数据结构

题型

一、填空题

二、选择题

三、判断题

四、综合题

五、算法填空题（给代码，说明功能，扣空来填）三道题

六、算法阅读题（阅读一段长代码，阐述各函数功能，指出程序输出结果）一道题

七、算法设计题 一道题

第一章 绪论

数据结构的基本概念

基本概念和术语

1. 数据：数据是信息的载体，是描述客观事物属性的数、字符及所有能输入到计算机中并被计算机程序识别和处理的符号的几何，是计算机程序加工的原料。
2. 数据元素：是数据的基本单位，通常作为一个整体进行考虑和处理。
3. 数据项：一个数据元素可由若干数据项组成，是构成数据元素的不可分割的最小单位。
4. 数据对象：具有相同性质的数据元素的集合，是数据的一个子集。
5. 数据结构：相互之间存在一种或多种特定关系的数据元素的集合。数据元素相互之间的关系称为结构
6. 数据类型：一个值的集合和定义在此集合上的一组操作
    a. 原子类型：其值不可再分的数据类型
    b. 结构类型：其值可以再分解为若干成分
    c. 抽象数据类型（ADT）：抽象数据组织及与之相关的操作

> 算法的设计依赖于逻辑结构，算法的实现依赖于存储结构

数据结构的三要素

- 逻辑结构——数据元素之间的逻辑关系
  - 线性结构
  - 非线性结构
    - 集合
    - 树形结构
    - 图形结构或网状结构

> 逻辑结构与数据的存储无关，是独立于计算机的

- 数据的运算——针对某种逻辑结构，结合实际需求，定义基本运算
  - 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
- 物理结构（存储结构）——如何用计算机表示数据元素之间的逻辑关系（又称映像）
  - 顺序存储：逻辑上相邻的元素在物理位置上也相邻
  - 链式存储：逻辑上相邻的元素在物理位置上可以不相邻
  - 索引存储：在存储数据元素的同时，还建立附加的索引表（关键字，地址）
  - 散列存储：根据元素的关键字计算元素的存储地址，又称哈希存储

> 顺序存储可能产生较多的外部碎片
>
> 链式存储不会产生碎片现象
>
> 散列存储可能会出现存储单元冲突的现象，解决冲突会增加时间和空间开销

算法的基本概念

程序 = 数据结构 + 算法

算法：针对特定问题求解步骤的一种描述，是指令的有限序列，其中的每条指令表示一个或多个操作。

算法的特性

- 有穷性：一个算法必须在执行有穷步之后结束，且每一步都可以在有穷时间内完成
- 确定性：对于相同的输入必须得出相同的输出
- 可行性：可以通过已实现的基本运算执行有限次来实现
- 输入：零个或多个输入
- 输出：一个或多个输出

算法必须是有穷的，程序可以是无穷的

“好”算法的特质

- 正确性
- 可读性
- 健壮性：对于非法输入，可以适当做出反应或处理
- 高效率和低存储量需求

算法效率的度量

时间复杂度

$$O(1) < O(log_2 n) < O(n) < O(nlog_2 n) < O(n^2) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$

$$T(n) = T(n_1) + T(n_2) = O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$$

空间复杂度

算法原地工作是指算法所需的辅助空间为常量，即O(1)

第二章 线性表

线性表的定义和基本操作

线性表的定义

相同数据类型、有限序列、表头元素、前驱、后继

基本操作

- 初始化
- 求表长
- 按值查找
- 按位查找
- 插入
- 删除
- 判空
- 销毁

顺序表

存储结构

逻辑上相邻的元素物理上也相邻

实现方式

- 静态分配
  - 数组大小和空间事先已固定
- 动态分配
  - 一旦数据空间占满，就另外开辟一块更大的存储空间，用以替换原来的存储空间

特点

- 支持随机访问：能在O(1)时间内找到第i个元素
- 存储密度高
- 拓展容量不方便
- 插入、删除数据元素不方便

顺序表上的基本操作

插入

最好情况：新元素插入到表尾，O(1)

最坏情况：新元素插入到表头，O(n)

平均情况：新元素插入到任何一个位置的概率都相同，O(n)

删除

最好情况：删除表尾元素，O(1)

最坏情况：删除表头元素，O(n)

平均情况：删除任何一个元素的概率均相同，O(n)

> 顺序表的插入和删除操作的时间主要耗费在移动元素上

查找

按位查找

获取表中第i个位置的元素的值

时间复杂度：O(1)

按值查找

查找具有指定关键字值的元素

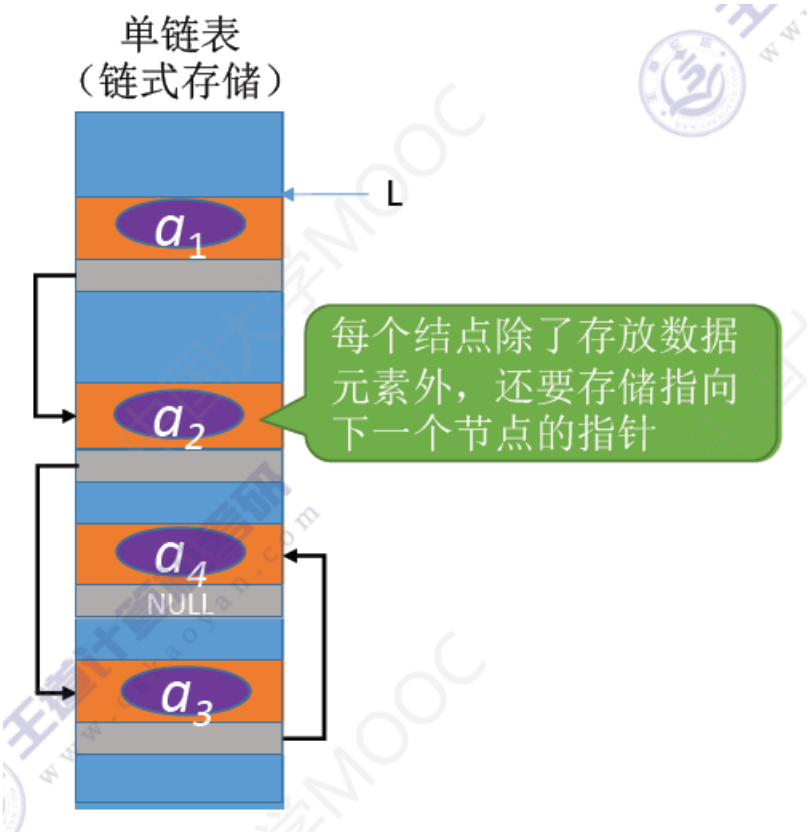
最好情况：目标元素在表头，O(1)

最坏情况：目标元素在表尾，O(n)

平均情况：目标元素出现在任何一个位置的概率相同，O(n)

单链表

存储结构



实现方式

- 带头节点
- 不带头节点

> 带头节点的判空操作
>
> L->next == NULL
>
> 不带头节点的判空操作
>
> L==NULL

特点：

- 优点：
  - 不要求大片连续空间，改变容量方便
- 缺点：
  - 不可随机存取，要耗费一定空间存放指针

单链表上的基本操作

初始化

- 头插法
  - 实现简单
  - 生成的链表的次序与输入数据顺序相反
- 尾插法
  - 需要维护一个尾指针

插入

按位序插入

在表中第i个位置上插入指定元素

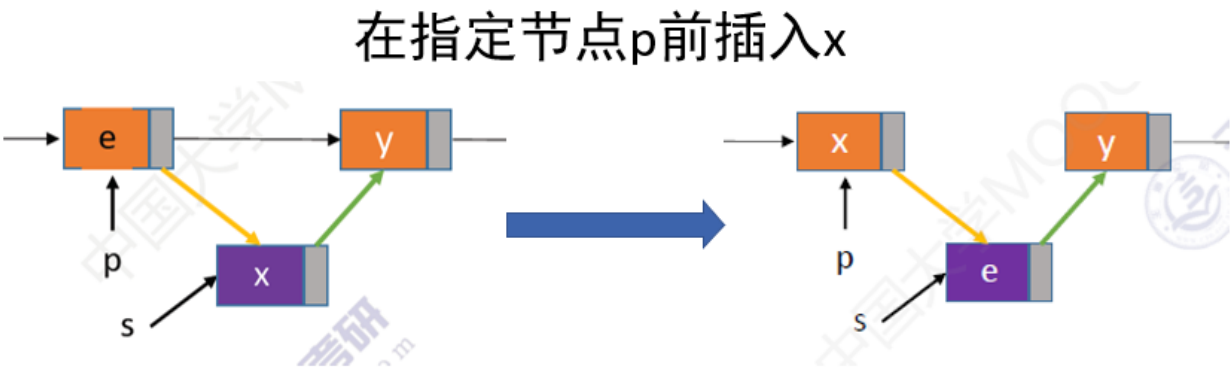
平均时间复杂度：O(n)

指定节点的后插入操作

时间复杂度：O(1)

指点节点的前插操作

时间复杂度：O(1)



删除

按位序删除

删除表中第i个位置的元素，返回被删除元素的值

最好情况：删除的元素是第一个元素，O(1)

最坏情况：删除的元素是最后一个元素，O(n)

平均情况：O(n)

指定节点的删除

删除给定节点p

在删除非尾节点时：O(1)

> 将p的下一个节点的值复制到p节点中，再将p的next指针指向p的下下个节点(可能是NULL)，然后释放下个节点

在删除尾节点时：O(n)

> 因为是尾节点，因此没有下一个节点，从而无法使用上述方法删除节点，因此只能迭代查找尾节点的父节点

查找

按位查找

获取表中第i个位置的元素的值

平均时间复杂度：O(n)

按值查找

查找具有给定关键字的元素

平均时间复杂度：O(n)

求表长

时间复杂度：O(n)

双链表

双链表结点中有两个指针prior和next，分别指向其前驱结点和后继结点

双链表的基本操作

插入

边界情况：新插入结点在最后一个位置，需要特殊处理

删除

边界情况：删除结点是最后一个结点，需要特殊处理

> 双链表的操作与单链表基本相同，但是要注意双链表指针的处理

遍历

从给定结点开始，后向、前向遍历（注意循环终止条件）

循环链表

表结尾的next指针指向头结点

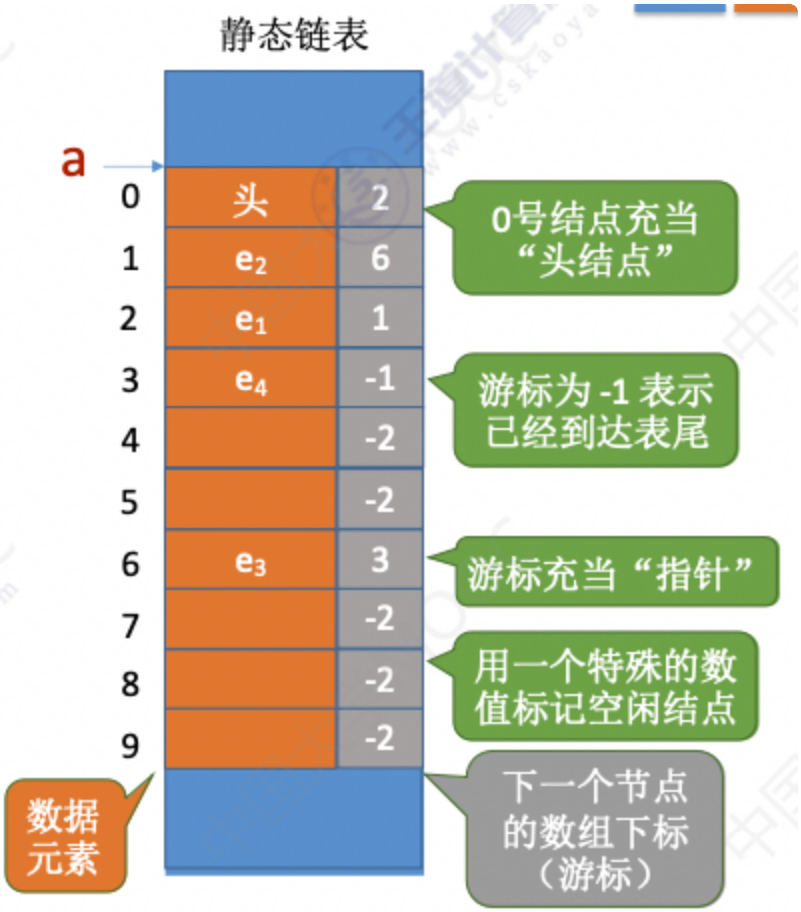
实现方式

- 循环单链表
- 循环双链表

> 循环链表初始时，头结点的prior和next指针都指向自身，可作为判空的条件

静态链表

静态链表借助数组来描述线性表的链式存储结构，结点也有数据域和指针域，这里的指针是指结点在数组中的下标



特点

- 优点
  - 增、删操作不需要大量移动元素
- 缺点
  - 不能随机存起，容量固定不可变
- 适用于不支持指针的低级语言

> 静态链表的第一个结点初始化时将其指针置为–1，其他结点指针初始化为一个特殊的值代表空闲
>
> 静态链表数组的第一个结点和最后一个结点一般作特殊处理，不存放元素

顺序表 vs. 链表

逻辑结构

均属于线性表

存储结构

	顺序表（顺序存储）	链表（链式存储）
优点	支持随机存取、存储密度高	离散的小空间分配方便，改变容量方便
缺点	大片连续空间分配不方便，改变容量不方便	不可随机存取，存储密度低

基础操作

	顺序表	链表
按值查找	O(n) ——当顺序表有序时可用折半查找，此时为O(log n)	O(n)
插入	O(n) ——时间开销主要来自于移动元素	O(n) ——时间开销主要来自于找到目标元素
删除	O(n) ——时间开销主要来自于移动元素	O(n) ——时间开销主要来自于找到目标元素

一元多项式的表示方法和基本运算实现

第三章 栈、队列和数组

栈

栈的的基本概念

栈是只允许在一端进行插入或删除的线性表

栈顶、栈底、空栈

栈的操作特性可概括为后进先出（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 $\frac{1}{n+1}C_{2n}^{n}$ ，该公式称为卡特兰数

基本操作：

- 初始化
- 判空
- 进栈
- 出栈
- 读栈顶元素
- 销毁栈

栈的顺序存储结构

采用顺序存储的栈称为顺序栈，附设一个指针指向当前栈顶元素的位置

顺序存储存在栈满上溢的情况

基本操作：

- 初始化

  top = –1 //初始化栈顶指针

- 判空

  if (top == –1)

  ...

- 栈满

  if (top == MaxSize – 1)

  ...

> 这里top指向的是栈顶元素，如果题目中给出的top指针的初始状态或数组的开始索引不同，则top指针的初始位置、栈空的条件、栈满的条件均会发生改变

共享栈

利用栈底位置不变的特性，可以让两个顺序栈共享一个一维空间，将两个栈的栈底分别设置在数组的两端，两个栈向数组中间延申。

基本操作：

- 初始化

  top1 = –1

  top2 = MaxSize

- 栈满

  top1 == top0 + 1　// 即两指针相邻

栈的链式存储

通常采用单链表实现，规定所有操作都是在链表表头进行的

如果要在初始化一个非空栈，则需要采用头插法构造链表

队列

队列的基本概念

尾入头出

只允许在一段进行插入，在另一端进行删除的线性表

操作特性：先进先出（FIFO）

出队、离队、入队、进队、队头、队尾、空队列

基本操作：

- 初始化
- 判空
- 入队
- 出队
- 读队头元素

队列的顺序存储结构

分配一块连续的存储单元存放队列元素，附设两个指针：头指针front和尾指针rear

- front头指针指向队头元素
- rear尾指针指向队尾元素的下一个位置

不能使用rear == MaxSize作为栈满条件

会出现”假溢出“现象

基本操作：

- 初始化：front = rear = 0
- 判空：front == rear
- 进队：rear ++
- 出队：front ++

循环队列

将顺序队列视为环形空间

基本操作：

- 初始化：front = rear = 0
- 队列长度：(rear + MaxSize – front)%MaxSize

区分队满和队空的三种处理方式

1. 牺牲一个单元来区分栈空和栈满

   约定队头指针在队尾指针的下一位置作为栈满条件

   队空：front==rear

   队满：(rear + 1)%MaxSize == front
2. 在队列中增设表示元素个数的成员变量
3. 增设tag成员，以区分队满还是队空

   若因删除导致front==rear，则tag置为1

   若因插入导致front==rear，则tag置为0

队列的链式存储结构

采用链式表示的链队列，附设头指针和尾指针的单链表，队头指针指向队头，尾指针指向队尾

基本操作：

- 判空：front==NULL && rear == NULL
- 初始化：front == rear == NULL

> 在不带头节点时，第一个元素入队和最后一个元素出队要特殊处理

双端队列

双端队列：允许两端都可以进行入队和出队操作，队列两端称为前端和后端，两端都可以入队和出队

输出受限的双端队列：只有一端可以输出，两端都可输入

输入首先的双端队列：只有一端可以输入，两端都可输出

栈和队列的应用

括号匹配——栈

算法思想：

1. 初始化一个空栈，顺序读入括号
2. 若是左括号，则作为一个新的更急迫期待入栈
3. 若是右括号，则尝试弹出一个栈顶元素表示匹配成功，若弹出失败（栈空）则表示插线了括号不匹配的情况

表达式求值——栈

运算符优先级：

1. []()
2. */
3. +-

中缀表达式转后缀表达式

算法思想：

1. 遇到操作数则直接输出（添加到后缀表达式中）

2. 栈为空时，遇到运算符则入栈
3. 遇到左括号则入栈
4. 遇到右括号：重复执行出栈操作，将出栈的元素依次弹出（加入后缀表达式），直到弹出第一个左括号
5. 遇到其他运算符：弹出所有优先级大于或者等于该运算符的栈顶元素（左括号视为最低级，加入后缀表达式），然后将该运算符入栈
6. 最终将栈中元素依次弹出（加入后缀表达式）

后缀表达式计算

算法思想：

1. 从左向右处理元素，直到处理完所有元素
2. 若扫描到操作数，则压入栈，并回到1
3. 若扫描到运算符，则弹出两个栈顶元素，执行相应运算并将运算结果压回栈顶，回到1

中缀表达式的计算

算法思想：

1. 初始化两个栈，操作数栈和运算符栈
2. 从左向右处理元素
3. 若扫描到操作数，则压入操作数栈
4. 若扫描到运算符，则按照”中缀转后缀“相同的逻辑压入运算符栈（期间每弹出一个运算符，都需要在从操作数栈弹出两个操作数做运算，并将运算结果压回操作数栈）

递归——栈

递归的精髓是将原始问题转化为属性相同的规模更小的问题

层次遍历——队列

算法思想：

1. 根节点入队
2. 若队空，则表示遍历结束
3. 第一个节点出栈，并访问之，将其左右孩子顺序入队

计算机中的应用——队列

缓冲区

数组

下标、维界、数组元素

数组与线性表的关系：一维数组可被视为一个线性表，二维数组可视为数据元素为一维数组的线性表

特殊矩阵的压缩存储

特殊矩阵：指具有许多相同矩阵元素或零元素，并且这些相同的矩阵元素或零元素有一定的分布规律的矩阵，如：上三角、下三角、对角

压缩存储：指为多个值相同的元素只分配一个存储空间，对零元素不分存储空间

特殊矩阵：

- 对称矩阵——只存放下三角区（含主对角）的元素

  $a_{i,j} = a_{j,i}$

- 三角矩阵——只存放主对角线、上三角区的所有元素和下三角区的常量一次（上三角矩阵）

  上三角矩阵、下三角矩阵

- 三对角矩阵——将三条对角线上的元素按行优先的方式存储在一维数组中

$$\begin{bmatrix} a[0][0] & a[0][1] & 0 & \cdots\cdots & \cdots\cdots & \cdots\cdots & 0 \\ a[1][0] & a[1][1] & a[1][2] & \cdots\cdots & \cdots\cdots & \cdots\cdots & \vdots \\ 0 & a[2][1] & a[2][2] & a[2][3] & \cdots\cdots & \cdots\cdots & \vdots \\ 0 & 0 & a[3][2] & a[3][3] & a[3][4] & \cdots\cdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \cdots\cdots & a[n-2][n-1] \\ 0 & \cdots\cdots & \cdots\cdots & \cdots\cdots & \cdots\cdots & a[n-1][n-2] & a[n-1][n-1] \end{bmatrix}$$

三对角矩阵

稀疏矩阵

矩阵中非零元素个数远小于矩阵元素个数

仅存储非零元素，即只存储三元组：（行坐标，列坐标，值）

三元组的存储可以采用一维数组，也可以采用十字链表

稀疏矩阵经压缩后失去了随机存储的特性

第四章 串

串的基本概念

计算机上非数值处理的对象基本都是字符串数据

空串、子串、主串、位置、空格串

串是线性结构

串的存储结构

顺序存储

- 静态数组
- 动态数组：动态分配数组存储串，动态分配管理的空间处于‘堆’中

块链存储

采用链式存储结构存储串

具体实现时，每个结点既可以存放一个字符，也可以存放多个字符，每个结点成为块，整个链表成为块链结构

块中没有字符的位置用‘#’或‘\0’补足

串的模式匹配

模式匹配：从主串中找到与模式串相同的字串，并返回其所在位置

朴素模式匹配算法

暴力算法

平均时间复杂度：O(mn)

KMP算法

主串不回溯（KMP算法的主要优点），子串根据next数组回溯

KMP算法的平均时间复杂度为：O(m+n)，理论上要优于朴素的匹配算法，但是一般情况下朴素模式匹配算法的时间复杂度也近似于O(m+n)，这是因为实际情况中并不会出现频繁回溯的情况。KMP算法仅在主串和子串有很多的”部分匹配“时才显得比普通算法快的多。

> 以下说明均以“数组索引从1开始”这一假设为基础

基本概念

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：字符串的前缀和后缀的最长相等前后缀长度

原理说明

公式：移动位数（子串回溯位数）=已匹配字符数–对应的部分匹配值

部分匹配值表（例：abcac）：

编号	1	2	3	4	5
S	a	b	c	a	c
PM	0	0	0	1	0

每当匹配失败，就找前一个元素的部分匹配值（对应公式里的“对应部分匹配值”）。但是这样有写不方便，因此将部分匹配值表右移一位得到next数组，第一个元素的PM值右移后的空缺用–1填充

编号	1	2	3	4	5
S	a	b	c	a	c

PM	–1	0	0	0	1

根据移动位数可知：

若设当前发生不匹配时子串的匹配指针为j，则移动位数Move = (j–1) – next[j]

从而可知移动后的指针j = j – Move = j– ((j–1) – next[j]) = next[j] + 1

综上，为了让公式更加简洁，因此在原有的next数组的基础上，将next数组整体+1（部分题目中并不会进行这项处理，此时第一个元素的next值就是–1 ），得到最终的next数组

编号	1	2	3	4	5
S	a	b	c	a	c
PM	0	1	1	1	2

此时next[j]的含义为：在子串第j个字符与主串发生失配时，将跳转到子串的next[j]位置重新与主串当前位置进行比较（当失配后指针j被置为0时，说明此时在子串的第一个字符除就发生了不匹配，这个时候就将子串和主串的指针都+1（主串指针+1：去匹配主串中下一个字符，子串指针+1：0–>1 下次匹配比较的是子串的第一个字符），后继续进行新一轮匹配）

> KMP算法原理的说明均以“数组索引从1开始”这一假设为基础

算法总结

KMP算法可分为两部分：next数组的求解 + 匹配算法

next数组

$$next[j] = \begin{cases} 0 & j = 1 \\ mac\{k|1 < k < j且”p1...p_{k-1}” = ”p_{j-k+1}...p_{j-1}”\} & 当此集合非空时 \\ 1 & 其他情况 \end{cases}$$

直接手算next数组：

- 如果是第1个元素，直接置为–1
- 如果是第二个元素，直接置为0
- 如果是其他元素，查看这个元素之前的串（不包含这个元素）的部分匹配值，置为匹配值
- 最后将根据题目，判断是否要将所有next数组的值+1

匹配算法

KMP C++

```
int i =1, j = 1;
while(i<=S.length && j<=T.length){
    if(j=0||S.ch[i]=T.ch[j]){
        ++i; ++j;
    }
    else {
        j=next[j];
    }
}
if(j>T.length)
    return i-T.length   //匹配成功
else
    return 0;
```

KMP算法的进一步优化

对于 $p_j = p_{next[j]}$ 这种情况，在j位置发生失配时，根据KMP算法原理，子串指针就会回溯到next[j]，此时会发生再次失配，即进行了一次无意义的回溯和一次无意义的匹配。

为了处理这种情况，提出了KMP算法的一种优化方式，即针对计算得出的next数组，从左向右依次检测是否有 $p_j = p_{next[j]}$ 这种情况，如果发现这种情况，就进行 $next[j] = next[next[j]]$ 处理得到新的nextval数组。

第五章 树与二叉树

树的基本概念

空树

任意非空树：

- 有且仅有一个根节点
- 当n>1时，其余节点可分为若干个不相交的有限集，每个集合本身又是一棵树

树的定义是递归的，树是一种递归的数据结构

树作为一种逻辑结构，同时也是一种分层结构

祖先、子孙、双亲、兄弟、叶子节点

结点的度：孩子的个数

度为n的树：所有的结点的度最多为n，且至少存在一个度为n的结点

结点的层次：根据题目，根节点可能是1层也可能是0层

结点的高度：从叶节点开始，逐层递加

结点的深度：从根节点开始，逐层递加

树的高度：结点的最大层数

有序树：各子树从左到右是有次序的，不能互换

无序树

路径和路径长度：树中的分支是有方向的，因此路径也是从上向下的（从根到叶），兄弟节点之间不存在路径

森林：互不相交的树的集合

树的基本性质：

- 树中的结点数 = 结点度数之和 + 1
- 度数为m的数中，第i层上最多有 m^{i-1} 个结点（i>=1）
- 高度为h的m叉树最多有 $(m^h-1)/(m-1)$ 个结点
- 具有n个结点的m叉树的最小高度为 $\lceil log_m(n(m-1)+1) \rceil$
- m叉树第i层至多有 m^{i-1} 个结点

二叉树

二叉树的基本性质

每个结点至多有两个子树

二叉树的子树有左右之分，即二叉树是有序树

哪怕某个结点只有一颗子树，也要区分他是左子树还是右子树

二叉树和度为2的有序树的区别：

- 度为2的树至少有3个结点，而二叉树可以为空

- 度为2的有序树在只有一个子树的情况下不需要区分左右

几类特殊的二叉树

满二叉树

高度为h，含有 2^h-1 个结点的二叉树，即树中每层都含有最多的结点

完全二叉树

高度为h，含有n个结点的二叉树

当且仅当其每个结点的编号都与满二叉树的前n个结点的编号一致时，被称为完全二叉树

完全二叉树的性质

- 若 $i \leq \lfloor \frac{n}{2} \rfloor$ ，则结点i为分支结点，否则结点为叶子结点
- 叶子结点只可能在层次最大的两层上
- 如果有度为1的结点，则只可能有一个，且该结点只有左孩子
- 若n为奇数，则每个分支都有左结点和右结点；若n为偶数，则编号最大的分支结点只有左孩子
- 完全二叉树可以根据节点数n，推出1度、2度、0度结点的个数

二叉排序树

$$左 < 根 < 右$$

构造：同样的数字，不同的顺序，构造出来的排序树也可能不同

删除：

1. 删除结点若为叶子结点，则直接删除
2. 删除结点只有左孩子或右孩子，则删除该结点，并用其孩子替代他
3. 删除结点即有左孩子又有右孩子：令删除结点与其直接前驱交换数值，后删除直接前驱（此时前驱中的数值为待删除数值，过程参考2）

平衡二叉树

树上任一结点的左右子树的深度之差不超过1

二叉树性质

- $n_0 = n_2 + 1$
- 非空二叉树上第k层上至多有 2^{k-1} 个结点
- 高度为h的二叉树至多有 $2^h - 1$ 个结点
- 具有n个结点的完全二叉树的高度为 $\lceil log_2(n+1) \rceil$ 或 $\lfloor log_2 n \rfloor + 1$

二叉树的存储结构

顺序存储结构

二叉树的顺序存储使用一组地址连续的存储单元自上而下、自左至右完全存储二叉树上的结点

顺序存储结构比较适合存储完全二叉树和满二叉树——树中结点的序号可以唯一地反映结点之间的逻辑关系

对于一般的二叉树，为了能让数组下标反应结点之间的逻辑关系，只能添加一些空结点

一定要把二叉树的结点编号与完全二叉树对应起来

链式存储结构

左指针域lchild + 数据域data + 右指针域rchild

含有n个结点的二叉链表中含有n+1个空域

二叉树的遍历和线索二叉树

二叉树的遍历

常见次序：

- 先序
- 中序
- 后序

不管那种次序，时间复杂度均为O(n)

空间复杂度为O(h)——h为数的深度，空间代价来自于递归操作用到的栈

> “序”是指根节点的何时被访问

先序遍历

前缀表达式

根左右

中序遍历

中缀表达式（结果缺少括号）

左根右

后序遍历

后缀表达式

左右根

递归算法和非递归算法的转换——借助栈或队列

一般性遍历思路：

- 从根结点出发，画一条路
- 如果左边还有路没走，则往左走
- 如果左边没路了就往右走
- 如果左右都没有路了，或者左右都走过了就往上走

先序遍历–>第一次路过时访问结点

中序遍历–>第二次路过时访问节点

后续遍历–>第三次路过时访问节点

中序：

- 迭代访问左孩子，并把当前结点入栈
- 如果当前结点为空，则出栈一个元素，访问它，后再访问它的右孩子

先序：

- 迭代访问当前结点，并把当前结点入栈，下一个访问的结点就是当前结点的左孩子
- 如果当前节点为空，则出栈一个元素，下一次访问的结点就是出栈结点的右孩子

后续：

1. 沿根的左孩子，依次入栈，直到左孩子为空

2. 读栈顶元素，若其右孩子不空且未被访问过，将右子树转执行1，否则栈顶元素出栈并访问

层序：

需要借助一个队列

- 从队列头获取一个结点，将其孩子结点入栈并访问该结点

由遍历序列构造二叉树

中序 + 先序/后续/层序 能唯一确定一颗二叉树（必有中序）

线索二叉树

链式存储结构的含有n个结点的二叉树，有n+1个空域

利用空域存放指向其前驱或后继结点的指针

若无左子树，则令lchild指向其前驱；若无右子树，则令rchild指向其后继

还需增加两个标志位ltag和rtag，用以标识其左右指针是指向孩子还是指向前驱后继

ltag或rtag取0,代表指向孩子

线索：用以指向前驱后继的指针

线索化核心：

- 访问一个结点时，连接该节点与前驱结点的线索信息
- 用一个指针pre存储当前结点的前驱结点

易错点：

- 最后一个结点的rchild==NULL和rtag==1
- 先序线索化中，只有ltag==0时，才能对左子树线索化

中序线索二叉树

中序遍历线索树线索化：

1. 如果当前结点非空,则将其左子树线索化
2. 执行完1后,如果其左孩子为空,则将左孩子指针指向当前结点的父节点
3. 如果当前结点父节点的右子树为空,则将父节点的右孩子指针指向当前结点
4. 线索化当前节点的右子树

中序线索二叉树的遍历:

先找到序列中第一个结点(最左结点),后依次查找其后继

找后继:

- 如果当前结点rtag为1,则直接返回右孩子指针
- 如果当前结点rtag为0,则访问右子树的最左结点

先序线索二叉树和后序线索二叉树

- 先序线索树在当前结点有左孩子时,找不到当前结点的前驱
- 后序线索树在当前节点有右孩子时,找不到当前节点的后继

> 只有中序线索树在任何情况下都能找到前驱后继

树 森林

树

树的存储结构

双亲表示法

用一维连续空间存储结点,每个结点中增设一个指针指向其父节点

孩子表示法

每个结点的孩子结点都用单链表连接起来,n个结点就有n个孩子链表

孩子兄弟表示法

每个结点包含三部分内容:结点值　指向该节点的第一个孩子结点的指针　指向该结点的下一个兄弟结点的指针

优缺点:

	优点	缺点
双亲表示法	找父节点方便	找孩子不方便
孩子表示法	找孩子方便	找父节点不方便

孩子兄弟表示法	可方便的实现树与二叉树的转换	找父节点不方便（可通过增设指向父节点的指针解决）

树 森林与二叉树的转换

左孩子 右兄弟

二叉树和树都可以用二叉链表作为存储结构,因此给定一棵树,可以找到唯一一颗二叉树与之对应

树转换为二叉树: 每个结点的左指针指向它的第一个孩子,右指针指向他的相邻右兄弟

树转化为二叉树的画法:

1. 兄弟之间连线
2. 对于每一个结点,只保留其与第一个孩子的连线

森林转化为二叉树:

1. 将每个树转化为二叉树
2. 每个树的根视为兄弟关系,连线
3. 以第一个树的根节点为根节点

树和森林的遍历

树的遍历

树的遍历有两种方式:

- 先根遍历-->对应二叉树的先序遍历
- 后根遍历-->对应二叉树的后序遍历
- 层次遍历

森林的遍历

- 先序遍历：逐个树进行先根遍历
- 中序遍历：逐个树进行后根遍历

树与二叉树的应用

哈夫曼树和哈夫曼编码

结点的带权路径长度：从根到任意结点的路径长度（经过的边数）与该节点上的权值的乘积

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：含有n个叶结点的二叉树中WPL最小的二叉树，也被称为最优二叉树

最优二叉树不唯一

哈夫曼树的构造

迭代：构造一个新节点，将当前最小的两个结点设为他的两个子树

哈夫曼编码

固定长度编码：每个字符用相同长度的二进制位表示

可变长度编码：允许对不用字符用不等长的二进制位

前缀编码：没有一个编码是另一个编码的前缀

出现的频度当作结点的值，构造哈夫曼树

构造出来的哈夫曼树不唯一

并查集

一种简单的集合表示

通常用树（森林）的双亲表示法作为并查集的存储结构，每个子集用一棵树表示

所有结点存储在一维数组中（顺序存储），索引代表元素序号，数组元素存储索引代表结点的父节点，根节点的父节点用负值表示，用根节点代表集合

并查集常用操作：

- 初始化：所有结点的父节点都置为-1，代表每个元素单独构成集合
- 合并两集合：将第二个集合的根节点的父节点置为第一个集合的根节点
- 查找元素所在集合：查找元素所在集合的根节点，时间复杂度 $O(log_2 n)$
- 判断两元素是否在同一集合：查找两元素所在集合的根节点是否相等

并查集中树高 $\leq \lfloor log_2 n \rfloor + 1$

第六章 图

图的基本概念

$G = (V, E)$

- V表示图G中顶点的非空有限集合
- E表示图G中顶点之间的关系（边）集合

图没有空图，至少要有一个顶点，可以没有边

有向图：E为有向边（也成为弧）的有限集合，|E|的取值范围为0~n(n–1)

无向图：E为无向边的有限集合，|E|的取值范围为0~n(n–1)/2

简单图：不存在重复边，且不存在顶点到自身的边

多重图：某两个顶点之间边数大于1，且允许顶点通过一条边与自身关联

> 数据结构只讨论简单图

完全图：任意两点之间都存在边（有向图中是方向相反的两条弧），即有n(n–1)条弧的有向图，有n(n–1)/2条边的无向图

子图：V'和E'均是V和E的子集，则G'是G的子图

生成子图：V'等于V

> 子图E'中的所有边关联的结点都要在V'中，否则也不能构成子图

连通：如果从v到w有路径存在，则称v和w是连通的

连通图：G中任意两点顶点都是连通的

连通分量：无向图的极大连通子图称为连通分量

> 连通、连通图、连通分量是对于无向图的概念

> 有n个顶点的无向图，如果其边数小于n–1，则必非连通

强连通：从v到w和从w到v都有路径

强连通图：任意一对顶点都是强连通的

强连通分量：有向图中极大强连通子图

> 强连通、强连通图、强连通分量是对于有向图的概念

> 有n个顶点的有向图，如果是强连通图，则最少有n个边

生成树：连通图的生成树是包含图中全部顶点的一个连通子图；若定点数为n，则生成树有n–1条边

生成森林：所有连通分量的生成树构成的集合

极大连通子图：根据子图的顶点，包含所有能包含的原图中的边

极小连通子图：保持图连通，且边数最少

顶点的出度：顶点作为起点的边

顶点的入度：顶点作为终点的边

顶点的度：对于无向图，就是与顶点关联的边的个数；对于有向图，则为出度+入度

> 有向图中顶点的出度之和==入度之和==顶点数

> 无向图中顶点的度之和 == 2*顶点数

带权图也称网

稀疏图、稠密图：$|E| < |V||logV|$ 为稀疏图，反之为稠密图

路径：v1到v2之间路径上的顶点序列

路径长度：路径上的边数

回路：第一个顶点和最后一个顶点相同的路径

简单路径：顶点不重复出现的路径

简单回路：除起点和终点外，顶点不重复出现的回路

距离：顶点之间最短路径的长度，若不存在则为无穷

有向树：一个顶点入度为0，其余顶点入度均为1的有向图

图的存储

邻接矩阵

用一个一维数组存储顶点信息，用一个二维数组存储边的信息

邻接矩阵性质：

- 对于无向图，邻接矩阵是对称阵（并且唯一），可考虑使用压缩存储
- 对于无向图，第i行非零元素个数正好是顶点的度
- 对于有向图，第i行非零元素个数，是第i个元素的出度；第i列非零元素的个数，是第i个元素的入度
- 计算|E|要遍历整个邻接矩阵
- 稠密图适合用邻接矩阵
- 对于矩阵 A ， A^n 的第i行第j列元素等于顶点i到顶点j的长度为n的路径数目

邻接表法

结合了顺序存储和链式存储方法，表示方式不唯一

链式存储部分是指为每个结点建立一个单链表，第i个单链表表示依附于顶点i的边（对于有向图是以第i个顶点为尾的弧），这些单链表称为边表。所有边表的头指针存储在一个顺序表中。

邻接表性质：

- 适用于稀疏矩阵
- 若为无向图，邻接表所需存储空间为O(|V|+2|E|)，对于有向图是O(|V|+|E|)
- 计算结点的入度不方便

十字链表

十字链表是针对有向图的一种链式存储方式

每个顶点和每条边都对应有一个结点，顶点结点集中用顺序表存储

顶点结点

data	firstin	firstout

弧结点

tailvex	headvex	hlink	tlink	info

顶点结点：

- data：数据
- firstin：第一个以该结点为弧头的弧
- firstout：第一个以该节点为弧尾的弧

弧结点：

- tailvex：弧尾结点索引
- headvex：弧头结点索引
- hlink：相同弧头的下一条弧
- tlink：相同弧尾的下一条弧
- info：注释

邻接多重表

邻接多重表是针对无向图的一种链式存储结构

与十字链表类似，每个边和每个顶点都有对应的结点，结点集中使用顺序表存储

顶点结点

data	firstedge

边结点

mark	ivex	ilink	jvex	jlink	info

顶点结点：

- data：数据
- firstedge：依附于顶点的第一条边

边结点：

- mark：标志域，用以在搜索时检测是否被搜索过
- ivex、jvex：边依附的两个结点索引
- ilink、jlink：依附于顶点ivex、jvex的下一条边

	邻接矩阵	邻接表	十字链表	邻接多重表
空间复杂度	$O(\|V\|^2)$	无向图 $O(\|V\|+2\|E\|)$ 有向图 $O(\|V\|+\|E\|)$	$O(\|V\|+\|E\|)$	$O(\|v\|+\|E\|)$
找相邻边	遍历行或列	必须遍历整个邻接表	方便	方便
删除边或顶点	删除边方便，删除点不方便	都不方便	很方便	很方便
适用于	稠密图	稀疏图	有向图	无向图
表示方式	唯一	不唯一	不唯一	不唯一

图的遍历

图的遍历可以用来判断图的连通性

- 无向图：DFS/BFS函数调用次数=连通分量数
- 有向图：对于强连通图，只需调用一次DFS/BFS

广度优先搜索（BFS）

需要借助一个辅助队列，需要一个数组来避免重复访问

类似于树的广度优先搜索

算法描述：

- 访问起点，并将起点的所有邻接结点加入队列
- 迭代从队列中拿出结点访问，并将其所有未访问过的邻接结点加入队列

性能

最坏时间复杂度：O(|V|)——所有结点都与起点邻接，此时队列需要存储|V|-1个结点

- 对于邻接表 每个顶点都需要搜索一次，每条边也需要访问一次

  O(|V|+|E|)

- 对于邻接矩阵 查找每个顶点的邻接点都需要扫描一行（或一列）一次

  $O(|V|^2)$

借助BFS求解单源最短路径

只能求解非带权图的单元最短路径问题

增加一个数组，用于存储起点到所有结点的距离

广度优先生成树

广度遍历的过程中访问的轨迹是一棵树——广度优先生成树

- 对于邻接矩阵，由于邻接矩阵有唯一性，因此此时广度优先生成树是唯一的
- 对于邻接表，由于邻接表并不唯一，因此此时广度优先生成树不是唯一的

深度优先搜索（DFS）

类似于先根遍历，是一个递归算法

算法概述：

- 访问起点
- 访问每一个结点的过程：
  a. 访问结点本身
  b. 访问结点的所有邻接结点

性能

空间复杂度：O(|V|)——递归工作站

- 对于邻接表，查找所有顶点的邻接点需要访问每个边两次次，访问所有顶点一次

  O（|V|+|E|）
- 对于邻接矩阵，查找每个顶点的临界点都需要扫描一行（或一列）一次

  $O(|V|^2)$

深度优先生成树

与广度优先生成树类似，在邻接表的情况下，生成树不唯一

图的应用

最小生成树

生成树包含所有的顶点和尽可能少的边

对于生成树，看去一条边则不连通，增加一条边则有回路

对于带权图，边的权值之和最小的生成树称为最小生成树

性质：

- 不唯一
- 边数为顶点数–1
- 权值之和唯一

Prim算法

不成环的加入相邻的权值最小的边，直到所有顶点都被纳入

时间复杂度： $O(|V|^2)$

> 时间复杂度不依赖于|E|，因此适用于求解稠密图

Kruskal算法

不成环的加入权值最小的边，直到所有点都联通

时间复杂度： $O(|E|log_2|E|)$

> 适用于边稀疏图

最短路径

Dijkstra算法

基于贪心策略，精确解

时间复杂度：对于邻接表和邻接矩阵都是 $O(|V|^2)$

不适用于带负权值的图

算法概述：

1. 初始化S集合、dist[]数组
    a. S为空
    b. dist[]数组元素为无穷，起点为0
2. 从集合V–S中挑出对应dist值最小的点v
3. 修改v的所有临界点的dist值，dist[i] = min{dist[i], arc[v][i] + dist[v]}，维护path数组，最后将v加入S集合中
4. 重复2和3直到所有的点都被加入到了S中

Floyd算法

Floyd算法支持带负权值的图，但不允许有带有负权值边的回路

时间复杂度: $O(|V|^3)$

Folyd算法代码紧凑，且不包含其他复杂度数据结构，因此算法的隐含的常数是很小的

算法概述：

1. 初始化方阵A(A=临界矩阵)，和路由矩阵B
2. 对每个点k，进行如下操作
    a. 对于每个元素i和每个元素j，如果A[i][k] + A[k][j] < A[i][j]，则更新A[i][j] = A[i][k] + A[k][j]
    b. 更新路由矩阵，如果a.中进行该修改，则执行B[i][j] = k

有向无环图（DAG）描述表达式

利用有向无环图可以实现对相同子式的共享

构造过程：

1. 把各操作数不重复地排成一排
2. 标出各个运算符的优先级
3. 按顺序加入运算符，注意“分层”
4. 自底向上逐层合并

拓扑排序

若图中有环，则不存在拓扑排序序列/逆拓扑排序序列

AOV网：顶点表示活动的网络，用DAG图表示一个工程，其顶点表示活动，用有向边<a,b>表示活动a必须先于活动b进行的这样一种关系，有向无环图

拓扑排序：满足以下条件的有向无环图的顶点构成的序列：

- 每个顶点仅出现一次
- 若顶点A在B前，则图中不存在从B到A的路径

每个AOV网都有一种或多种拓扑排序序列

一种拓扑排序算法（时间复杂度：邻接表O(|V|+|E|)，邻接矩阵 $O(|V|^2)$ ）：

1. 从AOC网中选择一个没有前驱的结点并输出
2. 删除该结点和所有以它为起点的边
3. 重复1和2，直到网为空或不存在无前驱结点

逆拓扑排序：

1. 从AOC网中选取一个无后继结点并输出
2. 删除该结点和所有以它为终点的边
3. 重复1和2直到网为空

DFS实现拓扑排序：参考先序遍历

DFS实现逆拓扑排序：参考后序遍历

补充：

- 无前驱的点可以当作工程起始点
- 若一个顶点有多个直接后继，则拓扑排序的结果通常不唯一
- AOC网边无权值，结点有权值

关键路径

AOE网：用边表示活动的网络，有向无环图

AOE网中有些活动是可以并行的

只有所有路径上的活动都完成，整个工程才能算结束

从源点到汇点的所有路径中，具有最大路径长度的路径成为关键路径（不唯一），关键路径上的活动称为关键活动

事件v的最早发生时间：

从源点到v的最长路径长度

计算：从源点开始逐层计算

事件v的最迟发生时间：

不推迟整个工程的前提下的最迟必须发生时间

计算：

- 从终点向前逐层计算
- 若vl[i] – Weight(i,j) < vl[k]，则vl[k] = vl[j] – Weight(i,j)

活动的最早开始时间：弧的起点代表的事件的最早发生时间

活动的最迟开始时间：活动弧终点的事件的最迟发生时间 – 活动时间

关键路径算法：

1. 计算所有事件的最早发生时间
2. 计算所有事件的最迟发生时间
3. 计算所有活动的最早开始时间
4. 计算所有活动的最迟开始时间
5. 找出所有最早开始时间=最迟开始时间的活动（关键活动）

第七章 查找

查找的基本概念

在数据集合中寻找满足特定条件的数据元素的过程称为查找，一般结果为查找成功/失败

查找表：用于查找的数据集合

静态查找表：不能进行修改的查找表

关键字：唯一标识元素的某个数据项的值

平均查找长度（ASL）

顺序查找和折半查找

顺序查找

又称线性查找

一般线性表的顺序查找

从一端到另一端，逐个检查

哨兵：在表的头部（或尾部）插入要查找的元素，这样在从后向前（或从前向后）的查找过程中就无需担心越界问题

$$ASL_{成功} = \frac{n+1}{2}$$

$$ASL_{失败} = n+1$$

有序表的顺序查找

可提早判断出查找失败，查找失败后便不在继续比较

$$ASL_{失败} = \frac{n}{2} + \frac{n}{n+1}$$

折半查找

又称二分查找，仅适用于有序的顺序表

折半查找过程可用二叉树表示（称为判定树）,是平衡二叉树, 只有最下层可以不满

在[low, heigh]之间查找关键字,每次与mid = (low + heigh)/2进行比较, 失败则根据mid的值调整low或heigh

失败条件: low > high

失败结点: 只存在于逻辑上的结点, 代表查找失败, 若树有n个结点, 则有n+1个失败结点

n个元素时树高 $h = \lceil log_2(n+1) \rceil$ (不包含失败结点)，折半查找时间复杂度 $O(log_2 n)$

$$ASL = log_2(n+1) - 1$$

分块查找

又称索引顺序查找

将查找表分为若干子块，块内无序，块间有序

第一块中的最大关键字小于第二块的所有关键字, 索引块表中记录着每块的最大关键字

分块查找：

1. 折半查找块
2. 顺序查找元素

$s = \sqrt{n}$ (s为块大小)时，平均查找长度取最小值 $ASL = \lceil log_2(b+1) \rceil + \frac{s+1}{2}$

树型查找

二叉排序树（BST）

左子树结点值 < 根节点值 < 右子树结点值

不允许两个结点的关键字相等

中序遍历能够得到一个递增的有序序列

二叉排序树的查找

O时间复杂度：O(h) （h：树高）

失败时的平均查找长度计算时最好补上失败节点, 不容易错

二叉排序树的删除

实现过程：

- 若被删除结点是叶节点，直接删除
- 若被删除结点只有一棵子树，则结点的子树代替结点原来的位置
- 若被删除的结点有两颗字数，则令结点的值与其直接后继（或直接前驱）结点进行交换，之后在根据直接后继（或直接前驱）请请跨国进行删除

平衡二叉树

平局查找长度 $O(log_2n)$

考点：高为h的平衡二叉树最少有几个结点?

递归求解: $$h(n)=h(n-1)+h(n-2)+1$$ $$h(0)=0\ h(1)=1\ h(2)=2$$

空树或左右子树高度之差的绝对值不超过1（–1、0、1）的树

左右子树的高度之差称为平衡因子

平衡二叉树的插入

核心问题：插入结点后如何保持平衡

每次插入后先查找距离插入点最近的平衡因子绝对值超过1的结点A，A有四种情况：

1. LL型

   A的左孩子的左子树的插入行为导致A不平衡

   方案：将A的左孩子右上旋

2. RR型

   A的右孩子的右子树的插入行为导致A不平衡

   方案：将A的右孩子左上旋

3. LR型

   A的左孩子的右子树的插入行为导致A不平衡

   方案：将A的左孩子的右孩子先左上旋，再右上旋

4. RL型

   A的右孩子的左子树的插入行为导致A不平衡

   方案：将A的右孩子的左孩子先右上旋，再左上旋

> 右旋时如果旋转结点的度为2，则可将其右子树嫁接到其父节点的左子树上
>
> 左旋时类似

平衡二叉树的删除

与平衡二叉树的插入类似，也是四种情况

步骤：

1. 先根据二叉树的删除方案删除指定结点
2. 从删除结点开始向上回溯，找到第一个不平衡的结点A
3. 根据A的情况进行处理：
    a. LL型：删除后，A结点的左子树较高（和A的右子树相比），且左子树的左子树也较高（和左子树的右子树相比）。右上旋（对左子树的左子树）
    b. RR型：删除后，A结点的右子树较高，且右子树的右子树也较高。左上旋
    c. LR型：删除后，A结点的左子树较高，且左子树的右子树较高。先左上旋，再右上旋
    d. RL型：删除后，A结点的右子树较高，且右子树的左子树较高。先右上旋，再左上旋

红黑树*

为了保持AVL树（平衡二叉树）的平衡性，插入和删除后都会频繁地调整树的结构，代价较大。因此提出红黑树，放宽了要求

红黑树查找操作时间复杂度同AVL树：$O(log_2 n)$

红黑树是满足以下特征的二叉排序树：

- 左 < 中 < 右
- 每个结点或是红色，或是黑色
- 根叶黑
- 不红红: 不存在两个相邻的红结点（即红结点的父节点和孩子结点均为黑色）
- 黑路同: 每个结点，从该结点到任一叶节点的简单路径上的黑结点个数相同

衍生结论：

- 从根节点到叶节点的最长路径不大于最短路径的二倍
- 有n个内部结点的红黑树的高度 $h \leq 2log_2(n+1)$
- 新插入的结点初始为红色
- 若根节点的黑高为h, 则 $2^h - 1 < \text{关键字个数} < 2^{2h} - 1$

红黑树的插入*

步骤：

1. 先查找，确定插入位置
2. 新节点如果是根节点则染黑
3. 新节点是非根结点则染红
    a. 若插入的新节点不破坏红黑树的性质，则结束
    b. 若插入的新节点破坏了红黑树的性质，则需调整
        i. 如果叔结点是黑色（父节点的兄弟结点）
            1. LL型：新节点是爷结点的左孩子的左孩子。右上旋（对父结点），父和爷结点颜色取反
            2. RR型：新结点是爷结点的右孩子的右孩子。左上旋（对父结点），父和爷结点颜色取反
            3. LR型：新节点是爷结点的左孩子的右孩子。左上旋，再右上旋（对新结点），新节点、父和爷结点颜色取反
            4. RL型：新节点是爷结点的右孩子的左孩子。右上旋，再左上旋（对新结点），新节点、父和爷结点颜色取反
        ii. 叔结点是红色：叔父爷结点颜色取反，后将爷结点当作新节点，从逻辑2开始重新执行

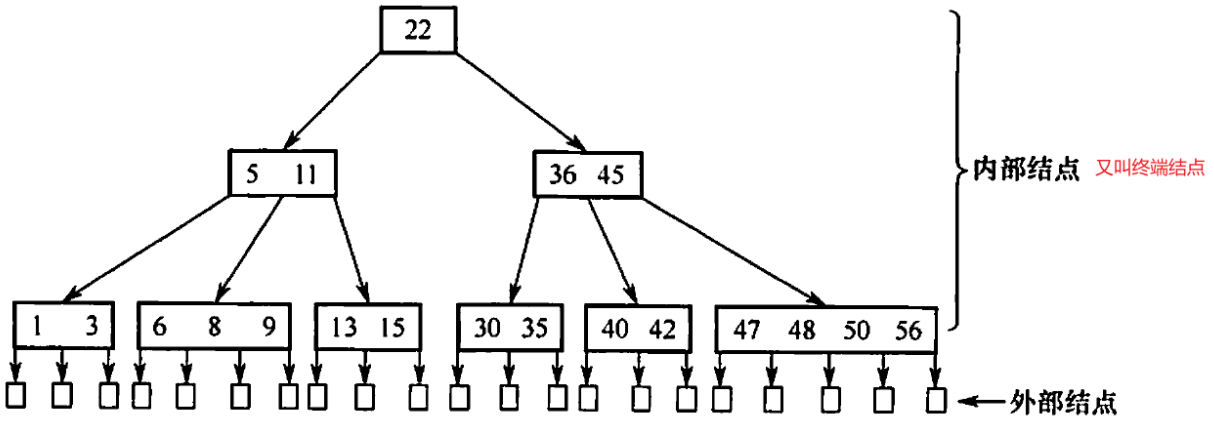
红黑树的删除

要点：

- 删除操作的时间复杂度为： $O(log_2 n)$
- 删除结点时的处理方式与”二叉排序树的删除“一样
- 再删除结点后，可能会破坏红黑树的特性，此时需要调整结点的颜色和位置

B树和B+树

B树



又称多路平衡二查找树

B树的阶：B树中所有结点的孩子个数的最大值

一颗m阶B树或为空树，或为满足以下特点的m叉树：

- 树中每个结点最多有m颗子树，最多含有m–1个关键字
- 若根节点不是终端结点，则至少有两颗子树
- 除根节点外的所有非叶节点至少要有 $\lceil \frac{m}{2} \rceil$ 颗子树，至少含有 $\lceil \frac{m}{2} \rceil - 1$ 个关键字
- 树要绝对平衡
- 所有叶节点都在同一层上，且不携带任何信息（只在逻辑上存在，物理上是NULL的指针），可视为失败结点
- n个关键字的B树有n+1个叶子结点

B树的非叶结点结构：

n	P_0	K_1	P_1	K_2	P_2	$\cdots$	K_n	P_n

- n为当前结点中关键字个数
- P为指向子树的指针
- K为关键字

B树的高度（磁盘存取次数）

B树的大部分操作所需存取磁盘的次数与B树的高度成正比

$$log_m(n+1) \leq h \leq log_{\lceil \frac{m}{2} \rceil}(\frac{n+1}{2}) + 1$$

让每个结点中的关键字个数达到最少，高度达到最大

B树的查找

分两步：

- 在B树中查找结点
- 在结点中查找关键字（顺序查找或折半查找）

B树的插入

步骤：

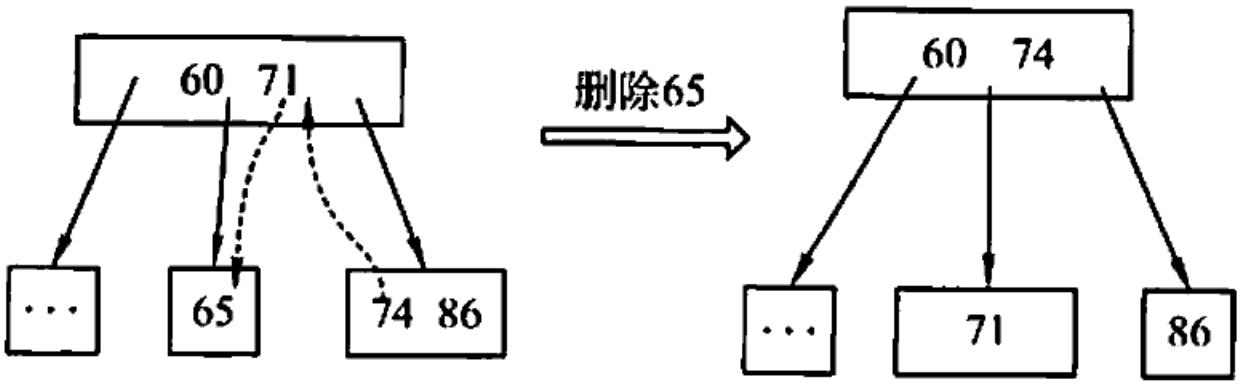
1. 定位。根据B树的查找算法，找出插入该关键字的最低层的某个非叶子结点
2. 插入。将新元素插入到定位到的结点中；如果插入后叶子中的结点个数大于m–1，则需要进行分裂
3. 分裂：申请一个新节点，从插入结点的中间位置（$\lceil \frac{m}{2} \rceil$）将其分为两部分；左部分包含的关键字放在原结点中，有部分包含的关键字放到新结点中，中间位置（$\lceil \frac{m}{2} \rceil$）的结点插入到插入节点的父结点中。若此操作导致父节点关键字个数也超过了m–1，则继续对父节点进行分裂操作。（如果分裂的结点为根结点，则需要再申请一个新节点当作新的根节点，这样也会导致B树的高度+1）

B树的删除

核心问题：要使删除后的结点中的关键字个数 $\geq \lceil \frac{m}{2} \rceil - 1$

删除的三种情况：

- 直接删除关键字：若被删除的关键字所在结点的关键字个数 $\geq \lceil \frac{m}{2} \rceil$，则说明删除后仍满足B树要求
- 兄弟够借：若被删除的关键字所在的结点在删除前的关键字个数为 $\lceil \frac{m}{2} \rceil - 1$，则说明删除后不满足要求。此时若结点的左兄弟（或右兄弟）的关键字个数 $\geq \lceil \frac{m}{2} \rceil - 1$，则可从该兄弟出获取一个关键字（父子换位法，参考下图）



- 兄弟不够借：此时左右兄弟中关键字个数均为 $\lceil \frac{m}{2} \rceil - 1$ ，此时将关键字删除后，需要把当前结点和左兄弟（或右兄弟）以及父节点中两结点之间的关键字合并

> 如果父节点不为根节点，且合并后父结点中的关键字个数不符合要求，则需要将父节点当作删除操作执行的结点再次进行合并。
>
> 如果父节点是根节点，且合并操作后根节点中的关键字个数为0，则直接删除根节点，用合并得到的新节点当作根节点

散列表

散列函数：把关键字映射成该关键字对应的地址的函数

散列表：根据关键字而直接访问的数据结构。建立了关键字和存储地址之间的直接映射关系

理想情况下查找的时间复杂度为O(1)，与表中元素个数无关

冲突：把多个不同的关键字映射到同一个地址，这些不同的关键字被称为同义词

散列表的构造方式

要点：

- 散列函数计算出来的地址应等概率、均匀地分布在整个地址空间中，从而减少冲突
- 散列函数定义域必须包含全部的关键字
- 散列函数应尽量简单

直接定址法

直接取关键字中的某个线性函数值作为散列地址

$H(key) = key$或$H(key) = a * key + b$

适用于关键字分布基本连续的情况

除留余数法

最简单、最常用的方法

$$H(key) = key\%p$$

关键：p的选取

数字分析法

分析关键字集合，根据其特点设计散列函数（比如只取中间几位作为关键字来映射）

适用于已知关键字集合的情况，更换关键字集合后就需要重新设计

平方取中法

取关键字的平方值的中间几位作为散列地址

处理冲突的方法

开放定址法

开放：新表项的空间即对同义词开放，又对非同义词开放

$$H_i = H(H(key) + d_i)\%m$$

- m为表长
- d_i 为增量数列

根据 d_i 的取法,又分四种方法:

- 线性探测法: $d_i = 0, 1, 2, ..., m-1$

  冲突发生时,顺序查看下一个单元,直到找出一个空闲元素或查遍全表

  会造成"聚集"现象,降低查效率

- 平方探测法: $d_i = 0^2, 1^2, -1^2, ..., k^2, -k^2 \quad (k \leq \frac{m}{2})$

  要求m必须是一个可以表示成4k+3的素数,又称二次探测法

  可避免"堆积"

  不能探测到散列表上的所有单元,但至少能探测到一半单元

- 双散列法: $d_i = Hash_2(key)$

  需要两个散列函数,当第一个散列函数H(key)得到的地址冲突时,利用第二个散列函数

  $H_i = (H(key) + i * Hash_2(key))\%m \quad (i为冲突次数)$

- 伪随机数法: $d_i =$ 伪随机数序列

拉链法

把所有同义词存储在一个线性链表中,这个链表由其散列地址唯一标识

散列查找及性能分析

散列表的查找过程同构造散列表的过程一致,出现冲突时通过散列函数查找"下一个地址"

散列表的查找效率取决与三个因素:

- 散列函数
- 处理冲突的方法
- 装填因子: $\dfrac{表中记录数n}{散列表长度m}$

> 散列表的平均查找长度依赖于装填因子,而不直接依赖于n或m
>
> 直观地看,装填因子越大,代表散列表越慢,发生冲突的可能性就越大

第八章 排序

重新排列表中的元素,使表中的元素满足按关键字有序

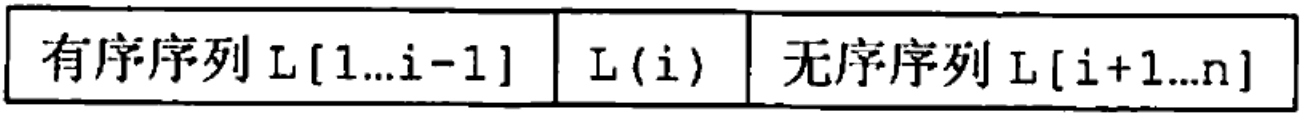
排序算法的稳定性: 当出现待排序的表中有两个元素的关键字相同的情况时, 若排序前后两元素的前后顺序一致则称稳定, 反之则称不稳定

补充：对任意n个关键字的基于比较的排序，比较次数至少为 $\lceil log_2(n!) \rceil$

插入排序

每次将一个待排序记录按其关键字大小插入前面已经排序好的子序列

直接插入排序



每次都是从后往前比较

主要步骤：

1. 找出应插入的位置
2. 移动数组元素

空间效率：O(1)

时间效率：

- 最好：O(n)，数组本身有序
- 最差：$O(n^2)$ 逆序
- 平局：$O(n^2)$

由于每次插入元素都是从后往前比较，所以是稳定的

实用性适用于顺序和链式

有序子序列在排序完成前可能都是局部有序的（不在最终位置上）

> 部分排序算法仅支持顺序存储

折半插入排序

查找有序子表时用折半查找的方法

时间复杂度仍为：$O(n^2)$

稳定

为了保证“稳定性”，在查找插入位置时，如果碰到了与待插入元素相等的元素，则应该继续向右搜索插入位置

仅支持顺序表

希尔排序

算法过程：

1. 去一个小于n的步长d，把表中的数据分成d组，所有距离为d的元素放在一组，组内进行插入排序

2. 不断取更小的步长，重复执行1，直到步长为1
3. 对整个表进行插入排序（由于此时表中数据已有较好的有序性，所以可以很快得出结果）

空间效率：O(1)

时间效率：

- 当n在某个特定范围时约为 $O(n^{1.3})$
- 最坏情况下 $O(n^2)$

不稳定

仅适用于顺序存储

交换排序

根据序列中两关键字的比较结果交换两元素的位置

冒泡排序

一次冒泡：从后往前（或从前往后）比较相邻两元素，若两元素为逆序，则交换

每次冒泡都能将一个元素放到最终的位置上，下次冒泡时就不需要对在最终位置上的元素进行比较了

空间效率：O(1)

时间效率：

- 最坏： $O(n^2)$
- 平均： $O(n^2)$

稳定

适用于顺序存储和链式存储

每次冒泡产生的子序列全局有序

快速排序

基本思想：分治法。

算法步骤：

1. 首先对待排序序列进行一趟快速排序；

2. 一趟排序下来之后，基准元素的左边都是比它小的元素，右边都是比它大的元素；
3. 再对基准元素左边的序列进行快速排序，对右边也进行快速排序；
4. 重复步骤2、3，直到序列排序完成。

一趟快排：

1. 附设两个指针 left 和 right，它们初始分别指向待排序序列的左端和右端；此外还要附设一个基准元素 pivot（一般选取第一个，本例中初始 pivot 的值为 20）
2. 首先从 right 所指的位置从右向左搜索找到第一个小于 pivot 的元素，然后将其记录在基准元素所在的位置。
3. 接着从 left 所指的位置从左向右搜索找到第一个大于 pivot 的元素，然后将其记录在 right 所指向的位置。
4. 然后再从 right 所指向的位置继续从右向左搜索找到第一个小于 pivot 的元素，然后将其记录在 left 所指向的位置。
5. 接着，left 继续从左向右搜索第一个大于 pivot 的元素，如果在**搜索过程中出现了 left == right，则说明一趟快速排序结束**。此时将 pivot 记录在 left 和 right 共同指向的位置即可。

空间效率（主要来自于递归工作栈）：

- 平均情况、最好情况 $O(log_2 n)$
- 最差情况 $O(n)$

时间复杂度： $nlog_2 n$

快排是内部排序算法中平均性能最佳的

不稳定

选择排序

每趟从待排序的元素中选取一个最小的元素，作为有序子序列中最大的元素

简单选择排序

第i趟：从后部分的无需序列中选取除最小的元素与整个数组中的第i个元素交换位置

空间效率：O(1)

时间效率： $O(n^2)$

不稳定

堆排序

空间复杂度： $O(1)$

时间复杂度： $O(nlog_2n)$

不稳定

堆

每个结点的值都大于（小于）其左右孩子的值（二叉树）——大根堆（小根堆）

如过用一维数组实现堆：

- 完全二叉树
- 大根堆有： $L(i) \geq L(2i)且L(i) \geq L(2i+1)$
- 小根堆有： $L(i) \leq L(2i)且L(i) \geq L(2i+1)$

堆排序思路

1. 首先将数组中存放的n个元素构建成一个堆，那么堆顶元素就是最大的数组元素
2. 取出堆顶元素，将一个堆底元素放在堆顶
3. 调整堆
4. 重复2，3直到排序完成

初始堆的构建

1. 将数组视为完全二叉树的顺序存储形式
2. 前 $\lfloor \frac{n}{2} \rfloor$ 个结点是非叶子结点，从第 $\lfloor \frac{n}{2} \rfloor$ 结点开始，从后向前逐个结点进行调整（比较结点与结点的孩子，将三者中的最大者放到结点中）

时间复杂度： $O(n)$

取出一个元素后调整堆(堆的删除)

1. 处理根节点
2. 若根节点小于其左右孩子的值，则将根节点与左右孩子中较大的结点进行交换
3. 以发生交换的子树为一个堆，对这个堆再进行调整

时间复杂度: $O(log_2n)$

堆的插入

1. 将新元素放到堆底（数组尾部）
2. 比较新元素与其父节点，若大于父元素，则交换；若小于父元素，则结束
3. 若发生交换，则重新再新的位置上比较新元素与其新的父元素，重复2

时间复杂度： $O(log_2 n)$

归并排序和基数排序

归并排序

"归并"：将两个或两个以上的有序表组合成一个新的有序表

初始时，将含有n个元素的待排序表视为n个有序子序列

递归2路归并算法逻辑：

1. 将还有n个元素的待排序表分成各含n/2个元素的子表，分别对两个子表进行归并得到两个有序子表
2. 合并两个有序子表

空间复杂度： $O(n)$ ——合并操作中的辅助空间

时间复杂度（2路归并）：

- 每次归并：O(n)，共需 $\lceil log_2 n \rceil$ 趟
- $O(nlog_2 n)$

稳定

基数排序

算法思路：

1. 在关键字前补0，使每个关键字都同位
2. 将关键字根据其个位的数值进行排序，相同的关键字则按顺序存放在其个位对应的链表中
3. 将排序结果写回数组中
4. 从后往前，根据个位、十位、百位...的顺序重复2和3

空间效率： $O(r)$ ——r个队列，对于十进制r=10，这个队列会重复使用

时间效率： $O(d(n+r))$ ——关键字最多d位，时间复杂度与序列初始状态无关

稳定（稳定是基数排序算法的基础）

内部排序算法的比较

算法种类	时间复杂度			空间复杂度	稳定
	最好情况	平均情况	最坏情况		
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
冒泡	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
简单选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
希尔				$O(1)$	否
快排	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(log_2n)$	否
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	否
2路归并	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	是
基数排序	$O(d(n+r))$			$O(r)$	是