数据结构与算法 课程复习提纲（一）

> 李明哲
>
> 计算学部金牌讲师团

第1章 绪论

- **算法时间复杂度和空间复杂度的分析与计算**

1.1 数据结构的基本概念

1.1.1 数据结构的基本概念

1.1.2 数据结构三要素

- 逻辑结构： 集合、线性结构、树形结构、图状或网状结构
- 存储结构：顺序存储、链式存储、索引存储和散列存储
- 运算的定义是针对逻辑结构的，运算的实现是针对存储结构的

1.2 算法和算法评价

- 算法的五个重要特性：有穷性、确定性、可行性、输入、输出
- 好的算法需要达到的目标：正确性、可读性、健壮性、高效率和低存储量需求
- **时间复杂度**
- **空间复杂度**

第2章 线性表

- 线性表的基本概念
- **线性表的实现（顺序存储、链式存储）**
- 线性表的应用

2.1 线性表的定义和基本操作

2.1.1 线性表的定义

- 线性表是具有**相同数据类型**的 $n\ (n \leqslant 0)$ 个元素的**有限序列**（a_1 为表头元素）
- 线性表是一种**逻辑结构**，表示元素之间一对一的相邻关系。顺序表和链表是指**存储结构**，两者属于不同层面的概念

2.1.2 线性表的基本操作

2.2 线性表的顺序表示

2.2.1 顺序表的定义

- 线性表的顺序存储又称顺序表
- 线性表中元素的位序是从 1 开始的，而数组中元素的下标是从 0 开始的
- 优点：
  - 顺序表**存储密度高**，每个结点只存储数据元素
  - 顺序表最主要的特点是**随机访问**，即通过首地址和元素序号可在时间 $O(1)$ 内找到指定的元素
- 缺点：
  - 顺序表逻辑上相邻的元素物理上也相邻，所以**插入和删除操作需要移动大量元素**
  - 当线性表长度变化较大时，难以确定存储空间的容量
  - 造成存储空间的“碎片”
- 顺序存储三个属性：存储空间的起始位置、线性表的最大存储容量、线性表的当前长度
- 顺序存储在静态存储分配情形下，一旦存储空间装满就不能扩充，若再加入新元素，则会出现内存溢出，因此需要预先分配足够大的存储空间。预先分配过大，可能会导致顺序表后部大量闲置；预先分配过小，又会造成溢出
- 动态存储分配虽然存储空间可以扩充，但需要移动大量元素，导致操作效率降低，而且若内存中没有更大块的连续存储空间，则会导致分配失败

2.2.2 顺序表上基本操作的实现

2.3 线性表的链式表示

2.3.1 单链表的定义

- 线性表的链式存储又称单链表
- 利用单链表可以解决顺序表需要大量连续存储单元的缺点，但单链表附加指针域，也存在浪费存储空间的缺点
- 单链表是**非随机存取**的存储结构（顺序访问）
- 通常用**头指针**来标识一个单链表，为操作上的方便，也可在第一个结点之前附加一个**头结点**

- 头指针域头结点区分：头指针始终指向链表的第一个结点，而头结点是带头结点链表中的第一个结点，结点内通常不存储信息
- 引入头结点的优点：
    - 由于第一个数据结点的位置被存放在头结点的指针域中，因此在链表的第一个位置上的操作和在表的其他位置上的操作一致，无须进行特殊处理
    - 无论链表是否为空，其头指针都是指向头结点的非空指针，因此空表和非空表的处理也就得到了统一

2.3.2 单链表上基本操作的实现

2.3.3 双链表

2.3.4 循环链表

- 循环单链表 / 循环双链表
- 可设置尾指针

2.3.5 静态链表

- 此时指针域中的指针是结点的相对地址（数组下标），又称游标
- 与顺序表一样，静态链表也要预先分配一块连续的内存空间
- 总体来说，**静态链表没有单链表使用起来方便**

2.3.6 顺序表和链表的比较

第3章 栈、队列和数组

- 栈和队列的基本概念
- **栈和队列的顺序存储结构**
- **栈和队列的链式存储结构**
- **多维数组的存储**
- 特殊矩阵的压缩存储
- 栈、队列和数组的应用

3.1 栈

3.3.1栈的基本概念

- **栈** $(Stack)$ 是只允许在一端进行插入或删除操作的**线性表**
- 栈的操作特性可以明显地概括为**后进先出** `(Last In First Out, LIFO)`
- 栈的数学性质：n 个不同元素进栈，出栈元素不同排列的个数为 $\frac{1}{n+1}C_{2n}^{n}$ ，称为卡特兰数

3.1.2 栈的顺序存储结构

- 采用顺序存储的栈称为**顺序栈**
- 由于顺序栈的入栈操作受数组上界的约束，当对栈的最大使用空间估计不足时，有可能发生栈上溢
- **共享栈：**
  - 利用栈底位置相对不变得特性，可让两个顺序栈共享一个一维数组空间，将两个栈的栈底分别设置在共享空间的两端，两个栈顶向共享空间中间延伸
  - 是为了更有效地利用存储空间，两个栈的空间相互调节，只有在整个存储空间被占满时才发生上溢

3.1.3 栈的链式存储结构

- 采用链式存储的栈成为链栈
- 优点是便于多个栈共享存储空间和提高其效率，且不存在栈满上溢的情况
- 通常采用单链表实现，并规定所有操作都是在单链表的表头进行的

3.2 队列

3.2.1 队列的基本概念

- 队列是一种操作受限的线性表，只允许在表的一端进行插入，另一端进行删除
- 队列的操作特性是**先进先出** `(First In First Out, FIFO)`
- 允许插入（也称入队、进队）的一端称为队尾，允许删除（也称出队）的一端称为队首
- **队列的顺序存储：**
  - 队列的顺序实现是指分配一块连续的存储单元存放队列中的元素，并附设两个指针：队头指针 `front` 和队尾指针 `rear`
- **循环队列：**
  - 解决普通顺序队列“假溢出”的问题
  - 把存储队列元素的表从逻辑上视为一个环，指针的移动可利用除法取余运算 `%` 实现

- 区分队空还是队满的三种处理方式：
  - 牺牲一个单元来区分队空和队满，入队时少用一个队列单元，约定以“队头指针在队尾指针的下一位置作为队满的标志”（较为普遍的做法）
  - 类型中增设表示元素个数的数据成员
  - 类型中增设`tag`数据成员，以区分是队满还是队空

3.2.3 队列的链式存储结构

- 队列的链式表示成为链队列，它实际上是一个同时带有队头指针和队尾指针的单链表，通常带有头结点
- 用单链表表示的链式队列特别适合于数据元素变动比较大的情形，而且不存在队列满且产生溢出的问题
- 假如程序中要使用多个队列，与多个栈的情形一样，最好使用链式队列，这样就不会出现存储分配不合理和“溢出”的问题

3.2.4 双端队列

- 双端队列是指允许两端都可以进行入队和出队操作的队列，其元素的逻辑结构仍是线性结构，队列的两端成为前端和后端
- 允许在一端进行插入和删除，但在另一端只允许插入的双端队列称为输出受限的双端队列
- 允许在一端进行插入和删除，但在另一端只允许删除的双端队列称为输入受限的双端队列
- 若限定双端队列从某个端点插入的元素只能从该端点删除，则该双端队列就蜕变为两个栈底相邻接的栈

3.3 栈和队列的应用

3.3.1 栈在括号匹配中的应用

3.3.2 栈在表达式求值中的应用

- **中缀表达式转化为后缀表达式**
  - 从左到右进行遍历
  - 运算数，直接输出
  - 左括号，直接压入堆栈（括号是最高优先级，无需比较；入栈后优先级降到最低，确保其他符号正常入栈）
  - 右括号（意味着括号已结束），不断弹出栈顶运算符并输出直到遇到左括号（弹出但不输出）
  - **运算符，将该运算符与栈顶运算符进行比较**
    - 如果优先级高于栈顶运算符则压入堆栈(该部分运算还不能进行), 如果优先级低于等于栈顶运算符则将栈顶运算符弹出并输出，然后比较新的栈顶运算符（低于弹出意味着前面部分可以运算，先输出的一定是高优先级运算符，等于弹出是因为同等优先级，从左到右运算）
    - 直到优先级大于栈顶运算符或者栈空,再将该运算符入栈
  - 如果对象处理完毕,则按顺序弹出并输出栈中所有运算符
- 后缀表达式与中缀表达式对应的表达式树的后序遍历有异曲同工之妙
- 通过后缀表示计算表达式值：

- 顺序扫描表达式的每一项，然后根据它的类型做如下相应操作：若该项是操作数，则将其压入栈中；若该项是操作符`<op>`，则连续从栈中退出两个操作数`Y`和`X`，形成运算指令`X<op>Y`，并将计算结果重新压入栈中。当表达式的所有项都扫描并处理完后，栈顶存放的就是最后的计算结果

3.3.3 栈在递归中的应用

- 若在一个函数、过程或数据结构的定义中又应用了它自身，则这个函数、过程或数据结构称为是递归定义的，简称递归
- 递归通常把一个大型的复杂度问题层层转化为一个与原问题相似的规模较小的问题来求解，递归策略只需少量的代码就可以描述出解题过程所需要的多次重复计算，大大减少了程序的代码量
- 但在通常情况下，由于递归调用过程中包含很多重复的计算，它的效率并不太高
- 可将递归算法转换为非递归算法，通常需要借助栈来实现这种转换
- **注意消除递归不一定需要使用栈，对于单向递归和尾递归，可以用迭代的方式来消除递归**

3.3.4 队列在层次遍历中的应用

3.3.5 队列在计算机系统中的应用

- 解决主机与外部设备之间速度不匹配的问题
  - 由于主机和打印机之间速度不匹配，故需设置一个打印数据缓冲区，这个缓冲区中所存储的数据就是一个队列
- 解决由多用户引起的资源竞争问题
  - 在一个带有多终端的计算机系统上，有多个用户分别通过各自的终端向操作系统提出占用 CPU 的请求。操作系统通常按照每个请求在时间上的先后顺序，把它们排成一个队列，每次把 CPU 分配给队首请求的用户使用

3.4 数组和特殊矩阵

3.4.1 数组的定义

- 数组是由 $n\ (n \geqslant 1)$ 个相同类型的数据元素构成的有限序列，每个数据元素称为一个数组元素
- 每个元素在 n 个线性关系中的序号称为该元素的下标，下标的取值范围称为数组的维界
- 数组与线性表的关系：**数组是线性表的推广**
- 数组一旦被定义，其维数和维界就不再改变，因此，除结构的初始化和销毁外，数组只会有存取元素和修改元素的操作
- 数组`A[n]`和`A[0…n-1]`的写法是等价的，如果数组写为`A[1…n]`，则说明指定了从下标`1`开始存储元素

3.4.2 数组的存储结构

- 对于多维数组，有两种映射方式：按行优先和按列优先
- 二维数组元素写为 $a[i][j]$，注意数组元素下标 i 和 j 通常是从 0 开始的；矩阵元素通常写为 $a_{i,\,j}$ 或 $a_{(i)(j)}$，注意行号 i 和列号 j 是从 1 开始的

3.4.3 特殊矩阵的压缩存储

- 压缩存储：指为多个值相同的元素只分配一个存储空间，对零元素不分配存储空间
- 特殊矩阵：指具有许多相同矩阵元素或零元素，并且这些相同矩阵元素或零元素的分布有一定规律性的矩阵
- 特殊矩阵的压缩存储方法：找出特殊矩阵中值相同的矩阵元素的分布规律，把那些呈现规律性分布的、值相同的多个矩阵元素压缩存储到一个存储空间中
- 对称矩阵
- 上（下）三角矩阵
- 三对角矩阵

3.4.4 稀疏矩阵

- 矩阵中非零元素（或为特定值的元素）个数 t，相对矩阵元素的个数 s 来说非常少，即 $s \gg t$ 的矩阵称为稀疏矩阵
- $\alpha = \frac{t}{s}$ ，称其为稀疏因子，通常认为 $\alpha \leqslant 0.05$ 时称为稀疏矩阵
- 将非零元素及其相应的行和列构成一个三元组（行标，列标，值）
- 稀疏矩阵压缩存储后便失去了随机存取特性
- 稀疏矩阵的三元组既可以采用数组存储，也可以采用十字链表法存储

第4章 串

- 字符串模式匹配

4.1 串的定义和实现

4.1.1 串的定义

- 串是由零个或多个字符组成的有限序列

4.1.2 串的存储结构

- 定长顺序存储表示
- 堆分配存储表示
- 块链存储表示

4.1.3 串的基本操作

- 串类型的最小操作子集：
  - 串赋值`StrAssign`
  - 串比较`StrCompare`
  - 求串长`StrLength`
  - 串联接`Concat`
  - 求子串`SubString`
- 除串清除`ClearString`和串销毁`DestoryString`外的其他串操作均可在该最小操作子集上实现

4.2 串的模式匹配

- 子串的定位操作通常称为串的**模式匹配**，它求得是子串（常称**模式串**）在主串中的位置

4.2.1 简单的模式匹配算法

- 最坏时间复杂度 $O(nm)$，其中 n 和 m 分别为主串和模式串的长度

4.2.2 串的模式匹配算法——KMP 算法

- 相关概念：
  - 前缀：除最后一个字符外，字符串的所有头部子串
  - 后缀：除第一个字符外，字符串的所有尾部子串
  - 部分匹配值`(Partial Match，PM)`：字符串的前缀和后缀的最长相等前后缀长度
- **移动位数 = 已匹配的字符数 − 对应的部分匹配值**
- 时间复杂度 $O(n+m)$，由于整个匹配过程中主串始终没有回退
- `next`**数组的计算**
  - 计算得到部分匹配值表
  - 将 PM 表的数值部分整体右移一位，左端补 -1
  - 将 PM 表的数值部分的所有元素 $+1$

    - 注：在实际 KMP 算法中，为了使公式更简洁、计算简单，如果串的位序是从 1 开始的，则 `next` 数组才需要整体加 1；如果串是从 0 开始的，则 `next` 数组不需要整体加 1
- **原理**
- 一般情况下，普通模式匹配的实际执行时间近似为 $O(m+n)$，因此仍被采用。 KMP 算法仅在主串与子串有很多“部分匹配”时才显得比普通算法快很多，其主要优点是主串不回溯

4.2.3 KMP 算法的进一步优化

- 避免出现 $P_j = P_{next[j]}$ 的情况
- 若出现此情况，则再次递归，将 `next[j]` 修正为 `next[next[j]]`，直至两者不相等为止