私塾在线《Java数据结构和算法——精讲版》

课程前言

- 什么是数据结构和算法

1：数据结构

对内存/磁盘上数据的安排，也就是数据的组织形式。基本的数据结构包括数组、链表、栈、队列、树、哈希表、图、堆等。

2：算法：是对数据结构中的数据进行处理的方式或过程，广义的说，就是解决问题的方法

- 数据结构和算法的关系

数据结构为算法服务，很多算法依赖于特定的数据结构，但不是全部算法，算法可以和数据结构没有关系

- 学习它们有什么好处

正确的选择数据结构和算法，可以使程序更健壮，运行效率大大提高

因此，数据结构和算法是每个软件开发人员必备的基本功，也是最重要的基本功之一

- 课程特点

1：主要放在数据结构和相应的基本的算法上，并不涉及更多的算法

2：不空洞的讲解概念和理论，简明扼要，重点放在代码的实现和示例上

3：不是专门的算法和算法分析课程，所以不涉及算法的推导以及详细的分析过程

课程整体概览-1

- 数组

  有序和无序数组的操作、二分法查找、存放对象、大O表示法

- 栈

  线形表、栈、栈的操作、栈的实例、后缀表达式(包括转换和计算)

- 队列

  队列、队列的实现、循环队列、双端队列、优先级队列

- 链表

  链表、单链表、双端链表、用链表实现栈和队列、有序链表、双向链表

- 递归算法

  递归、阶乘、分治算法、斐波那契数列、汉诺塔、背包问题、归并排序

- 排序算法

  冒泡、选择、插入法、希尔、快速、基数、对象排序

- 二叉树

  二叉树概念和性质、二叉树的实现、哈夫曼编码、哈夫曼树、哈夫曼算法、使用哈夫曼算法来实现压缩和解压的功能

课程整体概览-2

- 红黑树

  概念和特征、红黑树的规则和修正、红黑树的旋转、**红黑树的实现**

- 2-3-4树

  概念和规则、**2-3-4树的实现**、2-3-4树和红黑树的关系和转换规则

- B树

  概念和特性、B树的高度、**B树的实现**、B树的变形

- 堆

  概念和特点、**堆的实现**、**堆排序**

- 哈希表

  概念和优缺点、**Hash函数的构建**、**冲突解决（开放地址法和链地址法）**、**Hash化字符串**

- 图

  概念和基本术语、**深度和广度搜索**、**最小生成树**、**有向图的拓扑**、**有向图的连通**、 **Warshall算法**、带权图的最小生成树、**普里姆算法**、最短路径问题、**迪杰斯特拉算法**、 **弗洛伊德算法**

本部分课程概览

- 本部分课程，主要学习跟数组相关的内容，包括：

1：无序数组的操作
2：有序数组的操作
3：二分法查找
4：存放对象
5：大0表示法

数组-1

- 数组是什么

  数组是由相同类型的若干项数据组成的一个数据集合。也就是说数组是用来集合相同类型的对象并通过一个名称来引用这个集合，数组是引用类型。

- 对数组的插入、删除和查找操作

  说明：只研究最基本的算法思路，不引入其他复杂性，比如接口、参数校验、多线程环境运行等

- 无序数组

1：使用索引来操作
2：不使用索引，不存放重复值
3：不使用索引，存放重复值

数组-2

- 比较无序数组的重复存放值和不重复存放值

	不允许重复	允许重复
插入	无比较，一次移动	无比较，一次移动
删除	N/2次比较，N/2次移动	N次比较，多余N/2次移动
查找	N/2次比较	N次比较

- 有序数组

1：不使用索引，不存放重复值
2：不使用索引，存放重复值

- 二分法查找

思路：每次查找，将数据分为两个部分，逐次缩小查找范围，直到查找到数据。

- 比较有序和无序数组

1：有序数组的查找快、插入慢
2：无序数组的插入快、查找慢
3：删除都慢，都需要移动删除项后面的数据

数组-3

- 存放对象

- 大O表示法

  是一种粗略的度量算法运行时间的方法。

  大写字母O的含义是“大约是”的意思。它通常省去常数量，只描述跟算法运行时间主要相关的量值，比如：

1：无序数组的插入：常数

无序数组的插入跟数组大小无关，总是插入到最后一个，因此可以说插入时间为一个常数，可记为：T=K

2：线形查找：与N成正比

线形查找跟数组大小有关，假设有N项数据，那么找到某项数据的比较次数平均为：数据项总数的一半，可记为：T=K*N/2

这说明：平均线形时间和数组大小成正比

3：二分法查找：与log(N)成正比

二分法查找的次数，大约为log以2为底(N)次，可记为：T=K* log以2为底(N)

4：省去常数，只是估算量级，最终表示出来如下所示：

数组-4

算法	大O法表示的运行时间
无序数组插入	O(1)
有序数组插入	O(N)
无序数组删除	O(N)
有序数组删除	O(N)
线形查找	O(N)
二分法查找	O(logN)

- 说明

  大O表示法不是给出精确的运行时间，而是一个大约的量级。

  可以看出O(1)最优，O(logN)较优，O(N)一般，O(N的二次方)就比较差了。

本部分课程概览

- 本部分课程，主要学习跟简单排序相关的内容，包括：

1：冒泡排序
2：选择排序
3：插入法排序
4：对象排序

简单排序-1

- 冒泡排序

1：基本思路：对未排序的各元素从头到尾依次比较相邻的两个元素是否逆序（与欲排顺序相反），若逆序就交换这两元素，经过第一轮比较排序后便可把最大（或最小）的元素排好，然后再用同样的方法把剩下的元素逐个进行比较，就得到所要的顺序。

2：效率：比较和交换次数都为O(N的平方)

- 选择排序

1：基本思路：从所有元素中选择一个最小元素a[i]放在a[0]（即让最小元素a[i]与a[0]交换），作为第一轮；第二轮是从a[1]开始到最后的各个元素中选择一个最小元素，放在a[1]中；……依次类推。n个数要进行（n－1）轮

2：效率：交换次数减少到O(N)，但是比较次数仍为O(N的平方)

简单排序-2

- 插入法排序

1：基本思路：每拿到一个元素，都要将这个元素与所有它之前的元素遍历比较一遍，让符合排序顺序的元素挨个移动到当前范围内它最应该出现的位置。

2：效率：比较和交换次数都为O(N的平方)，大致为N*(N-1)/4，所以这个算法比冒泡大致快一倍，比选择排序略快，尤其是部分数据已经局部有序的情况下，这个算法效率会更高

- 对象排序

其实还是比较对象里面的某个或某些属性值来进行排序

本部分课程概览

- 本部分课程，主要学习跟栈相关的内容，包括：

1：线性表的概念
2：栈的概念
3：栈的基本操作
4：栈的基本实现
5：栈的应用实例
6：后缀表达式，包括：如何把中缀表达式转换成后缀表达式，以及如何计算后缀表达式

栈-1

- 什么是线性表

线性表也被称为顺序表，是一个线性序列结构，它是一个含有n≥0个结点的有限序列，对于其中的结点，有且仅有一个开始结点没有前驱但有一个后继结点，有且仅有一个终端结点没有后继但有一个前驱结点，其它的结点都有且仅有一个前驱和一个后继结点

- 线性表和数组的关系

1：是两种不同的数据结构，数组有纬度的概念，线性表没有；而线性表有前驱节点和后继节点的概念，线性表的数据是相互有关联的，而数组没有这些。

2：线性表可以使用数组，通常是一纬数组来作为其数据的存储结构

- 什么是栈(Stack)

栈是一种特殊的线性表，限定只能在表的一端进行插入和删除操作，俗称“后进先出”（FILO）。

操作数据的这端就是表头，称为栈顶；相应地，表尾称为栈底。不含任何元素的栈称为空栈。

栈-2

- 栈的基本操作

1：push：压栈或入栈操作
2：pop：弹栈或出栈操作
3：peek：查看栈顶数据，而不弹出数据，也就是不做出栈操作

- 栈的基本实现

- 栈的应用实例

1：字符串倒序
2：括号（小、中、大）匹配
3：计算算术表达式

栈-3

- 什么是后缀表达式

后缀表达式，也称波兰逆序表达式，由波兰数学家卢卡西维奇发明的一种表示表达式的方法。这种表示方式把运算符写在运算对象的后面，例如，把a+b写成ab+，所以也称为后缀表达式。

这种表示法的优点是根据运算对象和运算符的出现次序进行计算，不需要使用括号，也便于用程序来实现求值。

- 如何把中缀表达式转换成后缀表达式

说明：把中缀表达式转换成后缀表达式不用做算术运算，只是把操作符和操作数重新按照后缀表达式的方式进行排列而已。

1：从左到右顺序读取表达式中的字符
2：是操作数，复制到后缀表达式字符串
3：是左括号，把字符压入栈中
4：是右括号，从栈中弹出符号到后缀表达式，直到遇到左括号，然后把左括号弹出。
5：是操作符，如果此时栈顶操作符优先级大于或等于此操作符，弹出栈顶操作符到后缀表达式，直到发现优先级更低的元素位置，把操作符压入。
6：读到输入的末尾，将栈元素弹出直到该栈变成空栈，将符号写到后缀表达式中

栈-4

- 计算后缀表达式求值

1：从左到右顺序读取表达式中的字符
2：是操作数，就压入栈中
3：是操作符，就从栈中取出两个数据进行计算，然后把结果压入栈中
4：直到表达式计算结束

- 栈的效率

栈的出栈和入栈的时间复杂度都为常数，也就是O(1)，不涉及复制和移动操作

本部分课程概览

- 本部分课程，主要学习跟队列相关的内容，包括：

1：什么是队列
2：队列的基本操作
3：循环队列
4：队列的基本实现
5：双端队列
6：优先级队列及其实现

队列-1

- 什么是队列（Queue）

  队列是一种特殊的线性表，限定只能在表的一端进行插入（队尾），而在另一端进行删除操作（队头），特点是“先进先出”（FIFO）。

- 队列的基本操作

1：insert：在队尾插入数据

2：remove：从队头移走数据

3：peek：查看队头的数据

- 循环队列

  为了避免队列不满，却不能插入新数据项的问题，可以让队头队尾的指针绕回到数组开始的位置，这就是循环队列，也称为缓冲环。

- 队列的基本实现

- 队列的效率

  插入和移出数据项的时间复杂度均为O(1)

队列-2

- 双端队列

  所谓双端队列，就是一个两端都可以进行数据插入和移出操作的队列。它综合提供了栈和队列的功能，并不是很常用，了解即可。

- 优先级队列

  优先级队列就是数据项按照关键字排好顺序的队列。

- 优先级队列的实现

- 优先级队列的效率

  这里演示的优先级队列，插入需要O(N)，而删除需要O(1)，后面学习堆的时候，可以来改进优先级队列的插入速度

本部分课程概览

- 本部分课程，主要学习跟链表相关的内容，包括：

1：什么是链表
2：链表的基本操作
3：单链表的实现
4：双端链表及其实现
5：使用链表来实现栈和队列
6：有序链表及其实现
7：使用有序链表来实现插入排序
8：双向链表及其实现

链表-1

- 什么是链表（Linked List）

链表也是一种特殊的线性表，它由一系列结点组成，结点的顺序是通过结点元素中的指针链接次序来确定的。链表中的结点包含两个部分，一个是它自身需存放的数据，另一个是指向下一个结点的引用。

- 链表和数组

1：都可作为数据的存储结构
2：数组：固定长度、顺序存放
3：链表：没有容量限制，非连续和非顺序的存储数据
4：从效率上说，链表基本上完胜数组，这个在后面讲述每种链表的时候再分析
5：基本上，能用数组的地方，都可以使用链表来代替数组。链表的缺点是引入了复杂度。

- 链表的基本操作

1：向链表中插入数据
2：从链表中移走数据
3：查看链表中所有的数据
4：查找指定链结点
5：删除指定链结点

链表-2

- 单链表的基本实现

- 单链表的效率
  在表头插入和删除非常快，基本就是修改一下引用值，时间大约为常量，即为O(1)。
  查找、删除等功能，大约需要O(N)次比较，跟数组差不多，但仍然比数组快，因为它不需要移动或复制数据

- 双端链表
  所谓双端链表是，不仅仅记录着开始结点，同时也记录着结束结点。
  注意，不要把双端和双向搞混了。

- 双端链表的实现

- 使用链表来实现栈和队列

链表-3

- 有序链表
  就是链表中的数据是排好顺序的。

- 有序链表的实现

- 有序链表的效率
  插入和删除，大约需要O(N)次比较。

- 使用有序链表来实现插入排序
  思路：把数据依次插入到有序链表，然后再依次读取出来，这就已经排好序了。
  效率：比数组插入法更高，因为这种方式下，数据的复制次数要少一些，每个节点只需要2*N次复制，而数组中需要N的平方次复制。

链表-4

- 双向链表

  就是链表中的每个节点，同时记录着前驱结点和后继节点，这样就既能向后遍历数据，也可以向前遍历数据了。

- 双向链表的实现

本部分课程概览

- 本部分课程，主要学习跟递归算法相关的内容，包括：

1：什么是递归
2：阶乘
3：理解递归：调用顺序、和循环的关系
4：理解分治算法
5：斐波那契数列
6：汉诺塔（河内塔）问题
7：背包问题
8：归并排序

递归-1

- 什么是递归

  递归就是函数（方法）不断调用自身，直至求出结果的算法。其思路是把一个大问题转化为规模缩小了的子问题，通过解决小问题来解决最终的大问题。

- 递归示例：阶乘
- 理解

1：递归的运行顺序

2：递归和循环，把前面用循环实现的二分法查找，用递归来实现

3：分治算法：基本思想是将一个大的问题分解为N个较小的子问题，这些子问题相互独立且与原问题性质相同。求出子问题的解，就可得到原问题的解。

- 递归示例：斐波那契数列

  斐波那契数列，又称黄金分割数列，形如：0，1，1，2，3，5，8，13.....

1：第0项是0，第1项是第一个1

2：从第二项开始，每一项都等于前两项之和

递归-2

- 汉诺塔（河内塔）问题

在ABC三根柱子上，有n个圆形盘以从下到上、从大到小的次序叠置在A塔上。现要将A塔上的所有圆形盘，借助B搭，全部移动到C塔上。且仍按照原来的次序叠置。移动的规则如下：

1：这些圆形盘只能在3个塔问进行移动

2：一次只能移动一个盘子

3：任何时候都不允许将较大的盘子压在比它小的盘子的上面

递归-3

- 背包问题

  背包问题是一种组合优化的问题，一种简化形式如下：

  给定一组物品，重量各不相同，如何从中选择物品放入背包中，以使背包重量达到指定的重量。

- 归并排序

  思路：采用分治的思想，把数据序列分成两个子序列，排序每一半，然后再把排好序的两个子序列合并成为一个有序的序列。

- 归并排序的效率

  归并排序的时间是O(N*logN)，主要是复制和比较花费时间

本部分课程概览

- 本部分课程，主要学习跟高级排序相关的内容，包括：

1：希尔排序

2：快速排序

3：基数排序

高级排序-1

- 希尔排序

希尔排序是对插入排序的一种改进，减少了交换和移动次数，效率更高。

1：基本思路：先将数据按照一定的间隔进行分组，然后对每组数据进行排序，然后缩小分组间隔，再次对各组的数据进行排序，直到最后用分隔为1来对数据进行分组，并再次排序，最后得到排好序的数据。

2：希尔排序的重点，就在于选取合理的分组间隔，原始的希尔排序算法是N/2，现在更常用的是：使用h=h*3+1来产生增量序列，h=(h-1)/3来获取当前增量。

3：不管采用何种算法来产生分组增量，最后一个增量必须是1。

- 希尔排序的效率

希尔排序的效率，目前是基于大量的实验结果，大致是O(N的二分之三次方) 到O(N的六分之七次方)

高级排序-2

- 快速排序

1：基本思路：先根据分区枢纽值将数据序列分成两个子序列，使左边序列的所有值都小于枢纽值，右边序列的所有值都大于枢纽值，然后采用同样的方法来对每个子序列进行快速排序，最后得到排好序的数据。

2：快速排序的一个重点，就在于选取合理的分区枢纽值，也就是通过该值来把数据序列的数据分成两个序列。

3：目前常用的方式是“三数据项取中”的方式：就是对数据序列的第一个、中间一个、最后一个位置的数据项，找到这三个项的中间那个值。

4：快速排序的另外一个重点，就在于把数据分成两个序列，且满足条件。

- 快速排序的效率

快速排序的时间复杂度为O(N*logN)

高级排序-3

- 基数排序

  基数排序又称桶子排序(bucket sort)。

1：基本思路：

（1）根据数据项的个位值，把数据分成10组；

（2）对这10组数据项重新进行排列：0结尾的排前面，1结尾的排后面，以此类推，直到9结尾的派最后面。这就是第一趟排序

（3）再把所有的数据按照十位进行分组，这次分组不能改变前面已经排好的顺序

（4）再把这10组数据合并，十位为0的排最前面，十位为1的排后面，以此类推，直到十位为9的派最后面。这就是又一趟排序。

（5）对剩余位重复这个过程，如果某些数据项位数少于其他项，就认为其高位为0

- 基数排序的效率

  数据比较少的时候是O(N)，数据多了，倒退为O(N*logN)

本部分课程概览

- 本部分课程，主要学习跟二叉树相关的内容，包括：

1：树的基本知识
2：二叉树的概念和理解
3：二叉树的性质
4：二叉搜索树的查找、插入、遍历、查找最大最小值和删除操作的实现
5：哈夫曼编码和哈夫曼树的基本知识
6：哈夫曼算法
7：使用哈夫曼算法来实现压缩和解压的功能

二叉树-1

- 树是什么

  树是由边和节点构成的数据结构。节点通常就是存储数据的实体。

- 树的常见术语

1：根：树的顶端节点。一棵树只有一个根。
2：边：节点到节点的连接
3：路径：沿着边，从一个节点走到另一个节点，所经过的节点顺序称为路径
4：父节点、子节点
5：节点、叶节点
6：度：一个节点包含的子节点数
7：子树
8：层：就是从根开始到这个节点的“代”，层数也称为高度或深度
9：遍历：按照某个特定的顺序访问节点
10：关键字：节点对象域中的某个属性，用来识别节点对象

二叉树-2

- 二叉树是什么

  如果树中的每个节点，最多只能有两个子节点，这样的树称为二叉树。

- 二叉树理解

1：二叉树不是树的特殊情形，其区别为：

（1）树中节点的字节点树没有限制，而二叉树中限制节点数为不超过2个

（2）树的节点没有左右之分，二叉树节点是分左右的

2：二叉树有五种基本形态

（1）空二叉树，连根节点都没有

（2）只有一个根节点的二叉树

（3）只有左树

（4）只有右树

（5）完全二叉树：若设二叉树的高度为h，除第 h 层外，其它各层的结点数都达到最大个数，第h层有叶子结点，并且叶子结点都是从左到右依次排布，这就是完全二叉树。

二叉树-3

3：满二叉树：除叶节点外，每一个节点都有左右子节点，且叶子结点都处在最底层的二叉树

4：满二叉树肯定是完全二叉树，完全二叉树不一定是满二叉树。

5：二叉树常被用作二叉查找树、二叉排序树、二叉堆。

- 二叉树性质

性质1：在二叉树的第 i 层上至多有2^(i-1)个结点

性质2：深度为k的二叉树至多有 2^k - 1个结点

性质3：对任何一颗二叉树T，如果其终端结点数为n0，度为2的结点数为n2，则n0=n2+1

性质4：具有N个结点的完全二叉树的深度为[log2为底N]+1

性质5：如果对一棵有N个结点的完全二叉树的结点按层序编号，即从第1层到第[log2为底N]+1层，每层从左到右，对任一结点i(1<=i<=n)有：

（1）如果i=1，则结点i是二叉树的根；如果i>1，则其父结点是[i/2]

（2）如果2i>n，则结点i无左子节点；否则其左子节点是2i。也就是该节点是叶子节点。

（3）如果2i+1>n，则结点i无右子节点；否则其右子节点是2i+1

（4）若i为奇数且不为1，则Ki的左兄弟的编号是i-1；否则，Ki无左兄弟

（5）若i为偶数且小于n，则Ki的右兄弟的编号是i+1；否则，Ki无右兄弟

二叉树-4

- 二叉搜索树是什么

  如果一棵二叉树，满足：左子节点的值小于节点的值，而右子节点的值大于节点的值，就被称为二叉搜索树。

- 二叉搜索树的查找、插入、遍历、查找最大最小值和删除操作

  提示：删除节点有两个子节点的时候，要用它的中序后继来代替该节点，算法是：找到被删除节点的右子节点，然后查找这个右子节点下的最后一个左子节点，也就是这颗子树的最小值节点，这就是 被删除节点的中序后继节点。

- 二叉搜索树操作的效率

  常见的操作效率是：O(logN)，是以2为底的

- 用数组来表示树

  把树的节点遍上序号，按序存放到数组中。这样一来，查找节点就变成了查找相应的索引了。这种方法不常用，了解即可。

  这种方式效率不高，因为不满的节点，还有删除的节点，在数组中留下了洞，更糟糕的是删除节点时，如果需要移动子树的话，就更浪费时间了。

二叉树-5

- 哈夫曼(Huffman)编码

  霍夫曼编码是一种统计编码，属于无损压缩编码。其基本思路是：对每个字符编码的码长是变化的，对于出现频率高的，编码的长度较短；而对于出现频率低的，编码长度较长。

  编码规则还要求：每个编码都不能是其它字符编码的前缀。

- 哈夫曼树

  哈夫曼树又称最优二叉树，是一种带权路径长度最短的二叉树。所谓树的带权路径长度，就是树中所有的叶结点的权值乘上其到根结点的路径长度（若根结点为0层，叶结点到根结点的路径长度为叶结点的层数）。树的带权路径长度记为WPL= (W1*L1+W2*L2+W3*L3+...+Wn*Ln)，N个权值Wi(i=1,2,...n)构成一棵有N个叶结点的二叉树，相应的叶结点的路径长度为Li(i=1,2,...n)

  要使其WPL值最小，必须使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结点。

二叉树-6

- 哈夫曼树的构造方法——Huffman算法

1：对给定的n个权值{W1,W2,W3,...,Wi,...,Wn}，构成n棵只有根节点的二叉树，令起始权值为Wj，为方便处理，可以按照权值进行升序排列

2：在这些二叉树中，选取两棵根结点权值最小的树作为新构造的二叉树的左右子树，新二叉树的根结点的权值为其左右子树的根结点的权值之和。

3：从森林中删除这两棵树，并把这棵新的二叉树同样以升序排列加入到森林中。

4：重复2和3两步，直到森林中只有一棵二叉树为止

- 哈夫曼压缩的编码部分的实现步骤

1：统计：读入要压缩的源文件，统计字符出现的次数

2：建树：构建哈夫曼树

3：编码：对哈夫曼树的左边记0，右边记1，就可以得到字符的哈夫曼编码。

4：输出：把编码序列输出，这就是压缩后的数据

二叉树-7

- 哈夫曼压缩的解码部分的实现步骤

1：读取码表的信息，构建出码表来
2：读回具体的数据内容
3：把读回的字节还原成对应的整型数据
4：根据码表，把内容组成的哈夫曼编码，依次转换回原始的字符，从而得到原始的内容

本部分课程概览

- 本部分课程，主要学习跟红黑树相关的内容，包括：

1：平衡与非平衡树
2：红黑树是什么以及特征
3：红黑树的规则和修正手段
4：红黑树的旋转
5：红黑树的节点插入算法，并代码示例
6：红黑树的节点删除算法，并代码示例
7：红黑树的效率
8：了解其它的平衡树

红黑树-1

- 树的平衡

树的平衡指的是：树中每个节点的左边后代的数目，应该和其右边后代的数目大致相等。

对于随机数构成的二叉树，一般来说是大致平衡的，但是对于有序的数据，二叉树就严重不平衡了，极端情况下，退化成为链表，其时间复杂度下降到O(N)，而不再是平衡树的O(logN)。

- 红黑树（R-B Tree）是什么

红黑树是一种增加了某些特性的二叉搜索树，它可以保持树的大致平衡。

大致思路是：在插入和删除节点的时候，检查是否破坏了树的一定的特征，如果破坏了，就需要进行纠正，从而保持树的平衡。

- 红黑树的特征

1：节点都有颜色

2：在插入和删除过程中，要遵循保持这些颜色的不同排列的规则

红黑树-2

- 红黑树的规则（也称红黑规则）

1：每个节点不是红色就是黑色

2：根总是黑色

3：如果节点是红色的，那么它的子节点必须是黑色的（反之不必为真）

4：每个空子节点都是黑的

这里的空子节点指的是：对非叶子节点而言，本可能有，但实际没有的那个子节点。比如一个节点只有右子节点，那么它空缺的左子节点就是空子节点。

5：从根到叶节点或“空子节点”的每条路径，必须包含相同数目的黑色节点（这些黑色节点的数目也称黑色高度）

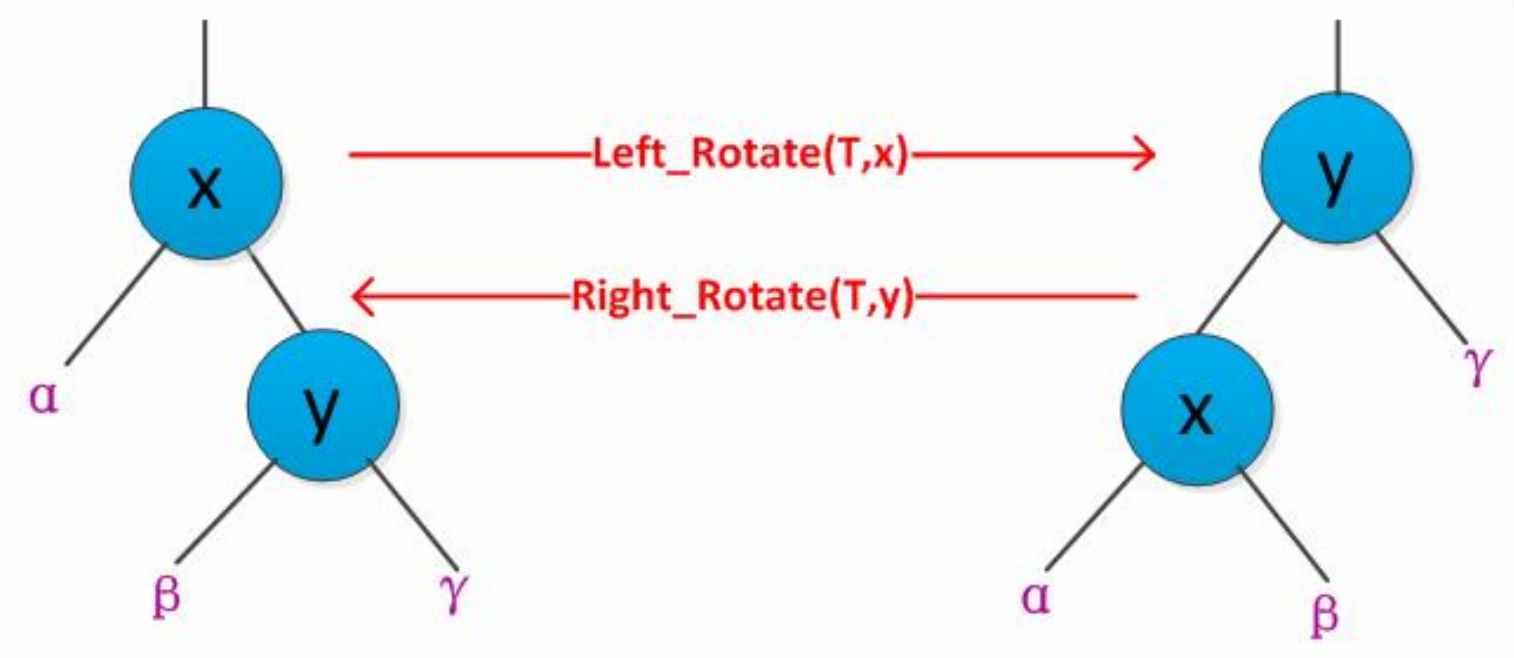
- 红黑树的修正手段

1：改变节点颜色

2：执行旋转操作

红黑树-3

- 红黑树的旋转



红黑树-4

- 红黑树的插入算法

红黑树的插入，前面跟二叉树插入一样，就是从根节点向下查找节点要插入的位置，然后插入节点；插入节点过后，后面添加了这样的操作：检查树是否平衡，如果不平衡，就要做修复，使树重新变得平衡。

插入新的节点，通常会设置这个节点为红色，因为这样违反红黑规则的几率较小，插入节点后，有以下几种情况：

1：如果插入的是根节点，那么违反规则2，就直接把节点修改为黑色

2：如果插入节点的父节点是黑色的，说明符合规则，什么都不做

3：如果插入节点的父节点是红色的，且祖父结点的另一个子节点（叔叔节点）也是红色的，那么：将祖父节点变红，而父和叔节点变黑，然后设置祖父节点为当前节点，然后重新开始判断。

4：如果插入节点的父节点是红色，而叔节点是黑色，且插入节点是其父的左子节点，而父节点是祖父节点的左子节点，那么：把父节点变为黑色，祖父节点变为红色，然后对祖父节点进行右旋，然后重新开始判断。

红黑树-5

5：如果插入节点的父节点是红色，而叔节点是黑色，且插入节点是其父的右子节点，而父节点是祖父节点的右子节点，那么：把父节点变为黑色，祖父节点变为红色，然后对祖父节点进行左旋，然后重新开始判断。

6：如果插入节点的父节点是红色，而叔节点是黑色，且插入节点是其父的右子节点，而父节点是祖父节点的左子节点，那么：把当前节点的父节点做为新的当前节点，对新的当前节点进行左旋，然后重新开始判断。

7：如果插入节点的父节点是红色，而叔节点是黑色，且插入节点是其父的左子节点，而父节点是祖父节点的右子节点，那么：把当前节点的父节点做为新的当前节点，对新的当前节点进行右旋，然后重新开始判断。

- 红黑树的节点删除

在前面学习二叉树的时候，我们知道做节点删除是很复杂的，同样，在红黑树里面做节点删除，也是很复杂的，甚至比二叉树的节点删除更复杂，因为还需要保证删除节点后，进行树的平衡。

因此，在实际开发中，多数情况下不用去做红黑树的节点删除，而是采用其它变通方法：比如仅仅标识这个节点被删除，并不真的删除，这样树就不用动，在进行业务处理的时候，判断一下，跳过这些节点就可以了。

红黑树-6

- 红黑树的删除算法

1：如果删除节点是叶子节点

（1）如果删除节点是红色的，那就直接删除，不做其它操作

（2）如果删除节点是黑色的，那么就创建一个空节点来顶替删除节点，然后按照后面的调整步骤进行调整

2：如果删除节点有一个子节点，把后来顶替被删节点的那个节点成为顶替节点，如果删除节点为黑，而且顶替节点也为黑，那么把顶替节点当作当前节点，然后按照后面的调整步骤进行调整。

3：如果删除节点有两个子节点，那么，找到其中序后继节点，把这两个节点的数据交换一下，不要复制颜色，也不改变其原有的父子等关系，然后重新进行删除。由于其中序后继节点只可能是 叶子节点 或者 只有一个子节点，因此回到前面两种情况。

红黑树-7

- 删除步骤后的调整步骤，有以下几种情况：

1：当前节点是红，那么：直接把当前节点变成黑色，结束

2：当前节点是黑且是根节点，那么：什么都不用做，结束

3：当前节点是黑且兄弟节点为红色，当前节点为父节点的左子节点，那么：把兄弟结点变成父节点的颜色，把父节点变成红色，然后在父节点上做左旋，再重新开始判断。

4：当前节点是黑且兄弟节点为红色，当前节点为父节点的右子节点，那么：把兄弟结点变成父节点的颜色，把父节点变成红色，然后在父节点上做右旋，再重新开始判断。

5：当前节点是黑且父节点和兄弟节点都为黑色，兄弟节点的两个子节点全为黑色，那么：把兄弟节点变红，然后把父节点当成新的当前节点，再重新开始判断

6：当前节点是黑且兄弟节点为黑色，兄弟节点的两个子节点都是黑色，但是父节点是红色，那么：就把兄弟节点变红，父节点变黑，结束

红黑树-8

7：当前节点是黑且兄弟节点为黑色，兄弟节点的左子是红色，右子是黑色，而且当前节点是父节点的左子节点，那么：把兄弟节点变红，兄弟左子节点变黑，然后对兄弟节点进行右旋，再重新开始判断

8：当前节点是黑且兄弟节点为黑色，兄弟节点的左子是黑色，右子是红色，而且当前节点是父节点的右子节点，那么：把兄弟节点变红，兄弟右子节点变黑，然后对兄弟节点左旋，再重新开始判断

9：当前节点是黑且兄弟节点为黑色，兄弟节点的右子是红色，左子的颜色任意，而且当前节点是父节点的左子节点，那么：把兄弟节点变成当前节点父节点的颜色，把当前节点父节点变黑，兄弟节点右子变黑，然后以当前节点的父节点为支点进行左旋，结束。

10：当前节点是黑且兄弟节点为黑色，兄弟节点的左子是红色，右子的颜色任意，而且当前节点是父节点的右子节点，那么：把兄弟节点变成当前节点父节点的颜色，把当前节点父节点变黑，兄弟节点左子变黑，然后以当前节点的父节点为支点进行右旋，结束。

红黑树-9

- 红黑树的效率

红黑树的查找、插入和删除的时间复杂度都是O(logN)，以2为底。跟二叉树是一样的，但实际上，由于红黑树在插入和删除的时候，需要保证树的平衡，所以会比二叉树慢。

另外一个，红黑树的节点需要多一点额外的空间，来存储颜色信息。

- 了解其它平衡树

AVL(发明者为：Adelson-Velskii和Landis)树是最早的一种平衡树，它要求节点左子树和右子树的高度相差不超过1。

当插入和删除节点的时候，都需要重新平衡树，也就是每次操作会扫描两趟树，一次向下查找节点，一次向上平衡树。

AVL树的效率不如红黑树，也不如红黑树常用，因此了解一下即可。

本部分课程概览

- 本部分课程，主要学习跟2-3-4树相关的内容，包括：

1：多叉树

2：2-3-4树是什么

3：2-3-4树的数据组织规则

4：2-3-4树的实现和代码示例

5：2-3-4树和红黑树的关系和转换规则

6：2-3-4树的效率

2-3-4树-1

- 多叉树

如果树中的节点可以包含更多的识别数据项，可以有更多的子节点，那么这样的树就是多叉树。

- 2-3-4树

2-3-4树是一种特殊的多叉树，跟红黑树一样，它也是一种平衡树，效率比红黑树稍差，其名字的含义是：

1：有一个识别数据项的节点可以有两个子节点
2：有两个识别数据项的节点可以有三个子节点
3：有三个识别数据项的节点可以有四个子节点

因此，2-3-4树的节点最多可以有四个子节点，所以也可以称为4叉树。

- 2-3-4树的数据组织规则

按照识别数据项分段存放

2-3-4树-2

- 2-3-4树的搜索算法

跟二叉树搜索差不多，从根开始查找，判断要搜索的数据处于哪个区间，找到相应的字节点，判断是否匹配，否则重复这些动作。

- 2-3-4树的插入算法

插入的时候，会有几种情况：

1：插入到未满的叶节点里面，直接插入数据即可，顶多会引起同一节点内数据项的移动，以保持数据项的顺序

2：插入到未满的子节点里面，子节点的编号就要发生变化，以此来保证树的结构

3：插入到已满的节点中，这就需要做节点分裂，算法如下：

（1）创建一个空节点，放置到分裂节点的右边

（2）最大数据项移动到新节点，中间数据项移动到父节点，最小的数据项不动

（3）重新设置最大和中间数据项的父子节点关系

2-3-4树-3

5：插入到已满的根节点中，这就需要做根节点分裂，算法如下：

（1）创建一个空节点，作为分裂节点的父节点

（2）再创建一个空节点，作为分裂节点的兄弟节点

（3）最大数据项移动到新的兄弟节点中，中间数据项移动到新的父节点，最小的数据项不动

（4）重新设置最大和中间数据项的父子节点关系

- 2-3-4树的节点删除

  这个需要做节点的合并和调整，比较复杂。由于没有太大的必要，因此建议采用最简单的做法：给删除节点打标记，然后在业务处理时跳过即可。

- 2-3-4树和红黑树

  2-3-4树和红黑树是同构的，可转化为红黑树。

2-3-4树-4

- 2-3-4树转换成红黑树的规则

1：把只包含1个识别数据项和2个子节点的节点，转化为红黑树的黑色节点

2：把只包含2个识别数据项和3个子节点的节点，转化成一个子节点和一个父节点，哪个节点做父或子都可以；子节点有两个自己的子节点，这两个子节点要求是原本相邻的子节点，父节点有另一个子节点，子节点涂成红色，父节点涂成黑色

3：把只包含3个识别数据项和4个子节点的节点，转化成一个父节点，就是原本中间的识别数据项，两个子节点和四个孙子节点，原来前面的两个子节点作为最小项的子节点，原来后面的两个子节点作为最大项的子节点，子节点涂成红色，父节点涂成黑色

- 2-3-4树的效率

2-3-4树的层数比红黑树少，但每个节点的数据更多，因此其效率跟红黑树差不多，都是O(logN)。

本部分课程概览

- 本部分课程，主要学习跟B树相关的内容，包括：

1：B树是什么
2：回忆磁盘存取数据的知识
3：B树的特性
4：B树的高度
5：B树的搜索、添加、删除节点的算法，并代码示例
6：了解B+树，B+树和B树的差异，B+树的优势

B树-1

- B树

B树（即是B-tree，B是Balanced，平衡的意思），是一种平衡的多路搜索树，主要用于磁盘等外部存储的一种数据结构，例如用于文件索引。

- 回忆磁盘存取数据的知识

1：磁盘存取数据的基本过程

（1）根据柱面号使磁头移动到所需要的柱面上，这一过程被称为定位或查找。

（2）根据盘面号来确定指定盘面上的磁道

（3）盘面确定以后，盘片开始旋转，将指定块号的磁道段移动至磁头下

2：磁盘读取数据是以块(block)为基本单位的，因此应尽量将相关信息存放在同一盘块，同一磁道中，以便在读/写信息时尽量减少磁头来回移动的次数，避免过多的查找时间。

3：在大量数据存储在外存磁盘中，如何高效地查找磁盘中的数据，就需要一种合理高效的数据结构了

B树-2

- B树的特性

对于一颗m阶（阶指的是子节点的最大值）的B树，有如下特性：

1：根结点要么是叶子，要么至少有两个儿子

2：每个结点最多含有m个孩子（m>=2）

3：除根结点和叶子结点外，其它每个结点至少有[ceil(m / 2)]个孩子

4：所有的叶子结点都出现在同一层上

5：有s个儿子的非叶结点具有 n = s － 1 个关键字，即：s = n + 1

6：每个非叶子节点中包含有n个关键字信息：

(n，C0，K1，C1，K2，C2，……，Kn，Cn)。其中：

（1） Ki (i=1...n)为关键字，且关键字按顺序升序排序K(i-1)< Ki

（2） Ci为指向子树根的接点，且指针C(i-1)指向子树种所有结点的关键字均小于Ki，但都大于K(i-1)

（3）关键字的个数n必须满足： [ceil(m / 2)-1]<= n <= m-1

B树-3

- B树的高度

B树的高度就是B树不包含叶节点的层数。

由于磁盘存取时的I/O次数，与B树的高度成正比，高度越小，那么I/O次数也就越小，因此理解如何计算B树的高度是很有必要的。

设若B树总共包含N个关键字，则此树的叶子节点可以有N+1个，而所有的叶子结点都在第K层，可以得出：

1：因为根至少有两个孩子，因此第2层至少有两个结点

2：除根和叶子外，其它结点至少有ceil(m/2)个孩子

3：因此在第3层至少有2* ceil(m/2)个结点

4：在第4层至少有2*(ceil(m/2)^2)个结点

5：在第K层至少有2*(ceil(m/2)^(K-2))个结点，于是有：N+1≥2*(ceil(m/2)^(K-2))，这就可以算出：K≤ log(ceil(m/2)((N+1)/2)+2

6：由于计算B树高度是不包含叶节点的层数，所以：

B树的高度≤ log(ceil(m/2)((N+1)/2)+1

B树-4

- B树的搜索算法

  跟2-3-4树的搜索一样，从根开始查找，判断要搜索的数据处于哪个区间，找到相应的子节点，判断是否匹配，否则重复这些动作。

- B树的插入算法

  插入的时候，会有几种情况：

1：插入到未满的叶节点里面，直接插入数据即可，顶多会引起同一节点内数据项的移动，以保持数据项的顺序

2：插入到未满的子节点里面，子节点的编号就要发生变化，以此来保证树的结构

3：插入到已满的节点中，这就需要做节点分裂，算法如下：

（1）创建一个空节点，放置到分裂节点的右边

（2）将一半较大的数据项移动到新节点，中间数据项移动到父节点，一半较小的数据项不动

（3）重新设置一半较大的数据项和中间数据项的父子节点关系

B树-5

4：插入到已满的根节点中，这就需要做根节点分裂，算法如下：

（1）创建一个空节点，作为分裂节点的父节点

（2）再创建一个空节点，作为分裂节点的兄弟节点

（3）将一半较大的数据项移动到新的兄弟节点中，中间数据项移动到新的父节点，将一半较小的数据项不动

（4）重新设置一半较大的数据项和中间数据项的父子节点关系

- B树的删除算法

1：若删除结点为非叶结点，且被删关键字为该结点中第i个关键字key[i]，则可从C[i]所指的子树中找出最小关键字Y，代替key[i]的位置，然后在叶结点中删去Y。这样一来，就把在非叶结点删除关键字k的问题就变成了删除叶子结点中的关键字的问题了。

2：要删除的关键字在叶子节点上的情况：先把这个关键字删除掉，然后再调整

B树-6

3：调整情况：

（1）如果被删关键字所在结点的原关键字个数n≥ceil(m/2)，说明删去该关键字后该结点仍满足B树的定义，直接删除，不做调整

（2）如果被删关键字所在结点的关键字个数n=ceil(m/2)-1，说明删去该关键字后该结点将不满足B树的定义，需要调整，过程为：如果其左右兄弟结点中有“多余”的关键字，即与该结点相邻的右（左）兄弟结点中的关键字数目大于ceil(m/2)。则可将右（左）兄弟结点中最小（大）关键字上移至父结点，而将父结点中小（大）于该上移关键字的关键字下移至被删关键字所在结点中

（3）如果左右兄弟结点中没有“多余”的关键字，需把要删除关键字的结点与其左（或右）兄弟结点以及父结点中分割二者的关键字合并成一个结点，即在删除关键字后，该结点中剩余的关键字，加上父结点中的关键字Ki一起，合并到Ci所指的兄弟结点中去。如果因此使父结点中关键字个数小于ceil(m/2)，则对此父结点做同样处理。以致于可能直到对根结点做这样的处理而使整个树减少一层。

B+树-1

- B+树

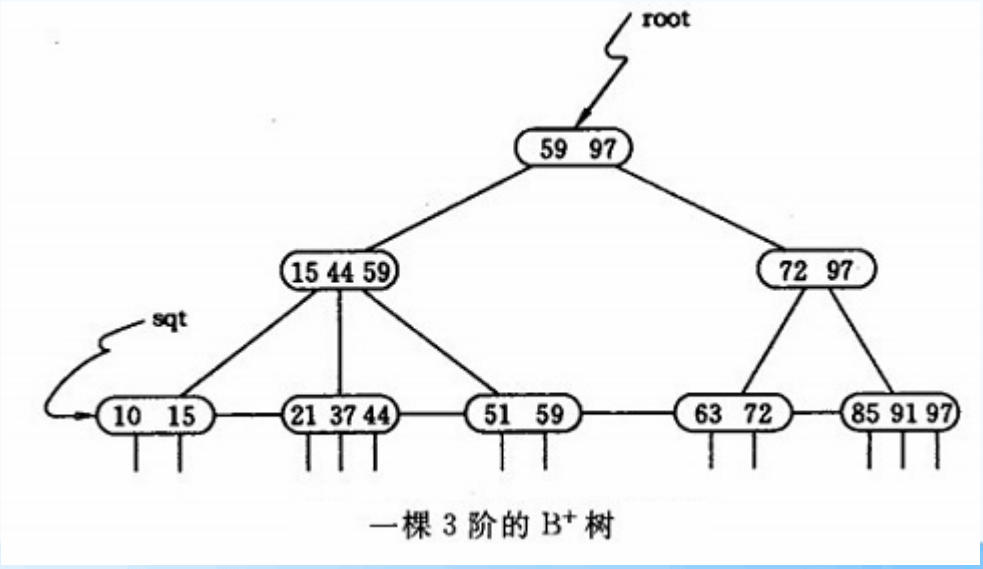
是一种B树的变形树，在实现文件索引方面比B树更好。

- B+树和B树的特性差异

1：有n棵子树的结点中含有n个关键码

2：所有的叶子结点中包含了全部关键码的信息，及指向含有这些关键码记录的指针，且叶子结点本身依关键码的大小自小而大的顺序链接

3：所有的非终端结点可以看成是索引部分，结点中仅含有其子结点中最大（或最小）关键码



一棵 3 阶的 B+ 树

B+树-2

4：叶结点中存放的是对实际数据对象的索引。

5：在B+树中有两个头指针：一个指向B+树的根结点，一个指向关键码最小的叶结点。因此可以对B+树进行两种查找运算：一种是从最小关键字起顺序查找，另一种是从根节点开始，进行随机查找。

- B+树的优势

B+树比B 树更适合实际应用中的文件索引和数据库索引，主要在于：

1：B+树的磁盘读写代价更低

B+树的非叶结点并没有指向关键字具体信息的指针，比B树更小。如果把所有同一非叶结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字更多，一次可读入内存中的关键字也就更多，从而降低了I/O读写次数

2： B+树的查询效率更加稳定

由于非叶结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找都是一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

B+树-3

3： B+树支持基于范围的查询

B+树只要遍历叶子节点就可以实现整棵树的遍历，可以很容易的支持范围查询，而B树不支持这样的操作（或者说效率太低），B+树比B树更适合做数据库的索引的数据结构。

本部分课程概览

- 本部分课程，主要学习跟Hash表相关的内容，包括：

1：Hash表是什么
2：Hash表的优缺点
3：什么是Hash化
4：Hash函数的过程以及构建
5：Hash冲突及其解决，包括：开放地址法和链地址法
6：Hash化字符串
7：Hash化的效率
8：了解Hash的应用

- Ant 1*27^2 + 15*27 + 18
- Apple 1*27^4 + 16*27^3 + 16*27^2 + 13*27 +5
- 200034 % 10000 = 34
- 5位 zzzzz = 130

Hash表-1

- Hash表

Hash表（也称散列表）是一种能实现快速插入和查找操作的数据结构，采用根据关键码值（key）来获取对应Value的方式。把关键码值映射到Value的位置的函数称为Hash函数，而存放记录的数组就是Hash表。

- Hash表的优缺点

1：优点：Hash表的查找、插入和删除操作很快，比树快很多

2：缺点：Hash表是基于数组的，一旦创建就很难改变大小，有些Hash表被基本填满时，性能会明显下降

3：缺点：不方便顺序遍历表中数据

因此Hash表适合于：不需要有序遍历数据，且可预测数据量大小的情况。

- Hash化

所谓Hash化指的就是：把数据中的关键字（关键字：可以用来识别或代表数据的内容，可以是数据本身，也可以是数据的一部分），转换成为对应的数组下标的过程，实现这个过程的函数就是Hash函数（或称散列函数）

Hash表-2

- Hash函数的基本过程

1：把关键字转换成为唯一的整数，通常这个整数范围非常大

2：把这些整数压缩到可接受的数组的索引范围

- 常用的构造Hash函数的方法

1：直接寻址法

取关键字或关键字的某个线性函数值为散列地址。即H(key)=key或H(key) = a*key + b，其中a和b为常数（这种散列函数叫做自身函数）

2：数字分析法

分析数据，找出其中数字的规律，尽可能利用这些数据来构造冲突几率较低的散列地址

3：平方取中法

取关键字平方后的中间几位作为散列地址

Hash表-3

4：折叠法

将关键字分割成位数相同的几部分，最后一部分位数可以不同，然后取这几部分的叠加和（去除进位）作为散列地址。

5：随机数法

选择一随机函数，取关键字的随机值作为散列地址，通常用于关键字长度不同的场合。

6：除留余数法

取关键字被某个不大于散列表表长的数除后所得的余数为散列地址。不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模。

- Hash冲突

就是不同的关键字，经过Hash化后，得到相同的数组下标。

Hash表-4

- Hash冲突的解决

1：开放地址法

当冲突发生时，不再使用Hash函数，而是采用其它的方式，来找到一个未使用的数组地址。

2：链地址法

数组内不直接存放数据，而是存放实际存放数据的链表，这样当冲突发生时，数据直接加入到数组下标所指的链表中。

- 开放地址法

1：线性探测

就是从冲突处开始，依次向下查找未使用的地址，到底了就回到开头继续查找，这种方式容易产生聚集，聚集越大，性能会降低。

哈希表中，一串连续的已填充单元叫做填充序列；如果加入新的数据项，使得填充序列也变长，这叫做聚集。

Hash表-5

2：二次探测

就是从冲突处开始，向下查找未使用的地址，每次增加步数的平方，从而加大探测的间隔，减少聚集的程度。它仍然会产生二次聚集的问题，毕竟步骤的平方也是固定的步长大小。

3：再哈希法

就是从冲突处开始，向下查找未使用的地址，但是每次增加的步数，是使用不同的Hash函数，对关键字再次Hash的值，这样就不采用固定步长了。

通常第二个Hash函数具有如下特点：

（1）和第一个Hash函数不同

（2）不能输出0，否则原地踏步，就死循环了

建议采用公式：stepSize=常量 -（key % 常量），

其中常量是小于数组容量的质数

Hash表-6

- 链地址法

1：装填因子：指的是装入的数据个数和数组容量的比值。

在开放地址法里面，装填因子会比1小，而链地址法里面，由于每个数组项指向一个链表，可以装载多个数据，因此这个装填因子可以比1大。

在开放地址法里面，当装填因子到了一半或2/3后，性能下降很快；在链地址法里面就没有这个问题，因此链地址法更健壮。

2：重复：在链地址法里数据可重复，所有相同关键字值的项都放在同一个链表中

- 设计好的Hash函数

1：一定要能快速运算

2：关键字是随机的情况：采用index=key%arraySize是一个不错的选择

3：关键字不是随机的情况

（1）不要使用无用数据，比如校验和就可以不要

（2）使用所有的数据，关键字的每个部分，都应该在Hash函数中有所反映

4：使用质数作为取模的基数

使用质数能保证关键字较为平均的映射到数组中

Hash表-7

- Hash化字符串

  Horner（英国数学家）数学恒等式，比如：

  var4*n^4+ var3*n^3+ var2*n^2+ var1*n^1+ var0 可以转换成：

  (((var4*n + var3)*n + var2)*n + var1)*n + var0

- Hash化的效率

  如果没有发生冲突，插入、查找和删除都是O(1)；如果发生冲突，就要依赖后面的探测长度了，而平均探测长度跟装填因子成正比。

- 了解Hash的应用

  Hash除了用在快速访问数据的场景，还常用在信息安全领域的加密上。比如大家熟知的一些Hash算法：MD5、SHA-1、SHA-256等等

本部分课程概览

- 本部分课程，主要学习跟堆相关的内容，包括：

1：堆是什么
2：堆的特点
3：堆的插入、删除、查找算法，并代码示例
4：堆的效率
5：堆排序的算法，并代码示例
6：堆排序的效率

堆-1

- 堆

  堆是一种特殊的二叉树，主要用来解决任务优先级调度类问题。

- 堆的特点

1：堆是完全二叉树

2：堆常常用一个数组来实现，由于堆是完全二叉树，因此这个数组是没有”洞”的

3：堆中每个节点都满足堆的条件

4：堆和二叉搜索树相比是弱序的，堆只要求从根到叶子的每条路径上，节点是按照顺序排列的，而不要求左边一定小于右边。将根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。

- 用数组来表示一颗树

  如果数组中节点的索引为x，那么有：

1：它的父节点索引是(x-1)/2

2：它的左子节点索引是 2*x+1

3：它的右子节点索引是 2*x+2

堆-2

- 堆的插入算法

1：先插入到数组末尾

2：然后向上筛选，直到它在一个大于它的节点之下，小于它的节点之上的位置

- 堆的删除算法

这里约定始终删除最大值，而且堆以最大值为根。

1：移走根

2：把最后一个节点移动到根的位置

3：向下筛选，直到它在一个大于它的节点之下，小于它的节点之上的位置

- 堆的查找

由于堆是弱序的，所以在查找的时候，没有足够的信息来判断应该走哪一个子分支，所以它的查找不是很方便，只能顺序的查找数组的每个项，较慢。

堆-3

- 堆的效率

  堆操作的时间复杂度是O(logN)，以2为底，比二叉搜索树略慢。

- 堆排序

  基本思想：先循环插入所有的无序数据，然后循环删除最大节点即可。

  问题：由于堆的新增和删除时间复杂度都是O(logN)，并且每个方法执行了N次，所以整个排序操作需要O(N*logN)，跟快速排序一样，但是不如快速排序快，因为在向下筛选数据时，循环次数更多。

  改进方法：去掉新增时的排序，然后对任意排列数据项的数组，再重新排列成堆，这样只需要调用N/2次向下的筛选方法

- 堆排序的效率

  堆排序操作的时间复杂度为O(N*logN)，比快速排序略慢，但是它对初始数据的分布不敏感，这点比快速排序优越。

本部分课程概览

- 本部分课程，主要学习跟图（不带权）相关的内容，包括：

1：图是什么
2：图的基本术语
3：在程序中表示图
4：图的搜索，包括深度搜索和广度搜索，算法并代码示例
5：最小生成树，算法并代码示例
6：有向图的拓扑顺序，算法并代码示例
7：有向图的连通性，算法并代码示例
8：Warshall算法并代码示例

图-1

- 图简介

图是一种用来描述多对多结构关系的数据结构，图中的每个顶点元素，可以有零到多个直接前驱元素，也可以有零到多个直接后继元素

- 图的定义

图G由两个集合构成，记作G= (V，{A})，其中V是顶点的非空有限集合，A是边的有限集合，其中边是顶点的无序或有序对(对应的图称为无向图或有向图)

- 图的实例

现实生活中有很多都可以抽象成为图的结构，比如：

1：交通图：顶点—地点；边—连接地点的路
2：电路图：顶点—元件；边—连接元件之间的线路
3：通讯线路图：顶点—地点；边—地点间的连线
4：各种流程图：顶点—活动；边—各活动之间的顺序关系

图-2

- 图的基本术语

1：邻接：如果两个顶点被同一条边连接，那么这两个顶点就是邻接的，这两个顶点可以被称为邻接点，连接的边可以称为关联边

2：顶点的度：与顶点相关联的边的数目

3：路径：边的序列

4：无向图：边没有方向的图

5：有向图：边有方向的图

6：连通图：至少有一条路径可以把所有的顶点连接起来的图

7：强连通图：任何两个顶点之间都存在连通路径的图

8：入度和出度：有向图中，以某个顶点为起始（终止）的有向边数，成为该顶点的出度（入度）

9：带权图：边被赋予一个权值的图

10：有环图：指的是存在路径的起点和终点是同一个顶点的图。不包含环的图就是树，在图中，树的顶点可以连接任意数量的顶点。

图-3

- 在程序中表示图

1：顶点：可以用一个对象来描述，要有一个识别码；多个顶点在图中，可以用一个顶点数组来表示

2：边：可以用邻接矩阵或者是邻接表的方式来描述

3：邻接矩阵：是一个二维数组，数据项表示两个顶点间是否存在边，如果有N个顶4点，那么邻接矩阵就是N*N的数组

4：邻接表：是一个链表数组，每个单独的链表记录了有哪些顶点与当前顶点邻接

- 图的搜索（遍历）

通常有两种搜索方法：深度优先 和 广度优先

- 深度优先搜索

深度优先，通常使用栈来实现，基本步骤如下：

1：访问一个邻接的未访问的顶点，把它压入栈中，并做标记，然后继续做这一步

2：第一步结束，如果栈不为空，从栈中弹出一个顶点，然后重复第一步

3：当最后栈中没有顶点了，那么搜索也就完成了

图-4

- 广度优先搜索

广度优先，通常使用队列来实现，基本步骤如下：

1：访问下一个将要访问的邻接顶点，把它放入队列中并做标记，然后继续这一步

2：第一步结束，如果队列不为空，从队列头取出一个顶点，作为当前顶点，然后重复这两步

3：当最后队列中没有顶点了，那么搜索也就完成了

- 最小生成树

包含图中所有顶点的最小连通子图称为图的最小生成树，如果在该子图中删除任意一条边，子图将不再是连通的。最小生成树有如下特性：

1：一棵n个顶点的最小生成树有且仅有足以构成树的n-1条边

2：若在一棵最小生成树上删除一条边，就不再连通

3：若在一棵最小生成树上添加一条边，必定构成一个环

创建最小生成树的算法和搜索算法是一样的，只不过需要记录走过的边。

图-5

- 有向图的拓扑排序

指的就是将有向图中的顶点以线性方式进行排序，其步骤为：

1：找到一个没有后继的顶点

2：从图中删除这个顶点，在列表前面插入顶点的标记

3：重复上述步骤，直到所有顶点都从图中删除，此时，列表显示的顶点顺序就是拓扑顺序

注意：拓扑排序只能在有向无环图中进行

- 有向图的连通性

有向图中一个有趣的问题是：从一个指定顶点出发，可以到达哪些顶点？

这个可以通过查看连通性表来得到，所谓连通性表，就是第一个是起点，后面就是这个顶点能到达的顶点了，简单的可以修改前面的程序得到。

图-6

- Warshall算法

解决问题的思路：如果我们修改原来的邻接矩阵，直接在上面标明一个顶点是否可以到达另外一个顶点的话，就可以直接通过这个修改过的邻接矩阵来得到一个顶点能到达的顶点了。

这个修改过的邻接矩阵表示的图，就称为原图的传递闭包。Warshall算法就是用来把邻接矩阵转变成图的传递闭包的，步骤如下：

1：使用一个三层嵌套循环，最外层循环每一行i，中间层循环列j

2：如果(i,j)为1，表明有一条从i到j的边，就执行内层循环

3：检查以i为列的每个单元k，如果(k,i)为1，表明有一条从k到i的边，进而推出存在一条从k到j的边，那么就可以设置(k,j)为1

4：如此循环完成，得到的新的邻接矩阵就是原图的传递闭包

本部分课程概览

- 本部分课程，主要学习跟带权图相关的内容，包括：

1：什么是权
2：带权图的最小生成树
3：普里姆算法，代码示例
4：最短路径问题
5：迪杰斯特拉算法，代码示例
6：弗洛伊德算法，代码示例
7：稀疏图和稠密图
8：图的效率

带权图-1

- 权

可以给图中的边设置权值，用来代表一定的含义，比如：交通图中可以设置边的权为距离，当然也可以是交通流量等等。

带权图可以用来处理很多有趣的实际问题，比如：两个城市之间的最短距离，或者造价最低的工程路线等等。

- 带权图的最小生成树

就是权值总和最小的生成树。

- 普里姆算法步骤：

1：任取一个顶点，放入树的集合中

2：找到这个顶点到其他顶点的所有的边，当然这些其他顶点不能在树的集合中，把这些边放入优先级队列中

3：找到权值最小的边，把它和它所到达的顶点放入树的集合中，重复2和3，直到所有顶点都在树的集合中。

带权图-2

- 最短路径问题

带权图的一个常见应用就是用来求从一个顶点到另外顶点的最小XX的问题，比如：最短路径、最便宜的费用、最少的总时间等等。

- 迪杰斯特拉（Dijkstra）算法

迪杰斯特拉算法是由荷兰计算机科学家狄克斯特拉于1959 年提出的，因此又叫狄克斯特拉算法。是求从一个顶点到其余各顶点的最短路径的算法，解决的正是有向图中的最短路径问题。算法步骤如下：

1：创建一个额外的数组sPath，用来保持源点到其他顶点的最短路径，这些值在算法开始的时候，就是相应边的权值，算法运行过程中是不断变化的，最后它存储的是从源点到其他顶点的真正的最短路径

2：把第一个顶点放入树中，同时给sPath赋初始值

3：进入循环，直到所有的顶点都放入树中就结束

4：在循环中，先选择sPath中的最小距离

5：把对应的顶点放入树中，设置这个顶点为当前顶点

6：根据新的当前顶点，更新sPath的内容，然后继续循环

带权图-3

- 每对顶点间的最短路径问题

  一个解决方案就是循环用每个顶点作为起始点，去使用迪杰斯特拉算法进行计算，最后把结果汇总起来。

  另外一个方案就是采用Floyd算法。

- 弗洛伊德（Floyd）算法

  又称为插点法，是一种用于在加权图中，求顶点之间最短路径的算法。其算法和Warshall算法类似，只不过在找到两端路径后，这里需要插入两端路径的权值之和，并保持这个权值始终最小即可。

带权图-4

- 稀疏图和稠密图

  边较少（多）的图称为稀疏图（稠密图）。

- 图的效率

1：用邻接矩阵实现的图，大多是O(N^2)，N是顶点个数

2：在稀疏图中，用邻接表的方式来代替邻接矩阵，可以提高效率

3：无权图，邻接表的深度优先搜索需要O(N+E)，N是顶点个数，E是边的个数

4：带权图，最小生成树算法和最短路径算法都需要O((E+N)logN)

5：Warshall算法和Floyd算法都需要O(N^3)