数据结构与算法(C++)

RenKai

1 class 1

1.1 后面的树和图里面是最后考试写程序的重点 把书上的习题练习完

最基础的东西要好好学

评分细则

- 课程网站 数据结构与算法分析 (scu.edu.cn)
- 考勤：抽查
- 平时作业（每一章都有 16周统一拷给老师） + 上机
- 考试：基本概念，算法设计，应用题(选择 + 问答设计算法 + 主观题) <重点全在课上讲>
- 成绩评定：平时成绩（50%） + 期末考试（50%）<有可能有期中考试 >

课程的主要内容

- 各种数据结构（线性结构、树、图）的概念、特点、存储、基本算法（重点）
    - 理清楚数据结构学习的思路
- 常用的排序、查找等各种算法的设计方法 *（重点）
- 文件的基本概念和各种结构（略）
- 程序性能分析：空间复杂性、时间复杂性
    - 有些可以直接记住
- 算法设计基础与算法分析

2 Chapter 1 绪论

2.1 1.1 数据结构的概念和学习数据结构的必要性

数值计算用计算数学范畴的数学模型来做

非数值计算问题（目前可以理解为离散问题）就要设计数据结构来建立数学模型

- 表
- 数
- 图

学习的最根本目的只有一个：对于一定的问题使用合适的数据结构，不要把问题搞复杂

2.2 1.2 数据结构的基本概念

数据 -> 数据元素 (由数据项组成) -> 数据结构（表示数据元素之间关系的集合）

数据（data）：所有能够加工处理对象的总称

数据对象（data object）：数据的一个子集

数据项（data item）：独立的不可以再分割的单位 （思维：先建立起来 再延伸）

数据元素（data element）：可以看成table里的一行 或者一个 结点

数据结构（data structure）：按一定逻辑关系组织起来的一批数据 || 按一定的存储表示方法存储起来（****） || 定义一个运算的集合（****） 重点就在于如何存储 如何表达。

<题目：1. 数据结构在计算机中的表示称为数据的（**存储结构**）>

四种基本数据结构

- 集合结构
- 线性结构
- 树状结构
- 图状结构

数据结构的表示

DataStructure = (D, S) D 表示数据元素集合， S 是定义在D中的数据元素**之间关系**的有限集合

总结：我们就学习数据结构以下三点：

- 数据项之间有什么**逻辑关系**
- 数据项怎么去存储（逻辑**存储结构**即**物理存储**结构）
- 数据项之间如何实现运算

在此做一个初步的交代(重点关注前两点)

- **顺序映像** （顺序存储结构）

顺序存储结构：借助相对位子来表示数据元素之间的逻辑关系 存储在一串连续的空间里

特点：只存储数据的值，不存去结点之间的关系。所以存储密度很大；可以直接计算第 i 个结点
的存储地址；插入和删除会引起大量结点的变化 <很麻烦>

- **非顺序映像**（非顺序存储结构）（反复强调）

  例如链式存储结构：借助指针来表示数据元素之间的逻辑关系，**可以存储在不连续的存储单元中**，但元素之间的关系
  可以通过地址确认。可能要操作许多个 points <其中可能不仅存储结点中的值 还要存储结点之间 的链接关系>

  特点：存储结构的存储密度小，存储空间利用率低<计算空间利用率只需要看一个结点数据域和指
  针域的对比>；可用性高，可以用于表、树、图等多种逻辑结构的存储；删除和插入操作灵 活方便，不必移动结点，只要改变结点中的指针值即可

- 索引存储结构：查字典
- 散列存储（每次都考）(Hash表)：根据结点的**关键字**直接计算出结点的存储地址。

  把关键字作为自变量 然后放到相应的函数中从而算出存储地址

  关键问题：如何找到这个函数；如何解决两个x同一个f(x)怎么办？

2.3 1.3 抽象数据类型及其实现

数据类型（`data type`）：相同性质的计算机<数据>的集合，以及定义在这个集合上的一组<操作>的总称。

抽象数据类型（`Abstract data type, ADT`） 这个概念和模板很类似，就是抽象出的能表示数据模型主体、以及相关操作的，能够随时适应各种具体变化的抽象概念。以后我们设计所有的数据结构的时候，都没有说是只针对某一个数据类型而做，而是说这个结构对于所有的数据类型都可以跑通。

2.4 1.4 算法和算法分析

概念：

- 算法：对特定问题求解步骤的一种描述，是指令的有效序列，其中一条指令表示一或多种操作

衡量方法：

- 时间复杂度 大概都是基本的函数 n n方 log(n) 等
- 空间复杂度

根据算式计算pi的算法

往后几章都会是这种学习方式

`step 1` 先学习一下该数据结构的逻辑结构（数学表达）

`step 2` 某种数据结构的顺序映像

`step 3` 某种数据结构的非顺序映像（链式）

3 Chapter 2 线性表

3.1 2.1 线性表的逻辑结构（数学表达）

线性表是由**类型相同的数据元素**组成的有限序列，不同线性表的数据元素类型才可以不同。当然像表格这种复杂的线性表可以把一行当作一个数据元素（由多个相同类型的数据项构成的一个相同类的数据元素）

> 考点：判断哪些序列不是线性表

3.2 2.2 线性表的顺序存储结构

本质为一个长度为maxSize 数据类型为ElemType 的数组 然后外加很多操作 其实本质上就相当于一个功能更加强大的数组罢了，但是很麻烦。

3.3 2.3 线性表的链式存储结构 <这个可能会考手写算法>

好处：不要求逻辑上相邻的数据元素在物理位置上也相邻。只要是能根据逻辑地址找到另外一个就ok了，这样可以避免大量的数据位置移动。

3.3.1 2.3.1 单链表（不管是哪种运算都要注意指针指向的顺序）（是后续所有的基础）

概念：构成单链表的不再是一个又一个普普通通的数据类型了，而是既有 data 又有 逻辑指向指针的一个 node 这是一个链表的数据元素，数据项为data和next。

线性表的链式存储结构有如下特点：

1、数据元素之间的<逻辑关系>由结点中的指针域表示；

2、每个元素的<存储位置>由其直接前驱的<指针域>所表示；

3、线性表的链式存储结构是非随机存取的存储结构；

4、线性表的链式存储结构中的尾结点的直接后记为NULL

实现：很简单 较比于顺序存储结构只不过基础核心变了

- 先构建一个Node结构体作为后续链表操作的数据元素

动态建立单链表的常用方法有如下两种

- 头插法
- 尾插法 （这样才能保证顺序输入顺序输出） 但是如何锁定尾部的位置呢？ 加一个标识指针r 始终指向尾部<我们的代码实现中用的是尾插法>

插入运算 <一定要记准确>

- step 1: 插入位置 i 找到 i - 1 位置的结点 pre
- step 2: newNode->next = pre -> next; pre->next = newNode;

删除运算

- step 1: 删除位置 i 先找到 i - 1 位置的结点 pre
- step 2: targetNode = pre -> next; pre -> next = targetNode-> next; delete targetNode; 一定要记得释放结点

3.3.2 2.3.2 循环链表

单循环链表是单链表的变形。

只有一个不同点：循环链表最后一个结点不为NULL，而是指向了表的前端，一般来说都要加一个头结点。即表示一个列表走完不再是 node->next = NULL 而是 node -> next = head;

- 之前的只有一个头节点 也只能对头或者尾进行操作
- 但是循环链表只需要有一个尾结点 rear rear->next 就是头节点 就都好找了（优化）

  有些选择题（下面哪一种情况最好啊？）一定要照顾每种情况的实现的要求

用循环链表求解**约瑟夫环**问题

约瑟夫环的普通解法在笔记中存在

3.3.3 双向链表

这个的实现和以上是差不多的 没有必要再次实现一遍但有必要将重点罗列于此：

1、Node这个结构体要改变，变成有back指针和next指针的一个node

同时：配套的拷贝构造以及 =重构也要改变

2、Insert

先构建三个Node

```
Node<ElemType> *preNode, *newNode, *nextNode;
preNode = GetElemPtr(position - 1); // Postion - 1很重要
nextNode = preNode -> next;
newNode = new Node<ElemType>(elem, backLink, nextLink);
nextNode->back = newNode;
preNode->next = newNode;
```

3、Delete

```
Node<ElemType> *tempNode; // 找的就是要删掉的结点
preNode = GetElemPtr(position);
tempNode->back->next = tempNode->next;
tempNode->next->back = tempNode->back;
e = tempNode->data;
delete tempNode;
```

3.3.4 2.3.3 顺序表和链表的比较（选择题）

基于时间的考虑

基于结构的考虑

3.3.5 2.3.4 在链表结构中保存当前位置和元素个数

这一部分的目的很简单：优化链表

方法：在链表种加入当前位置和元素个数 即在private中添加元素

- curPosition // 当前位置的序号
- curPtr // 当前位置的指针
- count // 当前的元素个数

分成三种情况来讨论， 简单线性表 循环链表 双向链表

- init()

  三个都一样

```
curPostion = 0;
curPtr = head;
count = 0;
```

- GetElemPter(int position)

```
if(curPosition > position) //如果说当前的位置超过了要查找的位置那只能从
```

头再查

```
        curPostion = 0; curPtr = head;
    for(; curPosition<position; curPostion++)//挨个找下去
        curPtr = curPtr -> next;

    // 注意双向链表就不一样了奥
    if(curPostion > postion)
    {
        for(; curPosition > position; curPosition--)
        {
            curPtr = curPtr->back;
        }
    }
    else if(curPosition < position)
    {
        for(; curPosition < position; curPosition++)
        {
            curPtr = curPtr->next;
        }
    }
- Length() // 直接 return count
    return count;

- Delete()
    // 只需要注意以下一个点 就是删除的为尾结点的时候
    if(position == Length()){
        curPosition = 0;
        curPtr = head;
    }
    else{
        curPosition = position;
        curPtr = tmpPtr->next;//(tmpPtr = GetPtr(position-1));
    }
    // 最后别忘了
    count--;

- Insert()
    curPosition = position;
    curPtr = newPtr;
    count++;
```

3.4 2.4 实例研究——一元多项式 <完完全全按书上写一遍>

3.4.1 多项式相加减相乘的运算规则

这个是基于改良后的 LinkList 实现的

其实多项式的实质就是：一个数据类型为“项”的链表；项中含有指数和系数

我们再把多项式封装到一个类中，类里面包含各种对多项式的操作。

以上就是整体实现的基础。

只再写一写重点部分的算法：

- 如何实现多项式的加？（要实现指数从低到高排列）

step1 比较要加的两个多项式相应项的次数大小

```
    if aItem.expn < bItem.expn
        then cItem = aItem and  ++aPos
    if bItem.expn < aItem.expn
        then cItme = bItem and  ++bPos
    else cItme.expn = aItem.expn;
            and cItem.coef = aItem.coef + bItem.coef
                and ++aPos ++bPos
```

step 2 将剩余的补充在后面

- 只要会实现了加那么必定就会实现减

只需要让每一项的系数变为原来的相反数即可

- 多项式的乘

实现的思想很简单

无非就是说让 a中的每一项和 b中的每一项相乘然后相加

因为加法已经写好了 所以完全不用再担心

4 Chpater 3 栈和队列（期末考试非重点 但是基础题型要了解 在基础题里面反复出现）

基础概念：栈和队列都是线性表，基本操作都是一些子表，但是栈和队列只不过是对数据的存储读取有一定的限制，所以能够满足一定的特殊需要。也就是说一个单链表完完全全可以做，只不过需要很多的注意事项。

4.1 3.1 栈 (Stack)

4.1.1 3.1.1 栈的基本概念

- 只能在一端操作的列表
- LIFO(后进先出) 有他的好处（看到栈就去想弹夹）（**重点**）

  题目：下面哪一项出入顺序是正确的？ 经常变化 1 2 3 4 5进去一定是 5 4 3 2 1出去吗？ 一定要记清操作的方法
- ADT Stack 转变成具体实现形式

4.1.2 3.1.2 顺序栈

本质也就是 insert 和 delete 受限的顺序单链表

- 多栈共享空间：多个栈共享一个或多个数组
- 主要是两个栈共用一个数组，一个用前部分，一个用后部分（两个栈顶相对）
- 最常用的是两个栈的共享技术，主要利用了栈“栈底位置不变 栈顶位置动态变化” 直到两个栈顶相遇时才发生溢出。

4.1.3 3.1.3 链式栈

本质也就是 insert 和 delete 受限的链式单链表

本质就是受限的单链表

例子

- 括号匹配检验（实验课的题目）

  最终要的是运用栈的思想，之前使用数组逢对儿就消零的思想和这里逢对儿就出栈的思想是一模一样的。
- 迷宫求解

  就用简单的遍历算法 重点在于有的地方走不通 所以要学会退步（怎么存这个步呢？就用栈呀 多合适）

4.2 3.2 队列 要仔细看一看

4.2.1 3.2.1 队列的基本概念

定义：队列是只允许在**一端删除**，在**另一端插入**的线性表（***）。允许插入的一段叫做队尾，允许删除的一端叫做队头

特性：FIFO（先进先出）。这是栈和队列的基本区别 就是八个字（研究生面试一定要抓住重点 问什么答什么 不要冗杂）

特殊队列（考的很少但是要注意读题 根据题中的要求来做）

- 双端队列，两端都可以实现插入或者删除

  输入受限的双端队列

  输出受限的双端队列

4.2.2　3.2.2 链式存储结构

实际操作（增删改查）

链式队列具有很强大的优越性，适合于数据元素变动比较大的情形，一般不会存在溢出的问题，如果程序中有多个队列，最好使用链队列，这样就不会存在存储分配问题，也没必要进行数据元素的移动。

删除结点的程序

- 删除时由 1 到 0 是什么情况

4.2.3　3.2.3 顺序存储结构（循环队列）

基本概念：

如果用顺序存储结构，实际上利用一个一维数组来存储元素，基本结构和LinkQueue没什么大的差别，多了一个maxSize

假溢出现象：

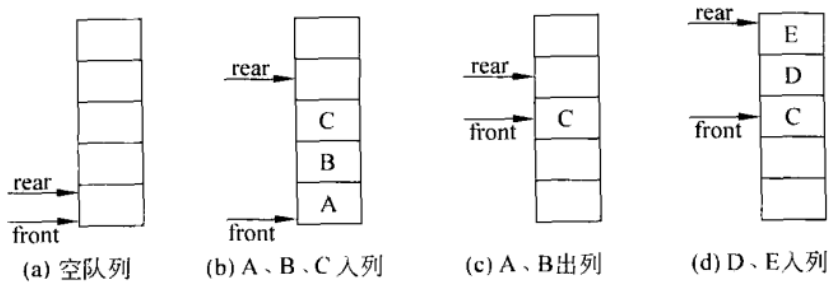


图 3.8　顺序队列的入队和出队示意图

就是如上图(d)所示尽管队列中还有空间，但 rear 到了 maxSize-1 所以也插不进去了，这种情况就叫做假溢出。如何解决呢？

最好的办法就是让队列循环起来，在逻辑上连成一个环。也就是说当front 和 rear 进入到 maxSize-1 时再进一个位置就自动地移动到 0

实现方法也很简单：

队头：front = (front + 1) % maxSize

队尾：rear = (rear + 1) % maxSize

逻辑上也就形成了下面的情况

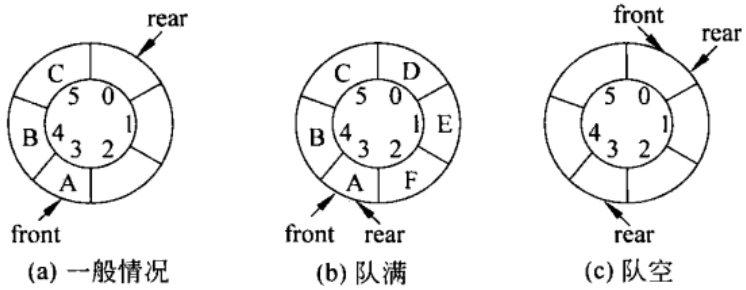


图 3.9　循环队列示意图

紧接着新的问题就又产生了—— rear == front 即是队空的标志也是队满的标志

解决方法：

way 1 另设置一个标志来区别队列是空还是满

第一种种方法也是我们实现时用到的方法：引入了一个 maxSize 来确定到底是满还是空 所以只有空的时候才会有 rear = front 满的时候用 length = maxSize - 1

way 2 少用一个元素空间，约定队头在队尾指针的下一个位置时作为队满的标志。也就是说不再是以rear == front 来判断队满

而是 front == rear - 1 来判断

在敲代码中发现本结构设计有几个特别好的地方值得吸取：

- 像这种有头有尾的循环式设计，在计算有多少个元素的时候 用+最大值再取余的方式，这样无论是头在尾前还是尾在头前都可以计算出来了
- 像这种循环设计 遍历的方式也要很讲究

```
for(int curPosition = front; curPosition != rear; curPosition = (curPosition+1) % maxSize)
```

另外以下几个方法的写法也很值得我们学习：

```
length = (rear - front + maxSize) % maxSize
```

4.3 3.4 实例研究——表达式求值 <自己写一下> <可以做到最终的大作业>

表达式求值是栈的典型应用，常用中缀表达式求值算法——算符优先法

4.3.1 基本逻辑的梳理

step 1确定规则

- 先乘除后加减
- 从左向右
- 先括号外再括号内

step 2 区分要操作的东西

总体需要操作的事物就分为三类：

- 操作符(operator)（在这只考虑 + - * /）
- 操作数(operand)（可为变量可为常量 这里只考虑常量）
- 界限符(delimiter)（这里只考虑 (和) 以及 =）

为了操作方便将界限符也当作操作数，得到如下优先级表格

解读：

- theta1表示左边出现的运算符 theta2表示右边出现的运算符
- e 表示不能相遇
- 在表达式左边虚设一个 "=" 遇到最终的 "=" 的时候说明表达式求值完毕

表 3.1 操作符优先关系表

theta1 \ theta2	'+'	'-'	'*'	'/'	'('	')'	'='
'+'	>	>	<	<	<	>	>
'-'	>	>	<	<	<	>	>
'*'	>	>	>	>	<	>	>
'/'	>	>	>	>	<	>	>
'('	<	<	<	<	<	=	e
')'	>	>	>	>	e	>	>
'='	<	<	<	<	<	e	=

step 3 具体算法实现——两个工作栈

设置两个工作栈----一个为操作符栈 optr 一个为操作数栈 opnd

step1 在 optr 栈中加入一个 '=' (也就是我们说的虚等号)

step2 从输入流中获取一个字符 ch 不断循环执行 (1) 至 (3) 直到求出表达式的值

end

(1) 取出 optr 栈顶 optrTop 当 optrTop等于'=' 并且 ch等于'='时，整个表达式求值完毕为止，这时候 opnd 栈栈顶的元素就是表达式的值(这就是出口)

(2) 若 ch 不是操作符就将字符放回输入流，将其读入操作数，加入 opnd 栈

(3) 若 ch 是操作符，将比较 ch 和 optrTop 的优先级

- case1: optrTop < ch 则 ch 入 optr 栈，从输入流中取下一个字符ch
- case2: optrTop > ch 则从opnd栈中退出两个操作数 right 和 left 从optr栈中退出操作符 theta，形成运算指令 (left) theta (right) 结果入opnd栈
- case3: optrTop为'(' 且 ch为')' 那就直接去个括号就行了 这个是完全没啥意义的
- case4: optrTop e ch 这是出现了语法错误，终止执行

5 Chapter 4 串（string）（考到的都是经典的）

这个开始和前三章出现断层 之前全是线性表的介绍

考点：就是字符串的一点点基础的表达等

5.1 4.1 串类型的定义

string 就是由**零个或多个字符（char）**构成的有限序列，如果再用线性表就太**浪费**了。

类型

- 空串：由零个字符组成的串 len == 0
- 空格串：由一个或多个空格组成的串，len != 0
- 子串
  - 连续的两个字符构成的子序列成为该字符串的子串
  - 空串是所有串的子串
- 主串：子串是主串的字串
- 位置：子串在主串中的位置以子串的**第一个字符**在主串中的位置表示
- 串相等：完完全全相同

基本操作

5.2 4.2 字符串的实现

5.2.1 4.2.1 字符串的实现

c实现

- 定长顺序存储表示
  - 定长字符数组长度直接用宏定义
  - 一般来说我们最终的终止符用C语言默认的 \0 来表示 如果自己不想的话还是要封装一个 len 来表示串的长度
- 堆分配存储表示
  - 就是我们书上 c++ 实现的方式
- 块链存储表示 <很笨重用的少> <只有很多经典教材上才会涉及>
  - 就是底层为链表形式的串
  - 所以为了提高存储密度 一个 Node 可能存多个字符 -> 块
    - 但又出现了很多很多问题，有时会很复杂不实用

5.3 4.3 字符串模式匹配算法 这个才是本章的重点

用法：搜索、屏蔽敏感文字等等

主串称为目标串，子串称为模式串

要想匹配成功就必须要每一位都相同

5.3.1 4.3.1 简单字符串模式匹配算法

```
// 这种算法是很简单的 逻辑思维也很简单
// 时间复杂度好的情况是 n 坏的情况是 n * m
```

5.3.2 4.3.2 首尾字符串匹配算法

```
// 这种是为了防止直到最后一个字符才完成匹配，所以我们采取左右开弓的架势
```

怎么加速呢？

- 要搞清楚BF算法的本质——回溯
- 所以我们的改进就是要避免回溯

另一种改善的匹配算法 **KMP 匹配算法**（有可能会考选择题）★

基本概念

目标串的指针永远不会回头，只会往前走——核心问题就在于你目标串只向前走直到不相等的时候才停下来，再次开始比较的时候该和哪一个比较呢？所以就引入了next数组这个概念。这个数组的意思就是说，如果上一次比较时在子串的j处不匹配了，那么下一次就让母串的 i 和子串的next[j] 元素做比较。

两个考察方向：

- nexj[j] = k 的求解 这个k怎么去解？
- next[j] 已知如何做更新？

程序编写方法：基本上不考

6 Chapter 5 数组和广义表（主要是理解基本概念）

数组和广义表可看成是一种特殊的线性表，其特殊在于，表中的数据元素本身也是一种线性表。

（因为如果一旦引入了矩阵的概念，作为二维数组，就像是一个由表构成的表）

6.1 5.1 数组

定义：由一组类型相同的数据元素构成，每一个元素可直接按序号寻址的线性表。

在n维数组中，每个元素受n个线性关系约束。（那么问题来了 数组虽然说可视出来好像很好存的样子 那他在计算机中到底是怎么存的呢？）

c++自带的数组已经很方便了也可以随便开维度，但是唯一不足的点是：如果数组下标越界不会报错，但是这个也很好解决，书上给出了一种解决方法，但是很复杂，我们只需了解即可。

6.1.1 5.1.2 数组的顺序存储方式

通常有两种顺序存储方式 考点也就是求一下物理地址罢了

- 行优先顺序（现在大部分用的就是这个）
    - 以二维数组为例子：一行行往里面存。第 i + 1 个元素紧接在第 i 个行向量后面;
    - 用的原因：一些外部设备所用的存储方式都是按行优先存的。
    - **考点：**求解以行为主的任意元素的物理地址
- 列优先顺序（基本不用）

6.2 5.2 矩阵

定义：

太简单了，就是二维数组，书上写的也是最基本的拓展，完全不用关心。

下面要讨论的就是——一些特殊矩阵让他不再是二维表达，而是变成一维表达（这样可能会节省时间和空间）

- 特殊矩阵：值相同或者零元素在矩阵中的分布有一定规律。这样的话我们若是能找到一定的特殊方法进行存储和处理，就能够提高空间和时间的利用率。
- 以对角矩阵为例子感受一下：

  我们要开辟一个一维数组 `elems` 来存储一个二维对角矩阵

  `step1` 开辟多大的空间呢？

  主对角线上有n个元素，两条次对角线上有 `n-1` 个元素，所以这三条对角线上共有：`2n-2` 个元素。但是千万别忘了还有一个填充其他区域的元素，一般是一个常数c -> 所以我们就要开辟 `2n-1` 个空间。

  `step2` 怎么存呢？

  我们规定第一个元素，即 `elems[0]` 始终存c。其余的可以行优先来存，也可以列优先来存，对于这个情况我们甚至可以选用对角线优先来存。

  - 对称矩阵

    关键是找到矩阵a[i] [j] 和向量s[i]之间的关系
  - 三角矩阵

    对称矩阵的存储方式可以应用于三角矩阵
  - 对角矩阵

    主对角线和对角线两侧的两个线的对称区域，想好需要多少数据元素
  - 总结：

    三种特殊的元素，非零元的分布总是有规律的，总能找到一种方法将他们压缩到一个向量中（找到对应关系嘛）

    整体过程和上述是没有任何区别的
- 稀疏矩阵：假设矩阵A中有s个非零元素，若 s 远远小于矩阵元素的总数，则称A为稀疏矩阵。因为这种情况下我们再采用二维数组这种普通存储方法就太浪费时间空间了。
  - 三元组顺序表

- 自己去编写架构 创建其了一个顺序表
- 这个要求非零元素比较多才有意义

```
// 自己搭建一下 思路和 由 Node -> List 是一样的
// step 1 基本结构 Triple
template <class ElemType>
struct Triple{
    int row, col;   // 行 列
    ElemType value; // 值

    // 两大构造函数
    Triple();
    Triple(int r, int c, ElemType v);
}

// step 2 基于基本结构开始搭建三元组顺序表类
template <class ElemType>
class TriSparseMatrix{
    protected:
        Triple<ElemType> * triElems;     // 存放稀疏矩阵的三元组表
        int maxSize;                    // 非零元素的最大个数
        int rows, cols, num;             // 原始稀疏矩阵的行列以及非零元素的个数
}

// step 3 填充方法就完喽
// 这里只写出一个最重要的。就是设定指定位置(r, c)处的元素值v，别看这个好像只有setElem 其实它涵盖了<增 删 改>三个功能(因为如果原本指定位置(r, c) v = 0 的话就是加了一个元素，如果把 v 变为 0 的话就是删去操作)
SetElem(int r, int c, const ElemType &v){
    if(r > row || c > col || r < 1 || c < 1){
        return Range_Error;
    }
```

```
    int i, j;
    for(j = num - 1;
        // 因为这个地方是顺序表，这个地方才采取这种条件方式
```

```
        j >= 0 && (r < triElems[j].row || r==triElems[j].row &&
c < triElems[j].col);
        j--);  // 这个地方出来的 j 就是我们搜索出的应该set的位置
```

```
    if(j >= 0 && triElems[j].row = r && triElems[j].col == c){
        if(v == 0){// 这种情况就是删除该元素
            for(i = j + 1; i < num; i++){
                triElem[i - 1] = triElem[i]; // 向前移动一个单位
            }
            num--; // 删去了一个三元组 非零元素个数--
        } else {
            triElem[j].value = v;
        }
       return SUCCESS;
    } else { // 这种情况就是原本在该处并没有非零元素
        if(v != 0){ // 这种情况就要添加元素了
            if(num < maxSize){
                for(i = num - 1; i > j; i--){
                    triElems[i + 1] = triElems[i];
                }
                triElems[j+1].row = r;
                triElems[j+1].col = c;
                triElems[j+1].value = v;
            } else {
                return OVER_FLOW;
            }
        return SUCCESS;
    }
}
```

- 十字链表
  - 十字链表中矩阵每一个非零元素用一个结点表示，除了data(row, clo, data) 还要有两个链域right 和 down 两个指针

这个本质和双向链表是一个道理，但是 right 和 down 是平面性指针了，而不再是线性指针。

这个地方只需要了解一下如何插入目的结点就可以了（为了保持和之前线性表的一致 我们依然采取带头节点的）

行表头 rightHead; 列表头 downHead

6.3 5.3 关键算法

6.3.1 5.3.1 三元组为起点的两大转置算法：

算法概要：

转置具体算法可分为两步实现：

(1) 将三元组表里的每一个三元组个元素行列互换

(2) 对三元组表重新排序，使其中元素按行序或者列序排列

上面的算法也是我第一时间想到的算法，实现起来很简单，但是时间复杂度很高，为了降低时间复杂度就不得不省略掉排序这个操作，所以我们假设原本的三元组表source 转置后放在目标三元组表 dest中

(1) 将三元组表source中的每一个元素取出来，交换其行、列号。

(2) 将变换后的三元组存入目标三元组表 dest 中的适当位置。

现在问题就变成了如何求解适当位置了

- 简单转置算法
- 快速转置算法

简单转置算法

如果非零元素很多的话就会造成其时间复杂度更大

实现的方法很简单：

```
detPos = 0; // 从 source 中取出放在 dest 中的位置
for(col = 最小列号; col <= 最大列号; col++){
    // 在source中<查找>有无列号为 col 的三元组; 若有，则交换
    // 然后存到dest中
}
```

快速转置算法

找到简单转置算法慢的原因：不仅需要扫描整个三元组表，还需要扫描三元组表的每一列。所以我们就想能不能只扫描三元组表后就直接找到相应元素在dest中的位置呢？

依按三元组表A的次序进行转置，转置后直接放在三元组表B的正确位置上，这种转置算法叫做快速转置算法 核心问题是在于如何计算 destPosition

step 1 计算每一列的非零元个数
存放在 cNum[] 中
一个 for 时间复杂度 O(n)
step 2 计算原始矩阵每一列第一个非零元素在目的三元组中的正确位置 用递推关系得出
并存放在 cPos[] 中 即 cPos[col] = source 第 col列上第一个非零元素在dest中应放置的位置

6.4 5.4 广义表

基本不考 只考概念

6.4.1 基本概念

广义表是线性表的推广，就是 python 里面 list 的概念，每一个元素的属性完全可以不同。

广义表通常称为表，或者 list 每一个ai其实也可以是一个表可以是任何原子的东西

- 当ai为仍然为表的时候我们就将其称为子表

最重要

$n = 0$ 的广义表为空表， 没有表头表尾的概念

$n > 0$的时候表的第一个表元素称为广义表的表头

除此以外其它表元素组成的表称为广义表的表尾

解读：

- $n = 0$ 的时候没有表头表尾
- $n > 0$的时候表头是一个元素
- 表尾肯定是一个表

一些习惯：

- 用大写表示广义表
- 用小写表示原子（可能就是一个简单的元素也可以是一张表）

广义表的定义是一个递归的定义，也因此有了深度的概念

广义表举例

- A = ()
- B = (c)
- C = (a, (b, c, d))
- D = (A, B , C)

广义表的长度和深度

广义表的长度：就是原始表有多少个点（不论是原子还是表）

GL的深度Depth(GL)定义如下

depth(GL) = 0 if GL 为原子元素

= 1 if GL 为空表

= 1 + Max(depth(ai)(1 <= i <= n)) 其他情况

例如 2 深度为0

() 深度为1

(1, (2, 3))深度为 3 (1 + 2)

6.5 广义表的基本操作

6.5.1 广义表的存储结构

- 由于广义表中的数据元素可以具有不同的结构(或是原子，或是列表)因此难以用顺序表的存储结构来表示，通常用链式存储结构
- 这里引用了数链式存储结构——引用数法广义表
- 广义表一般需要两种结构的结点：一种是表结点->表示列表； 一种是原子结点用来表示原子
- 为了方便起见：**引用数法广义表**，还在列表的前面加上头结点，形成第三种结点
    - 因为表结点是一种引用指向，所以当你删除的时候一定要想清楚能不能回收，如果说有一部分指针明明有指向你就直接删除了，那他指向的东西就没办法删除掉了
    - 结点就可以分为三类：每一个里面都有标志域(分类)， 数据域
        - 头节点：tag = HEAD，数据域存储引用数
        - 原子结点：tag = ATOM
        - 表结点：tag = LIST，数据域里面存的也是指针
    - 引用数法广义表的特点：

        - 有头节点的好处
            - 便于操作：删除该表的第一个元素，不用改变引用他的指针，只用改变表内的结构即可
- 使用空间法广义表 <这个回去多去了解一下基本概念>

第 6 章往后的所有程序全部都要会<背>

7 Chapter 6 树和二叉树 <究极重点> <每一小节两个算法> <手写程序的出处>

7.1 6.1 树的基本概念

7.1.1 6.1.1 树的基本概念（我们画树就是从上往下 不要从左往右）

树型结构式一类重要的非线性结构

树型结构是结点之间有分支，并且具有层次关系的结构，非常像一个倒着的树

定义：

- 树是n个节点的有限集合T
- 每个结点最多只能有一个直接前驱，可以零个或多个直接后继
- n = 0 -> 空树
- n > 0
    - n有且只有一个根(root) 其只有直接后继 没有直接前驱。
    - n > 1时 m个互不相交的有限集T1 ~ Tm 其中每一个子集本身又是复合本定义的树。

7.1.2 6.1.2 树的基本术语

- 结点(node)

  数据元素 + 若干指向子树的分支
- 结点的度(degree)<往下走的分支>

  一个结点**子树**的个数 就是有多少个分叉
- 树的度

  树中所有节点的度的**最大值**
- 叶子结点

  度为零的结点，即无后继的结点，也称为终端结点
- 分支结点

度不为零的node 也称为非终端结点

- 结点的层次

  从根结点开始定义，根结点的层次为1 直接后继层次为2 依次类推

- 树的深度（也称高度）

  最大的层次 **只有一个根结点高度就为 1**

- 孩子结点

  一个**结点子树**的根就是它的孩子结点

- 双亲结点

  一个结点的直接前驱

- 兄弟结点

  同一个双亲结点的子结点

- 祖先结点

  从**根结点**到该结点**路径上**的**所有结点**

- 子孙结点

  以某结点为根的所有子树上的结点

- 有序树

  子树之间存在关系，从左至右是有序的(不能互换)，不能变序

- 无序树 <通常来讲是无序的>

  结点没有顺序，可以任意更改

- 森林

  森林是由互不相交的树构成的

  根结点拿掉 子树又变成一个森林 -> 子树森林

  注意：0棵或1棵树都可以组成森林

7.2 6.2 二叉树

7.2.1 6.2.1 基本概念

- 二叉树是一种简单的特殊的树，任何的树和森林**都可以表达成二叉树**。

  > 所以后面才有那么多的修改算法

- 二叉树的子树有左右之分，是有序的，次序不能任意颠倒。即使有一棵树也要画出来是左还是右。
- 定义：**结点的度**不大于2， 有左右之分。
- 二叉树有**五种**基本形态
  - 空树
  - 只有**根结点**的二叉树
  - **右子树为空**的二叉树
  - **左子树为空**的二叉树
  - **左右子树都非空**的二叉树
- code 中的基本方法
- **满二叉树**

  结点全部满了，这样编号就很好编
- **完全二叉树**

  只有可能最后一行不完整。这样编号就和满二叉树在刚开始是一一对应的。可能只有最后一层有一些区别

7.2.2 6.2.2 二叉树的性质

性质 1：二叉树的第 i 层上至多有 **2 的 i - 1 方**个结点 <考点>

性质 2：深度为 k 的二叉树上总结点个数最多为 **2 的 k 次方 - 1**个 <考点>

性质 3：对任何一棵二叉树，若有 n0 个叶子节点，n2 个度为2的结点，则必存在 $n0 = n2 + 1$ <经常考计算题>

证明：

总结点数：n；总分支数：b；结点度区分结点 n0，n1，n2

- n = n0 + n1 + n2
- 两种方法算 b

  从下向上 b = n - 1

  从上向下 b = n1 + 2*n2
- n0 = n2 + 1

性质 4： 具有 n 个结点的**完全二叉树**的深度为 **log2n下取整 + 1**

证明：

设深度为 k

得到结点个数范围

性质 5：对于具有 n 个结点的**完全二叉树**，如果按照从上到下和从左到右的顺序对二叉树中所有结点从1开始顺序编号，则对于任意的序号为 i 的结点有：

- 如果 i = 1 则为根结点，无双亲结点。如 i > 1 则序号为 i 的结点的双亲结点序号为 i/2 向下取整
- 如 2 x i > n 则序号为 i 的结点无左孩子; 如 2 x i <= n 则序号为 i 的结点的左孩子为 2 x i
- 同上 如果有 2 x i + 1 > n 则序号为 i 的结点无右孩子，否则有右孩子为 2i + 1

7.2.3 6.2.3 二叉树的存储结构

顺序存储结构：用一组连续的存储单元存储二叉树的数据元素。因此要找到合理的编号方法。所以方法是：**将所有的二叉树都安排成完全二叉树来编号。**

> 很少使用

链式存储结构：

> 访问起来不太方便 但大多数都是使用这个

- 二叉链表和三叉链表的不同

7.3 6.3 二叉树遍历 <核心算法——考试>

每一种算法均分为递归算法(比较简单)和非递归算法(要选对使用的数据结构)

遵从某种次序，顺着这种次序把所有的结点都访问到并且只访问一次。

想法：寻找某一种规律使二叉树上的结点能排列在一个**线性队列**上从而便于遍历。

- 先上后下按层次遍历
- 先左后右的遍历
- 先右后左的遍历

对于我们来讲只分为 三个部分 **左边(L) 根(D) 右边(R)** 规则一定是统一的——就是说先左后右这个股则不变

- DLR 先序遍历

  遇到一个(子)树就先访问根，再访问左边，再访问右边

```
template <class ElemType>
void BinaryTree<ElemType>::PreOrderHelp(BinTreeNode<ElemType>*
r, void (*visit)(const ElemType&)) const
// 操作结果：先序遍历以r为根的二叉树  √
{
    if (r != NULL) {
        (*visit)(r->data);                  // 访问根结点
        PreOrderHelp(r->leftChild, visit);  // 遍历左子树
        PreOrderHelp(r->rightChild, visit); // 遍历右子树
    }
}
```

- LDR 中序遍历

  遇到一个(子)树就先访问左边，再访问根，再访问右边

- LRD 后序遍历

  遇到一个(子)树就先访问左边，再访问右边，再访问根

- 层次遍历

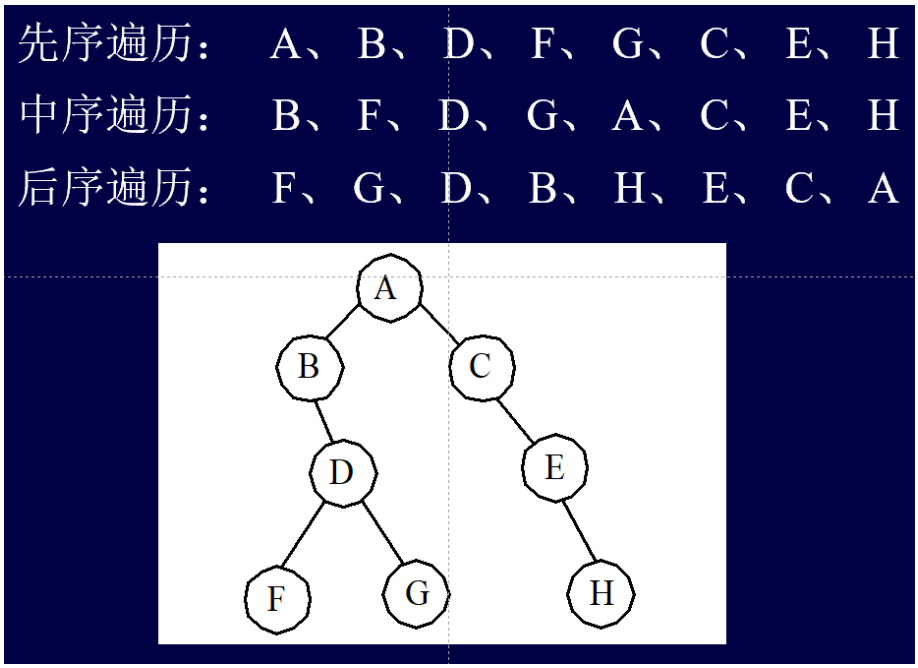
  每一层从左往右 这个用到的是队列

```
template <class ElemType>
void BinaryTree<ElemType>::LevelOrder(void (*visit)(const
ElemType&)) const
// 操作结果：层次遍历二叉树 √ 刚开始该功能实现时有一定的问题 但后来解决掉
了
{
    LinkQueue<BinTreeNode<ElemType>*> q;     // 创建一个二叉结点元
素队列
```

```
    BinTreeNode<ElemType>* t = root;          // 从根结点开始进行层
次遍历

    if (t != NULL) q.InQueue(t);              // 如果根非空,则入队
    while (!q.Empty()) {
        // q非空,说明还有结点未访问
        q.OutQueue(t);
        (*visit)(t->data);
        if (t->leftChild != NULL)             // 左孩子非空
            q.InQueue(t->leftChild);          // 左孩子入队
        if (t->rightChild != NULL)            // 右孩子非空
            q.InQueue(t->rightChild);         // 右孩子入队
    }
}
```

例子：(必须会！) 只要会了这个什么都会了



7.3.1 6.3.3 二叉树遍历的应用 <递归思想在其中 十分重要>

> 基本上所有的都是递归 参看实验课中二叉树的实现

- 统计二叉树中结点的个数
- 求二叉树的高

```
int BinaryTree<ElemType>::HeightHelp(const
BinTreeNode<ElemType>* r) const
```

```
// 操作结果：返回以r为根的二叉树的高（一定要先明确高的定义）√
{
    // 递归思路
    if (r == NULL) {
        // 空二叉树高为0
        return 0;
    }
    else {
        // 非空二叉树高为左右子树的高的最大值再加1
        int lHeight, rHeight;
        lHeight = HeightHelp(r->leftChild);      // 左子树的高
        rHeight = HeightHelp(r->rightChild);     // 右子树的高
        return (lHeight > rHeight ? lHeight : rHeight) + 1;
        // 非空二叉树高为左右子树的高的最大值再加1
    }
}
```

- 二叉树的节点数

```
int BinaryTree<ElemType>::NodeCountHelp(const
BinTreeNode<ElemType>* r) const
// 操作结果：返回以r为根的二叉树的结点个数√
{
    // 递归思路
    if (r == NULL) return 0;              // 空二叉树结点个数为0
    else return NodeCountHelp(r->leftChild) + NodeCountHelp(r-
>rightChild) + 1;
    // 非空二叉树结点个为左右子树的结点个数之和再加1
}
```

- 二叉树的双亲

```
BinTreeNode<ElemType>*
BinaryTree<ElemType>::ParentHelp(BinTreeNode<ElemType>* r,
const BinTreeNode<ElemType>* cur) const
// 操作结果：返回以r为根的二叉树，结点cur的双亲 √
{
    if (r == NULL) return NULL;      // 空二叉树
    else if (r->leftChild == cur || r->rightChild == cur)
return r; // r为cur的双亲
    else {
```

```
        // 在子树上求双亲
        BinTreeNode<ElemType>* tmp;

        tmp = ParentHelp(r->leftChild, cur);    // 在左子树上求
cur的双亲
        if (tmp != NULL) return tmp;            // 双亲在左子树上

        tmp = ParentHelp(r->rightChild, cur);   // 在右子树上求
cur的双亲
        if (tmp != NULL) return tmp;            // 双亲在右子树上

        else return NULL;                       // 表示cur无双亲
    }
}
```

- 二叉树的销毁

```
void BinaryTree<ElemType>::DestroyHelp(BinTreeNode<ElemType>*&
r)
// 操作结果：销毁以r的二叉树 √
{
    if (r != NULL) {
        // r非空,实施销毁 迭代调用
        DestroyHelp(r->leftChild);      // 销毁左子树
        DestroyHelp(r->rightChild);     // 销毁右子树
        delete r;                       // 销毁根结点
        r = NULL;
    }
}
```

- 二叉树的复制

```
BinTreeNode<ElemType>*
BinaryTree<ElemType>::CopyTreeHelp(BinTreeNode<ElemType>* r)
// 操作结果：将以r为根的二叉树复制成新的二叉树,返回新二叉树的根
{
    if (r == NULL) {
        // 复制空二叉树
        return NULL;                    // 空二叉树根为空
    }
    else {
```

```
        // 复制非空二叉树
        BinTreeNode<ElemType>* lChild = CopyTreeHelp(r->leftChild);     // 复制左子树
        BinTreeNode<ElemType>* rChild = CopyTreeHelp(r->rightChild);    // 复制右子树
        BinTreeNode<ElemType>* rt = new BinTreeNode<ElemType>(r->data, lChild, rChild);
        // 复制根结点
        return rt;
    }
}
```

- 二叉树的显示

```
void DisplayBTWithTreeShapeHelp(BinTreeNode<ElemType>* r, int level)
// 操作结果：按树状形式显示以r为根的二叉树，level为层次数，可设根结点的层次数为1
{
    if (r != NULL) {
        // 空树结点不显示，只显示非空树
        DisplayBTWithTreeShapeHelp<ElemType>(r->rightChild, level + 1);//显示右子树
        cout << endl;                    //显示新行
        for (int i = 0; i < level - 1; i++)
            cout << " ";                 //确保在第level列显示结点
        cout << r->data;                 //显示结点
        DisplayBTWithTreeShapeHelp<ElemType>(r->leftChild, level + 1);//显示左子树
    }
}
```

- 依靠中序 + 前序 或者 后序就可以获得唯一的树
- 用二叉树实现前缀中缀和后缀表达式

> 其实就是同一颗二叉树 不同的读取形式罢了

  - 前缀 >> 波兰式 读取的方法就是前序
  - 中缀 >> 就是最普通的形式 读取的方法是中序
  - 后缀 >> 逆波兰式 读取的方法式后序

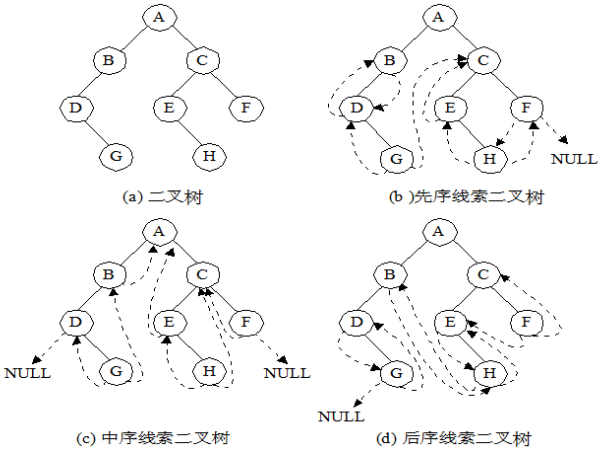
线索化

7.4 6.4 线索二叉树

7.4.1 线索的概念

当我们将原本的二叉树利用先中后序遍历出来变成了一个序列后，我们在原本树上看不到这种前驱和后驱关系，所以我们加上线索就可以很好的展现出这种前驱和后驱关系了。

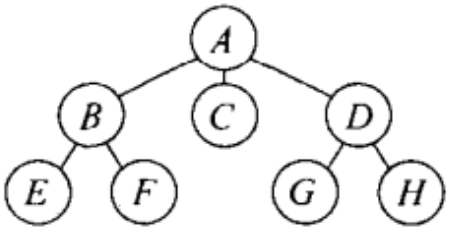
空出来的左指针指向前驱，空出来的右指针指向后继



7.5 6.5 树和森林 普适性

7.5.1 6.5.1 树的存储方法

双亲表示法 好找父亲 但不好找孩子

- 

下标		
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1
6	G	3
7	H	3

图 6.17 树双亲表示法示意图

孩子双亲表示法 既好找父亲，又好找孩子

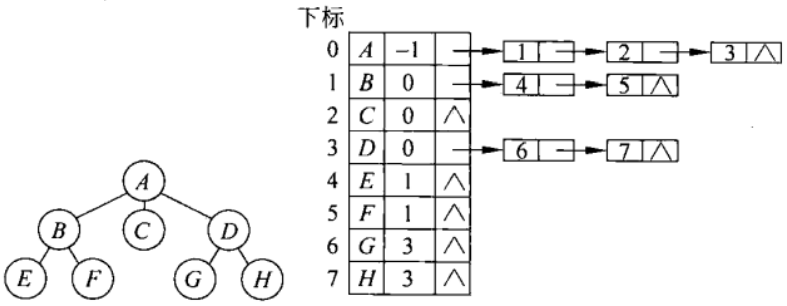
- 

图 6.18　树的孩子双亲表示法示意图

上述两种方式均有顺序存储结构，但是我们想要用链式存储结构，也就是说把学的二叉树用上 所以就需要下面的方法

孩子兄弟表示法（最便捷最常用） -> 向二叉树转变的方法

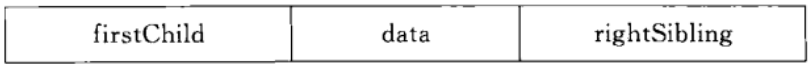


图 6.19　孩子兄弟表示树的结点示意图

树的孩子兄弟表示法示意图。

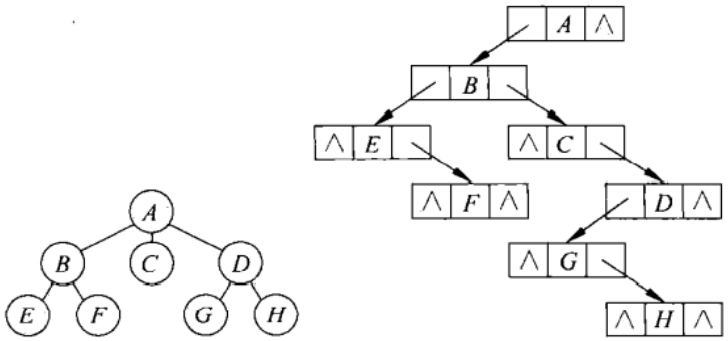
- 

图 6.20　树的孩子兄弟表示法示意图

7.5.2　6.5.3 森林的存储表示

一样有三种方式 同上

7.5.3　6.5.4 树和森林的遍历

和 二叉树 不同的地方这个地方只有两种种方式，没有了中根访问

只有 **先根遍历** 和 **后根遍历** 分别叫做**先序**和**中序** (基本上不会去考)

> 考到的话就是从左到右一颗一颗树去做

7.5.4　6.5.5 树和森林与二叉树的转换

就是用刚才说的孩子双亲表示法来回转换 so easy

只要会底层的指向关系 就不会错，不要管什么添加连线和删除连线

注意：二叉树有右子树就展开为森林 没有就展开为树

7.6 6.6 哈夫曼树与哈夫曼编码<必考点>

7.6.1 6.6.1 基本概念

哈夫曼树是一种**最优**二叉树

路径：从树的一个结点到每一个结点

路径长度：走了几个线段**<就是走了几段路>**

权： <由具体问题而来>**<只有叶子结点上有权重>**

树的带权路径长度(WPL)：所有**叶子结点**的带权路径长度之和。

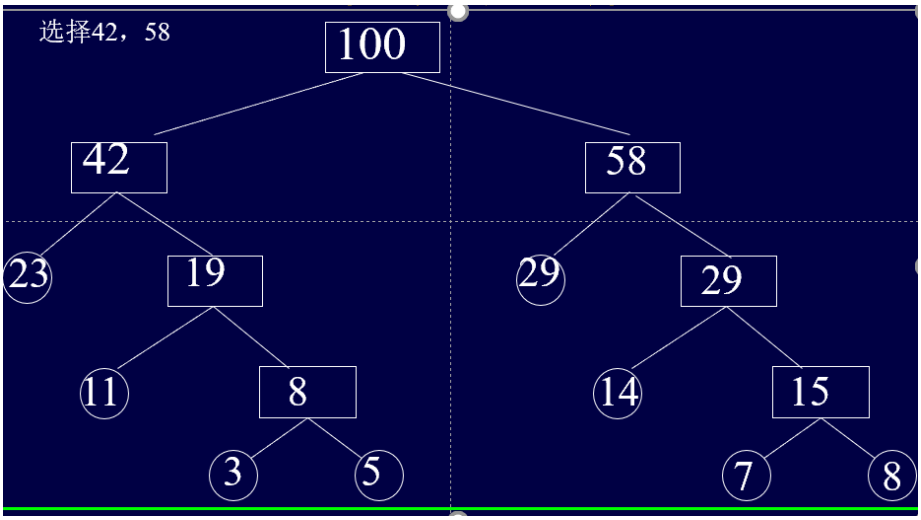
哈夫曼树：n个叶子结点、并带相同权值的二叉树，必存在一棵带权路径长度最小的二叉树叫做最优二叉树即哈夫曼树。

7.6.2 6.6.2 哈夫曼树构造算法

只要会构造就行，没必要非要根据标准算法来。永远要记住我们排列的是**叶子结点**

一定要把WPL算出来

一个例子 {5, 29, 7, 8, 14, 23, 3, 11}



7.6.3 6.6.3 哈夫曼编码

要想设计一个变长度编码 + 就必须要有一定的规则

前缀码：任何一个编码都不是其他任何编码的前缀，则称此编码系统中的编码是前缀码。<发现根本不好设计> 其实用任何一个二叉树都可以实现一个前缀编码

编码规则<利用哈夫曼树来实现>：阴阳调和就是前缀码了。

哈夫曼编码只是一个编码的方式，根据不同的情况还要进行调节，它表示使得一定报文编码**总长度的最短**二进制前缀编码。最终WPL就表示报文总长度。

<最终复习倒着去做>

7.7 6.7 树的计数 <也有可能有设计题>

只用记住公式就行了

Gatalan 公式 n个结点到底有多少棵二叉树

$$b_n = \frac{1}{n+1} C_{2n}^n$$

7.7.1 树和森林与二叉树的转换 由二叉树的计数可推得树的计数

具有 n 个结点的树的数目 == 具有 n-1 个结点的二叉树的数目相同

具有 n 个结点的森林数目 == 具有 n 个结点的二叉树的数目相同

8 Chapter 7 图

8.1 7.1 图的定义和术语

权

子图 <可以单独拿来一个点作为子图 但没有说拿来边做子图>

度：牢记握手定理（**出度&&入度**）

路径：路径长度指的是这一条路径上边的条数。

- 路径上各个顶点都不同 >> 简单路径
- 回路（环） >> 简单回路

连通：

> n 个结点，少于n - 1条边则一定非连通，如果多于 n - 1 则一定有环

连通图：

- 强连通图
- 强连通分量<极大强连通子图>

连通图的生成树： 一个连通图有n个顶点 e条边，其中n-1条边和n个顶点构成一个极小联通子图 >> 此连通图生成树（如果在一棵生成树上添加一条边，必定构成一个环）

8.2 7.2 图的存储表示

> 由于图的结构比较复杂，任意两个结点之间可能凑存在联系，因此没有办法用顺序影像来存储。所以常用的存储结构包括数组表示(邻接矩阵)和邻接表。

8.2.1 7.2.1 邻接矩阵表示法

关键看代码，弄明白如何加边、如何找邻接点等

8.2.2 7.2.2 邻接表表示法

> 图的一种链式存储结构

产生了一个很大的问题：这样虽然很好就找到了儿子，但是找不到父亲。还必须要去遍历。

- 解决方式1：可以针对一个有向图分别构建一个**邻接表**，和**逆邻接表**来表达。

8.3 7.3 图的遍历

> 这个问题是后续解决很多问题

8.3.1 7.3.1 深度优先搜索(Deep - First - Search) <有可能会考写程序>

> 类似于先根遍历，是树先根遍历的推广

过程看一下 PPT

code：

```
template <class ElemType>
```

```
void DFS(const AdjMatrixUndirGraph<ElemType>& g, int v, void
(*visit)(const ElemType&))
// 操作结果：从顶点v出发进行深度优先搜索图g
{
    g.SetTag(v, VISITED);            // 设置访问标志
    ElemType e;                      // 临时变量
    g.GetElem(v, e);                 // 顶点v的数据元素
    (*visit)(e);                     // 访问顶点v的数据元素
    for (int w = g.FirstAdjVex(v); w != -1; w = g.NextAdjVex(v, w))
{
        // 对v的尚未访问过的邻接顶点w 先进行的是邻接矩阵上的逻辑第一个临结点 递
归调用DFS
        if (g.GetTag(w) == UNVISITED) {
            DFS(g, w, visit);
        }
    }
}

template <class ElemType>
void DFSTraverse(const AdjMatrixUndirGraph<ElemType>& g, void
(*visit)(const ElemType&))
// 操作结果：对图g进行深度优先遍历 即对每一个顶点都调用 DFS
{
    int v;
    for (v = 0; v < g.GetVexNum(); v++) {
        // 对每个顶点作访问标志
        g.SetTag(v, UNVISITED);
    }

    for (v = 0; v < g.GetVexNum(); v++) {
        // 对尚未访问的顶点按DFS进行深度优先搜索
        if (g.GetTag(v) == UNVISITED) {
            // 从v开始进行深度优先搜索
            DFS(g, v, visit);
        }
    }
}
```

8.3.2 7.3.2 广度优先搜索(Breadth - First - Search)

做图片处理 一般都用广度优先算法

> 类似于层次遍历

最核心的就是：先访问点才能先访问它的邻结点 <队列结构>

code：

```
template <class ElemType>
void BFS(const AdjMatrixUndirGraph<ElemType>& g, int v, void
(*visit)(const ElemType&))
// 初始条件：存在图g
// 操作结果：从第顶点v出发进行广度优先搜索图g
{
    g.SetTag(v, VISITED);                          // 作访问标志
    ElemType e;                                    // 临时变量
    g.GetElem(v, e);                               // 顶点v的数据元素
    (*visit)(e);                                   // 访问顶点v的数据元素
    LinkQueue<int> q;                              // 定义队列
    q.InQueue(v);                                  // v入队
    while (!q.Empty()) {
        // 队列q非空，进行循环
        int u, w;                                  // 临时顶点
        q.OutQueue(u);                             // 出队
        for (w = g.FirstAdjVex(u); w >= 0; w = g.NextAdjVex(u, w)) {
            // 对u尚未访问过的邻接顶点w进行访问
            if (g.GetTag(w) == UNVISITED) {
                // 对w进行访问
                g.SetTag(w, VISITED);              // 作访问标志
                g.GetElem(w, e);                   // 顶点w的数据元素
                (*visit)(e);                       // 访问顶点w的数据元素
                q.InQueue(w);                      // w入队
            }
        }
    }
}

template <class ElemType>
void BFSTraverse(const AdjMatrixUndirGraph<ElemType>& g, void
(*visit)(const ElemType&))
// 操作结果：对图g进行广度优先遍历
```

```
{
    int v;
    for (v = 0; v < g.GetVexNum(); v++) {
        // 对每个顶点作访问标志
        g.SetTag(v, UNVISITED);
    }

    for (v = 0; v < g.GetVexNum(); v++) {
        // 对尚未访问的顶点按BFS进行深度优先搜索
        if (g.GetTag(v) == UNVISITED) {
            BFS(g, v, visit);
        }
    }
}
```

8.4 7.4 图的最小代价生成树<每年都会考> (minimum cost spanning tree, MST)

> 有很好的用处 比如后期做最短路线

最小生成树的一个基础性质：

保证能找到一棵

8.4.0.1 Prim 算法 （加点法）（普雷姆）

设 $G=(V,\{E\})$是连通网，TE 是最小代价生成树的边的集合，算法如下：
(1) TE=$\{\}$，$U=\{u_0\}$。
(2) 在所有$<u,v>\in E, u\in U, v\in V-U$ 的边中选择权值最小的边$<u',v'>$。
(3) 将$<u',v'>$并入 TE 中，v'并入 U 中。
(4) 重复(2)和(3)直到 TE 有 $n-1$ 条边(n 为G 的顶点个数)，这时 $T=(T,\{TE\})$便是最小代价生成树。

Prim 算法生成最小代价生成树示例如图 7.13 所示。

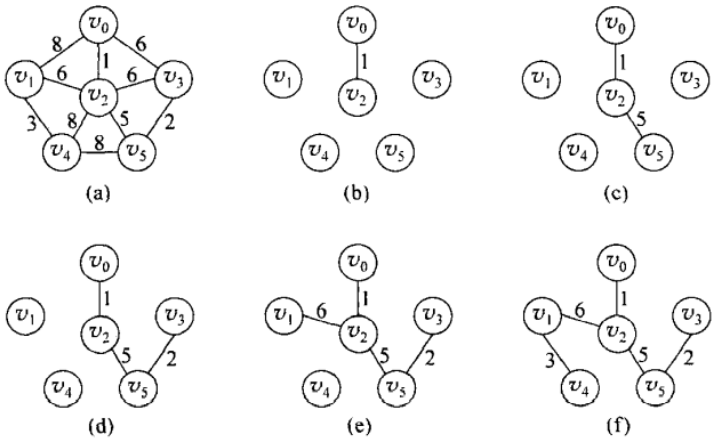


图 7.13　Prim 算法构造最小生成树的过程

考试的时候每一步都要写，关键是看清楚从哪个点出发。得满分。

每一步都要去遍历邻接表

8.4.0.2　Kruskal 算法 （加边法）（克鲁斯卡尔）

Kruskal 算法从另一途径构造最小代价生成树，算法的初态为只有 n 个顶点而无边的非通图 $T=(V,\{TE\})$，此处 TE=｛｝，这时每个顶点自成一个自由树，在 E 中选择权值最小的边$<u,v>$，如果此边所依附的顶点分别落在两个不同的自由树中，便将$<u,v>$并入 TE 中，也就是将 u 和 v 所在的自由树合并为一棵新的自由树，否则舍去此边选择下一条权值最小的边，依次类推直到 T 中所有顶点都在同一棵自由树上为止。

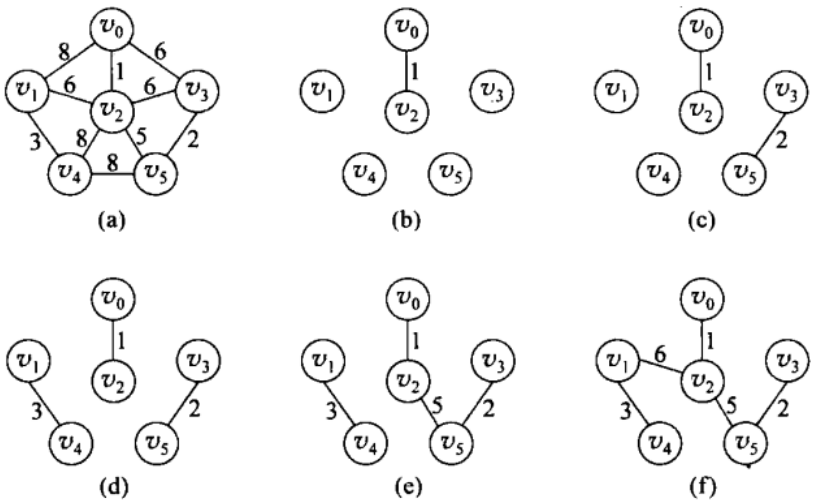


图 7.14　Kruskal 算法构造最小生成树的过程

8.5　7.5 有向无环图及其应用 （DAG）

> 就是一种很特殊的树 一个点可能有很多个入度

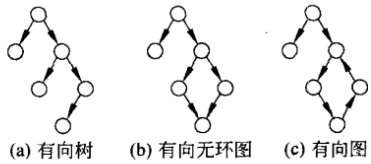


图 7.15　有向树、有向无环图和有向图的示例

8.5.1　7.5.1 拓扑排序

就是将顶点一个个排列起来 对AOV图来讲

找到入度为零的点后擦掉 然后再找再擦掉

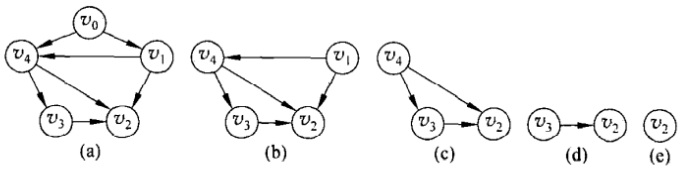


图 7.17　AOV 图及其拓扑排序的过程

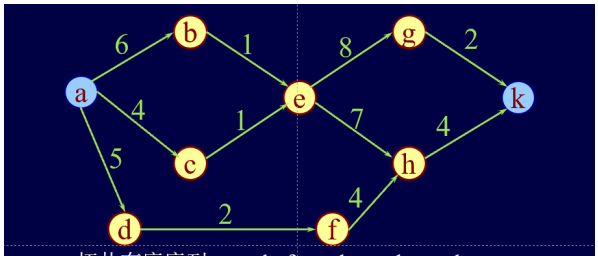
8.5.2　7.5.2 关键路径

> 这个地方考虑的问题就是完成一项工程需要很多个部分，每一个部分都要完成。ppt上有相关算法的可视化图像，其实这些所有的都是一个问题。

AOE 网

> 1. 确定整个工程的最少完成时间
> 2. 为缩短工程的完成时间，哪些活动起到关键作用，也就是说为了加快工程要从哪儿下功夫。

求关键路径：



拓扑有序序列：a - d - f - c - b - e - h - g - k

	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

	ab	ac	ad	be	ce	df	eg	eh	fh	gk	hk
权	6	4	5	1	1	2	8	7	4	2	4
ee	0	0	0	6	4	5	7	7	7	15	14
el	0	2	3	6	6	8	8	7	10	16	14
	✓			✓				✓			✓

某一个点的最早发生时间ve: AOE网中**最短完成时间**应该是从源点到汇点的**最长路径** ve

> 也就是说最关键的一部分需要多久才能完成 那么这一部分完成了全部的就该完成

- 计算方法：从源点往后推 慢慢来

某一个点的最迟发生时间 vl

> 无非就是偷懒的时间

- 计算方法：从汇点往前推 vl(i)=Min{vl(j)- g.GetWeight (i, j)}

由上 ve 和 vl 得下面关于边的内容

某一个活动最早开始时间 ee（i, j） 表示点i到点j的最早开始时间

- ee = ve(i, j)

某一个活动最迟开始时间 el 表示点i到点j的最晚开始时间

- el = vl(j) - weight(i, j)

结论：

- vl = ve 的点肯定在关键路径上 el = ee 的活动肯定在关键路径上
- el - ee 表示活动的时间余量

8.6 7.6 最短路径 <很有用 到时候看ppt>

> 和关键路径还不同
>
> 这一章十分 useful 可以做路径规划建模

8.6.1 7.6.1 单源点最短路径问题 Dijkstra

> 其实求从一个点 d 到点 s 的最短路径还不好求
>
> 反而求 d 到其他所有点的路径还比较好求

Dijkstra 算法求解该问题

性质： 现在找到的最短路径的终点集合S。则下一个最短路径 要么从 v0 直接到 要么从 S 中找一个点比较后取出较小值。

算法过程在 ppt 上

也就是两个数组在不断改变

8.6.2 7.6.2 每个点之间的最短路径 <不太容易考>

> 如果出了题目还不如直接 n 次 Dijkstra 算法

9 第 8 章 查找

9.1 8.1 查找的基本概念

查找表的概念：由同一类型的数据元素所组成的集合

对查找表基本操作：

静态查找：

- 查询某个 特定的 数据元素是否在查找表中
- 检查某个 特定的 数据元素的公众属性

动态查找：

- 在查找表中插入一个数据元素
- 从查找表中删除某个元素

查找时间效率的度量 —— 很重要

平均查找长度，（平均查找长度）

总查找长度 / 总字符数

9.2 8.2 静态表的查找

9.2.1 8.2.1 顺序查找

概念很简单

但是要注意平均查找长度（ASL）的计算

对于查找表的效率，一般通过平均查找长度(ASL)进行衡量，此处的平均查找长度是指为确定元素在查找表中的位置进行执行关键字比较次数的平均值。对于有 n 个元素的查找表，查找成功的平均比较次数如下：

$$\mathrm{ASL}_{\mathrm{succ}}=\sum_{i=0}^{n-1}p_i c_i$$

其中，p_i 为查找表中查找第 i 个对象的概率，满足条件 $\sum_{i=0}^{n-1}p_i=1$，c_i 是查找到第 i 个对象所需关键字的比较次数。对于前面的顺序查找，查找第 i 个元素要比较 $i+1$ 次，所以 $c_i=i+1$，这时

$$\mathrm{ASL}_{\mathrm{succ}}=\sum_{i=0}^{n-1}p_i\cdot(i+1)$$

如果是等概率查找，也就是 $p_i=\frac{1}{n}$，这时有

$$\mathrm{ASL}_{\mathrm{succ}}=\sum_{i=0}^{n-1}\frac{i+1}{n}=\frac{n\cdot(n+1)}{2}\cdot\frac{1}{n}=\frac{n+1}{2}$$

也就是对于顺序查找，在等概率的情况下平均查找长度为 $\frac{n+1}{2}$。

9.2.2 8.2.2 有序表的查找 <常用的就是我们所说的二分法>

> 如果对于有序数组 这个就是最好的

折半查找算法很好实现 一定要注意始终折半操作的是下标而不是数值本身

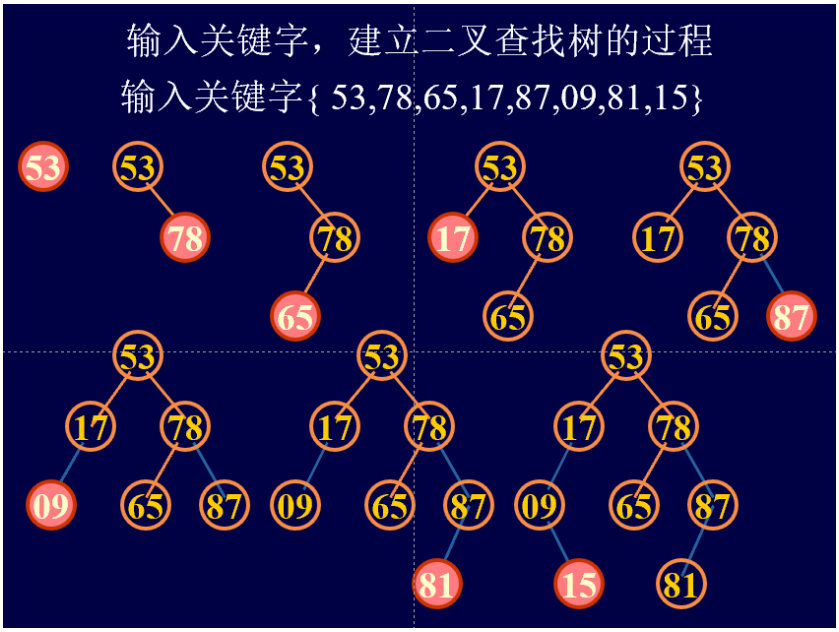
判定树 其平均查找长度为

$$\frac{(\log_2 n-1)\cdot n/2+1}{n}\leqslant \mathrm{ASL}_{\mathrm{succ}}(n)<\frac{2n\log_2 n+1}{n}$$

9.3 8.3 动态查找表 操作比较麻烦

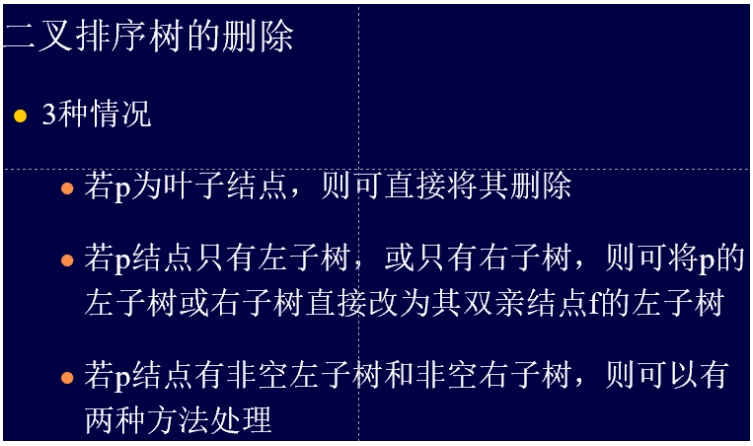
9.3.1 8.3.1 二叉排序树

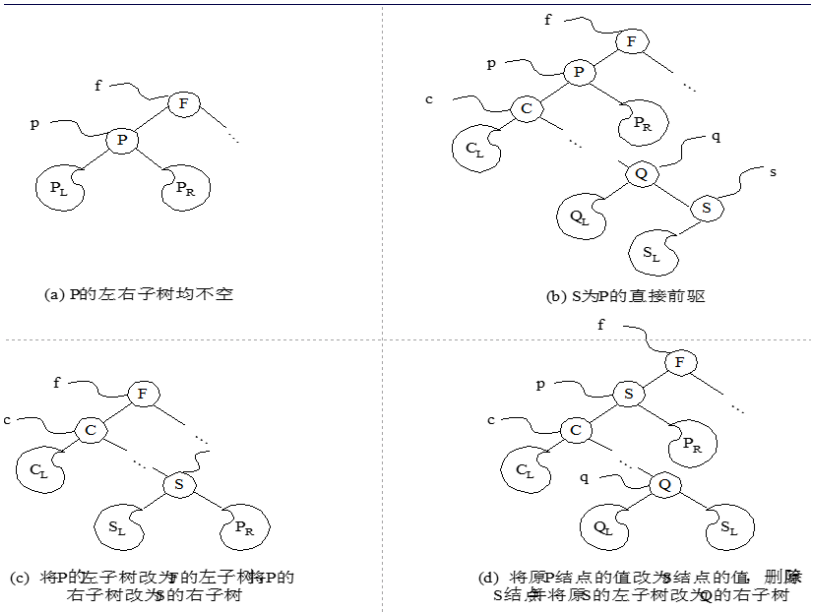
创建二叉排序树



ASL怎么算（必考），对于一个特定的问题，只要数一数就可以了 按照定义直接算。注意不同的构造方法 ASL 可能不一样。

删除结点问题比较重要——不经常考<基本上不会考>





9.3.2 8.3.2 二叉平衡树 AVL -- 这个是二叉排序树的进阶

性质： 每一个结点 的高度差 <= 1 平衡因子 -1 0 1

平衡化旋转：

- 单旋转
  - 左旋
  - 右旋
- 双旋转
  - 左旋加右旋
  - 右旋加左旋

看例子 必须会构造 —— 考的概率不高 但还是要掌握

> 核心是找到最近的可能失衡的点

9.3.3 8.3.3 B树 B+ 树

9.4 8.4 散列表 重点！！！！一定要掌握

> 计算式的查找

9.4.1 8.4.1 散列表的概念

无非就是创建了一个散列表H，H(Key) 为key对应的地址

冲突： H(k1) == H(k2) 这种现象叫做**冲突**

哈希表：函数 + 解决冲突的方法

两大问题

- 怎么选函数？
    - 本节是直接给出映射规则
        - 平方取中法
        - 除留余数法(**最重要**)
- 出现了冲突该怎么办？
    - 只有两种处理方法

9.4.2 8.4.2 构造散列函数的方法

> 基本上用不着自己构造

平方取中 最常用

除留取余 —— 首选

随机数法

直接定址法

9.4.3 8.4.3 处理冲突的方法 最重要的考点但其实也很简单

> 为冲突的找一个新的地址

开放定址法

无非就是取余若冲突再加一取余 以此类推

- 线性再探测

    > 很简单 ppt上有例子 必考点！！！
    >
    > 会构造并会画结构 + 会算 ASL
    >
    > 指定除数 就按照除数算 没有指定 就尽量大一点

- 随机探测法

链地址法

> 更是重点 必考点 去年考了 今年应该也会考

very easy 太好构造了

哈希表的查找及其分析

装填因子也要会算

10 第九章 内部排序

> 每一种排序算法 都要会做 特别特别重要
>
> 每一个算法的时间复杂度和空间复杂度都要 背下来

10.1 9.1 概述

将无序的数据变为有序状态

就是将数据不断地比较和移动 但由于比较和移动的方式不同可以分为不同的排序方法。

搞清楚学会分类

- **插入：和打扑克类似**

  将无序子序列中的一个元素“插入”到有序序列中，从而**增加**元素的有序子序列的长度。

- **交换：理解冒泡排序**

  交换无序序列中的元素从而得到其中关键字最小或最大的元素，并将它加入到有序子序列中，从而增加元素的有序子序列长度。

- **选择：**

  选一个最小的放在某块空间。

- **归并：** 唯一一个可以用在外部排序中的

  将多个已经有序的序列进行排列

- **基数：** 最特殊的一种

  连比较都不需要了，只需要分配和收集来进行排序。

排序算法的最重要的属性：**稳定性** 排序的效果，而非严格的数学证明<背住结论>

排序算法的效率：

- 看比较次数
- 移动次数
- 辅助存储空间

大的区分：

- 内部排序
- 外部排序

10.2 9.2 插入排序

10.2.1 9.2.1 直接插入排序 <一般来说不会考>

就像打扑克一样

性能：

- 稳定
- 时间复杂度 O（n方）

改进的思路：增大组数 减少直接插入排序的数据

10.2.2 9.2.2 Shell 排序 <一定要记得如和操作的>

> 缩小增量排序

Step 1 取 d 一般取 d = n/2

Step 2 然后再缩小间隔 d 例如 d = d/2 重复上述的分组并进行插入排序 直到 d = 1

> ppt上有相关例子
>
> 有时间看一下代码怎么写的

10.3 9.3 交换排序

最简单的就是**冒泡排序**

最nb的就是快速排序

10.3.1 9.3.2 快速排序（最优秀之一）

Step 1 从记录序列中任取一个元素作为枢轴

性能：

- 不稳定
- O(ln<2n>)

- 很适合原始性能杂乱无章的情况

还有很多的方法 可以让快速排序变得更优秀

10.4 9.4 选择排序

简单选择排序

- 操作很简单
- 是简单排序里面唯一一个不稳定的

堆排序

学会手工**筛选** 这个很重要

每一次只需要保证这一个节点的堆为目标堆就行

10.5 9.5 归并排序——唯一能用于外部排序的算法

基本上不用手工操作（递归算法实现 前后切分）

是稳定的 没有交换位置 直接继承下来的

10.6 9.6 基数排序——关键的狠

并不是说不比较了，而是说比较其他的

各个排序数字之间的不比较 这是本质的不同

10.6.1 9.6.1 多关键字排序

就像扑克牌一样 有一个高位关键字

通常来讲选择最低位优先法 LSD

其实就只有两个操作 = **分配** + **收集**

10.6.2 9.6.2 链式基数排序

一般用 r 表示关键字的基数

用 d 表示比较的位数

在ppt上有清晰的过程

10.7 9.7 排序性能比较

表 9.1 各种排序方法性能

排序分类	排序名称	时间复杂度		辅助空间	稳定性
		平均时间	最坏时间		
简单排序	直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
	起泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
	简单选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
高级排序	快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
	归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
其他排序	Shell 排序	由增量序列确定		$O(1)$	不稳定
	基数排序	$O(d(2n+r))$	$O(d(2n+r))$	$O((n+r)n)$	稳定

10.8 9.8 外部排序

基本不会考 了解一下即可

##

不仅仅要去理解算法本身 还要学会编写算法

实验课

1　大作业

1.1　计算器

	+	-	*	/	^	%	(	)	=	s	S	C	T	l	e
+					<	<				<	<	<	<	<	<
-					<	<				<	<	<	<	<	<
*					<	<				<	<	<	<	<	<
/					<	<				<	<	<	<	<	<
^	>	>	>	>	<	>	<	>	>	>	<	<	<	<	<
%	>	>	>	>	<	>	<	>	>	<	<	<	<	<	<
(					<	<				<	<	<	<	<	<
)					>	>				>	>	>	>	>	>
=					<	<				<	<	<	<	<	<
s	>	>	>	>	<	>	<	>	>	<	<	<	<	<	<
S	>	>	>	>	>	>	<	>	>	<	<	<	<	<	<
C	>	>	>	>	>	>	<	>	>	<	<	<	<	<	<
T	>	>	>	>	>	>	<	>	>	<	<	<	<	<	<
e	>	>	>	>	>	>	<	>	>	<	<	<	<	<	<
l	>	>	>	>	>	>	<	>	>	<	<	<	<	<	<

乘方先算后面

取余先算前面