数据结构

#0. Abstract

数据结构

基础数据结构 **线性结构** 线性表
栈
队列
串
非线性结构 数组
广义表
树
二叉树
图

应用数据结构 查找
内部排序

#1. Introduction

第一章 绪论
1.1 数据结构及其讨论的范畴
1.2 基本概念和术语
1.3 抽象数据类型
1.4 算法和算法分析

1.1 数据结构及其讨论的范畴

1. 数据结构的兴起和发展

客观世界与计算机世界的关系
利用计算机解决问题的过程
计算机求解问题-数值问题、非数值问题

2. 非数值计算问题

表结构
数据结构
图结构

3. 什么是数据结构

数据结构是一门研究**非数值计算**的程序设计问题中计算机的操作对象（数据）间的**逻辑结构**和**物理结构**以及它们之间相互关系，并对这种结构进行定义和实现相应运算的学科

数据结构课程主要讨论：
数据的**逻辑结构**，指数据元素之间的逻辑关系
数据的**存储结构**，指数据元素及其关系在计算机存储器中的存储方式，也成为数据的物理结构
数据的**运算**，指施加在数据上的基本操作

1.2 基本概念和术语

1. 基本概念

数据
数据元素
数据项
数据对象
数据结构

2. 数据的逻辑结构

数据的逻辑结构：数据元素之间的抽象关系，数据元素之间**逻辑关系**的整体

数据的逻辑结构是从具体问题抽象出来的**数据模型**

数据结构从逻辑上分为四类，即四种基本的逻辑结构：
集合结构：数据元素属于同一个集合
线性结构：数据元素属于一对一的线性关系
树结构：数据元素属于一对多的层次关系
图结构：数据元素属于多对多的关系

3. 数据的存储结构

数据的存储结构：
又称**物理结构，**是数据及其逻辑结构在计算机中的表示
实质是内存分配，以确定元素及元素之间关系的表示

两种基本的存储结构：
顺序存储结构：用一组**连续**的存储单元依次存储数据元素，数据元素之间的逻辑关系由**元素的存储位置**来表示
链接存储结构：用一组**任意**的存储单元存储数据元素，数据元素之间的逻辑关系用**指针**来表示

4. 数据的运算

5. 逻辑结构与存储结构的关系

数据的逻辑结构属于**用户视图**，是**面向问题**的，反映了数据内部的构成方式；

数据的存储结构属于具体**实现视图**，是**面向计算机**的。

一种数据的逻辑结构可以用多种存储结构来表示，而采用不同的存储结构，其数据处理的效率往往是不同的

任何一个**算法的设计**取决于选定的数据（逻辑）结构，而**算法的实现**依赖于采用的存储结构

1.3 抽象数据类型

1. 数据类型
2. 抽象数据类型（ADT）

定义：一个数学模型和在该模型上定义的操作集合的总称
ADT 的定义仅取决于它的一组**逻辑特性**，而与其在计算机内部如何表示和实现无关。
ADT 是程序语言中数据类型概念的进一步推广和进一步抽象
抽象的意义在于数据类型的数学抽象特性
同一数学模型上定义不同的操作集，则代表不同的 ADT

抽象数据类型形式化定义：ADT（D,S,P）
D 是数据对象
S 是 D 上的关系集
P 是对 D 的基本操作集

定义形式：
ADT 抽象数据类型名{
数据对象：…
数据关系：…
基本操作：…
} ADT 抽象数据类型名

3. 数据类型、数据结构和 ADT

数据类型、数据结构和 ADT 相互关系
数据类型是根据数据结构分类的，同类型的数据元素的数据结构相同
数据结构则是抽象数据类型中数学模型的表示
ADT 是数据类型的进一步推广和进一步抽象

1.4 算法和算法分析

1. 算法的概念

算法是解决特定问题求解步骤的描述

2. 算法设计的原则

3. 算法分析

时间复杂性
空间复杂性

算法的时间成本

一个算法是由**控制结构**（顺序、分支和循环）和**原操作**（指固有数据类型的操作）构成的，算法时间取决于两者的综合

比较同一问题的不同算法，通常选取一种对于所研究的问题来说是基本操作的原操作进行分析

一般情况下，算法中基本操作重复执行的次数是问题规模 n 的某个函数 f(n)

4. 时间复杂度分析

渐进分析：大 O 记号

若存在 c>0，当 n>=n0 时都满足 **T(n)<=c*f(n)**，则 T(n)=O(f(n))表示随着问题规模 n 的增大，算法执行时间的增长率与 f（n）的增长率相同，成为算法的**渐进时间复杂度**，简称**时间复杂度**

当问题规模充分大时在渐进意义下的阶。称 f(n)是 T(n)的**同数量级函数。**一般情况下，**只求出 T(n)的最高阶（数量阶），忽略其低阶项和常系数**，这样既简化 T(n)的计算，又能比较客观地反映出 n 很大时算法的时间特性

最坏情况与平均情况

如果问题规模相同，时间代价与**输入数据（的分布）**有关，则需要分析**最好情况、最坏情况、平均情况**

常数阶 O（1）

一个没有循环的算法的基本运算次数与问题规模 n 无关，记作 **O（1）**，也称作**常数阶**

对数阶 O（logn）

例：对于任意非负整数，统计其二进制数位为 1 的总数

线性阶 O（n）

一个只有一重循环的算法的基本运算次数与问题规模 n 的增长呈线性增大关系，记作 O(n)

其他常用阶

O（nlogn）近似线性阶//归并排序算法

O（n^2）二次阶 双重循环

O（n^3）三次阶 三重循环

5. 时间复杂度的分析方法

时间复杂度的运算法则：

若两段算法复杂度 T1（n）=O（f1（n））和 T2（n）=O（f2（n）），则

加法规则：

T1(n)+T2(n)=max(O(f1(n)),O(f2(n)))

乘法规则：

T1(n)*T2(n)=O(f1(n))*O(f2(n))

时间复杂度的分析方法：首先求出程序中各语句、各模块的运行时间，再求整个程序的运行时间

各种语句和模块分析应遵循的规则

（1） 赋值语句或读/写语句

运行时间通常取 **O(1)**

有函数调用的除外，此时要考虑函数的执行时间

（2） 语句序列

运行时间由加法规则确定，即该序列中**耗时最多的语句**的运行时间

（3） 分支语句

运行时间由条件测试（通常为 O（1））加上分支中**运行时间最长的语句**的运行时间

（4） 循环语句

运行时间是对输入数据重复执行 **n 次循环体所耗时间的总和**

每次重复所耗时间包括两部分：一是循环体本身的运行时间；二是计算循环参数、测试循环终止条件和跳回循环头所耗时间

（5） 函数调用语句

若程序中只有**非递归调用**，则从没有函数调用的被调函数开始，计算所有这种函数的运行时间。然后考虑有函数调用的任意一个函数 P，在 P 调用的全部函数的运行时间都计算完之后，即可开始计算 P 的运行时间

#2. LinearList

第二章 线性表

2.1 线性表的类型定义

2.2 线性表的顺序表示和实现

2.3 线性表的链式表示和实现

2.4 一元多项式的表示

2.1 线性表的类型定义

1. 线性表的类型定义

线性表是 n（n>=0）个类型相同数据元素的有限序列，通常记作(a1,a2,a3,a4…ai-1,ai,ai+1…an)

ai 是数据元素，n 是表的长度

各元素由“编号唯一确定

一般要求，同一序列中元素类型相同

线性结构的基本特征

集合中必存在**唯一**的一个“**第一元素**“

集合中必存在唯一的一个“**最后元素**“

除最后元素在外，均有唯一的**后继**

除第一元素之外，均有唯一的**前驱**

2. 线性表的 ADT

3. 基本操作的应用

合并 a*b

归并 a+b

2.2 线性表的顺序表示和实现

1. 线性表的顺序表示和实现

顺序表—线性表的顺序存储

线性表的顺序存储指用一组地址连续的存储单元依次存储线性表的数据元素

存储结构特点

存储单元地址连续（需要一段连续空间）

逻辑上相邻的数据元素其物理位置也相邻

存储**密度大**（存储空间全部用于存储元素）

随机存取：元素序号与存储位置存在如下关系 LOC(ai)=LOC(a1)+(i-1)*C

2. 基本操作的实现：顺序表

查找 O(L.length)

插入 O(n)

删除 O(n)

3. 顺序表小结

顺序表优点

不需要额外的存储空间来表示元素间的逻辑关系

可以随机地存取表中的任意一个元素

顺序表缺点

插入和删除元素时要移动大量的元素

插入元素时必须事先分配足够的空间

2.3 线性表的链式表示和实现

1. 链表—线性表的链式存储

一个线性表由若干个结点组成，每个结点至少含有两个域：**数据域**和**指针域**，由这样的节点形成存储的线性表称作**链表**

存储结构特点

逻辑次序和物理次序**不一定相同**

元素之间的**逻辑关系**用指针表示

需要**额外**空间存储元素之间的关系

非随机访问存取结构（顺序访问）

2. 单链表的表示和实现

每个结点只包含一个指针域的链表称作单链表

一般情况下，单链表中的每个结点存储其直接**后继节点**的位置信息

指向单链表第一个结点的**头指针**可确定一个单链表

线性表最后一个数据元素没有后继，指针为**空指针 NULL**

带头结点的单链表

带头结点的空链表

头结点是链表的第一个结点，其指针域指向第一个元素结点，其**数据域可以存储链表的其他信息**，如链表的长度、链表的说明等信息，也可以不存储任何信息

头结点作用：空表和非空表表示统一；使对第一个元素的操作与对其他元素的操作保持一致

注意：是否带表头结点在存储结构定义中无法体现，由操作决定

单链表上的查找运算 O(ListLength(L))

单链表上的插入运算 O(ListLength(L))

单链表上的**删除**运算 O(ListLength(L))

生成链表：头插法

链表是一个动态的结构，它不需要预分配空间，因此生成链表的过程是一个**结点逐个插入**的过程

头插法：逆位序输入 n 个数据元素的值，建立带头结点的单链表

生成链表：尾插法

尾插法：顺序输入 n 个元素的值，建立带头结点的单链表

单链表的运算实例

两个单链表归并 O（a+b）

单链表小结

结点中只设一个指针域（指向当前元素的后继）

用头指针标识一个单链表（可能含头结点）

在单链表中只能按顺序访问元素（顺序存取）

顺序查找是最基本的操作，**插入、删除**操作在查找的基础上仅需修改结点间的链接关系

优点

插入、删除时无须移动元素，只需修改指针

根据需要申请存储空间，且不要求连续的存储空间

缺点

对表中的元素只能进行顺序访问，查找第 i 个元素的时间复杂度为 O（n）

用指针指示元素之间的逻辑关系（直接前驱、后继），存储空间利用率低

3. 循环链表

将单链表中终端结点的指针端由空指针改为指向**头结点**，就使整个单链表形成一个循环，这种头尾相接的单链表称为**单链的循环链表**，简称**循环链表**

从任一结点出发、沿任一个方向都可以到达所有结点

存储结构及插入、删除和查找等操作，与单链表基本一致

尾指针

设两个链表 La 和 Lb，讨论：

*La、Lb 都是带头结点的单链表

实现将 Lb 接在 La 之后 O（a）

* La、Lb 都是带头结点头指针的单循环链表

实现将 Lb 接在 La 之后形成一个循环链表 O（a+b）

* La、Lb 都是带头结点尾指针的单循环链表

实现将 Lb 接在 La 之后形成一个循环链表 O（1）

带尾指针的循环单链表的特点

*设立尾指针 tail（可以不用头指针和头结点）

*尾结点的指针域指向表头结点，快速得到表头指针（tail->next）

*操作和单链表基本一致，主要差别在于**循环条件判别**不是（p）或（p->next）是否为空，而是它们**是否等于表头指针**

*对于两个表的简单合并，可通过简单操作完成 O(1)

应用实例

约瑟夫问题

4. 双向链表

结点结构

一个链表的每一个结点含有两个指针域：一个指针指向其前驱结点，另一个指针指向其后继结点

双向链表的 C 语言实现

双向链表中地址关系：

*在双向链表中：

p->prior 指向 p 的前驱，p->next 指向 p 的后继

以下都是 p 结点的地址：

p->prior->next, p->next->prior

*双向链表的操作特点

查询和单链表相同

插入和删除操作需要同时修改两个方向上的指针

5. 双向循环链表

带头结点的空双向循环链表

带头结点的非空双向循环链表

2.4 一元多项式的表示

1. 一元多项式的表示

* P(x)=p0+p1x+p2x^2+…pnx^n 表示为序列 P=（p0,p1,…,pn）

* 加法

若 P=（p0,p1,…,pn），Q=（q0,q1,…,qm），m<n

则 P+Q=（p0+q0,p1+q1,…,pm+qm,pm+1,…,pn）

可采用顺序存储结构来实现

*空间浪费

多项式的次数不确定

不同多项式的次数可能相差悬殊

多项式的次数很高，但大多数项的系数为 0

*在用**顺序表存储**时，如果是稀疏多项式则浪费空间，可改用**链式存储**

只保留非零项

Pn(X)=p1x^e1+p2x^e2+…+pmx^em

pi 是指数为 ei 的项的非零系数

Pn 表示为长度为 m 且每个元素有两个数据项的线性表：

((p1,e1),(p2,e2),…,(pm,em))

*存储空间

最坏情况：n+1=m 个系数均不为零，则存储加倍

一般情况：存储空间大大节省//稀疏

#3. Stack & Queue

第三章 栈和队列

3.1 栈

3.2 栈的应用

3.3 递归

3.4 队列

3.1 栈

1. 栈的定义

栈是限定仅在表尾进行插入、删除操作的线性表

栈的术语

栈顶：允许插入和删除的一端//栈的表尾

栈底：是栈顶的另一端//栈的表头

空栈：不含任何数据元素的栈

入栈：在栈顶压入元素

出栈：从栈顶弹出元素

栈的特点

后进先出

2. 栈的抽象数据类型

3. 栈的表示和实现：顺序栈

栈的初始化

入栈操作

如果栈满，给栈增加容量

将数据存入栈顶位置栈顶后移一位

*S.top++=e;//*S.top=e;S.top++

出栈操作

e=*--S.top;//S.top--,e=*S.top

取栈顶元素操作

4. 栈的表示和实现：链栈

链栈的实现与链表的实现基本相同

注意：链栈中指针的方向以及栈顶和栈底的位置

3.2 栈的应用

1. 数制转换
2. 括号匹配的检验
3. 行编辑程序
4. 迷宫求解
5. 表达式求解

3.3 递归

递归函数：一个直接调用自己或者通过一系列的调用语句间接地调用自己的函数

递归实现需要有**边界条件、递归前进段和递归返回段**

当边界条件不满足时，递归前进（压栈）
当边界条件满足时，递归开始返回（出栈）

汉诺塔问题

时间复杂度 O（2^n）

优点

递归易于实现、代码简洁、易于理解
例如在树的遍历算法中，递归地实现明显要比循环简单的多

缺点

空间效率低

调用栈可能会溢出，其实每一次函数调用会在内存栈中分配空间，而每个进程的栈的容量是有限的，当调用的层次太多时，就会超出栈的容量，从而导致栈溢出

时间效率低

递归由于是函数调用自身，而函数调用是有时间和空间的消耗的；每一次函数调用，都需要在内存栈中分配空间以保存参数、返回地址以及临时变量，而往栈中压入数据和弹出数据都需要时间

递归中很多计算都是重复的，由于其本质是把一个问题分解成两个或多个小问题，多个小问题存在相互重叠的部分，则存在重复计算，如斐波那契数列的递归实现

建议

在讲求效率时，最直接的办法是将**递归**改写为**迭代**模式

3.4 队列

1. 队列

仅限定在表尾进行插入和表头进行删除操作的线性表

*队尾 队列的表尾，即只允许插入的一端
*队头 队列的表头，即只允许删除的一端
*入队 向队尾插入元素
*出队 从表头删除元素

队列的存储结构和实现

*链队列 队列采用链式存储结构
*循环队列 队列采用顺序存储结构

2. 链队列

队列采用链式存储结构（单链表）

3. 链队列的基本操作

链队列初始化
入队
出队

4. 顺序队列

顺序队列不循环时的问题：**队列假溢出**

5. 循环队列

循环队列是顺序队列的一种特例，它是把顺序队列构造成一个**首尾相连**的**顺序表**，指针和队列元素之间关系不变

*循环队列初始化
*循环队列入队
*循环队列出队

队列空队列满判断

方法一 设置一个队列长度计数器
Q.length

方法二 令队列空间中的一个单元闲置，使得在**非空**队列中，**Q.rear 与 Q.front 不同**

*队满: (Q.rear+1) & maxsize == Q.front
*队空: Q.rear == Q.front
*队列长度:
（Q.rear-Q.front+MAXSIZE）%MAXSIZE
*队尾元素:
Q.base[（Q.rear-1 +MAXSIZE）%MAXSIZE]

#4. String

4.1 串的定义
4.2 串的表示和实现
4.3 串的模式匹配算法

4.1 串的定义

串是由零个或多个字符组成的有限序列

***串的长度**：串中所包含的字符个数
***空串**：长度为“零”的串，记为“”
***非空串**记为：s=”a1 a2…an”

s 是串名，双引号是定界符，用括号括起来的字符序列是串值

字符集：ASCII 码、Unicode 字符集

***子串**：串中任意个连续的字符组成的子序列
***主串**：包含子串的串
***串相等**：只有当两个串的长度相等，并且各个对应位置的字符都相等
***空格串**：由一个或多个空格组成的串。要和“空串”区别，空格串有长度就是空格的个数

串的操作通常以**串的整体**为操作对象，操作的一般都是子串；

线性表则以**单个元素**为操作对象

4.2 串的表示和实现

1. 定长顺序存储表示

*类似于**线性表**的存储结构
*用一组地址连续的存储单元存储串值的字符序列
*按照预定义的大小，为每个定义的串变量分配一个**固定长度**的存储区

截断：串的实际长度可在预定义长度的范围内随意，超过预定义长度的串值则被舍去

串的联接操作
求子串操作
字串的删除操作

2. 堆分配存储表示

采用不限定串长的最大长度//动态分配串值的存储空间

堆分配存储表示

*仍以一组**地址连续**的存储单元存放串值字符序列
*其存储空间是**动态分配**的，在C语言中，存在一个称之为堆的自由存储区，可利用函数 malloc()为串分配一块实际串长所需的存储空间
*串操作仍是基于字符序列的复制来完成

串插入操作
串联接操作
求子串操作

定长顺序存储结构

*如果出现串值序列的长度超过上界，约定用截尾法处理
*这种情况不仅在联接串时可能出现，也可能出现于插入、置换等操作中

堆分配存储结构

*动态分配存储空间，克服了顶层顺序存储限定串长的缺点
*既有顺序存储结构的特点，处理方便，操作中对串长又没有任何限制，更显灵活

3. 串的块链存储表示

链表存储字符串

*字符串本身就是一个线性表，可以用链表存储

*如果每个结点存储一个字符，如用 32 位地址，字符按 8 位记，则存储密度：

存储密度 = 数据元素所占存储位 / 实际分配的存储位

存储密度 = 8/40 = 20%

*采用普通链表存储字符串，存储密度小，存储占用量大

串的块链存储：一个结点存储多个字符

*输入字符串："ILOVESUFE"，设链表结点大小为 4，则：

*最后一个结点不全被串值占满时可补上#

串的块链存储 C 语言实现

*对串操作，一般只需从头向尾顺序扫描即可，无需双向链表

*设置尾指针可方便进行串的联接操作

*对一些特定的操作（如联接），采用链式存储结构可方便处理

*链式存储结构需占有大量的存储空间，且操作复杂

*定长顺序存储与堆分配存储方式更灵活

4.3 串的模式匹配算法

1. 串的模式匹配

子串定位运算**又称为**模式匹配**或**串匹配

*在串匹配中，一般将**主串称为目标串**，**子串称为模式串***

Index(S, T, pos)

初始条件：主串 S 和模式串 T 存在，T 非空，1<=pos<=StrLength(S)

操作结果：若主串 S 中存在和串 T 值相同的子串（匹配成功），返回它在子串 S 中第 pos 个字符之后第一次出现的位置；否则（匹配不成功）函数值为 0

例如：S="abcdefg", T="cde", pos=1

匹配成功，返回位置 3

2. 朴素模式匹配算法

朴素模式匹配算法-Brute-Force 算法（BF 算法）

BF 基本思想（回溯法）

(1) 从主串 S 的第一个字符开始和模式 T 的第 1 个字符进行比较，若相等，则继续比较两者的后续字符；否则，从主串的第 2 个字符开始和模式 T 的第 1 个字符进行比较

(2) 重复上述过程，直到 T 中的字符全部比较完毕，说明本串匹配成功；或 A 中字符全部比较完，则说明匹配失败

BF 算法时间复杂度（第 1 遍就匹配成功）

*设 S 串长为 n，T 串长为 m，pos=1

S: a b c d e f g h j k l l k c d e

T: a b c

*仅需比较 T 与 S 的前 m 个字符，总比较次数为 **m** 次

*第一遍就匹配成功时时间复杂度为 O(m)

BF 算法时间复杂度（每遍失败，失配均发生在 T 的第 1 位）

*设 S 串长为 n，T 串长为 m，pos=1：

S: a b c d e f g h j k l l k c d e

T: l m n

*假设从第 i 个位置开始匹配成功，前 i-1 次共比较了 i-1 次，第 i 次比较了 m 次，共比较了 i+m-1 次，总的执行次数为：

*此时，平均时间复杂度为 O（n+m）

BF 算法时间复杂度（每遍失败，失配均发生在 T 的最后 1 位）

*设 S 串长为 n，T 串长为 m，pos=1:

S: 00000000000000001

T: 0000001

*若第 i 趟比较成功，共比较了多少次？前 i-1 趟每次比较 m 次，第 i 趟比较 m 次，共 i*m 次，因为每次均回溯，共比较了（n-m+2）m/2

*此时，时间复杂度为 O（(n-m+1）*m）若当 n>>m，故略去 m^2 与 m，时间复杂度为 O（n*m）

3. 改进的模式匹配算法

讨论一：主串 S 游标 i 指向 5，子串 T 中游标 j 指向 5，按照朴素模式匹配算法两处相等时 j 回到 1，i 回到 2，继续比较，分析在这种情况下的效率

主串 S：abcdccdde

子串 T：abcde

(1) 通过匹配得知：S{1.2.3.4}=T{1,2,3,4}

(2) 已知：T[1]≠T[2]≠T[3]≠T[4]

(3) 那么 T[1]≠S[2], T[1]≠S[3], T[1]≠S[4]

因此按照朴素模式匹配方法让 i 回到 2，j 回到 1 位置进行比较会发生重复比较，影响效率

讨论二：主串 S 游标 i 指向 8，子串 T 中 j 指向 8，按照朴素模式匹配算法两处不相等时 j 回到 1，i 回到 2，继续比较，分析在这种情况下的效率

主串 S：abcdabcdabcfe

子串 T：abcdabcf

(1) 通过匹配得知：S{1…7}=T{1…7}

(2) 已知：T[1…3]=T[5…7]，T[1]T[2]T[3]≠T[4]，T[5]T[6]T[7]≠T[4]

(3) 那么 T[1]≠S[2]，T[1]≠S[3]，T[1]≠S[4]，此时若让 i 回 2，j 回 1 会发生重复的比较，影响效率，由于 T 中有重复子串 T[1…3]=T[5…7]，在第一次比较时已得 T[5…7]=T[5…7]，那么 T[1…3]=S[5…7]，即让 i 回到 5，j 回到 1 也是会发生重复的比较，影响效率

BF 算法时间性能低：在每遍匹配不成功时存在大量回溯，没有利用已经部分匹配的结果

*利用部分匹配的结果，主串 S 指针将完全不必回退，若比对成功，则前进一个字符，否则，可直接确定应与 T 中哪个字符继续比对

*改进的算法思想：主串指针不回溯，模式串向后滑动至某个位置上

KMP 算法

*思想：利用已经得到的"部分匹配"的结果，**避免主串回退**

(1) 依次比对当前字符对 S[i]和 T[j]

(2) 若 T[i]==T[j],主串模式串都前进一个字符

(3) 否则，指示串回退到某个指示的位置//主串不回退

(4) 直到整个模式串匹配

主串指针不回溯，如何确定模式串滑动到某个位置？

(1) 根据模式串 T 本身即可确定

(2) 根据 T，构造一张查询表 next[]

*当 S[i]≠T[j]时

*已经得到的结果 S[i-j+1…i-1]=T[1…j-1]

*若已知 T[i…k-1]=T[j-k+1…j-1]

*则有 S[i-k+1…i-1]=T[1…k-1]

模式串的 next[]数组值计算

表明当模式中第 j 个字符与主串中相应字符"失配"时，在模式中需重新和主串中该字符进行比较的字符的位置

next[j]=

0 当 j=1 时

Max{k|1<k<j 且 T[i…k-1]=T[j-k+1…j-1] 此集合不空时

1 其他情况

例：计算 T=abcabx 的 next[]

当 j=1 时，按照定义地 next[1]=0

当 j=2 时，j 由 1 到 j-1 则只有一个字符

a，属于其他情况，next[2]=1
当 j=3 时，j 由 1 到 j-1 是 ab，显然 a≠b，属于其他情况，next[3]=1
当 j=4 时，j 由 1 到 j-1 是 abc，显然 a≠b≠c，属于其他情况，next[4]=1
当 j=5 时，j 由 1 到 j-1 是 abca，得到 T[1]=T[4]，由定义推出 k=2，next[5]=2
当 j=6 时，j 由 1 到 j-1 是 abcab，得到 T[1,2]=T[4,5]，由定义推出 k=3，next[6]=3
可得到 next[j]=011123

讨论：模式匹配 T=aaab 与主串 S=aaabaaaab

*模式串 T 中含有重复子串，可先计算其 next[]数组值，next[j]=01234
*i=j=4 时：T[4]≠S[4]，j 回退到 next[4]=3，此时发现 T[3]≠S[4]
*继续回退到 j=next[3]=2，发现 T[2]≠S[4]
*最后回退到 j=next[2]=1，发现 T[1]≠S[4]
*此时，i++，比较 i=5 和 j=1
问题以上几次 j 的回退，其实是没有必要的、多余的比较

计算 next[]数组的改进

*由于模式串 T 中 T[4]=T[3]=T[2]=T[1]，且有意义的操作是：i=5 与 j=1 位置比较，即实际可将 next[4]，next[3]，next[2]的值赋为 next[1]的值
*因此，将 next[]求值改进为求解一个新数组的值，记为 **nextval[]**

BF 算法

*时间复杂度 O（m*n）
*在一般情况下的执行时间复杂度近似降为 O（m+n）
*多数情况下较为常用

KMP 算法

*时间复杂度（m+n）
*特点：充分利用以往的比对所提供的信息
*适用于模式与主串之间**存在若干部分匹配**的场合
*可有效处理外部设备输入的庞大文件匹配，而无需回头重读

#5.Array & GList

5.1 数组
5.2 数组的顺序表示和实现
5.3 矩阵的压缩存储
5.4 广义表

5.1 数组

数组是我们很熟悉的一种数据结构，可以将它看作线性表的推广
*是由**下标和值**组成的序对的序列
*也可以定义为是由**相同类型**的数据元素组成有限序列
*每个元素受 **n（n≥1）个线性关系**的约束，每个元素在 **n 个线性关系**中的序号 i1,i2,…in 称为该元素的下标，并称该数组为 **n 维数组**

数组特点

*数组整体上可以看作一个线性表
*数组中的元素本身可以是具有某种结构的数据
*数组中的不同元素属于同一数据类型

5.2 数组的顺序表示和实现

数组一般不做**插入和删除**操作，数组一旦建立，结构中的元素个数和元素间的关系就不再发生变化。因此，一般采用**顺序存储**的方法来表示数组

数组的顺序存储

用一组连续的存储单元来实现（多维）数组的存储
高维数组可以看成是由多个低维数组组成的

需要将多维数组**映射到一维内存**
按某种次序将多维数组中的元素排成线性序列，然后将这个线性序列顺序存放在存储器中

二维数组的存储与寻址
N 维数组的存储与寻址

随机存储结构

数组中的任一元素可以在常量时间内存取，即顺序存储的数组是一个随机存储结构

只要知道**开始结点的存放地址（即基地址）、维数和每维的维界**，以及**每个数组元素所占用的单元数**，就可以将数组元素的存放地址表示为其下标的线性函数

5.3 矩阵的压缩存储

1. 特殊矩阵的压缩存储

***矩阵**是很多科学与工程计算问题中研究的数学对象
*用高级语言编程时，一般是用**二维数组**来存储矩阵元素
***特殊矩阵：**矩阵中很多值相同的元素或零元素在矩阵的分布具有一定的规律
常见的特殊矩阵有：**对称矩阵、三角矩阵、1 对角矩阵**等
***压缩存储**：为多个值相同的非零元素只分配一个存储空间；对零元素不分配空间
***特殊矩阵压缩存储的两个要素**
识别存储哪些矩阵元素
如何将这些矩阵元素映射到一维数组

对称矩阵

只存储上（或下）三角部分的元素

三角矩阵

*下三角矩阵是指矩阵的上三角（不包括对角线）中的元素均为常数或零的 n 阶矩阵。上三角矩阵反之
*下三角矩阵与对称矩阵相同，再加一空间存储常数 c 即可

对角矩阵

所有的非零元素都集中在以主对角线为中心的带状区域中

上述各种特殊矩阵，其非零元素的分布都是**有规律的**，因此总能找到一种方法将它们压缩到**一位数组**中

并且一般都能找到矩阵中的元素与该一维数组元素的对应关系，通过这个关系，依然能对矩阵中的元素**随机存取**

2. 稀疏矩阵的压缩存储

稀疏矩阵：矩阵中有很多特定值（如零）的元素

*分布没有规律
*在 m 行 n 列
*通常认为δ≦0.05 时为稀疏矩阵

稀疏矩阵的抽样数据类型

如何只存储非零元素

*稀疏矩阵中的非零元素的分布没有规律
*对稀疏矩阵进行压缩存储时，对于非零元素，必须同时记下它的值及其所在的位置（行号 i，列号 j）

将稀疏矩阵中的每个非零元素表示为**（行号，列号，非零元素值）-三元组表**

一个稀疏矩阵可由表示**非零元及矩阵的行列数**唯一确定

假设以**顺序存储结构**来表示三元组表，则可得到稀疏矩阵的一种压缩存储方法-**三元组顺序表**

如何求转置矩阵?

*一个 m*n 的矩阵 M，其转置矩阵 T 是一个 n*m 的矩阵，且 M[i][j]=T[j][i]
***不压缩存储的矩阵转置**
O（mu*nu）

采用**三元组顺序表**求转置矩阵
*在稀疏矩阵的三元组表中，非零矩阵元

素按行存放。当行号相同时，按列号递增的顺序存放

*稀疏矩阵的转置运算转化为对应三元组表的变换

*时间复杂度为：O（nu*tu），最坏 O（mu*nu^2）

此算法仅适用于稀疏矩阵 tu<<mu*nu 的情况

改进的稀疏矩阵转置算法-以时间换空间

*改进思路：若能预先确定 M 中每一列（即 T 中每一行）的第一个非零元在 T.data 中的恰当位置。那么在对 M.data 中的三元组依次做转置时，便可直接放到 T.data 中恰当的位置上去

*设两个向量：num 和 cpot

*num[col]表示矩阵 M 中第 col 列中的非零元个数

*cpot[col]表示 M 中第 col 列的第一个非零元在 T.data 中的恰当位置

cpot[1]=1

cpot[col]=cpot[col-1]+num[col-1]

2≦col≦M.nu

改进的稀疏矩阵转置算法

十字链表

当矩阵的非零元个数和位置在操作过程中变化较大时，适合采用链式存储结构来表示三元组的线性表

*每个非零元结点含有 5 个域

*i, j, e：非零元所在的行，列，非零元的值

*right, down 指针：链接同一行（同一列）中的非零元结点

*可用两个存储行链表和列链表的头指针的一维数组表示

5.4 广义表

1. 广义表

广义表是线性表的推广，也称列表

广义表是 n 个元素的有限序列，记作 **LS=（a1,a2,…an）**

其中 LS 是广义表的**表名**，n 是广义表的**长度**

ai 是广义表的元素，它可以是**单个元素**，也可以是**广义表**

*__原子__：ai 是单个元素，习惯用小写字母表示

*__子表__：ai 是广义表，习惯用大写字母表示

*__表头__：非空广义表的**第一个元素 a1**

*__表尾__：除表头外的**其余元素组成的广义表**

*__深度__：广义表中括号的**最大嵌套层数**

*__长度__：广义表中**元素的个数**

A=（）空表，长度为 1，深度为 1

B=（e）只含一个原子，长度为 1，深度为 1

C=（a,（b, c, d））有一个原子，一个子表，长度为 **2**，深度为 2

D=（B，C）两个元素都是列表，长度为 **2**，深度为 3//广义表可以共享

E=（a, E）是一个递归表，长度为 2，深度无限，相当于（a,（a,（a,（a,…））））

广义表的抽象数据类型

广义表 A=(a,b,c), B=(A,(c,d)), C=(a,(B,A)，（e,f）),写出下列各运算结果：

*Head(A)=a

*Tail(B)=((c,d))

*Head(Head(Head(Tail(C))))

=Head(Head(Head(((B,A),(e,f)))))

=Head(Head((B,A),(e,f)))

=Head(B)

=Head((A,(c,d)))

=A

*广义表是一个**多层次的线性结构**

例如：D=（E,F）

其中：E=（a,(b,c)）F=(d,(e))

2. 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（原子或列表），因此难以用顺序存储结构表示，常采用**链式存储结构**

结点结构

tag 标志域；hp 表头指针域，tp 表尾指针域

表结点：| tag=1 \| hp \| tp |

原子结点：| tag=0 \| atom |

构造存储结构

*空表 ls=NIL

*非空表 ls

tag=1 prt.hp 指向表头的指针

prt.tp 指向表尾的指针

*若表头为原子，则为

#6.Tree & Binary Tree

6.1 树的定义和基本术语

6.2 二叉树

6.3 遍历二叉树

6.4 树和森林

6.5 赫夫曼树及其应用

6.1 数的定义和基本术语

树是 n 个结点的有限集合（可以是空集），在任一棵非空树中：

*__有且仅有一个__成为根的结点

*其余结点可分为互不相交的有限集合，而且这些集合中的每一集合本身又是一棵树，称为根的**子树**

*根据树的**递归定义**，每个子树又由它的根和它的子树构成

树的抽象数据类型

树的逻辑结构特点

*树是一种**分枝**结构

*树中只有**根节点没有前驱**

*除根节点外，其余结点有**零个或多个**直接后继

*除根节点外，其余结点都有且仅有一个直接**前驱**

*除根节点外，其余结点都存在**唯一一条**从根到该节点的路径

*__一对多__的关系，反映了结点之间的层次关系

树型结构的应用实例

树可表示具有分支结构关系的对象。有些应用中数据元素之间并不存在分支结构关系，但是为了便于管理和使用数据，将它们用树的形式来组织。如：文件系统

树的其他表示形式

嵌套集合

广义表表示

基本术语

***结点的度：**结点拥有的**子树个数**

*__树的度__：树内各节点度的最大值

*__结点__：包含一个数据元素及若干指向其子树的分支

***结点的分类**

非终端结点或分支结点：度不为 0 的结点

内部节点：除根节点之外的分支节点

叶结点或终端结点：度为 0 的结点

*__孩子结点、双亲结点__：树中某结点子树的根节点称为这个结点的孩子结点（子节点），这个结点称为它孩子结点的双亲节点（父节点）

*__兄弟结点：__具有同一个双亲的孩子结点互称为兄弟

*__路径：__如果树的结点序列 n1,n2,…nk 有

如下关系：结点 ni 是 ni+1 的双亲（1≦i<k），则把 n1,n2,…nk 称为一条由 n1 至 nk 的路径

*__路径长度：__路径上经过的边的个数

*__结点的祖先：__从根到该节点所经分支上的所有结点

*__结点的层次：__根节点的层定义为 1，若某结点在第 k 层，则其子树的根节点在第 k+1 层

*__树的深度或高度：__树中结点的最大层次

*__有序树、无序树：__如果一棵树中结点的各子树从左到右是有次序的，称这棵树为有序树；反之，称为无序树

*__森林__：m（m≧0）棵互不相交的树的集合

*任何一颗非空树是一个二元组 **Tree=(root,F)**,其中 root 称为根节点，F 称为**子树森林**

6.2 二叉树

二叉树的定义

*__二叉树__一个是 n(n≧0)个结点的有限集合，该集合或者为空（称为__空二叉树__）；或者是由一个根节点和两棵互不相交的、分别称为__左子树__和__右子树__的二叉树组成

二叉树的特点

*每个结点至多只有两棵子树

*且子树有左右之别，子树的次序（位置）不能颠倒

*即使某结点只有一颗子树，也有左右之别

二叉树的五种形态

（1） 空树

（2） 仅有根节点

（3） 根节点只有左子树

（4） 根节点只有右子树

（5） 根节点同时有左右子树

二叉树的基本操作

性质 1：在二叉树的第 i 层上至多有 **2^（i-1）**个结点（i≧0）

性质 2：深度为 k 的二叉树最多有 2^k-1 个结点

性质 3：对任何一棵二叉树 T，如果其终端结点数为 n0，度为 2 的结点树为 n2,则 n0=n2+1

满二叉树

*深度为 k 且含有 2^k-1 个结点的二叉树

*特点：每一层上的结点数都是最大结点数

完全二叉树

*深度为 k 的包含 n 个结点的二叉树中每个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应

*__特点：__叶子结点只能在层次最大的两层上出现；最后一层的叶子结点必须从左往右依次排列

性质四：具有 n 个结点的完全二叉树的深度为**[log2n]+1**

性质五：如果对一棵有 n 个结点的完全二叉树的结点按层序编号，则对任一结点 i(1≦i≦n)，有

（1） 若 **i=1**，则 i 是根节点，无父节点；
若 **i>1**，则 i 的父结点为**[i/2]**

（2） 若 2i>n，则 i 无左孩子（i 为叶结点）；
若 2i≦n，则 i 有左孩子且为 2i

（3） 若 2i+1>n，则 i 无右孩子
若 2i+1≦n，则 i 有右孩子且为 2i+1

二叉树的顺序存储结构

完全二叉树（或满二叉树）的顺序存储

采用一维数组，按层序自上而下，自左而右依次存储完全二叉树的每一个结点元素

二叉树的顺序存储表示

一般二叉树的顺序存储

*通过虚设部分结点，使其变成相应的完全二叉树//用 0 或空符号表示

*根据性质 5，如已知某结点的层序编号 i，则可求得该节点的父结点、左孩子结点和右孩子结点，然后检测其值是否为虚设的特殊结点

特殊二叉树（最坏情况）

*一个深度为 k 且只有 k 个结点的单支树需要长度为 2^k-1 的一维数组

*需增加很多空结点，造成存储空间的浪费

*非完全二叉树不适合进行顺序存储

二叉树链式存储结构

*一般用__二叉链表__存储二叉树（每个结点有两个指针域）

二叉链表的 C 语言类型描述

具有 n 个结点的二叉链表中，有多少指向孩子结点的指针？有多少个空指针？

*具有 n 个结点的二叉链表中，总的指针域数为 2n；除根节点外，其余 n-1 个结点均有指针域所指向，即有 n-1 个指向孩子结点的指针；

空指针个数为：2n-(n-1)=n+1

三叉链表增加一个指向其双亲结点的指针域

6.3 遍历二叉树

二叉树的遍历

*顺着某一条搜索路径访问二叉树中的所有结点

*使得每个结点均被访问依次，而且仅被访问一次

遍历

*按某种搜索路径访问二叉树的每个结点，而且每个结点仅被访问一次

*访问是指对结点进行各种操作的简称，包括输出节点的信息、查找、修改等操作

*遍历是各种数据结构最基本的操作，许多操作可在遍历的基础上实现

*遍历对__线性结构__来说是个容易解决的问题，对于__非线性结构__需要寻找一种规律：把二叉树的结点排列在一个线性队列上

*遍历的结果是二叉树结点的线性序列。

非线性结构线性化

二叉树的递归

*二叉树由根、左子树、右子树 3 个基本单元组成

*遍历二叉树可分解为：访问根、遍历左子树和遍历右子树

先序遍历二叉树

*若二叉树为空，则空操作；否则

（1）访问根节点

（2）先序遍历左子树

（3）先序遍历右子树

所得到的线性序列称为**先序序列**

中序遍历二叉树

*若二叉树为空，则空操作；否则

（1）中序遍历左子树

（2）访问根节点

（3）中序遍历右子树

所得到的线性序列称为**中序序列**

后序遍历二叉树

*若二叉树为空，则空操作；否则

（1）后序遍历左子树

（2）后序遍历右子树

（3）访问根节点

所得到的线性序列称为**后序序列**

层序遍历二叉树

*若二叉树为空，则从根结点开始，从上到下从左到右按层次遍历所有结点

所得到的线性序列称为**层序序列**

*算法思想：

（1）将二叉树的根入队列

（2）将队头元素出队列，并判断此元素是否有左右孩子，若有，则将其左右孩子入列，否则转步骤 3

（3）重复步骤 2，直到队列为空

表达式与二叉树

表达式的前缀、中缀、后缀表示：操作符放于两个操作数之前、之间和之后

回顾栈结构

逆波兰表达式求值

*逆波兰表示：一种不需要括号的后缀表达法，所有符号都是在要运算的数字的后面出现

*算法思想

（1）从左到右遍历表达式的每个数字和符号

（2）遇到是操作数就进栈

（3）遇到是运算符就将处于栈顶的两个操作数出栈

（4）出栈元素进行运算，运算结果进栈，一直到最终获得结果

中缀表达式转换为逆波兰表达式

*算法思想：

（1）从左到右遍历中缀表达式（包含括号）的每个操作数和运算符

（2）若是操作数就输出，即成为后缀表达式的一部分

（3）若是运算符，则判断其与栈顶运算符的优先级，若是右括号或低于优先级低于栈顶运算符则栈顶元素出栈并输出，并将当前运算符进栈，一直到最终输出后缀表达式为止

递归算法转换为非递归算法

*理论上而言，所有递归算法都可以用非递归算法来实现，其基础是递归算法的计算总能用**一棵树形结构**来表示

*递归算法转化为非递归算法有以下基本方法

（1）通过分析，跳过分解过程，直接用**循环结构**的算法实现

（2）由用户定义的栈模拟系统的运行时栈，通过分析在栈中只保存必须的信息，从而用非递归算法替代递归算法

运行时栈：存储了某一结点及其所有祖先结点（有些可能已访问过）；递归返回到某一结点时，通过函数中执行到的语句来决定是否访问此节点

自定义栈：在访问一个结点时，只保存此节点的还未被访问的祖先结点；每弹出一个结点，则立即对其执行访问

按先序序列建立二叉树的二叉链存储结构

写出要创建的二叉树的先序序列，其中空指针用字符#来表示

按前缀表达式建立表达式二叉树的二叉链存储结构

计算二叉树叶子结点的个数（先序）

*先序（或中序、后序）遍历二叉树

*在遍历过程中查找叶子结点，并计数

*在遍历算法中增添一个计数参数

*并在算法中访问结点的操作改为：若是叶子，则计数器增 1

计算二叉树的深度（后序）

*二叉树的深度应为其左、右子树深度的最大值加 1

*先序（或中序、后序）遍历二叉树

*在遍历过程中记录左、右子树深度

思考：通过一种遍历序列可以恢复原始二叉树吗

先序+中序 中序+后序

6.5 树和森林

1. 树和森林

***森林**：m 棵互不相交的树的集合

*对树中每个结点而言，其子树的集合也是森林

*树的操作是可以通过对二叉树的操作来实现

2. 树的存储结构

*树的三种存储结构

（1）双亲表示法

（2）孩子链表表示法

（3）树的二叉链表（孩子兄弟）表示法

双亲表示法

*采用一组连续空间存储树的结点，并保存每个结点的双亲（父节点）结点的位置（下标）

*特点：可以在常量时间得到指定结点的父亲结点；反复求父亲结点可以找到根结点

*缺点：求**指定结点的孩子结点**需遍历整个结构

孩子链表表示法

*用数组保存所有结点，并用线性链表表示每个结点的所有孩子

*便于找结点的孩子，但要找结点的双亲则需遍历整个结构

孩子兄弟表示法

*即二叉链表表示法，结点的两个指针域分别指向该结点的第一个孩子和下一个兄弟，所以此二叉链表也称为孩子兄弟链表

*树和二叉树的存储表示方式是一样的，只是左右孩子表达的逻辑关系不同，它们的二叉链表完全一样，只是右指针的解释不同

3. 树、森林与二叉树的转换

树转换为二叉树

*由于二叉树和树都可用二叉链表作为存储结构，则以二叉链表作为媒介可导出树与二叉树之间的一个对应关系

*给定一棵树，可以找到唯一的一棵二叉树与之对应，从物理结构来看，它们的二叉链表完全一样，只是右指针的解释不同

*树→孩子兄弟链表→二叉树的二叉链表→二叉树

森林转换为二叉树

*将二叉树中的每一树转换二叉树

*将各二叉树的根结点视为兄弟连在一起

二叉树转换为森林

*二叉树→二叉树的二叉链表→孩子兄弟链表→树（森林）

4. 树的遍历

树的先根遍历

*若树不空，则先访问根结点，然后依次先根遍历各棵子树

*树的先序遍历等于对转换所得的二叉树进行先序遍历

树的后根遍历

*若树不空，则先依次后根遍历各棵子树，然后访问根结点

*树的**后序遍历**等于对转换所得的二叉树进行**中序遍历**

5. 森林的遍历

森林的先序遍历

*访问森林中第一棵树的根结点

*先序遍历第一棵树的根结点的子树森林

*先序遍历除第一棵树后剩余的树构成的森林

*森林的先序遍历等于对转换所得的二叉

树进行先序遍历

森林的中序遍历

*中序遍历第一棵树的根结点的子树森林

*访问第一棵树的根节点

*中序遍历除第一棵树外剩余的树构成的森林

*森林的中序遍历等于对转换所得的二叉树进行中序遍历

6. 树和森林的遍历

*在森林转换成二叉树过程中：

第一棵树的子树森林→二叉树的左子树

剩余树森林→二叉树的右子树

*森林的先序、中序遍历都是**对各个树逐个访问**，森林的后序遍历不是对每个树逐个访问，故此处不考虑其后序遍历

树的遍历和二叉树遍历的对应关系

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

对树和森林的遍历可以调用二叉树对应的遍历算法实现

7. 树的遍历的应用

建树的存储结构

*假设以二元组（F，C）的形式自上而下、自左而右依次输入树的各边，建立树的孩子兄弟链表

*算法需要一个队列保存已建好的结点的指针

计算树的深度

6.6 赫夫曼树及其应用

1. 赫夫曼树

相关术语

***叶子结点的权值**

对叶子结点赋予的一个有意义的数值量（实数）

***路径和路径长度**

从树中的一个结点到另一个结点之间的分支构成这两个结点之间的路径，路径上的分支数目称作路径长度

***结点的带权路径长度**

从根到该结点的路径长度与该结点权的乘积

***树的带权路径长度**

树中所有叶子的带权路径长度之和

赫夫曼树

*给定一组具有权值的叶子结点，带权路径长度最小的二叉树，称为赫夫曼树，亦称最优二叉树

赫夫曼树的特点

*权值越大的叶子结点越靠近根结点，而权值越小的叶子结点越远离根节点（构造赫夫曼树的核心思想）

*只有度为 0（叶子节点）和度为 2（分支节点）的结点，不存在度为 1 的结点

*n 个叶结点的赫夫曼树的结点总数为 2n-1 个

*赫夫曼树不唯一，但 WPL 唯一

最优二叉树（赫夫曼树）可以用来得到某些判定问题的最佳判定算法

将百分制表示的成绩 score 转换为五级分制

*判定树将一个 score 转化为 grade（比较次数）

*判定树将多个 score 转化为 grade（考虑性能）

*实际上学生的成绩在 5 个等级上的分布不均匀（权值不同）

*WPL 最小的判定树，转换多个成绩的平均比较次数

*有些结点包含了两次比较

*求解平均比较次数最小的判定树就是求解给定的叶结点权值最优二叉树

赫夫曼树的构造算法

*根据给定的 n 个权值{w1,w2,…wn}，构造一棵有 n 个叶子结点的二叉树，每个叶子结点带权为 wi，使 WPL 达到最小

*步骤：

（1）根据给定的 n 个权值{w1,w2,…wn}构造 n 棵二叉树的集合 F={T1,T2,…TN}，其中每棵二叉树 Ti 中只有一个带权为 wi 的根结点，其左右子树均空

（2）在 F 中选取两棵根节点的权值最小的树作为左右子树构造一棵新的二叉树，且置新的二叉树的根节点的权值为其左右子树上根节点的权值之和

（3）在 F 中删除这两棵树，同时将新得到的二叉树加入 F 中

（4）重复 2 和 3,直到 F 中只含一棵树为止。这棵树就是赫夫曼树

*n 个结点需要进行 n-1 次合并，每次合并都产生一个新结点，因此赫夫曼树共有 2n-1 个结点

2. 赫夫曼编码

*在进行数据通讯时，涉及**数据编码**问题。所谓数据编码就是将信息原文转换为二进制字符串，译码则是将信息的编码形式转换为原文

原文→电文→原文

***编码方案**

等长编码方案

不等长编码方案

等长编码方案

*在传送电文时，希望总长尽可能地短

*如果对每个字符设计长度不等的编码，且让电文中**出现次数较多**的字符采用尽可能短的编码，则总长可减少

不等长的编码方案

*设计不等长编码时，任意字符的编码都不能是另一个字符的前缀，这种编码称为**前缀编码**

*利用二叉树得到使电文总长最短的二进制前缀编码

（1）以各个字符出现频次作权，设计一颗赫夫曼树

（2）形成前缀编码的原理：一个叶子结点的路径不可能是另一叶子路径的前缀，这样得到的编码称为**赫夫曼编码**

赫夫曼树和赫夫曼编码的存储结构

赫夫曼编码算法

译码算法

#7.Graph

7.1 图的定义和基本术语

7.2 图的存储结构

7.3 图的遍历

7.4 最小生成树

7.5 有向无环图的应用

7.6 最短路径

7.1 图的定义和基本术语

1. **图**

***顶点集 V 和边集 E** 组成的二元组 G=（V,E）,V 是顶点的有穷非空集合，E 是两个顶点间的关系的集合，若边是顶点的有序对（弧 arc）的集合，那么称该图为**有向图**；若边是顶点的无序对（边 edge）集合，那么称该图为**无向图**

***无向完全图：**在无向图中，如果任意两个顶点之间都存在边，则称该图为无向完全图

***有向完全图**：在有向图中，如果任意两个顶点之间都存在方向相反的两条弧，则称该图为有向完全图

*含有 n 个顶点的无向完全图的边数为 n(n-1)/2

含有 n 个顶点的有向完全图的弧数位

n(n-1)

*稀疏图：有很少条边或弧（e<nlogn）的图省委稀疏图，否则称作稠密图

*子图：设有两个图 G=（V，{E}）、G1=（V1，{E1}），若 V1∈V 且 E1∈E，则称 G1 是 G 的子图

*邻接点及关联边：

邻接点：边的两个顶点互为邻接点

关联边：若边 e=（v,u）,则称顶点 v,u 关联边 e

*顶点的度：一个顶点 V 的度是与它相关联的边的条数，记作 TD（v）

*在有向图中：

V 的入度：以 v 为终点的有向边数，记作 ID（v）

V 的出度：以 v 为始点的有向边数，记作 OD（v）

顶点 V 的度=V 的入度+V 的出度

*设图 G 的顶点数为 n，边数为 e，满足如下关系

*路径、回路或环

无向图 G=(V,{E})中的顶点序列 v1,v2,…vk，若(vi,vi+1)∈E(i=1,2,…k-1),v=v1,u=vk,则称该序列是从顶点 v 到顶点 u 的路径

有向图 G=(V,{E})中的顶点序列 v1,v2,…vk，若<vi,vi+1>∈E(i=1,2,…k-1),v=v1,u=vk,则称该序列是从顶点 v 到顶点 u 的有向路径

若 v=u,则称该序列为**回路或环**

路径上边或弧的数目称作**路径长度**

*简单路径、简单回路或简单环

简单路径：序列中顶点不重复出现的路径称为简单路径

简单回路：除了第一个和最后一个顶点外，其余顶点不重复出现的回路，称为简单回路或简单环

*连通图与联通分量

顶点的连通性：在无向图中，若从顶点 v 到顶点 u 有路径，则称顶点 v 和 u 是联通的

连通图：在无向图 G 中，若任意两个顶点 v，u 是联通的，则称 G 是连通图

连通分量：非连通图的极大连通子图称为连通分量

（1） 含有极大顶点数

（2） 依附于这些顶点的所有边

*强连通图与强连通分量

强连通图：在有向图 G 中，若每一对顶点 v,u（v≠u），从 v 到 u 和从 u 到 v 都存在路径，则称 G 是强连通图

*强连通分量：非强连通图的极大强连通子图称为强连通分量

*权：某些图的边或弧具有与它相关的系数

*网：边或弧带权的图称为网

*生成树：一个连通图的生成树是一个极小连通子图，它包含图中全部顶点，但只有足以构成一棵树的 n-1 条边

在生成树上添加一条边，必定构成一个环，这条边使得它依附的那两个顶点之间有了第二条路径

有 n 个顶点的生成树有且仅有 n-1 条边

有 n 个顶点和小于 n-1 条边的图是非连通图，若多于 n-1 条边，那一定有环

有 n-1 条边的图不一定是生成树

7.2 图的存储结构

*图的存储结构至少包括两类信息：顶点的数据+顶点间的关系

*多重链表表示：由于各结点的度数各不相同而造成空间浪费，实际应用不宜采用

邻接矩阵表示（数组表示法）

*基本思想：用一个一维数组存储图中顶点的信息，用一个二维数组（称为邻接矩阵）存储图中个顶点之间的邻接关系

*无向图的邻接矩阵存储结构特点

主对角线为 0 且一定是对称矩阵

*有向图的邻接矩阵存储结构特点

一定不对称

带权图（网）的邻接矩阵

*假设带权图 G（V,{E}）有 n 个顶点，则邻接矩阵是一个 n*n 的方阵，定义为：

A[i][j] = Wij

∞

邻接表表示

*无向图的邻接表

对于无向图的每个顶点 vi，将所有与 vi 相邻的顶点链成一个**单链表**，称为顶点 vi 的边表（顶点 vi 的邻接表）

再把所有边表的指针和顶点信息的**一维数组**构成顶点表

空间需求 O(n+2e)

*有向图的邻接表-正邻接表

对于有向图的每个顶点 vi，将邻接于 vi 的所有顶点链成一个**单链表**，称为顶点 vi 的出边表

再把所有出边表的指针和顶点信息的**一维数组**构成顶点表

空间需求 O(n+e)

*有向图的邻接表-逆邻接表

对于有向图的每个顶点 vi，将邻接到 vi 的所有顶点链成一个**单链表**，称为顶点 vi 的入边表

再把所有入边表的指针和顶点信息的**一维数组**构成顶点表

空间需求 O(n+e)

7.3 图的遍历

图的遍历

*从图的某个顶点出发，访问图中的所有顶点，且使每个顶点仅被访问依次。这一过程叫做**图的遍历**

*图的遍历是求解图的**连通性问题、拓扑排序**等问题的基础

图的遍历要解决的关键问题

*从某个七点开始可能到达不了所有的顶点，怎么办?

多次调用从顶点出发遍历图的算法

*图中可能存在回路，且图的任一顶点都可能与其它顶点“相通”，在访问完某个顶带你之后可能会沿着某些边又回到了曾访问过的顶点。如何避免某些顶点可能会被重复访问

附设访问标志数组 visited[0…n-1]

*在图中，一个顶点可以和其它多个顶点相连，当这样的顶点访问过后，如何选取下一个要访问的顶点

深度优先搜索和**广度优先搜索**

深度优先搜索

*类似于树结构的先根遍历，是树的先根遍历的推广

*深度优先遍历图的方法是，从图中某顶点 v 出发：

（1）访问顶点 v

（2）依次从 v 的未被访问的邻接点出发，对图进行深度优先遍历，知道图中和 v 有路径相通的顶点都被访问

（3）若此时图中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行深度优先遍历，直到图中所有顶点均被访问过为止

深度优先遍历算法复杂度

*邻接矩阵表示

查找每一个顶点的所有边所需时间

O（n）
则遍历图中所有的顶点所需时间 O(N^2)
适合**边稠密**的图（因复杂度跟边五官）
*__邻接表表示__
沿每个边链表可以找到某个顶点 v 的所有临界顶点 w
由于共有 2e 个边结点，所以扫描边的时间为 O（e）
对所有顶点递归访问一次，所需时间 O（n+e）
适合**边稀疏**的图

广度优先搜索

*类似于树的按层次遍历过程
深度优先遍历图的方法是，从图中某顶点 v1 出发：
（1）访问顶点 v1
（2）访问 v1 的所有未被访问的邻接点 w1,w2,…wk
（3）依次从这些邻接点（在步骤 2 中访问的顶点）出发，访问它们的所有未被访问的邻接点；以此类推，直到图中所有邻接点均被访问过为止

广度优先遍历算法复杂度

*邻接矩阵表示
查找每一个顶点的邻接点所需时间 O（n^2）
则遍历图中所有的顶点所需时间 O(N^2)
*__邻接表表示__
由于共有 2e 个边结点，所以扫描边的时间为 O（e）
对所有顶点递归访问一次，所需时间 O（n+e）
适合**边稀疏**的图

7.4 最小生成树

生成树

*一个连通图的生成树是一个极小连通子图，它含有图中全部顶点，但只有足以构成一棵树的 n-1 条边（无环）
*__生成树的代价__：一个带权的连通图 G 的生成树上**各边权值之和**称为该生成树的代价

最小生成树

*对于带权图（网），在所有生成树中，各边的权值总和最小的生成树称作最小代价生成树
*实际问题
*性质：假设 G=(V,E)是一个联通网，U 是顶点集 V 的一个非空子集。若(u，v)是一条具有最小权值（代价）的边，比存在一棵最小生成树包括它
*基于最小生成树性质的算法：普里姆算法和克鲁斯卡尔算法

普里姆算法

*普里姆算法构造最小生成树的过程是从一个顶点 U={u0}作初态，不断寻找与 U 中顶点相邻且代价最小的边的另一个顶点，扩充到 U 集合直至 U=V 为止
*设置一个**辅助数组**，对当前 V-U 集中的每个顶点，记录和顶点集 U 中顶点相连接的代价最小的边 **closedge[i]**
*__最小值的权值 lowcost__ 有 0、正整数、无穷大三种可能的取值，分别表示 i 在 U 中、i 与 U 中顶点邻接、i 与 U 中顶点不邻接
*__时间复杂度__主要体现在两层循环上，时间复杂度为 **O（n^2）**
*与网中的边数无关，适用于求边稠密的网的最小生成树
*空间复杂度主要体现在两个辅助数组，**空间复杂度**为 **O（n）**

克鲁斯卡尔算法

*假设连通图 N={V，{E}}，则令最小生成树的初始状态为只有 n 个顶点而无边的非连通图 T={V，{}}，图中每个顶点自成一个连通分量
*在 E 中选择代价最小的边，若该边依附的顶点落在 T 中不同的**连通分量**上，则将此边加入到 T 中，否则舍去此边而选择下一条代价最小的边。以此类推，直至 T 中的所有顶点都在**同一连通分量**上为止
*__时间复杂度：O(eloge)__，与图网中的边数有关，适用于求边稀疏的网的最小生成树

7.5 有向无环图的应用

1. 有向无环图

*描述含有公共子式的表达式的有效工具
*可节省存储空间
*关心问题：
（1）工程能否顺利进行
（2）完成整项工程至少需要多少时间，哪些子工程是影响工程进度的关键子工程

AOV 网：

*用顶点表示子工程（也称活动），用弧表示子工程间的顺序关系，工程流程可用 AOV 网表示
*一个 AOV 网不应该存在环，因为存在环意味着某项活动的进行应该以本活动的完成作为先决条件

2. 拓扑排序

*将有向图中的顶点排成一个序列
*__拓扑序列__
满足如下关系的有向图 D 的一个顶点序列称作一个拓扑序列，对于该序列中任意两点 v、u，若在 D 中 v 是 u 的前驱，则在序列中 v 也是 u 的前驱
*__偏序__：集合上的关系 R 是自反、反对称、传递的，则 R 是集合上的偏序关系，即包括：可比较的、不可比较的、以及传递的
*__全序__：则是对任意的 x 和 y 必有 xRy 或 yRx,，即几何上任意两点都是可比较的、传递的
*由集合上的一个偏序得到该集合上的一个全序，这个操作称为拓扑排序。即将不可比较的通过得到的前后序列关系强制生成一个前后的比较关系

拓扑排序算法

（1） 在有向图中选一个无前驱的顶点 v，输出之，
（2） 从有向图中删除 v 与以 v 为尾的弧
（3） 重复 1.2，直到输出全部顶点或当前图中不存在无前驱的顶点为止（此时有向图中存在环）

*一个有向图的拓扑序列不一定唯一

拓扑排序算法分析

*初始建立入度为 0 的栈，要检查所有顶点一次，执行时间为 O（n）
*排序中，若 AOV 网无回路，则每个顶点入栈、出栈各一次，每个边表结点被检查一次，执行时间为 O（n+e）
*因此，拓扑排序算法的时间复杂度为 O（n+e）

利用深度优先搜索遍历进行拓扑排序

*在 DAG 图中，由于图中无环，则可从图中某点出发利用**深度优先搜索**进行遍历，最先退出 DFS 函数的顶点即为**出度为零**的顶点，是拓扑有序序列的最后一个顶点
*由此，按退出 DFS 函数的先后记录下来的顶点序列即为**逆向的拓扑有序序列**

3. 关键路径

AOE 网

*带权的有向无环图，顶点表示时间，边表示活动，边上权表示活动持续的时间，称有向图为边表示活动的网络

AOE 网的特点

*只有在某个顶点所代表的事件发生后，从该顶点出发的各有向边代表的活动才能开始

*只有在进入某一顶点的各有向边代表的活动已经结束，该顶点所代表的事件才能发生

*表示实际工程计划的 AOE 网应该是无回路的

*只有一个入度为零的顶点（源点）表示整个活动开始

*只有一个出度为零的顶点（汇点）表示整个活动结束

路径长度、关键路径、关键活动

***路径长度**：从源点到汇点路径上所有活动的持续时间之和

***关键路径**：在 AOE 网中，由于有些活动可以并行，所以完成工程的最短时间是从源点到汇点的最大路径长度。因此把从源点到汇点具有最大长度的路径称为关键路径

***关键活动**：关键路径上的活动称为关键活动

***算法分析**

在拓扑排序求顶点最早开始时间和逆拓扑有序求顶点最迟开始时间时，所需时间 **O（n+e）**

求各个活动的最好开始时间和最迟开始时间时，所需时间 O(e)

利用拓扑排序算法求关键路径总的时间复杂度为 O(n+e)

7.6 最短路径

最短路径问题

*如果图中从一个顶点可以到达另一个顶点，则称这两个顶点间存在一条路径

*从一个顶点到另一个顶点间可能存在多条路径，而每条路径上经过的边数并不一定相同

*如果图是一个**带权图**，则路径长度为路径上各边的权值的总和，两个顶点间路径最短的那条路径称为两个顶点间的最短路径，其路径长度称为最短路径长度

最短路径问题解法

#8. Search

8.1 基本概念和术语

8.2 静态查找表

8.3 动态查找表

8.4 哈希表

查找的性能：

*时间性能由关键字的比较次数来度量

*查找算法的时间复杂度是问题规模 n 和待查关键字在查找集合中的位置 k 的函数，记为 T（n,k）

*平均查找长度：把给定值与关键字进行比较的次数的期望值称为查找算法在查找成功时的平均查找长度

cha

8.2 静态查找表

1. 顺序查找表

监视哨：

免去查找过程中每一步都要检测整个表是否查找完毕，通过这一实现技巧的改进，查找所需的平均时间几乎减少一半

平均查找长度

查找成功 n+1/2

查找失败 n+1

2. 折半查找或二分查找

顺序存储 按关键字有序

折半查找的判定树

折半查找的过程可以用二叉树来描述，树中每个结点对应有序表一个记录，结点的值为该纪录在表中的位置。称描述折半查找过程的二叉树为折半查找判定树，简称判定树

平均查找长度

*查找成功

*查找失败

*折半查找成功时进行比较的关键字个数最多不超过树的深度，因为判定树（不包括外部结点）的度小于 2 的结点只能出现在下面两层上，所以 n 个结点的完全二叉树的高度相同，即 log2n 下取整 +1

*查找不成功时和给定值进行比较的关键字个数最多也不超过 log2n+1,折半查找的最坏性能与平均性能相当接近

不可以采用链表作为查找表的存储结构，不支持随机访问

8.3 动态查找表

1. 二叉排序树

二叉排序树或是一棵空树，或是具有如下特性的二叉树

*若它的左子树不空，则左子树上所有结点的值均小于根节点的值

*若它的右子树不空，则左子树上所有结点的值均大于根节点的值

*它的左、右子树也都分别是二叉排序树

二叉排序树的结构特点

*任意一个结点的关键字，都大于（小于）其左右子树中任意结点的关键字，因此个结点的关键字各不相同

*同一个数据集合，二叉查找树不唯一，中序序列相同

插入

新插入的结点一定是查找不成功时，查找路径上最后一个结点的左孩子或有孩子

构造

循环调用插入算法

注意：在建立二叉查找树的时候，若按关键字有序顺序插入记录，则退化为单链表

删除

叶子 直接删除

只有左子树或右子树，子树替代

既有左子树又有右子树，前驱节点代替再删除前驱结点一一处理

平均查找长度

不同形态平均查找长度不同

O（logn）和 O（n）之间

平均时间复杂度 O（logn）

就平均性能而言，二叉排序树的查找与折半查找差不多

3. 平衡二叉树

最小不平衡子树

在平衡二叉树插入结点时，其祖先结点的平衡因子可能发生变化，离插入节点最近，且平衡因子变为 2 或-2 的祖先结点作为根的子树称为最小不平衡子树

插入

删除

多次平衡化

查找性能

在平衡树上进行查找的过程和二叉排序树相同，关键字比较次数不超过平衡树深度

查找插入删除均为 O（logn）

8.4 哈希表
ASL=1 查找成功
无需进行关键字的比较
除留余数法
选取比较理想的 p 值
素数
1.1n~1.7n，n 为哈希表中待存储的元素个数
开放定址法
线性探测再散列 容易二次聚集
二次探测再散列 表长 4j+3 覆盖所有地址
哈希表查找的性能分析
由于冲突的存在 ASL>1
三个因素：
哈希函数、处理冲突方法、装填因子
装填因子：哈希表的饱和程度，存入元素的个数和哈希表大小 m 比较

#9. Sorting

9.1 基本概念
9.2 插入排序
9.3 交换排序
9.4 选择排序
9.5 归并排序
9.6 基数排序
9.7 内部排序方法的比较

9.1 基本概念

排序

*假设 n 个元素的序列{R1,R2,…Rn}，相应的关键字序列为{ k1,k2,…,kn}，确定 1，2，…，n 的一种排列 p1,p2,…pn，使相应的关键字满足非递减（或非递增）关系 kp1,kp2,…kpn，从而得到一个关键字有序的序列

排序的目的：方便查询和处理

排序方法的稳定性

*当需要排序的关键字都不相同时（ki≠kj）,排序的结果是唯一的，否则排序结果可能不唯一

*若 ki=kj，且在排序前的序列中 Ri 领先于 Rj，排序后 Ri 仍领先于 Rj，则称所用的排序方法是稳定的；反之，若可能使排序后的序列中 Rj 仍领先于 Ri，则称所用的排序方法是不稳定的

内部排序和外部排序

*待排序的记录存放在计算机的内存中所进行的排序操作称为内部排序

*待排序的记录数量很大，以致内存一次不能容纳全部记录，在排序过程中需要访问外存的排序过程称为外部排序

排序方法度量

*排序过程主要是比较记录的关键字和移动记录。因此排序的时间复杂性可用算法执行中的**数据比较次数**及**数据移动次数**来衡量

*排序过程在最坏或平均情况下所进行的比较和移动次数越少，则时间复杂性就越好

*针对一种排序方法，不仅要分析它的**时间复杂性**，而且要分析它的**空间复杂性**（**辅助存储空间**是指在数据规模一定的条件下，除了存放待排序记录占用的存储空间之外，执行算法所需要的其他额外存储空间）、**稳定性**

基于比较的排序

插入排序：将无序子序列中的一个或几个记录“插入“到有序序列中，从而增加记录的有序子序列的长度

交换排序：通过“交换“无序序列中的记录从而得到其中关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度

选择排序：从记录的无序子序列中“选择“关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度

归并排序：通过“归并“两个或两个以上的记录有序自序列，逐步增加记录有序序列的长度

不基于比较的排序：根据组成关键字的分量及其分布特征，如基数排序

9.2 插入排序

***插入排序**：
将一个记录插入到已排好顺序的序列中，形成一个新的、记录数增 1 的有序序列

***一趟插入排序的插入过程**

（1）在 R[1,…i-1]中查找 R[i]的插入尾注，R[1,…j].key≤R[i].key< R[j+1,…i-1].key 将

（2）R[j+1,…i-1]中的所有记录均后移一个位置

（3）将 R[i]插入到 R[j+1]的位置上

1. 直接插入排序

*利用顺序查找实现在 R[1,…i-1]中查找 R[i]的插入位

***算法概述**

（1）将序列中的第 1 个记录看成是一个有序的子序列

（2）从第 2 个记录起逐个进行插入，直到整个序列变成按关键字有序序列为止

*从 R[i-1]起向前进行**顺序查找**

*设置**监视哨 R[0[=R[i]**

*整个排序过程需要进行**比较、后移记录、插入恰当位置**

*从第 2 个记录到第 n 个记录共需 n-1 趟

时间复杂度 O（n^2）

*实现排序的基本操作：比较两个关键字的大小+移动记录

*最好的情况（关键字在记录序列中**顺序有序**）

比较的次数：n-1
移动的次数：0

*最坏的情况（关键字在记录序列中**逆序有序**）

比较的次数：（n-1）*（n+2）/2
移动的次数：（n-1）*（n+4）/2

空间复杂度 O(1)

只需要 1 个辅助单元（监视哨&备份待插入的元素）

稳定的排序方法

适用情况：元素数目少，或者元素的初始序列基本有序

4. **折半插入排序**

*在寻找位置时，因为 R[1,…i-1]是一个按关键字有序的有序序列，则可利用折半查找实现在 R[1,…i-1]中查找 R[i]的插入位置

*折半插入排序减少了关键字的**比较次数**，而记录的移动次数不变；

时间复杂度为 O（n^2），空间复杂度为 O（1）,稳定的排序方法

5. **2-路插入排序**

*目的是减少在排序过程中移动记录的次数，但需 n 个记录的**辅助空间**

*将 d 看成一个**循环变量**，指针 first 和 final 分别指示排序过程中得到的有序序列中的第一个记录和最后一个记录在 d 中的位置

***时间复杂度为 O（n^2），空间复杂度为 O（n）,稳定的排序方法**

*当 L.r[1]是待排记录中关键字的最小或最大的记录时，2-路插入排序就会失去其优越性

6. 希尔排序

*希尔排序又称**缩小增量排序**其改进思想主要基于以下两点

(1)元素基本有序时，直接插入排序的时间复杂度接近于 O（n）

(2)对小序列（n 较小）进行直接插入排序的效率更高

***希尔排序的算法思想**

（1）先将整个待排元素序列分割成**若干个小的子序列**（等间距的元素分在同一个子序列），在每个子序列内进行**直接插入排序**

（2）缩小间距：增加子序列的长度（子序列个数减少再对每个子序列进行直接插入排序

（3）在整个序列基本有序情况下，最后在对全体元素进行一次直接插入排序

*希尔排序的算法分析是一个复杂的问题

（1）时间复杂度依赖于增量序列的设置，尚没有正式的依据说明如何设置最佳的增量序列，大量的实验表明希尔排序所需的比较和移动次数可达到 **O（n^1.3）**

（2）按希尔最初建议的增量序列设置 $n/2^i$,则**最坏情况下**时间复杂度为 **O（n^2）**

空间复杂度 O（1），因为希尔排序内部采用的是直接插入排序

希尔排序是**不稳定**的排序方法

希尔排序易于实现，一般较为常用

9.3 交换排序

1. 冒泡排序

基本思想：将相邻位置的关键字进行比较，若为逆序则交换之

*通过比较与交换得到待排序列的一个最值元素“**上浮**”到序列一端，然后**缩小排序范围**

*若在一趟排序过程中**没有进行过交换记录**的操作，则整个排序过程**终止**

时间复杂度 O（n^2）

*最好的情况（关键字在记录序列中**顺序有序**）只需进行一趟起泡

比较的次数：n-1

移动的次数：0

*最坏的情况（关键字在记录序列中**逆序有序**）

比较的次数：（n-1）*n/2

移动的次数：3（n-1）*n/2

空间复杂度 O(1)

只需要 1 个辅助单元进行交换

稳定的排序方法

适用情况：元素数目少，或者元素的初始序列基本有序

2. 快速排序

*取序列中的某个元素作为基准（也称枢轴、主元素或划分元素，一般取第一个元素）

*通过一趟划分，将待排序列分为左右两个子序列

左子序列元素的关键字均不大于基准元素，右子序列元素的关键字均不小于基准元素

然后分别对左、右子序列进行快速排序，直到整个序列有序

*当键值各不相等时，快速排序的划分过程与一棵二叉排序树完全对应

时间复杂度 O（n^2）

***最好**的情况为 O（nlogn）

*快速排序的**平均时间复杂度**也为 O（nlogn）

*最坏的情况（**逆序或正序**），时间复杂度为 O（n^2）

空间复杂度 O(logn)

一般情况下栈的深度为 logn，在最坏情况下深度为 n

不稳定的排序方法

快速排序**不适合对小规模**的序列进行排序

快速排序方法的改进和思考

*枢轴元素的选取

为避免蜕化为冒泡排序，随机性

最好的一种内部排序方法

9.4 选择排序

基本思想：每趟在 n-i+1 个记录中选取关键字最小的记录作为有序序列的第 i 个记录

1. 简单选择排序

*第 1 趟在 n 个记录中选取**最小记录**作为有序序列的第 1 个记录

*第 i 趟在 n-i+1 个记录中选取**最小**的记录作为有序序列中的第 i 个记录

时间复杂度 O（n^2）

*对 n 个记录进行简单选择排序，所需进行的关键字间的

比较次数：

总计为 n(n-1)/2

移动次数：

最小值为 0，最大值为 3（n-1）

空间复杂度 O(1)

只需要 1 个辅助单元进行交换

不**稳定**的排序方法

适用情况：元素数目少，无需完全排序的情况，比如选出第 i 小的元素（i 要较小）

2. 树形选择排序

*首先对 n 个记录的关键字两两进行比较，然后在 n/2 个较小者之间再进行两两比较，如此重复，直至选出最小关键字的记录

*选出最小记录后，将树中的该最小记录修改为∞，然后从该叶子结点所在子树开始，修改到达树根的路径上的结点

*以后每选出一个小元素，都只需进行 logn 次比较

*目的是减少简单选择排序的重复比较次数

时间复杂度 O（nlogn）

执行 n 趟，每趟比较 O（logn）

空间复杂度 O(n)

增加了 **n-1** 个额外的**存储空间存放中间比较结果**

稳定的排序方法

*缺陷：

需要较多的辅助空间

存在与∞进行比较的冗余比较

3. 堆排序

*对一组待排序记录的关键字，首先把它们按堆的定义建成初始小(大)顶堆

*然后输出顶堆的最小（大）关键字所代表的记录，在调整剩余的关键字仍为堆，直到全部关键字排成有序序列为止

*称这个自堆顶至叶子的调整过程为“筛选”

时间复杂度 O（nlogn）

*对深度为 k 的堆，筛选所需进行的关键字比较的次数至多为 2(k-1)

*对 n 个关键字，建成深度为 **h=[log2n]+1** 的堆，所需进行的关键字

比较的次数至多为 **4n**

空间复杂度 O(1)

仅需一个记录大小供交换的辅助空间

不**稳定**的排序方法

堆排序适用于记录数较大的排序，记录数较少时不提倡

9.5 归并排序

*将两个或两个以上的有序子序列合并为一个新的有序序列

*可将 n 个记录的一个无序序列看成是由 n 个长度为 1 的有序子序列组成的序列，然后进行两两归并，得到 2 或 1 的有序子序列，再两两归并，直到最后形成包含 n 个记录的一个有序序列的排序方法称为 2-路归并排序

时间复杂度 O（nlogn）

每一趟归并的时间复杂度为 O(n)，总共需进行 log2n 趟

空间复杂度 O(n)

一个辅助数组 O（n）以及递归形式的栈空间 O（logn）

稳定的排序方法

9.6 基数排序

时间复杂度 O（d(n+rd)）

每一趟分配：O（n）

每一趟收集：O（rd）

空间复杂度 O(rd)

2*rd 个队列指针

稳定的排序方法

适用于元素数目 n 很大且关键字很小的情况

9.7 内部排序方法的比较