分X-Y：10,111 需要右归  
 右规后 Z=011,1011，符合规格化

求-0.875\*21的补码的较快方式：  
 -0.875\*2=-1.75=(-1.11)2=(10.01)2=1.001\*21

假设机器数用反码表示，当机器数为负时，左移时最高数位丢0，结果出错；右移时，最低数位丢0，影响精度

在算数移位的情况下，反码、补码左移的前提条件时其原最高有效位与原符号位要相同

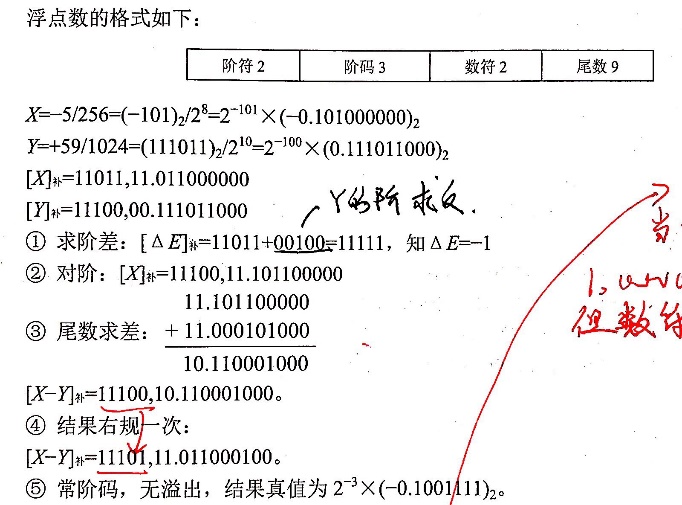
在算数移位的情况下，双符号位的移位操作只有低符号位需要参加移位操作

在采用舍入到最接近且可表示的值时，若要舍入成两个有效数字形式，(12.5)D应该舍入为12  
由于最后一位是5，采用取偶数的方式，即有两个最接近值时，选择偶数，此题有12，13两个数，取12

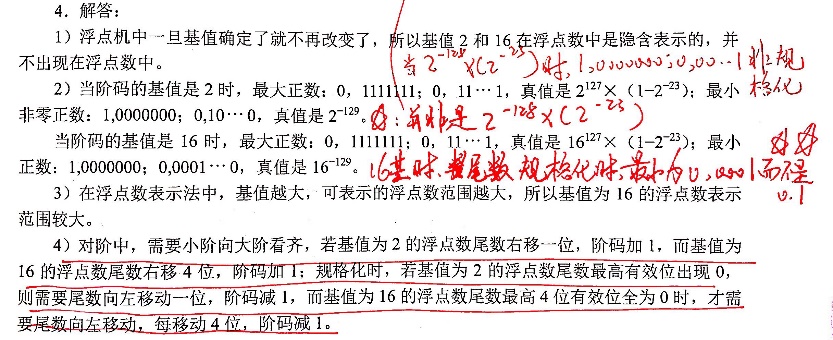
舍入时浮点数的概念，定点数没有舍入

浮点数舍入的情况有两种：对阶、右规格化

舍入不一定产生误差

已知十进制数X=-5/256，Y=+59/1024，按机器补码浮点运算规则计算X-Y,结果用二进制表示，浮点数格式如下：阶符取2位，阶码取3位，数符取2位，尾数取9位

设浮点数长32位，其中阶码部分8位（含一位阶符），尾数部分24位（含一位数符），当阶码的基值分别是2和16时：

* + 说明基值2和16在浮点数中如何表示
  + 当阶码和尾数均用补码表示，且尾数采用规格化形式时，给出两种情况下所能表示的最大正数真值和非零最小真值
  + 在哪种情况下，数的表示范围大
  + 两种基值，对阶和规格化操作有何不同

Float型变量在计算机中都被表示成IEEE754单精度格式

IEEE754在进行运算时，对阶操作，尾数右移，不要忘了隐藏的”1”,也是很要跟着右移的

IEEE754在初始化的时候，要求数值第一位必须是1，然后隐藏；其他浮点数格式定义，按照原码/补码规格化定义来，如补码就要求符号位和第一数值位相反

两个规格化浮点数进行加减运算，最多右规一次；左规次数无法确定，但不会超过尾数的字长n位次

IEEE754

* + 阶码全0 & 符号位1 & 尾数全0：-0
  + 阶码全0 & 符号位0 & 尾数全0：+0
  + 阶码全1 & 符号位1 & 尾数全0：负无穷大
  + 阶码全1 & 符号位0 & 尾数全0：正无穷大
  + 阶码全0 & 尾数非全0：非规格化数
    - 非规格化数，无隐含的”1”，只有规格化数才隐含”1”，如
    - 1,00000000,10000000…实际值为  
      (-0.1)2\*2-126=-2-127 (20-127=-126)

#### 算数逻辑单元ALU

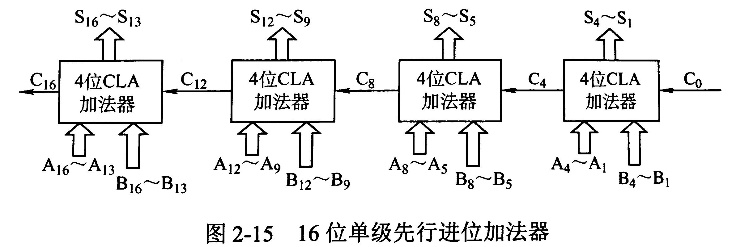
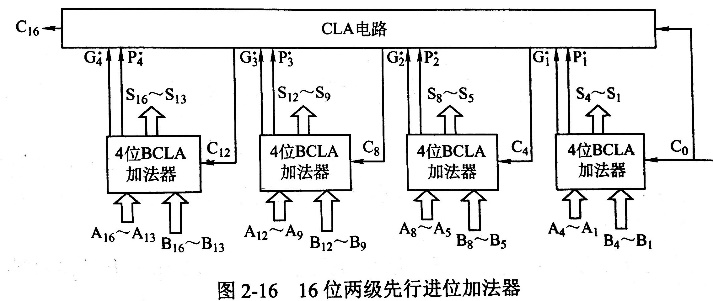
##### 串行加法器和并行加法器

串行加法器只有一个全加器。如果操作数长n位，加法就要分n次进行，每次产生一位和，并且串行逐位地送回寄存器

并行加法器

* + 串行进位：把n个全加器串接起来，就可可以进行两个n位数相加，即串行进位地并行加法器。串行进位又称**行波进位**，每一级直接依赖于前一级地进位，即进位信号是逐级形成地。最长运算时间主要是由进位信号地传递时间决定的。
  + 并行进位：又称**先行进位**，同时进位。特点是各级进位信号同时产生

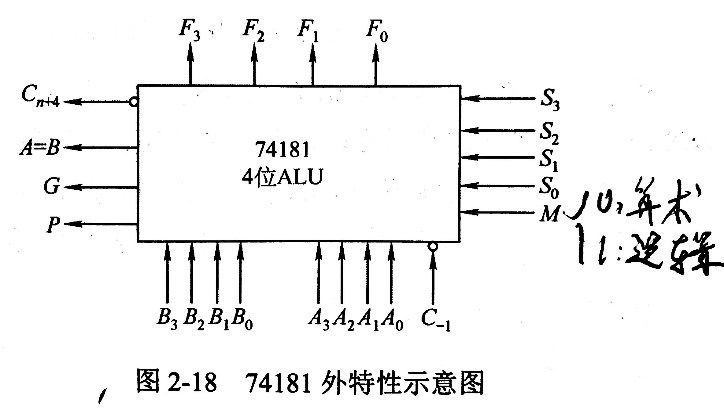
分组并行进位方式

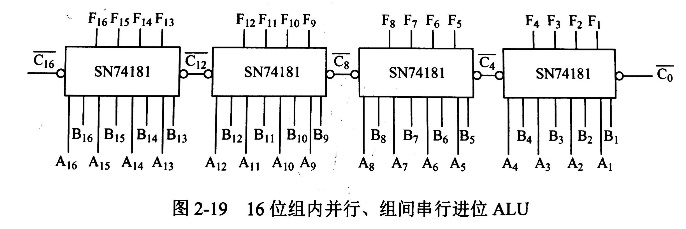
* + **单级先行进位**方式又称组内并行、组间串行
    - CLA：先行进位电路
  + **多级先行进位**方式又称组内并行、组间并行
    - BCLA：成组先行进位电路

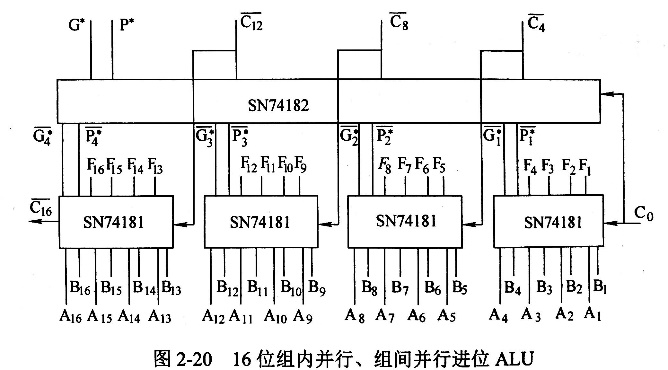
##### 算术逻辑单元的功能和结构

ALU的核心首先应当是一个并行加法器，同时也能执行与或非等逻辑运算

4位ALU芯片（74181，4位并行加法器）

74181能执行16种算数运算，16种逻辑运算，可工作在正负逻辑下。M用来区分算数运算和逻辑运算，S3~S0的不同取值实现不同操作

74182先行进位芯片



##### 习题知识点

ALU是由组合逻辑电路构成的，最基本的部件是并行加法器。单纯的ALU不能够存储运算结果和中间变量，往往将ALU和寄存器或暂存器相连

串行进位的并行加法器，影响加法器运算速度的关键因素是进位传递延迟

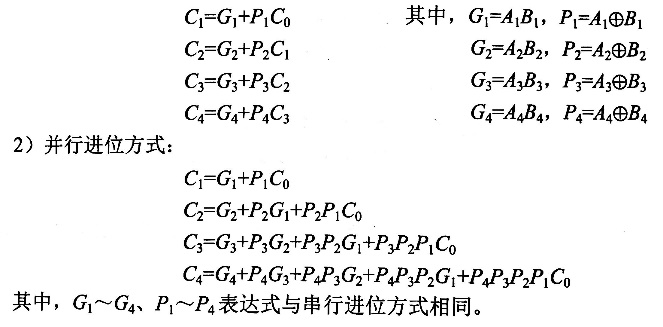
在设计多维加法器时，为了加快运算速度而采用了快速进位链，即对加法器每一位都生成两个信号：进位信号g和进位传递信号p，

用8片74181和两片74182可组成二级先行进位结构的32位ALU  
74181：4位内部先行进位的ALU  
74182：4位先行进位ALU  
每4片74181与1片74182相连，可组成一个两级先行进位结构的16位ALU，两个这种结构的16位ALU串行进位构成两级先行进位的32位ALU

状态寄存器，ALU，数据总线都是组成运算器的部件

某加法器进位链小组信号为，低位来的进位信号为，请分别写出的逻辑表达式

* + 串行进位方式
  + 并行进位方式



#### 易混知识点

字长相同时，浮点数和定点数表示范围与精度

* + 浮点数范围大，定点数精度大

现代计算机用IEEE754标准表示浮点数

* + Float:32位单精度浮点数
  + Double:64位双精度浮点数

C语言中的int和unsigned型变量的存储方式没有区别，都是按照补码的形式存储，在不溢出的范围内的加减法运算也是相同的

对无符号定点整数来说，若寄存器尾数不够，计算运算过程中一般保留低n位，舍弃高位

* + 保留的低n位数不能正确表示运算结果，即有效位超过了n位，发生溢出现象
  + 保留的低n位能正确表达计算结果，即高位的舍去并不影响其运算结果

如何判断一个数是否时规格化数

* + 规格化浮点数的尾数小数点后的第一位一定是个非零数。因此，
  + 对于原码编码的尾数来说，只要看尾数的第一位是否为1就行
  + 对于补码表示的尾数，只要看符号位和尾数最高位是否相反。
  + IEEE754是原码表示的尾数，且1隐藏

对于尾数相同的定点数和浮点数，可表示的浮点数个数和定点数个数应该一样多

* + 可表示的个数取决于编码所采用的位数，编码数一定，则编码出来的数据个数就是一定的。N位编码只能表示2n个数
  + 对于相同位数的浮点数和定点数，可表示的数据个数应该一样多（有时可能由于一个值由两个或多个编码对用，编码个数会有少量差异，如补码和原码在0的表示）

IEEE754四种舍入方式：

* + 就近舍入：舍入为最近可表示的数，若结果值正好落在两个可表示数的中间，一般选择偶数
  + 正向舍入：朝方向舍入，即取右边的那个数
  + 负向舍入：朝方向舍入，即取左边的那个数
  + 截去：朝0方向舍入，即取绝对值较小的那个数

原码加/减法运算可以有以下两种实现方式：

* + 转化为补码后，用补码加减法实现，再转化为原码
  + 直接用原码进行加/减运算，符号和数值部分分开进行

长度为n+1的定点数，按照不同的编码方式，表示的数值范围：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 编码方式 | 最小值编码 | 最小值 | 最大值编码 | 最大值 | 数值范围 |
| 无符号定点整数 |  |  |  |  |  |
| 无符号定点小数 |  |  |  |  |  |
| 原码定点整数 |  |  |  |  |  |
| 原码定点小数 |  |  |  |  |  |
| 补码定点整数 |  |  |  |  |  |
| 补码定点小数 |  |  |  |  |  |
| 反码定点整数 |  |  |  |  |  |
| 反码定点小数 |  |  |  |  |  |
| 移码定点整数 |  |  |  |  |  |
| 移码定点小数 | 小数没有移码定义 | | | | |

设阶码和位数均用补码表示，阶码部分共K+1位（含1位阶符），位数部分共n+1位（含1位数符），则这样的浮点数的表示范围：

|  |  |  |  |
| --- | --- | --- | --- |
| 浮点数 | 浮点表示 | | 真值 |
| 阶码 | 尾数 |
| 最大正数 | 01…1 | 0.11..11 |  |
| 绝对值最大负数 | 01…1 | 1.00..00 |  |
| 最小正数 | 10…0 | 0.00..01 |  |
| 规格化的最小正数 | 10…0 | 0.10..00 |  |
| 绝对值最小负数 | 10…0 | 1.11..11 |  |
| 规格化的绝对值最小负数 | 10…0 | 1.01..11 |  |