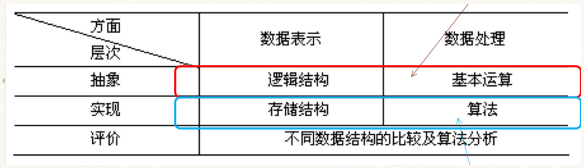
数据结构

- Ch1绪论
  - 1.1 引言
    - 程序=算法+数据结构
      - 数据结构关心数据如何有效的组织
      - 算法关注数据如何用运算解决问题
    - 冯诺伊曼体系结构
      - 输入、存储、运算、控制器、输出
  - 1.2 基本概念和术语
    - 数据
      - 信息的载体，被识别、储存加工处理的对象
    - 数据元素
      - 数据集合的一个个个体、数据的基本单位、由若干个数据项组成
    - 数据项
      - 具有独立含义的单位、是数据不可分割的最小单位
      - 原子项：
        - 例：学号、入学成绩
      - 组合项：
        - 例：出生日期–年、月、日
    - 数据对象:
      - 性质相同的数据元素的集合
      - 可以无限也可以有限
      - 例：整数的数据对象{...–3,–2,–1,0,1,2,3,...}
    - 数据结构：
      - 存在一种或者多种特定关系的数据元素的集合
    - 结构：
      - 数据元素之间的关系、可以看作是抽象的数学模型
      - 与计算机无关、与数据存储无关，也叫做逻辑结构
      - 可以用二元组表示
        - D：数据元素的有穷集合，如d1～dn
        - S：关系的有穷集合，如<d1，d2>...
      - 结构的类型
        - 集合结构
          - 数据元素属于一个集合
        - 线性结构
          - 数据元素一对一
        - 树形结构
          - 数据元素一对多
        - 图形结构
          - 数据元素多对多

- 存储结构
  - 计算机中的映像表示
  - 数据元素的映像
    - 数据元素：结点
    - 数据项：数据域
  - 关系的映像
    - 顺序
      - 数据元素依次放在连续的存储单元
    - 链式
      - 结点中增加若干指针域
- 运算操作（常用）
  - 建立数据结构（加工）
  - 清除数据结构（加工）
  - 插入数据元素（加工）
  - 删除数据元素（加工）
  - 排序（加工）
  - 检索（引用）
  - 更新（加工）
  - 判空/判满（引用）
  - 求长（引用）
  - 抽象面向用户，概念层；实现面向计算机，实现层



- 数据类型
  - 基本类型：int、float、char...
  - 构造类型：数组、结构、指针...
- 抽象数据类型
  - 如队列、栈、树...

1.3 算法和算法分析

- 算法
  - 指令的有限序列
  - 五个特性：
    - 有穷性：有穷时间内有穷步完成
    - 确定性：有确切含义、相同输入相同输出
    - 可行性：可以通过已经实现的基本运算执行有限次来实现
    - 输入：零个或者多个输入
    - 输出：一个或者多个
  - 算法与程序的区别
    - 程序可以无穷，例如不关机一直执行
    - 程序的指令必须机器可执行，算法无所谓
    - 算法代表了问题的解，程序是算法在计算机上特定的实现
    - 算法可以通过伪代码实现，清晰易懂
  - 好的算法具有的标准

    - 正确性：满足需求
    - 可读性：好读，便于理解
    - 健壮性：有容错处理，输入非法数据也会做出反应而不是输出莫名其妙
    - 效率和存储量需求合理
- 算法分析
  - 一般从时间和空间来评价优劣
  - 方法
    - 事后统计
    - 事前分析（有价值
  - 有关因素
    - 算法采用何种策略
    - 问题的规模
    - 程序语言
      - 语言的级别越低、效率越高
    - 机器代码的质量
    - 机器执行指令的速度
  - 时间复杂度T(n)
    - 所有语句执行时间之和
    - 一般k层循环就是O(n*k)的时间复杂度
    - 只看最高次项
    - O(n)就是最优解
    - 常用算法时间排序
      - O(1)<O(logn)<O(n)<O(nlogn)<O(n^2)<O(n^3)
      - O(2^n)<O(n!)<O(n^n)
  - 具体问题1:最大子序列和
    - 改进穷举与联机算法

算法2：穷举法-改进

```
int MaxSubSequenceSum(const int A[], int N){
    int ThisSum, MaxSum, i, j,k;

    MaxSum = 0;
    for(i=0;i < N; i++){  //以i为子序列起点，遍历所有的N个可能
        ThisSum = 0;
        for(j=i;j < N;j++){ //j为子序列终点，遍历所有的i~N个可能
            ThisSum += A[j];       //计算从i到j的子序列和时利用已经计算
            if(ThisSum > MaxSum){  出的从i到j-1的子序列和
                MaxSum = ThisSum;
            }                      例如：A={1， 3， -2， 4， -5}
        }
    }                              计算1、 3、 -2三个元素和时，利用已经计
    return MaxSum;                 算出的1、 3两个元素的和。
}
```

算法4：完美的联机算法

解析：

假设a[i]为负数，则a[i]不可能为此子序列的起始，同理，若a[i]到a[j]的子序列为负，则a[i]到a[j]不可能为子序列的起始，则可以从a[j+1]开始推进。

```
int MaxSubSequenceSum(const int A[ ], int N){
    int ThisSum, MaxSum, j;

    ThisSum = MaxSum = 0;
    for (j= 0; j < N; j++)
    {
        ThisSum += A[j];

        if (ThisSum > MaxSum)
        {
            MaxSum = ThisSum;
        }
        else if(ThisSum < 0){
            ThisSum = 0;
        }
    }

    return MaxSum;
```

  - 存储空间
    - 算法存储空间
      - 输入数据
      - 程序本身
      - 辅助变量

        - 固定部分：程序代码、常量、简单变量、定长结构变量
        - 可变部分：与问题规模有关的存储空间
    - 1.4 C语言补充知识
        - 定义别名：typedef struct{...}book/*bookptr
        - 引用传递：void Exchange(int &x,int %y)
            - 这里的x和y即使退出函数也依旧会保留改变
        - C++命名空间
            - using namespace std;
            - namespace first-space里有function
            - namespace second-space里有function
            - namespace third-space里有function
            - 调用时：first-space::funcyion();即可
        - C++的标准输入输出
            - cin>>x;
            - cout<<"Hello,world!"<<endl;
            - 可以自动检测数据类型
- Ch2线性表
    - 2.1线性表（List）
        - 线性结构：数据元素有序的集合
            - 四个特征
                - 唯一的第一元素
                - 唯一的最后元素
                - 除最后元素都有唯一后继
                - 除第一元素都有唯一前驱
            - 常见的线性结构：
                - 线性表、栈、队列、循环队列、数组、串
        - 线性表：具有相同数据类型的n个数据元素的有限序列
    - 2.2线性表的顺序存储与实现
        - 进行各种操作需要注意：
            - 表空间是否满了
            - 位置是0还是1开始计算
            - 数据的移动方向
            - 表长一定要修改
        - 线性表的声明

```
typedef struct{
    int *elem;
    int length;
    int listsize;
}Sqlist
```

        - 主要操作的思路

        - 顺序表初始化：malloc一个线性表，用L.elem指即可
            - 要考虑如果存储分配失败的情况
        - 按位序取元操作：直接数组访问
        - 插入元素：顺序后移，修改长度
        - 删除元素：顺序前移，修改长度
            - 需要检查删除位置的有效性（是否存在、表是否为空）
        - 元素定位：while循环
    - 2.3 线性表的链式存储与实现
        - 单链表：结点包含数据与指向下一个元素的指针
            - 结点定义

```
typedef struct
{
    int data;
    int *nextptr;
}LNode, *LinkList;
```

            - 建立单链表
                - 尾插法：用r记录尾指针位置
                - 头插法
                - 加入空的头结点
            - 求表长：while循环
            - 查找：while循环
            - 插入操作
                - 后插结点：直接插
                - 前插结点：先用while找到前驱、再插
            - 删除：直接指向下一个结点
                - 记得free
        - 循环链表：尾指针指向头结点
        - 双向链表：一个结点有两个指针，一个指向前驱一个指向后继
    - 2.4 顺序表和链表的比较
        - 顺序存储优点：
            - 方法简单，容易实现
            - 节省空间
            - 可以按照元素序号直接访问
        - 顺序存储缺点：
            - 移动等操作对于n较大的顺序表效率低
            - 不好预估预先分配的存储空间
    - 2.5 单链表的应用：多项式及其运算
        - 通过存储系数、指数的方式，保存下来多项式的信息
        - 通过比较指数、运算系数的方式，完成多项式的运算
- Ch3 栈/队列
    - 3.0 线性表与栈、队列区别

- 线性表允许在任意位置插入删除
- 栈只允许在栈顶进行插入和删除
- 队列只允许在队尾插入、队头删除

- 3.1 栈 （stack）
  - 3.1.1 栈的定义
    - 特殊的线性表，插入和删除操作只能在栈顶进行，后进先出
  - 3.1.2 顺序栈
    - 类型定义

```
typedef struct
{
    int *topptr;
    int *baseptr;
    int stacksize;
}SqStack
```

      - 栈顶指针top指向栈顶元素下一个位置（待装载位置）
        - top-base就是栈中元素数量，top=base就是空栈
    - 具体操作
      - 生成空栈；base=malloc，top=base，设置stacksize
      - 判断是否为空栈：top是否等于base
      - 入栈：*S.top=e,然后S.top++
        - 如果栈满了且需要扩容
          - S.base=(int *)realloc(S.base,(stacksize+n)*sizeof(int));
          - S.top=S.base+stacksize(防止realloc取了新的地方丢失top)
          - stacksize+=n；
      - 出栈：S.top--, e=*S.top
        - 空栈则返回error
      - 取栈顶元素：e=*（S.top-1）指针不动所以不能用S.top--
      - 双头栈
        - top1、top2、base ，base2可以通过base+stacksize取到
  - 3.1.3 链栈
    - 栈顶指针指向队尾元素，和单链表相反
      - 链栈不需要事先分配空间
      - 进行入栈操作不需要判断是否满栈
      - 不需要头结点，因为指针方向从栈顶指向栈底
    - 具体操作
      - 入栈：先创建结点s=malloc(sizeof(LinkStack)),再赋值，最后插入top后面
      - 出栈：top=top->next
        - 一定记得free
- 3.2 栈的应用举例
  - 数制转换（辗转相除）
    - 依次除n，余数压栈

- n=0时依次出栈
- 括弧匹配
  - 左括弧直接入栈，右括弧则进行匹配，正确则出栈左括弧继续，错误则结束
- 表达式求值
  - 中缀表达式求值，如：3*2^(4+2*2-1*3)-5
  - 处理方法
    - 一个运算对象栈s1，一个运算符栈s2
  - 处理过程
    - 如果读取对象，入对象栈
    - 如果读取运算符
      - 比栈顶运算符高阶：入栈继续
      - 比栈顶运算符低阶：对象栈中取出两个对象，算符栈中取出一个运算符运算，结果存入队两战
    - 直到遇到结束符
    - 为了让第一个运算符入栈，会先预设一个“（”
  - 后缀表达式求值，如：32422*+13*-^*5-
    - 对象入栈，遇到运算符则取两个对象运算后入栈
  - 中缀表达式与后缀表达式转换（一个数组一个栈）
    - 运算对象顺序向B数组中存放
    - 遇到运算符
      - 比栈顶高级：入栈
      - 比栈顶低级：栈顶出栈放入数组B
  - 转换手算法
    - 按照优先级为所有运算单位加括号
    - 所有运算符移动到对应括号的后面然后去除括号

3.4 队列

3.4.1 队列的定义

- 先进先出，有队头队尾

3.4.2 队列的存储以及运算实现

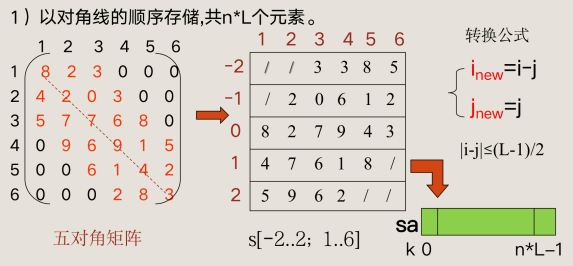
- 1. 顺序队列
  - 需要两个指针front和rear
  - 队尾指针指向队列尾元素的下一个位置
  - 随着出队入队的进行，会出现假溢出
- 2. 循环队列
  - 将队列的数据区看成头尾相接的循环结构
  - 循环的实现：
    - 队尾插入一个元素：rear=（rear+1）%n
    - 队头删除一个元素：front=（front+1）%n
    - 判断队满：（rear+1）%n=front
    - 判断队空：rear=front
    - front和rear并不是指针，只是记录位置

    - 存储结构：
        - typedef struct
        - {
            - int *base
            - int front
            - int rear
        - }SeQueue
    - 具体操作：
        - 初始化：申请地址，front=rear=0
        - 判队空：front=rear
        - 判队满：front−1=rear
        - 求队长：（rear−front+n）%n
        - 入队：Q.base[Q.rear]=e
        - 出队：Q.front=(Q.front+1)%n
- 链队
    - 一个头指针一个尾指针
    - 类型定义
        - typedef struct
        - {
            - Qnode *front
            - Qnode *rear
        - }headnode
        - typedef struct
        - {
            - int data
            - int *next
        - }Qnode
        - 头尾指针一开始都指向一个空结点
    - 具体操作：
        - 初始化链队：创建空头结点，被指，指空
        - 入链队：插入，重制尾结点
        - 出链队：如果出最后一个则直接rear=front，记得free
    - 队列应用
        - 计算出迷宫最短路径：
            - 深度优先（优先对最近结点探索到底）——栈
            - 广度优先（优先把结点四周搞清楚再外扩）——队列

- Ch5 数组和广义表
    - 5.1 数组和线性表的关系以及数组的运算
        - 任何一个数组A都可以看成一个线性表A=(a1,a2,...an)
            - 对于二维数组m*n时，每一个元素是一个一维数组
            - 对于三维数组时，每一个元素是一个二维数组

        - 对于n维数组时，每一个元素是一个（n-1）维数组
    - 数组与线性表的关系
        - 线性表的拓展，数据元素本身也是线性表
        - 数组的特点
            - 每个元素类型同意
            - d维非边界元素有d个直接前驱和d个直接后继
            - 维数确定后，数据元素个数和元素之间的关系不再发生改变，适合顺序存储
            - 每组有意义的下表都有一个对应的值
        - 数组的基本操作
            - 给一组下标，取/改相应的数据元素值
        - 具体操作
            - 构造n维数组
            - 销毁数组
            - 取指定下标数组元素值
            - 改指定下标数组元素值
- 5.2 数组的顺序存储结构
    - 一维数组: a[n]
        - 基本信息：
            - 基地址LOC[0]=b
            - 步长（一个元素所占的单元）：L
        - LOC[i]=LOC[0]+i*L=c0+c1*i
            - c0=b=LOC[0]
            - c1=L
    - 二维数组: a[b1][b2]
        - 步长信息：
            - c2=L（第二维的步长为一个元素所占单元）
            - c1=n*L=b2*c2（第一维步长为一整行元素所占单元）
            - c0=b=LOC[0,0]
        - LOc[i,j]=c0+c1*i+c2*j
    - n维数组: a[b1][b2]...[bn]
        - 步长信息：
            - cn=L(第n维的步长为一个元素所占单元)
            - ci-1=cn*bn（第i-1维步长为上一维全部填满所占单元）
            - c0=b=LOC[0,0,...,0]
    - 遇到求数组访问的某个元素地址
        - 求出各维的维数b1～bn，若有非零开始的一律转化成零开始计算
        - cn会由题目给出，由cn和b1～bn求出c1～cn-1
        - 由LOC[x1,x2,...,xn]=b+c1j1+c2j2+...+cnjn得到答案
- 5.3 矩阵的压缩存储
    - 目的是节省空间
    - 5.3.1 对称矩阵：主对角线右上与左下对应元素相等

- 只存上三角（包括主对角线）的n(n+1)/2个数据元素
  - 从sa[0]开始存
- 5.3.2 三角矩阵：对角线以下的数据元素全部为c
  - 一共n(n+1)/2+1个数据元素
    - sa[0]用来存c，其余按序往后存
- 5.3.3 带状矩阵：非零元素沿着对角线呈等宽带状分布
  - 存储方法
    - 1.按照对角线顺序存储

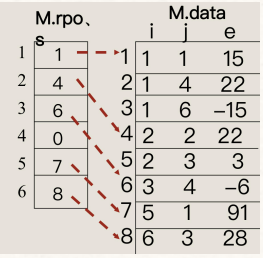


    - 2.只存储带状区域

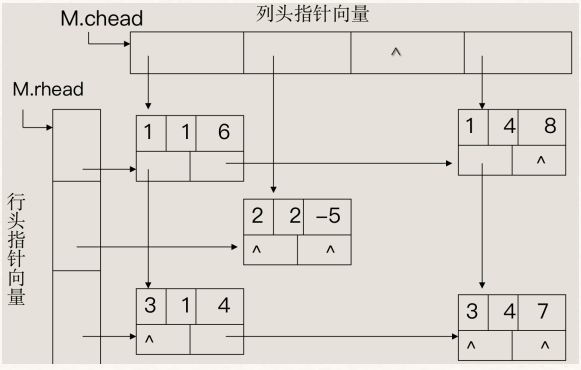
- 5.3.4.1 随机稀疏矩阵
  - 特点：大多数元素为零（稀疏因子=t/（m*n）<=0.05）
  - 顺序存储：三元组表
    - 类型定义
      - typedef struct
      - {
        - int i
        - int j
        - int e
      - }Triple
      - typedef
      - {
        - Triple data[MAX+1]
        - int rn
        - int cn
        - int num

        - }TSMatrix
        - data[0]不用，从data[1]开始存
    - 实际应用
        - 求转置矩阵（仍然行优先）
            - 找列为1的元素
            - j=i, i=j, e=e
- 5.3.4.2 行逻辑链接的顺序表
    - 再加一个指针数组来标记不同行的起点
        - 可以便于某些运算如稀疏矩阵相乘

```
typedef  struct  {
    Triple  data[MAXSIZE+1];
    int       rpos[MAXRC+1];
    int       mu, nu, tu;
}RLSMatrix;
RLSMatrix  M;
```



- 5.3.4.3 十字链表
    - 在行列方向上把元素链接在一起
    - 类型定义
        - typedef struct
        - {
            - int i,j,e
            - OLnode *down
            - OLnode *right
        - }OLnode
        - typedef struct
        - {
            - OLnode **rhead(需要申请内存来形成指针数组
            - OLnode **chead
            - int mu,nu,tu
        - }CrossList



    - 操作流程

        - 赋值初始化mu，nu，tu
        - 申请指针数组
        - 读入非零元素信息
            - 建立结点复制三元组
            - 寻找行表中合适位置插入
            - 寻找列表中合适位置插入
            - 存下一个元素，无则结束
- 5.4 广义表的定义
    - 概念：由多个原子或者子表组成的有限序列
        - 原子：逻辑不可再分
        - 子表：广义表元素中的广义表
    - 与线性表的关系：线性表的推广
    - 表示方法：
        - 大写字母表示广义表名称，小写字母表示原子
            - 如：A=(),B=(a,A)=(a,())
    - 相关术语

	表长	表深	表头	表尾
A=( )	0	1	–	–
B=(a, A)=(a, ( ))	2	2	a	(( ))
C=((a,b), c, d)	3	2	(a,b)	(c,d)
D=(a, D)=(a, (a, (a, ...))	2	∞	a	(D)

        - 表长：表的第一层的元素个数
        - 表深：最深嵌套数（括号对的数量）
        - 表头：表中第一个元素
        - 表尾：除了表头外，剩下元素再封装形成的新的广义表
    - 常用操作
        - 获取表头GetHead（L）
        - 获取表尾GetTail（L）
    - 广义表的分类
        - 递归表：允许递归生成
        - 要求结点共享的递归表：再入表
        - 必须是树形结构的再入表：纯表
        - 必须是一叉树的纯表：线性表
    - 广义表的应用
        - m元多项式的表示

$$P(x, y, z) = x^{10}y^3z^2 + 2x^6y^3z^2 + 3x^5y^2z + 2yz + 15$$
$$= (x^{10}y^3 + 2x^6y^3)z^2 + (3x^5y^2 + 2y)z + 15$$

P = z ((A,2), (B,1), (15, 0))
A = y((C, 3))
C = x ((1, 10), (2, 6))
B = y((D, 2), (2, 1))
D = x ((3, 5))

        - 先找到第一主元（z）并整理多项式

        - 根据不同次数存成不同元素，每个元素前键存系数、后键存次数
            - 若系数仍为多项式，则存成下一级广义表
        - 然后继续寻找主元以此类推直到所有变元均被当过主元结束
- 5.5 广义表的存储结构
    - 5.5.1 头尾链表
        - 分不同的结点类型
        - 类型定义

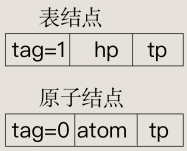
```
typedef enum {ATOM, LIST} ElemTag;
// ATOM==0:原子；LIST==1：子表
typedef struct GLNode{
        ElemTag  tag;
        union {
            AtomType  atom;
            struct  {
               struct GLNode  *hp, *tp;
            }ptr;
        }
} * GList1;
```

hp：指示表头的指针

tp:指示表尾的指针

        - 类型是一，右侧指针指表尾，表尾的类型一定也是一，下侧指针指表头，类型不定
    - 5.5.2 扩展的线性链表形式

```
typedef enum {ATOM, LIST} ElemTag;
// ATOM==0:原子；LIST==1：子表
typedef struct GLNode{
        ElemTag  tag;
        union {
            AtomType  atom;
            struct  {
                struct GLNode  *hp;
            }
        }
        struct GLNode  *tp;
} * GList2;
```



hp：指向表头的指针

tp：指向同一层的下一个结点

        - 任何类型都有指向同层指针，子表拥有向下层指的指针
- 5.6 广义表的递归算法
    - 计算广义表的深度
        - 方法一分析各个元素
            - 求深度的递归函数
                - 基本项：depth（ls）=1/0 当ls为空表/原子
                - 归纳项：depth（ls）=1+max{depth(ai)}
            - 注意：最后要+1，元素最深的深度+1才是广义表的深度
        - 方法二分析表头和表尾
            - 求深度的递归函数
                - depth（ls）=
                    - 1:空表
                    - 0:原子
                    - max{depth(head(ls))+1,depth(tail(ls))}
                        - 表尾白嫖了一层深度，所以剥夺+1资格
    - 复制广义表

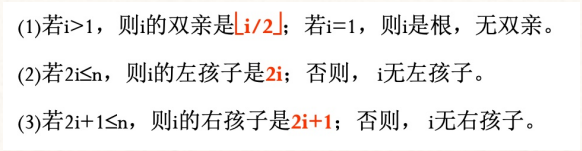
- 递归定义
  - 基本项：
    - 置空：ls为空表
    - 复制单原子结点：ls为原子
  - 递归项：
    - 建表结点
    - 复制表头
    - 复制表尾

- Ch6 树结构
  - 6.1 引言
    - 6.1.1 问题提出
      - 需要处理一个前驱元素多个后继元素的情况
    - 6.1.2 树的定义
      - 以分支关系定义的层次结构
      - 树是由n个结点组成的有限集合T
      - 非空树满足：
        - 有一个根结点，该结点没有前驱结点
        - 除了根以外其余结点被分成了m个互不相交的集合，其中每一个集合本身也是一棵树，称为根的子树
      - 树的递归定义：一颗非空树由若干颗子树构成
      - 特点：
        - 根结点没有前驱结点，除了根结点之外所有结点有且只有一个前驱结点
        - 所有结点可以有0或者多个后继结点
      - 基本术语：
        - 结点的度：结点拥有子树数量
        - 树的度：最大结点度
        - 叶子结点：度为0的结点
        - 分支结点：不是叶子结点的结点
        - 内部结点：不是根结点的分支结点
        - 父与子结点：有直接上下级关系
        - 兄弟：同一双亲的孩子
        - 结点的祖先：根到该结点分支上所有的结点
        - 结点的子孙：该结点为根的子树中的所有结点
        - 结点的层次：结点在树中的相对位置，根为第一层，每个父子关系层数加一
        - 树的高度：最大结点层次
        - 有序树：左右不能互换，否则为无序树
        - 路径长度：从Ni能通过树中结点到Nj则称两者间存在路径，长度为边的个数
        - 森林：多棵互不相交的树的集合
  - 6.2 二叉树
    - 6.2.1 二叉树的概念
      - 二叉树的定义

- 由左子树右子树的二叉树组成
- 二叉树的特点
  - 定义是递归的
  - 结点的度
  - 是有序树
    - 即使只有一棵树也要区分左子树和右子树
- 满二叉树
  - 每一层的结点个数达到了最大
  - 每个结点都存在左子树和右子树
  - 所有叶子都在最下面一层
- 完全二叉树
  - 满二叉树差最低下右往左几片叶子

6.2.2 二叉树的性质

1. 二叉树第i层至多有2^i−1个结点
2. 深度为k的二叉树至多有2^k−1个结点
3. 对于任何一棵二叉树，如果叶子结点数为n0，度为2的结点数为n2
   - 则n0=n2+1
4. 具有n个结点的完全二叉树的深度为【log2n】
5. 一棵具有n个结点的完全二叉树对其结点按照从上到下从左到右进行编号，则对于任意结点



- 二叉树的存储结构
  - 顺序存储
    - 用连续存储单元，从上到下从左到右存储
    - 为了解决无法表示左右子树关系的问题需要添加不存在的空结点使其变成完全二叉树的形式再进行存储
      - 可能会造成大量空间的浪费
    - 还可以存储左右孩子的序号或者存储父结点的序号
  - 链式存储
    - 每个结点除了数据域还有两个指针域指向左孩子和右孩子
    - 根结点存放在头结点的左孩子
  - 三叉链式存储
    - 除了前三个指针域再多一个指针来指向双亲结点，从而方便找到双亲
      - 由lchild,data,rchild,parent构成

6.3 二叉树的遍历

- 需要按照某条搜索路径访问树中的每一个结点
  - 使得每个结点都被访问一次且仅被访问一次
- 分类方法：根据二叉树的基本组成单元根结点、左子树、右子树

- 先序遍历（根左右）
  - 先访问根结点
  - 先序遍历左子树
  - 先序遍历右子树
- 中序遍历（左根右）
  - 中序遍历左子树
  - 访问根结点
  - 中序遍历右子树
- 后序遍历（左右根）
  - 后序遍历左子树
  - 后序遍历右子树
  - 访问根结点

▸ 非递归方法实现二叉树的三种遍历

- 包络线
  - 由左到右绕一圈（空结点也要画个尖尖）
  - 则每遇到结点就访问——先序遍历
  - 从左子树返回就访问——中序遍历
  - 从右子树返回就访问——后序遍历
- 栈的使用
  - 思路
    - p是否为空
      - 不为空则p入栈然后p指向左子树
      - 为空则出栈一个地址给p然后p指向右子树
    - 若入栈前访问该结点为先序遍历，出站后访问为中序遍历
- 队列方法实现二叉树的层次遍历
  - 即从上到下从左到右的遍历
  - 实现方法
    - 根结点入队
    - 从队头取出结点访问，如果该结点左右孩子非空则顺序入队
    - 重复上面过程直到队列结束

○ 6.4 二叉树遍历的应用

▸ 6.4.1 构造二叉树的二叉链表存储

- 先序输入序列
  - 读取ch
  - 如果ch=0则制空
  - ch不为零则新建结点
  - ch等于data
  - 先序构造左子树
  - 先序构造右子树

▸ 6.4.2 在二叉树中查找值为x的数据元素

- 如果data就是，返回

- 如果左子树存在，查找左子树
- 如果右子树存在，查找右子树

- 6.4.3 统计给定二叉树中叶子结点的数目
  - 如果是叶子结点，返回1
  - 如果是空结点，返回0
  - 其余返回左子树叶子结点数目+右子树叶子结点数目
- 6.4.4 由遍历序列恢复二叉树
  - 定义
    - 先序序列和中序序列可唯一确定该二叉树
    - 中序序列和后序序列可唯一确定该二叉树
    - 先序序列和后序序列不可唯一确定该二叉树
  - 操作过程（先序序列+中序序列）
    - 先序序列的第一个元素建立根结点
    - 中序序列中找到该元素
    - 再配合先序序列确定左右子树的根结点
    - 递归调用分别恢复左子树、右子树

- 6.5 线索二叉树
  - 直接记录每个结点的直接前驱和直接后继
  - 分类
    - 先序线索二叉树
    - 中序线索二叉树
    - 后序线索二叉树
  - 方法一
    - 直接再加两个指针域fwd和bkwd指向前驱和后继
      - 这些指针被称为“线索”
    - 问题：储存效率低
  - 方法二
    - 2n个指针域中有n−1个储存孩子信息，n+1个空的，可以拿他们做线索
      - 空的左指针域指向直接前驱
      - 空的右指针域指向直接后继
      - 增设ltag和rtag
        - tag为0则该指针存储的是左/右孩子
        - tag为1则该指针存储的是前驱/后继
      - 结点结构为ltag,lchild,data,rchild,rtag
  - 线索二叉树的具体构建
    - 中序线索化步骤
      - 如果当前二叉树非空
      - 左子树线索化
      - 访问当前结点p
        - 设置ltag和rtag
        - 建立p的前驱线索pre（前驱：上一个）

        - 建立pre的后驱线索
        - p=pre
    - 右子树线索化
    - 注意
        - 要先构造头结点
            - ltag是0，rtag是1
            - 初始化右指针指回自己
            - 左指针指根结点，头结点设为最初的pre（头结点上一个）
        - 线索化T
        - 最后一个pre的rtag为1，后续指向头结点
        - 头结点右指针指向最后一个结点
    - 算法描述
        - p存在
        - 左子树线索化
        - p没有左孩子则存储前驱pre
        - pre没有右孩子则存储后继p
        - pre=p调整位置
        - 右子树线索化
- 查找指定次序下的前驱/后继
    - 中序线索二叉树
        - p的前驱
            - ltag=1，则lchild就是前驱
            - ltag=0，则是左子树的右下角
        - p的后继
            - rtag=1，则rchild就是后继
            - rtag=0，则是右子树的左下角
    - 后序线索二叉树
        - p的前驱
            - ltag=1，则lchild就是前驱
            - ltag=0，有右子树就是右子树根，没右子树就是左子树根
        - p的后继
            - 很麻烦 垃圾
    - 先序线索二叉树
        - 与后序本质一样
- 中序线索二叉树遍历算法步骤
    - 寻找中序序列第一个结点（一路向左）
    - 访问
    - 寻找后继，若存在则重复上一步，不存在则结束
        - 具体实现：如果右指针是线索，循环访问遍历右孩子，如果不是线索，则指向右孩子，等待下一轮一路向左

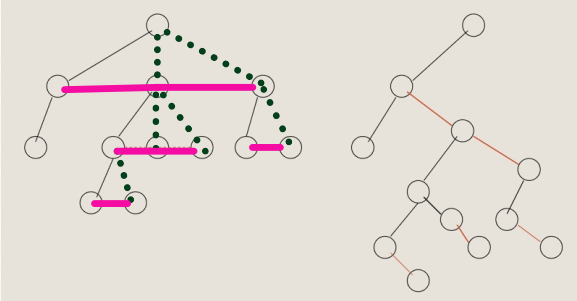
- 6.6 最优二叉树

- 输入值分布不均，为了让操作时间最短而调整二叉树的结构
- 术语
  - 结点权值：和叶子结点对应的某种有意义的实数
  - 结点带权路径长度：路径与权值之积
  - 树的路径长度：树根到每个叶子结点的路径长度之和
  - 树的带权路径长度：书中所有叶子结点带权路径长度之和
  - 最优二叉树（哈夫曼树）：带权路径长度最小的二叉树
    - 权值越大的叶子距离根越近
- 建立最优二叉树
  - 基本思想
    - 构造权值从小到大排列的森林
    - 两棵权值最小的二叉树有序合并成新的二叉树加入森林
      - 新二叉树权值为两棵树权值之和
    - 重复上面操作直至只剩一棵树
  - 存储结构
    - 构造2n−1个结点顺序存储
    - 流程
      - 申请空间并初始化（weight=lchild=rchild=parent=0）
      - 设置1～n的weight
      - 在parent=0度结点中选取两个最小的结点修改parent为i
      - 修改i的weight、rchild、lchild
      - 重复，直到空间填满
- 最优二叉树的应用
  - 最佳判定树
  - 哈夫曼编码，用于传输二进制代码且让电文编码长度最短
    - 以出现系数为权值，字符集合为叶子结点构建
  - 性质
    - 不等长编码
    - 前缀编码
    - 没有度为1的结点，结点总数为2n−1
    - 发送过程：根据编码送出字符数据
    - 接受过程：按照左零右一规则从根结点走到叶子结点完成单个字符译码
  - 操作
    - 分配HC空间
    - 分配临时空间cd
    - 置最后一位为"\0"（倒着编码）
    - 依次根据HT向上爬到根结点，图左左孩子编0，右孩子编1
    - 为第i个字符申请n−start的字符空间，把cd复制进去
    - 释放cd

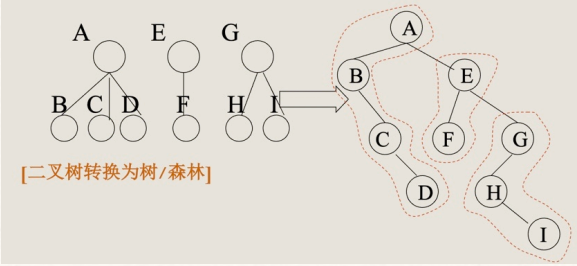
6.7 树和森林

- m棵互不相交的树的集合

- 6.7.1 树的存储
  - 双亲存储
    - 每个结点除了data还存储双亲结点序号
    - 求孩子结点很复杂
  - 孩子存储（指向孩子的链表）
    - 多重链表表示法
      - 指针域个数为该结点度数
        - 操作困难
      - 树的度数
        - 浪费空间
    - 孩子链表表示法
      - 每个结点存储数据以及该结点孩子链表的头指针（跟一串孩子）
  - 双亲指针加孩子链表存储法
  - 孩子兄弟链表存储法
  - 每一个结点有两个指针，第一个指向孩子结点，第二个指向下一个兄弟结点
- 6.7.2 树的转换
  - 树与二叉树转换

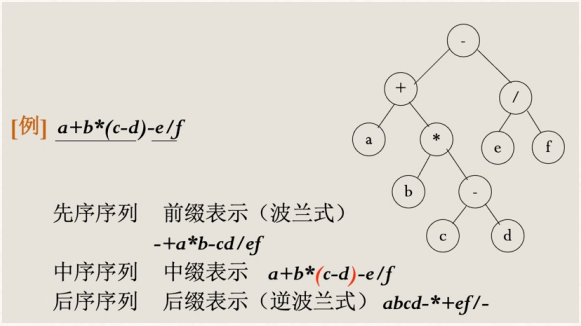


    - 所有兄弟之间连线
    - 双亲只保留与长子的连线
    - 顺时针旋转45度（？谨防形式主义
  - 森林转换为二叉树



    - 每一颗树先转换为二叉树
    - 把所有树的根看作兄弟相连
      - 若长子并没有双亲结点，说明是森林的树根们
  - 二叉树转换为树、森林

    - 所有兄弟都与长子所连的双亲结点相连（如果有
    - 清除兄弟结点连线
- 6.7.3 树的遍历
  - 先序遍历
    - 访问树的结点
    - 依次先序遍历每颗子树
    - 对应的二叉树先序
  - 后序遍历
    - 依次后序遍历每颗子树
    - 访问树的结点
    - 对应的二叉树中序
- 6.7.4 森林的遍历
  - 先序遍历
    - 访问第一棵树的根结点
    - 先序遍历第一棵树的根的子树森林
    - 先序遍历除第一棵树之外剩余的树构成的森林
    - 对应的二叉树先序
  - 后序遍历
    - 中序遍历第一棵树的根的子树森林
    - 访问第一棵树的根结点
    - 中序遍历除第一棵树之外剩余的树构成的森林
    - 对应的二叉树中序
- 6.7.5 二叉树和树的应用实例
  - 二叉树表示的表达式
    - 表达式=【操作数1】【运算符】【操作数2】



    - 回溯求解问题
      - 定义一个解空间包含问题的解
      - 用适合搜索的方式组织该空间
      - 在约束条件下先序遍历搜索，遍历过程中减去不满足条件的分支
    - 应用实例——四皇后问题
      - 累了 开摆！

[回溯法的一般形式]

```
void  trial(ElemType r, …)
{    if  (当前所得解r为所求)   printf( r );
     else  for (i=1; i<=n; ++i)   {
                将当前解r在第i维修改;
                if (修改后的r合法)  then trial(r);
                将r恢复为未修改;
           }
}//trial
```