《大话数据结构》读书笔记

一.数据结构绪论

- 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
- 数据结构是一门研究非数值计算的程序设计问题中的操作对象，以及它们之间的关系和操作等相关问题的学科。
- 程序设计 = 数据结构 + 算法
- 数据：描述客观事物的符号，是计算机中可操作的对象，是能被计算机识别，并输入给计算机处理的符号集合
    - 数值类型：整型、实型等
    - 非数值型：字符、声音、图像、视频等
- 数据元素：组成数据的、有一定意义的基本单位，在计算机中通常作为整体处理，也被称为记录
- 数据项：一个数据元素可以由若干个数据项组成，是数据不可分割的最小单位
- 数据对象：性质相同的数据元素的集合，是数据的子集
- 逻辑结构：数据对象中数据元素之间的相互关系
    - 集合结构：集合结构中的数据元素除了同属于一个集合外，它们之间没有其他关系
    - 线性结构：线性结构中的数据元素之间是一对一的关系
    - 树形结构：树形结构中的数据元素之间存在一种一对多的层次关系
    - 图形结构：图形结构的数据元素是多对多的关系
- 物理结构：是指数据的逻辑结构在计算机中的存储形式
    - 顺序存储结构：把数据元素存放在地址连续的存储单元里，其数据间的逻辑关系和物理关系是一致的
    - 链式存储结构：把数据元素存放在任意的存储单元里，这组存储单元可以是连续的，也可以是不连续的
- 数据类型：一组性质相同的值的集合及定义在此集合上的一些操作的总称
- 抽象：抽取出事物具有的普遍性的本质
- 抽象数据类型：一个数学模型及定义在该模型上的一组操作
- 抽象的意义在于数据类型的数学抽象特性
- 抽象数据类型体现了程序设计中问题分解、抽象和信息隐藏的特性

二.算法

- 算法：解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作
- 算法的基本特性：
    - 输入：算法具有零个或多个输入
    - 输出：算法至少有一个或多个输出
    - 有穷性：算法在执行有限的步骤之后，自动结束而不会出现无限循环，并且每一个步骤在可接受的时间内完成
    - 确定性：算法的每一步骤都具有确定的含义，不会出现二义性
    - 可行性：算法的每一步必须是可行的，即每一步都能够通过执行有限次数完成
- 算法设计的要求：

    - 正确性：算法至少应该具有输入、输出和加工处理无歧义性、能正确反映问题的需求、能够得到问题的正确答案
        - 算法正确性的四个层次
            1. 算法程序没有错误
            2. 算法程序对于合法的输入数据能够产生满足要求的输出结果
            3. 算法程序对于非法的输入数据能够得到满足规格说明的结果
            4. 算法程序对于精心选择的，甚至刁难的测试数据都有满足要求的输出结果
    - 可读性：便于阅读、理解和交流
    - 健壮性：当输入数据不合法时，算法也能够做出相关处理，而不是产生异常或莫名其妙的结果
    - 时间效率高和存储量低
- 算法效率的度量方法
    - 事后统计法：主要是通过设计好的测试程序和数据，利用计算机计时器对不同算法编制的程序的运行时间进行比较，从而确定算法效率的高低（此方法缺陷较大）
    - 事前分析估算法：在计算机程序编制前，依据统计方法对算法进行估算
- 函数的渐近增长：给定两个函数f(n)和g(n)，如果存在一个整数N，使得对于所有的n>N，f(n)总是比g(n)大，那么，我们说f(n)的增长渐近快于g(n)。
    - 可以忽略加法常数
    - 最高次项相乘的常数并不重要
    - 最高次项的指数大的，函数随着n的增长，结果也会变得增长特别快
    - 判断一个算法的效率时，函数中的常数和其他次要项常常可以忽略，而更应该关注主项（最高阶项）的阶数
- 算法时间复杂度：在进行算法分析时，语句总的执行次数T(n)是关于问题规模n的函数，进而分析T(n)随n的变化情况并确定T(n)的数量级。记作：T(n) = O(f(n))。表示随问题规模n的增大，算法执行时间增长率和f(n)的增长率相同，称作算法的渐近时间复杂度，其中f(n)是问题规模n的某个函数
- 推导大O阶方法：
    1. 用常数1取代运行时间中的多有加法常数
    2. 在修改后的运行次数函数中，只保留最高阶项
    3. 如果最高阶项存在且不是1，则去除与这个项相乘的常数。
- 常见的时间复杂度：

表 2-10-1

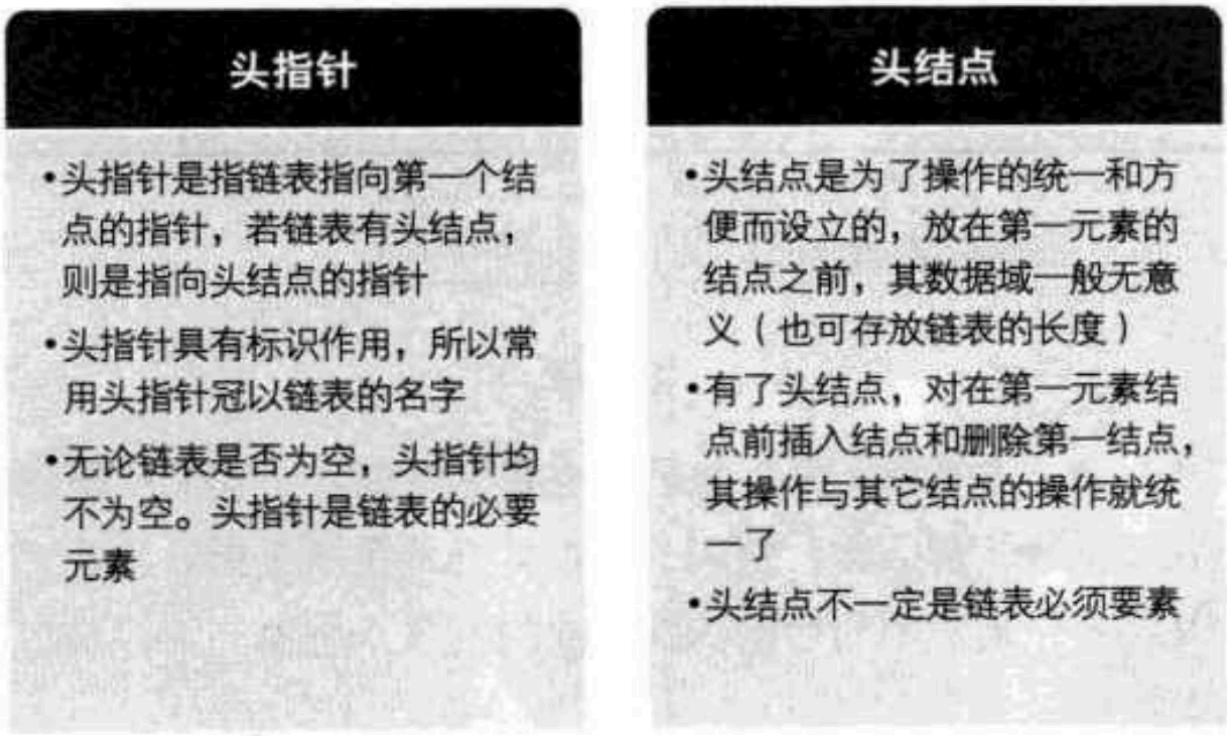
执行次数函数	阶	非正式术语
12	O(1)	常数阶
2n+3	O(*n*)	线性阶
$3n^2+2n+1$	$O(n^2)$	平方阶
$5\log_2 n+20$	O(log*n*)	对数阶
$2n+3n\log_2 n+19$	O(*n*log*n*)	nlogn 阶
$6n^3+2n^2+3n+4$	$O(n^3)$	立方阶
2^n	$O(2^n)$	指数阶

    - O(1)<O(logn)<O(n)<O(nlogn)<O(n^2)<O(n^3)<O(2^n)<O(n!)<O(n^n)
- 最坏情况：最坏情况运行时间是一种保证。通常，除非特别指定，我们提到的运行时间都是最坏情况的运行时间
- 平均情况：平均运行时间是期望的运行时间
- 算法空间复杂度：通过计算算法所需的存储空间实现，算法空间复杂度的计算公式记作：S(n) = O(f(n))，其中，n为问题的规模，f(n)为语句关于n所占存储空间的函数
- 若算法执行时所需的辅助空间相对于输入数据量而言是个常数，则称此算法为原地工作，空间复杂度为O(1)

三.线性表

- 线性表(List)：零个或多个数据元素的有限序列
- 数学语言定义：若将线性表记为（a_1，……，a_{i-1}，a_i，a_{i+1},……，a_n），则表中a_{i-1}领先于a_i，a_i领先于a_{i+1},称a_{i-1}是a_i的直接先驱元素，a_{i+1}是a_i的直接后继元素。当i=1，2，……，n-1时，a_i有且仅有一个直接后继，当当i=2，3，……，n时，a_i有且仅有一个直接前驱。
- 线性表元素的个数n（n≥0）定义为线性表的长度，当n=0，成为空表
- 在复杂线性表中，一个数据元素可以由若干个数据项组成
- 线性表的顺序存储结构：用一段地址连续的存储单元依次存储线性表的数据元素
    - 一维数组来实现顺序存储结构
    - 描述顺序存储结构的三个属性：
        1. 存储空间的其实位置：数值data，它的存储位置就是存储空间的存储位置
        2. 线性表的最大存储容量：数组长度MaxSize
        3. 线性表的当前长度：length

- 在任意时刻，线性表的长度应该小于等于数组的长度
- 地址：存储器中的每个存储单元都有自己的编号，这个编号称为地址
    - $LOC(a_i)=LOC(a_1)+(i-1)*c$
- 顺序存储结构的插入与删除
    - 插入算法：
        - 如果插入位置不合理，抛出异常
        - 如果线性表长度大于等于数组长度，则抛出异常或动态增加容量
        - 从最后一个元素开始向前遍历到第i个位置，分别将它们都向后移动一个位置
        - 将要插入元素填入位置i处
        - 表长加1
    - 删除算法：
        - 如果删除位置不合理，抛出异常
        - 取出删除元素
        - 从删除元素位置开始遍历到最后一个元素位置，分别将它们都向前移动一个位置
        - 表长减1
- 线性表顺序存储结构的优缺点
    - 优点：
        - 无须为表示表中元素之间的逻辑关系而增加额外的存储空间
        - 可以快速地存取表中任一位置的元素
    - 缺点：
        - 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”
- 线性表链式存储结构定义：为了表示每个数据元素a_i与其直接后继数据元素a_{i+1}之间的逻辑关系，对数据元素a_i来说，除了存储其本身的信息之外，还需要存储一个指示其直接后继的信息。我们把存储数据元素信息的域称为数据域，把存储直接后继位置的域称为指针域。指针域中存储的信息称做指针或链。这两部分信息组成数据元素a_i的存储映像，称为结点Node。n个结点链结成一个链表，即为线性表的链式存储结构，因为此链表的每个结点中只包含一个指针域，所以叫做单链表
    - 头指针：链表中第一个节点的存储位置。线性链表的最后一个结点指针为“空”（NULL）
    - 头结点：单链表的第一个结点前附设的一个结点。头结点的数据域可以不存储任何信息



    - 如果p->data = a_i,那么p->next->data = a_{i+1}

- 单链表的读取
    - 获得链表第i个数据的算法思路：
        1. 声明一个结点p指向链表第一个结点，初始化j从1开始
        2. 当j<i时，就遍历链表，让p的指针向后移动，不断指向下一结点，j累加1
        3. 若到链表末尾p为空，则说明第i个元素不存在
        4. 否则查找成功，返回结点p的数据
- 单链表的插入与删除：
    - 单链表第i个数据插入结点的算法：
        1. 声明一结点p指向链表第一个结点，初始化j从1开始
        2. 当j<i时，就遍历链表，让p的指针向后移动，不断指向下一结点，j累加1
        3. 若到链表末尾p为空，则说明第i个元素不存在
        4. 否则查找成功，在系统中生成一个空结点s
        5. 将数据元素e赋值给s->data
        6. 单链表的插入标准语句：s->next = p->next; p->next=s
        7. 返回成功
    - 单链表第i个数据删除结点的算法：
        1. 声明一结点p指向链表第一个结点，初始化j从1开始
        2. 当j<i时，就遍历链表，让p的指针向后移动，不断指向下一结点，j累加1
        3. 若到链表末尾p为空，则说明第i个元素不存在
        4. 否则查找成功，将欲删除的结点p->next赋值给q
        5. 单链表的删除标准语句p->next=q->next
        6. 将q结点中的数据赋值给e，作为返回
        7. 释放q结点
        8. 返回成功
    - 对于插入或删除数据越频繁的操作，单链表的效率优势就越明显
- 单链表的整表创建：

1. 声明一结点p和计数器表量i
2. 初始化一空链表L
3. 让L的头结点的指针指向NULL，即建立一个带头结点的单链表
4. 循环
    - 生成一新结点赋值给p
    - 随机生成一数字赋值给p的数据域p->data
    - 将p插入到头结点与前一新结点之间

- 单链表的整表删除：
    1. 声明一结点p和q
    2. 将第一个结点赋值给p
    3. 循环
        - 将下一结点赋值给q
        - 释放p
        - 将q赋值给p
- 单链表结构与顺序存储结构优缺点

- 静态链表：用数组描述的链表
- 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环，这种头尾相接的单链表称为单循环链表
- 双向链表：在单链表的每个结点中，再设置一个指向其前驱结点的指针域

四.栈与队列

- 栈：限定仅在表尾进行插入和删除操作的线性表，后进先出（LIFO）
    - 栈顶：允许插入和删除的一端
    - 栈底
    - 空栈：不含任何数据元素的栈
- 进栈（push）出栈（pop）变化形式
    - 最先进栈的元素，不一定只能最后出栈

    - 312的出栈次序不会出现（只有三个元素的情况下）
- 栈的顺序存储结构
    - 数组下标为0的一端作为栈底
    - 进栈和出栈的时间复杂度均为O（1）
    - 缺点：事先必须确定数组空间的大小，万一不够用了，就需要用编程手段扩展数组的容量
- 两栈共享空间
    - 数组有两个端点，两个栈有两个栈底，让一个栈的栈底为数组的始端，即下标为0处，另一个栈为栈的末端，即下标为数组长度n-1处。如果两个栈增加元素，就是两端点栈向中间延伸（只针对两个具有相同数据类型的栈的技巧）
- 栈的链式存储结构
    - 把栈顶放在单链表的头部
    - 进栈和出栈的时间复杂度都为O（1）
- 如果栈的使用过程中元素变化不可预料，有时很小，有时非常大，那么最好是用连战，反之，如果它的变化在可控范围内，建议使用顺序栈
- 栈的作用
    - 简化了程序设计的问题，划分了不同关注层次，使得思考范围缩小，更加聚焦于要解决的核心问题
- 栈的应用
    - 递归：把一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，称作递归函数。每一个递归定义必须至少满足一个条件，满足时递归不再进行，即不再引用自身而是返回值退出
        - 斐波那契数列：前面相邻两项之和，构成了后一项
        - 递归和迭代
            - 迭代：使用的是循环结构，不需要反复调用函数和占用额外的内存
            - 递归：使用的是选择结构，能使程序的结构更清晰、更简洁、更容易让人理解，从而减少读懂代码的时间，但大量的递归会建立函数的副本，耗费大量的时间和内存
    - 四则运算表达式求值
        - 后缀表示法：从左到右遍历表达式的每个数字和符号，遇到数字就进栈，遇到符号就将处于栈顶两个数字出栈，进行运算，运算结果进栈，一直到最终获得结果
        - 中缀表达式（标准四则运算表达式）转后缀表达式：从左到右遍历中缀表达式的每个数字和符号，若是数字就输出，即称为后缀表达式的一部分；若是符号，则判断其与栈顶符号优先级，是右括号或优先级低于栈顶符号则栈顶元素依次出栈并输出，并将当前符号进栈，一直到最终输出后缀表达式为止
- 队列：只允许在一端进行插入操作、而在另一端进行删除操作的线性表，先进先出（FIFO）
- 循环队列
    - 队列顺序存储不足：出队列的时间复杂度为O（n）
    - 定义：头尾相接的顺序存储结构
    - 队列计算长度公式：(rear-front+QueueSize)%QueueSize
    - 不足：面临数组可能会溢出的问题
- 队列的链式存储结构
    - 线性表的单链表，只不过只能尾进头出
- 循环队列和链队列的比较：
    - 时间上，它们的基本操作都是O（1），如果入队出队频繁，则还是有差异
    - 空间上，链队列更加灵活

- 在可以确定队列长队最大值的情况下，建议用循环队列，如果无法预估队列长度，则用链队列

五.串

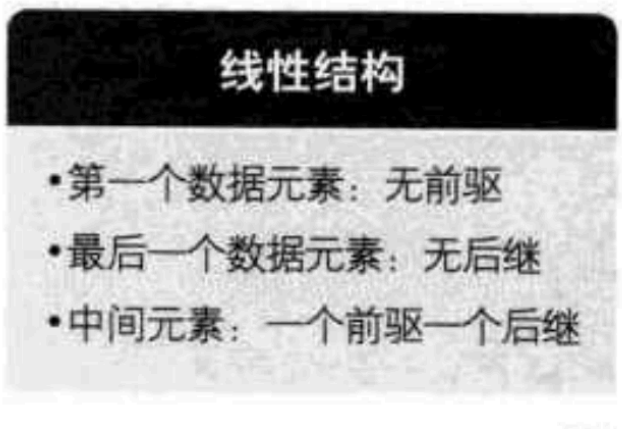
- 串是由零个或多个字符组成的有序序列，又名字符串
  - 记为：$s=a_1a_2.....a_n$(n大于等于0)
  - 串中的字符数目n称为串的长度
  - 零个字符的串称为空串
- 串的比较
  - 通过组成串的字符之间的编码来进行比较
    - n<m,且$a_i=b_i$(i=1,2,.....，n)
    - 存在某个k≤min（m,n）,使得$a_i=b_i$(i=1,2,.....，k-1)，$a_k=b_k$
- 串的抽象数据类型
  - 串的逻辑结构和线性表相似，不同之处在于串针对的是字符集
  - 串的基本操作与线性表有很大差别，线性表更关注单个元素，串中更多的是查找子串位置、替换子串、得到指定位置子串等
- 串的存储结构
  - 串的顺序存储结构是用一组地址连续的存储单元来存储串中的字符序列的。一般用定长数组定义
  - 串的链式存储结构与线性表相似，一个结点可以存放一个或多个字符，最后一个结点未被占满时，可以用？“#”或其他非串值字符补全
  - 串的链式存储结构除了在连接串与串操作时有一定方便之外，总的来说不如顺序存储灵活，性能也不如顺序存储结构好
- 朴素的模式匹配算法：对主串的每一个字符作为子串开头，与要匹配的字符串进行匹配。对主串做大循环，每个字符开头做要匹配的字符串的长度的小循环，直到匹配成功或全部遍历完成为止
- KMP模式匹配算法
  -

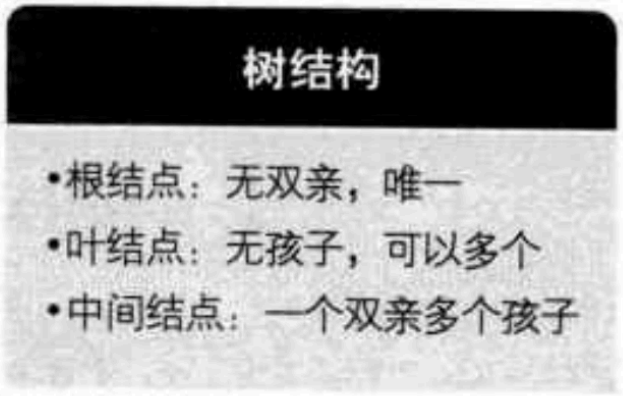
/璇玑图/

六.树

- 树是n（n≥0）个结点的有限集。n=0时称为空树。在任意一颗非空树中：（1）有且仅有一个特定的称为根（root）的结点；（2）当n>1时，其余结点可分为m（m>0）个互不相交的有限集T_1、T_2、.....、T_m，其中每一个集合本身又是一棵树，并且称为根的子树（SubTree）
  - n>0时根结点是唯一的，不可能存在多个根结点
  - m>0时，子树的个数没有限制，但它们一定是互不相交的
- 结点的度：结点拥有的子树数
  - 叶结点（终端结点）：度为0的结点
  - 分支结点（非终端结点）：度不为0的结点
- 树的度（degree）：树内各结点的度的最大值
- 结点的子树的根称为该结点的孩子（child），相应的，该结点称为孩子的双亲（parent）。同一个双亲的孩子之间互称兄弟。
  - 结点的祖先是从根到该结点所经分支上的所有结点
  - 以某结点为根的子树中的任一结点都称为该结点的子孙

- 结点的层次（level）：根为第一层，根的孩子为第二层。若结点在第l层，则其子树的根就在第l+1层。其双亲在同一层的结点互为堂兄弟。
- 树的深度（depth）/高度：树中结点的最大层次
- 如果将树中结点的各子树看成从左至右是有次序的，不能互换的，则称该树为有序树，否则称为无须树
- 森林是m（m≥0）课互不相交的树的集合





- 树的存储结构
  - 双亲表示法：以一组连续空间存储树的结点，同时在每个结点中，附设一个指示器指示双亲结点到链表中的位置
  - 孩子表示法：把每个结点的孩子结点排列起来，以单链表作存储结构，则n个结点有n个孩子链表，如果是叶子结点则此单链表为空。然后呢n个头指针又组成一个线性表，采用顺序存储结构，存放进一个一维数组中。
  - 孩子兄弟表示法:设置两个指针，分别指向该结点的第一个孩子和此结点的右兄弟
  - 存储结构的设计是一个非常灵活的过程。一个存储结构设计得是否合理，取决于基于该存储结构的运算是否适合、是否方便、时间复杂度好不好等
- 二叉树：n（n≥0）个结点的有限集合，该集合或者为空集（称为空二叉树），或者由一个根结点和两颗互不相交的、分别称为根结点的左子树和右子树的二叉树组成
- 二叉树特点
  - 每个结点最多有两颗子树，所以二叉树中不存在度大于2的结点
  - 左子树和右子树是有顺序的，次序不能任意颠倒
  - 即使树中某结点只有一颗子树，也要区分它是左子树还是右子树
- 二叉树的5种形态
  - 空二叉树
  - 只有一个根结点
  - 根结点只有左子树
  - 根结点只有右子树
  - 根结点既有左子树又有右子树
- 特殊二叉树
  - 斜树：所有结点都只有左子树的二叉树叫左斜树，所有结点都是只有右子树的二叉树叫右斜树。两者的统称为斜树
  - 满二叉树：所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上的二叉树
  - 完全二叉树：一颗具有n个结点的二叉树按层序编号，如果编号为i（1≤i≤n）的结点与同样深度的满二叉树中编号为i的结点在二叉树中位置完全相同，则称为完全二叉树
- 二叉树的性质

- 在二叉树的第i层上至多有2^{i-1}个结点（i≥1）
- 深度为k的二叉树至多有2^k-1个结点（k≥1）
- 对任何一颗二叉树T，如果其终端结点数为n_0,度为2的结点树为n_2,则$n_0=n_2+1$
- 具有n个结点的完全二叉树的深度为⌊log2n⌋+1,⌊x⌋表示不大于x的最大整数
- 如果对一颗有n个结点的完全二叉树（其深度为⌊log2n⌋+1）的结点按层序编号，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，无双亲；如果i>1,则其双亲是结点⌊i/2⌋
    2. 如果2i>n,则结点i无左孩子；否则其左孩子是结点2i
    3. 如果2i+1>n,则结点i无右孩子，否则其右孩子是结点2i+1

- 二叉树的存储结构
    - 二叉树顺序存储结构
    - 二叉链表
- 二叉树遍历：从根结点出发，按照某种次序依次访问二叉树中所有结点，使得每个结点被访问一次且仅被访问一次
    - 前序遍历：先访问根结点，然后前序遍历左子树，再前序遍历右子树

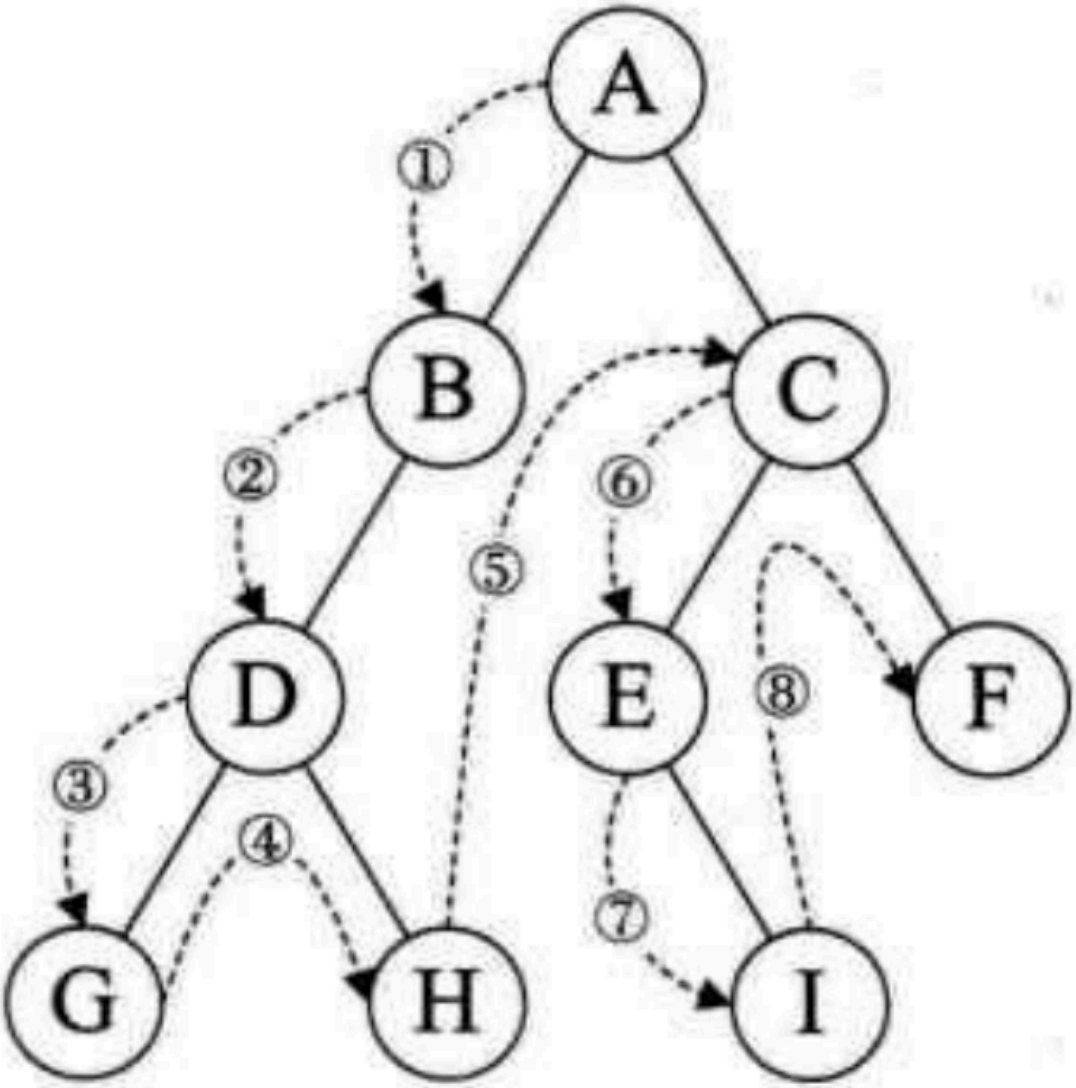


图 6-8-2

- 中序遍历：从根结点开始，中序遍历根结点的左子树，然后访问根结点，最后中序遍历右子树

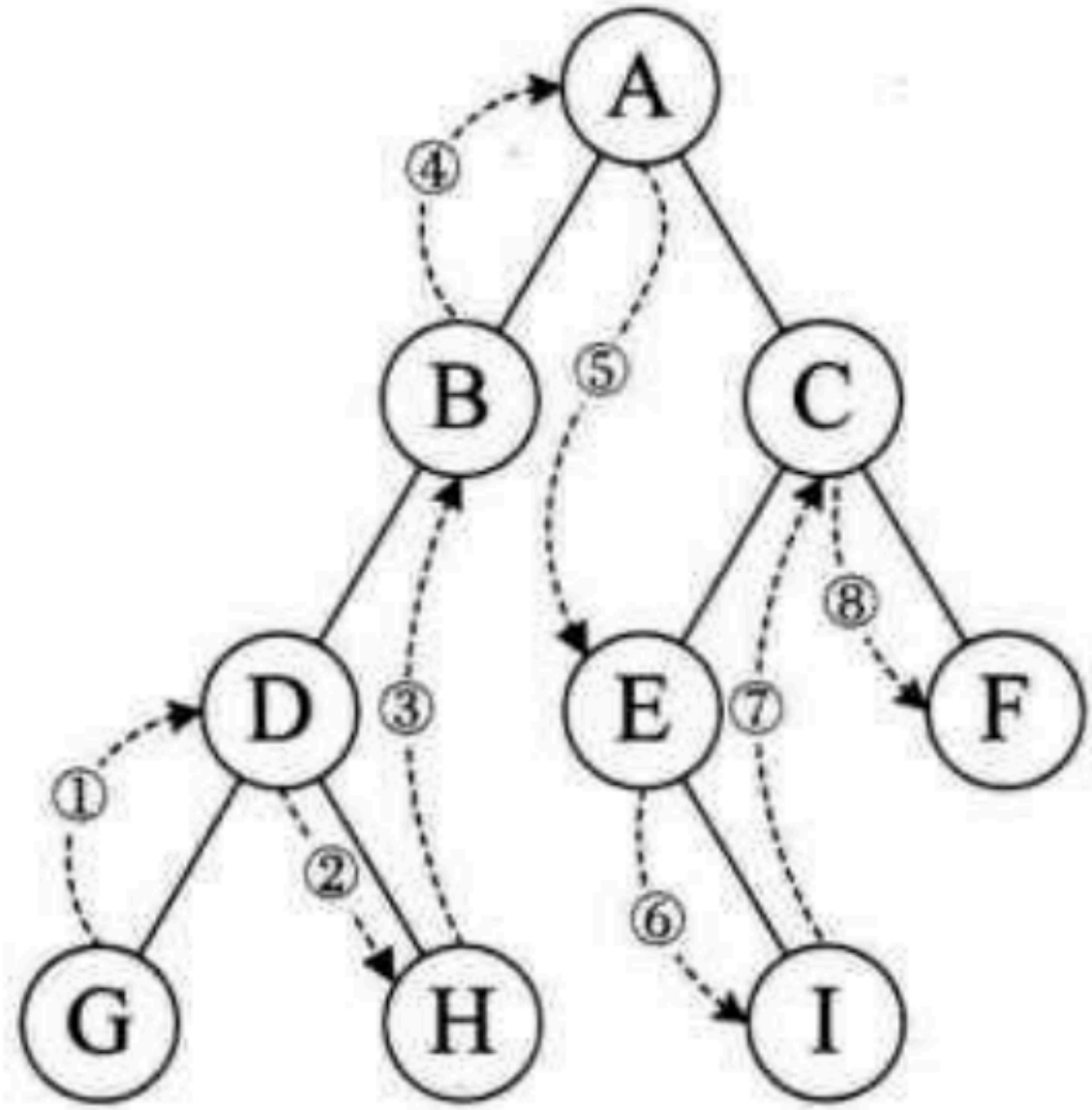


图 6-8-3

- 后序遍历：从左到右先叶子后结点的方式遍历访问左右子树，最后访问根结点

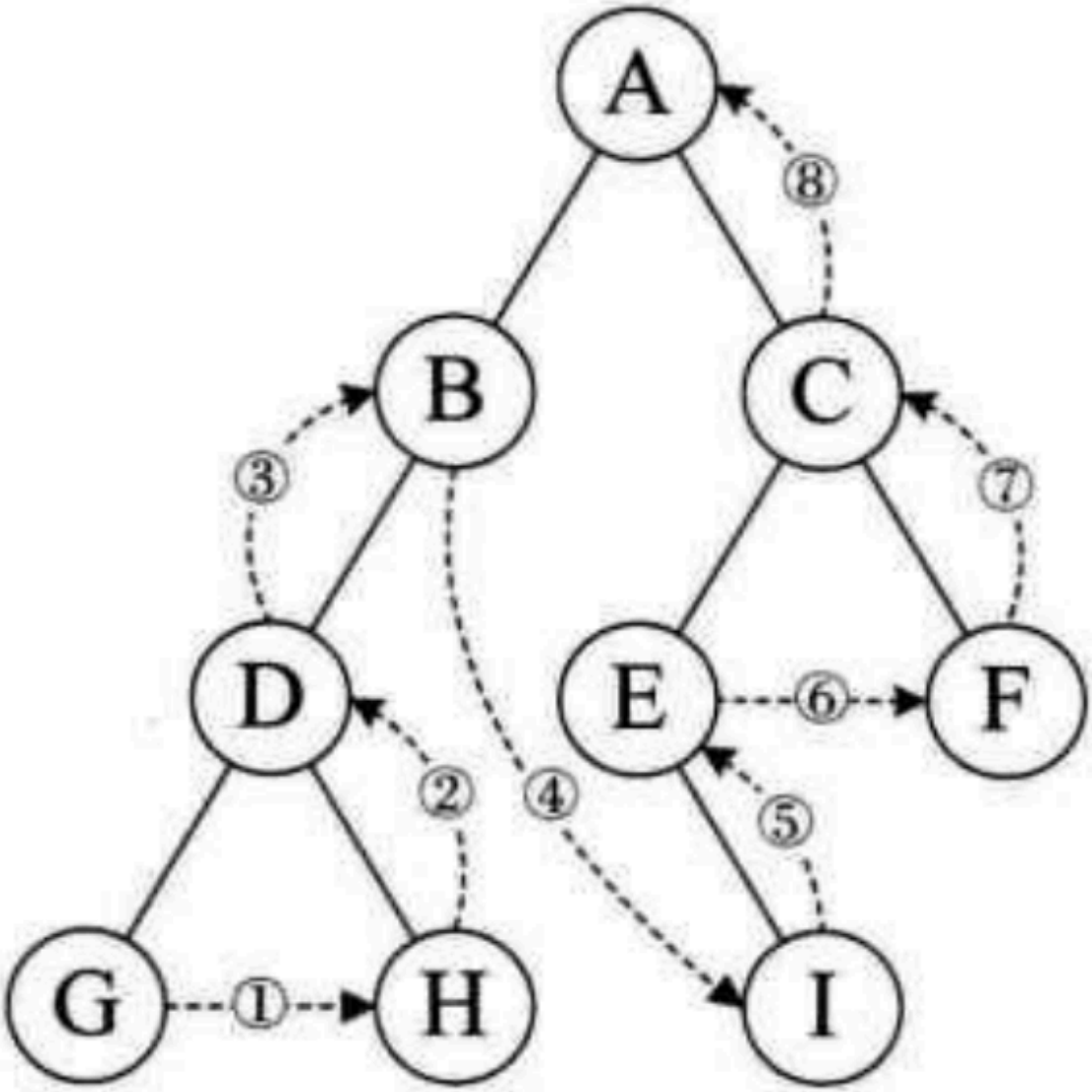
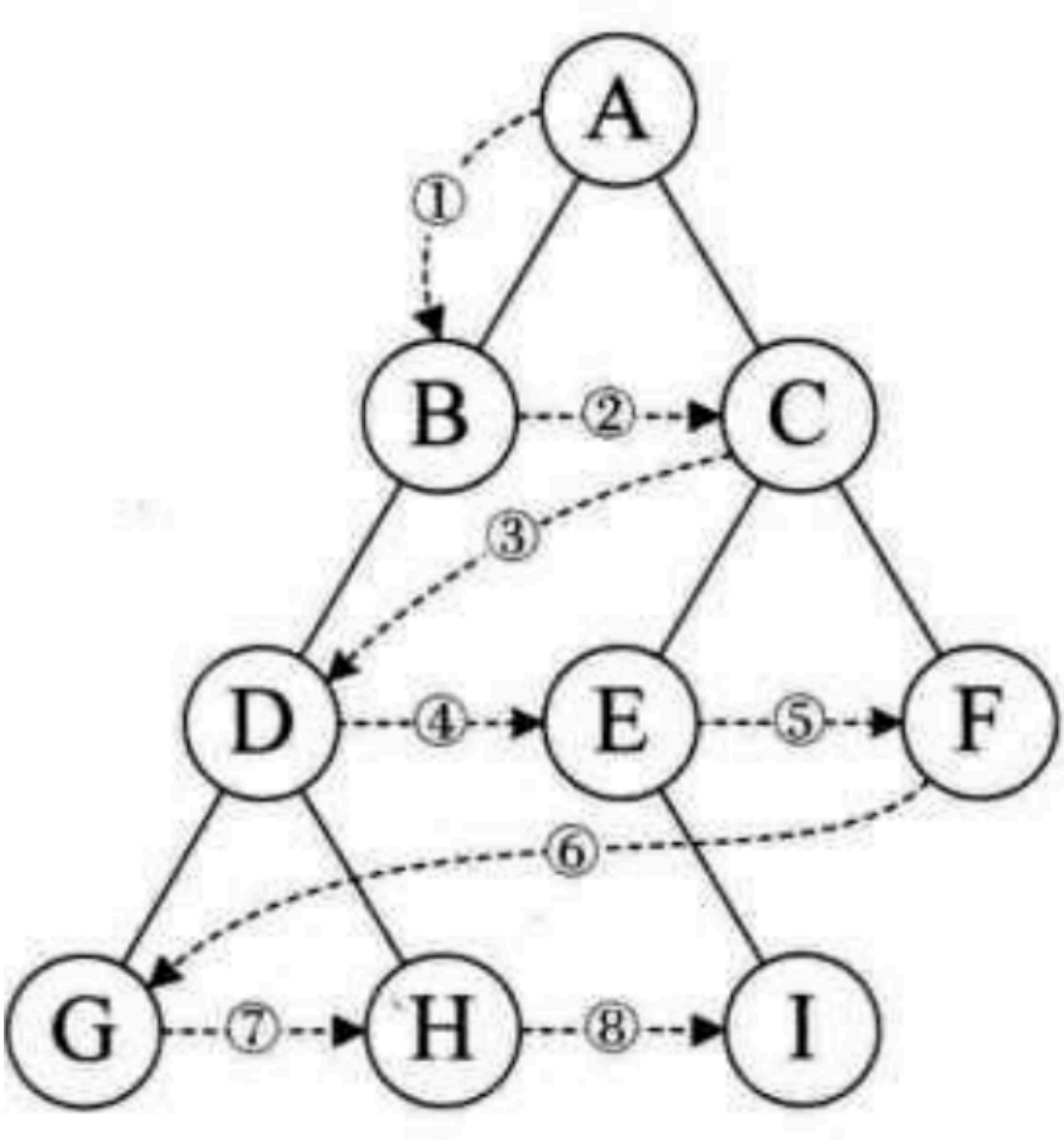


图 6-8-4

- 层序遍历：从树的第一层，也就是根结点开始访问，从上而下逐层遍历，在同一层中，按从左到右的顺序对结点逐个访问



    - 已知前序遍历序列和中序遍历序列，可以为宜确定一颗二叉树
    - 已知后序遍历序列和中序遍历序列，可以唯一确定一颗二叉树
    - 已知前序和后序遍历，不能确定一颗二叉树
- 二叉树的建立
    - 将二叉树中每个结点的空指针引出一个虚结点，其值为一特定值，如“#”。称这种处理后的二叉树为原二叉树的扩展二叉树
    - 利用递归的原理
- 线索二叉树
    - 指向前驱和后继的指针称为线索，加上线索的二叉树表称为线索链表，相应的二叉树称为线索二叉树
    - 对二叉树以某种次序遍历使其变为线索二叉树的过程称作线索化
    - lchild ltag data rtag rchild
- 树、森林、二叉树的转换
    - 树转换为二叉树
        1. 加线。在所有兄弟结点之间加一条连线
        2. 去线。对树中每个结点，只保留它与第一个孩子结点的连线，删除它与其他孩子结点之间的连线

3. 层次调整。以树的根结点为轴心，将整颗树顺时针旋转为一定角度，使之结构层次分明。注意第一个孩子是二叉树结点的左孩子，兄弟转换过来的孩子是结点的右孩子

- 森林转换为二叉树
  1. 把每个树转换为二叉树
  2. 把第一棵二叉树不动，从第二颗二叉树开始，依次把后一颗二叉树的根结点作为前一颗二叉树的根结点的右孩子，用线连接起来。当所有的二叉树连接起来以后就得到了由森林转换来的二叉树
- 二叉树转换为树
  1. 加线。若某结点的左孩子结点存在，则将这个左孩子的n个右孩子结点都作为此结点的孩子。将该结点与这些右孩子结点用线连接起来
  2. 去线。删除原二叉树所有结点与其右孩子结点的连线
  3. 层次调整。使之结构层次分明
- 二叉树转换为森林
  1. 从根结点开始，若右孩子存在，则把与右孩子结点的连线删除，再查看分离后的二叉树，若右孩子存在，则连线删除……，直到所有右孩子连线都删除为止，得到分离的二叉树
  2. 再将每棵分离后的二叉树转换为树即可
- 树的遍历
  - 先根遍历，即先访问树的根结点，然后依次先遍历根的每棵子树。
  - 后根遍历，即先依次后根遍历每棵子树，然后再访问根结点。
- 森林的遍历
  - 前序遍历：先访问森林中第一棵树的根结点，然后再依次先根遍历根的每棵子树，再依次用同样方式遍历除去第一棵树的剩余树构成的森林
  - 后序遍历：先访问森林中第一棵树，后根遍历的方式遍历每棵子树，然后再访问根结点，再依次同样方式遍历除去第一棵树的剩余树构成的森林

- 赫夫曼树及其应用
  - 从树中一个结点到另一个结点之间的分支构成两个结点之间的路径，路径上的分支数目称作路径长度。
  - 树的路径长度就是从树根到每一结点的路径长度之和
  - 带权路径长度WPL最小的二叉树称作赫夫曼树
  - 赫夫曼算法
    1. 根据给定的n个权值{w_1,w_2,……，w_n}构成n颗二叉树的集合F={T_1,T_2,……，T_n}，其中每棵二叉树T_i中只有一个带权为W_i根结点，其左右子树均为空
    2. 在F中选取两颗根结点的权值最小的树作为左右子树构造一颗新的二叉树，且置新的二叉树的根结点的权值为其左右子树上根结点的权值之和
    3. 在F中删除这两颗树，同时将新得到的二叉树加入F中
    4. 重复2和3步骤，直到F只含一棵树为止。这棵树就是赫夫曼树
  - 赫夫曼编码：设需要编码的字符集为{d_1,d_2,……，d_n}，各个字符在电文中出现的次数或频率结合为{w_1,w_2,……，w_n}，以d_1,d_2,……，d_n作为叶子结点，以w_1,w_2,……，w_n作为相应叶子结点的权值来构造一颗赫夫曼树。规定赫夫曼树的左分支代表0，右分支代表1，则从根结点到叶子结点所经过的路径分支组成0和1的序列变为该结点对应的编码
    - 前缀编码：若要设计长短不等的编码，则必须是任一字符的编码都不是另一个字符的编码的前缀

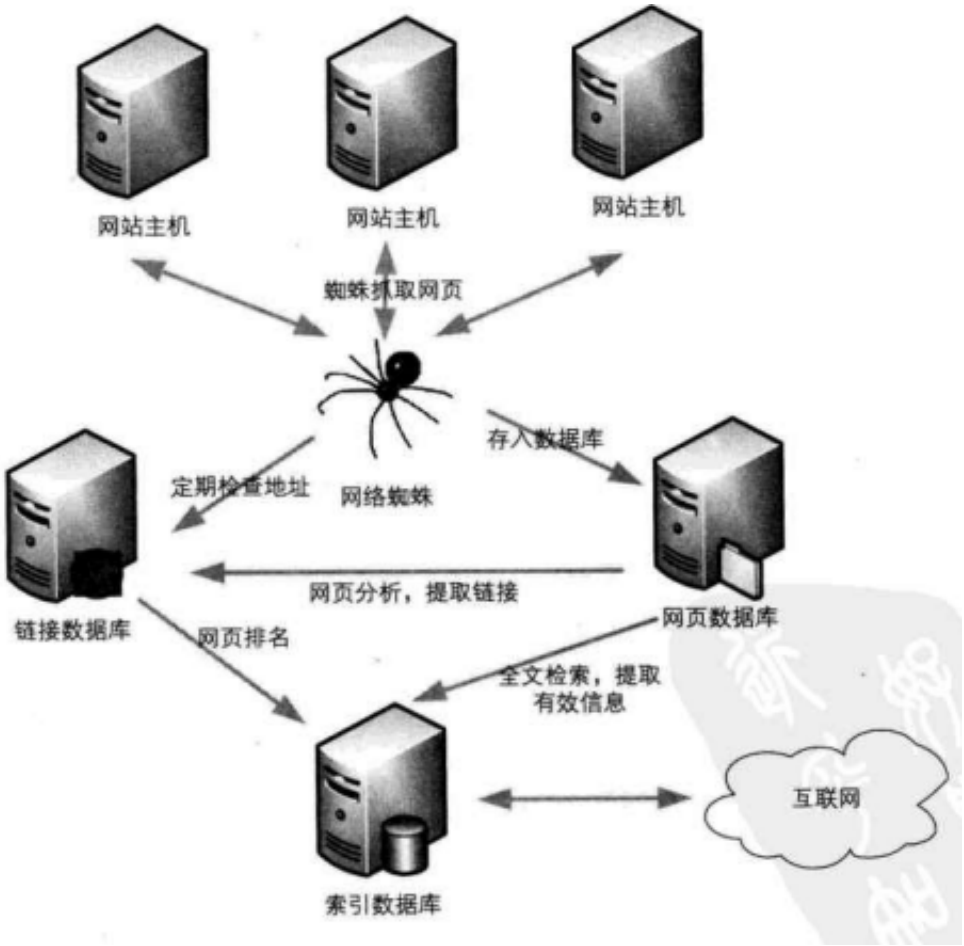
七.图

- 图（graph）：是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为G（V，E），其中，G表示一个图，V是图G中顶点的集合，E是图G中边的集合
  - 线性表中我们把数据元素叫元素，树中将数据元素叫结点，在图中数据元素则称之为顶点（Vertex）
  - 线性表中可以没有数据元素，称为空表；树中可以没有结点，叫做空树；在图结构中，不允许没有顶点
  - 线性表中，相邻的数据元素之间具有线性关系；树结构中，相邻两层的结构具有层次关系；在图中，任意两个顶点之间都可能有关系，顶点之间的逻辑关系用边来表示，边集可以为空
  - 无向边：若顶点v_i到v_j之间的边没有方向，则称这条边为无向边，用无序偶对表示（v_i，v_j）
  - 无向图：图中任意两个顶点之间的边都是无向边
  - 有向边：若顶点v_i到v_j之间的边有方向，则称这条边为有向边，也称为弧。<v_i，v_j>
  - 有向图：图中任意两个顶点之间的边都是有向边
  - 简单图：若不存在顶点到其自身的边，且同一条边不重复出现，则称这样的图为简单图
  - 无向完全图：在无向图中，如果任意两个顶点之间都存在边，含有n个顶点有n*（n-1）/2条边
    - 对于无向图 G=（V，{E}），如果边（v，v'）∈ E，则称顶点v和v'互为邻接点，即v和v'相邻接。边（v，v'）依附于顶点v和v'，或者说（v，v'）与顶点v和v'相关联。顶点v的度是和v相关联的边的数目
  - 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互为相反的两条弧边、，含有n个顶点有你n*（n-1）条边
    - 对于有向图 G=（V，{E}），如果弧<v，v'>∈ E，则称顶点v邻接到v'，顶点v'邻接自顶点v。弧<v，v'>和顶点v和v'相关联。以顶点v为头的弧的数目称为v的入度，记为ID（v）；以顶点v为尾的弧的数目称为v的出度，记为OD（v）；顶点v的度为TD（v）= ID（v）+ OD（v）
  - 稀疏图、稠密图
  - 权：与图的边或弧相关的数
  - 网：带权的图
  - 假设有两个图G=（V，{E}）和G'=（V'，{E'}）,如果V'⊆V且E'⊆E，则称G'为G的子图
  - 路径的长度是路径上的边或弧的数目
- 图的抽象数据类型
- 图的存储结构
  - 邻接矩阵：用两个数组来表示图，一个一维数组存储图中顶点信息，一个二维数组存储图中的边或弧的信息
  - 邻接表：数组与链表相结合的存储方法。图中顶点用一个一维数组存储，每个顶点v_i的所有邻接点构成一个线性表（用单链表存储）
  - 逆邻接表：对每个顶点v_i都建立一个链接为v_i弧头的表
  - 十字链表：把邻接表和逆邻接表结合起来，既容易找到以v_i为尾的弧，也容易找到以v_i为头的弧，因而容易求得顶点的出度和入度
  - 邻接多重表：同一条边在邻接表中用两个结点表示，而在邻接多重表中只有一个结点
  - 边集数组：由两个一维数组构成。一个是存储顶点的信息，另一个是存储边的信息，这个边数组每个数据元素由一条边的起点下标（begin）、终点下标（end）和权（weight）组成
- 图的遍历：从图中某一顶点出发访遍图中其余顶点，且使每一个顶点仅被访问一次的过程

    - 深度优先遍历（DFS）：一个递归的过程，像一棵树的前序遍历。连通图：从图中某个顶点v出发，访问此顶点，然后从v的未被访问的邻接点出发深度优先遍历图，直至图中所有和v有路径相通的顶点都被访问到。非连通图：在先前一个顶点进行一次深度优先遍历后，若图中尚有顶点未被访问，则另选图中一个未被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。
    - 广度优先遍历（BFS）：类似树的层序遍历。
- 最小生成树：构造连通网的最小代价生成树
    - 普里姆算法：时间复杂度为O（n^2）。对于稀疏图有优势
    - 克鲁斯卡尔算法：时间复杂度为O（eloge）。对于稠密图有优势
- 最短路径：非网图没有边上的权值，所谓的最短路径其实就是指两顶点之间经过的边数最少的路径；网图的最短路径是指两顶点之间经过的边上权值之和最少的路径，并且我们称路径上的第一个顶点是源点，最后一个顶点是终点。
    - 迪杰斯特拉算法：时间复杂度O（n^ 3）
    - 弗洛伊德算法：时间复杂度O（n^ 3）
- 拓扑排序：主要是为解决一个工程是否能顺序进行的问题
    - AOV网：在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系，这样的有向图为顶点表示活动的网称为AOV网。AOV网中不能存在回路
    - 设G=（V，E）是一个具有n个顶点的有向图，V中的顶点序列v_1,v_2,……，v_n，满足若从顶点v_i到v_j有一条路径，则在顶点序列中顶点v_i必在顶点v_j之前，则我们称这样的顶点序列为一个拓扑序列。拓扑排序其实就是对一个有向图构造拓扑序列的过程。时间复杂度为O（n+e）
- 关键路径：解决工程完成需要的最短时间问题
    - AOE网：在一个表示工程的带权有向图中，用顶点表示事件，用有向边表示活动，用边上的权值表示活动的持续时间，这种有向图的边表示活动的网，称之为AOE网。把AOE网中没有入边的顶点称为始点或源点，没有出边的顶点称为终点或汇点。
    - 把路径上各个活动所持续的时间之和称为路径长度，从源点到汇点具有最大长度的路径叫关键路径，在关键路径上的活动叫关键活动
    - 关键路径算法：时间复杂度O（n+e）
    - AOE和AOV都是用来对工程建模的，它们之间有很大的不同。主要体现在AOV网是顶点表示活动的网，他只描述活动之间的制约关系，而AOE网是用边表示活动的网，边上的权值表示活动持续的时间，因此AOE网是建立在活动之间制约关系没有矛盾的基础之上，再来分析完成整个工程至少需要多少时间，或者缩短完成工程所需时间应当加快哪些活动等问题

八.查找

- 查找就是根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素或记录



- 查找表是由同一类型的数据元素或记录构成的集合
- 关键字是数据元素中某个数据项的值，用它可以标识一个数据元素，也可以标识一个记录的某个数据项。若关键字可以唯一地标识一个记录，则称此关键字为主关键字；对于那些可以识别多个数据元素或记录的关键字，称为次关键字
- 静态查找表：只作查找操作的查找表
    - 查询某个“特定的”数据元素是否在查找表中
    - 检索某个“特定的”数据元素和各种属性
- 动态查找表：在查找过程中同时插入查找表中不存在的数据元素，或者从查找表中删除已经存在的某个数据元素
    - 查找时插入数据元素
    - 查找时删除数据元素
- 顺序表查找（线性查找）：时间复杂度O(n)
    - 从表中第一个（或最后一个）记录开始，逐个进行记录的关键字和给定值比较，若某个记录的关键字和给定值相等，则查找成功；若直到最后一个（或第一个）记录，其关键字和给定值比较都不等时，则查找不成功
- 有序表查找
    - 折半查找（二分查找）：时间复杂度O(logn)

      前提是线性表中的记录必须是关键码有序，线性表必须采用顺序存储。基本思想：在有序表中，取中间记录作为比较对象，若给定值与中间记录的关键字相等，则查找成功；若给定值小于中间记录的关键字，则在中间记录的左半区继续查找，反之则在右半区继续查找。不断重复上述过程，直到查找成功，或所有查找区域无记录，查找失败为止
    - 插值查找：时间复杂度O(logn)

根据要查找的关键字key与查找表中最大最小的关键字比较后的查找方法，核心在于插值计算公式：

```
mid = low + (key - a[low]) / (a[high] - a[low])(high-low)
```

- 斐波那契查找：时间复杂度O(logn)
  1. 当key=a[mid]时，查找就成功
  2. 当key<a[mid]时，新范围是第low个到第mid-1个，此时范围个数为F[k-1]-1个
  3. 当key>a[mid]时，新范围是第m+1个到第high个，此时范围个数为F[k-2]-1个

- 线性索引查找
  - 索引是为了加快查找速度而设计的一种数据结构，是把一个关键字与它对应的记录相关联的过程。一个索引由若干个索引项构成，每个索引项至少包含关键字和其对应的记录在存储器中的位置等信息。
  - 索引按照结构可以分为：线性索引、树形索引和多级索引。重点讲三种线性索引
  - 稠密索引：讲数据集中的每个记录对应一个索引项，索引项一定是按照关键码有序的排列
  - 分块索引：把数据集的记录分成了若干块，并且这些块需要满足两个条件1.块内无序2.块间有序
  - 倒排索引：记录号表存储具有相同关键字的所有记录的记录号（可以是指向记录的指针或是该记录的主关键字）。根据属性的值来查找记录
- 二叉排序树：它或者是一棵空树，或者是具有以下性质的二叉树
  - 若它的左子树不空，则左子树上所有的节点的值均小于它的跟结构的值
  - 若它的右子树不空，则右子树上所有的节点的值均大于它的跟结构的值
  - 它的左、右子树也分别为二叉排序树
- 平衡二叉树：一种二叉排序树，其中每一个节点的左子树和右子树的高度差至多等于1
- 多路查找树：每一个结点的孩子数可以多以两个，且每一个结点处可以存储多个元素
- B树：一种平衡的多路查找树，结点最大的孩子数目称为B树的阶
- 散列表查找
  - 散列技术是在记录的存储位置和它的关键字之间建立一个确定的对应关系f，使得每个关键字key对应一个存储位置f(key)
  - 采用散列技术将记录存储在一块连续的存储空间中。这块连续存储空间称为散列表或哈希表
  - 散列函数的构造方法：1.计算简单2.散列地址分布均匀
    - 直接定址法
    - 数字分析法
    - 平方取中法
    - 折叠法
    - 除留余数法
    - 随机数法
  - 处理散列冲突的方法
    - 开放定址法
    - 再散列函数法
    - 链地址法
    - 公共溢出区法
  - 散列表查找性能分析：1.散列函数是否均匀2.处理冲突的方法3.散列表的装填因子

九.排序

- 假设含有n个记录的序列为{r_1,r_2,……,r_n},其相应的关键字分别为{k_1,k_2,……,k_n},需确定1,2，……，n的一种排列p_1,p_2,……，p_n,使其相应的关键字满足$k_{p1}≤k_{p2}≤……≤k_{pn}$（非递减或非递增）关系，即使得序列称为一个按关键字有序的序列{r_{p1},r_{p2},……,r_{pn}},这样的操作就称为排序
- 多个关键字的排序最终都可以转化为单个关键字的排序
- 排序的稳定性：假设$k_i=k_j$(1≤i≤n，1≤j≤n，i≠j)，且在排序前的序列中r_i领先于r_j(即i＜j)。如果排序后r_i仍领先于r_j，则称所用的排序方法是稳定的；反之，若可能使得排序后的序列中r_j领先于r_i，则称所用的排序方法是不稳定的
- 内排序：在整个过程中，待排序的所有记录全部被放置在内存中
    - 时间性能
    - 辅助空间
    - 算法复杂性
- 外排序：由于排序的记录个数太多，不能同时放置在内存，整个过程需要内外存之间多次交换数据
- 冒泡排序：一种交换排序。基本思想：两两比较相邻记录的关键字，如果反序则交换，直到没有反序的记录为止。时间复杂度O（n^2）
- 简单选择排序：通过n-i次关键字间的比较，从n-i+1个记录中选出关键字最小的记录，并和第i（1≤i≤n）个记录交换。时间复杂度O（n^2）
- 直接插入排序：将一个记录插入到已经排好序的有序表中，从而得到一个新的、记录数增1的有序表。时间复杂度O（n^2）
- 基本有序：小的关键字基本在前面，大的基本在后面，不大不小的基本在中间
- 希尔排序：将相距某个“增量”的记录组成一个子序列，这样才能保证在子序列内分别进行直接插入排序后得到的结果是基本有序而不是局部有序。不同增量序列的时间复杂度不同，但优于O（n^2）
- 堆排序：（假设利用大顶堆进行排序）基本思想是，将待排序的序列构造成一个大顶堆。此时，整个序列的最大值就是堆顶的根节点，将它移走（就是将其与堆数组的末尾元素交换，末尾就是最大值），然后将剩余n-1个序列重新构造成一个堆，这样就会得到n个元素中的次小值，如此反复执行。时间复杂度O（nlogn）
- 归并排序：假设初始序列含有n个记录，则可以看成是n个有序的子序列，每个子序列的长度为1，然后两两归并，得到[n/2]（[x]表示不小于x的最小整数）个长度为2或1的有序子序列；再两两归并……直到得到一个长度为n的有序序列为止。时间复杂度O（nlogn）
- 快速排序：通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序。时间复杂度O（nlogn）

十. 总结

虽然我是一名前端工程师，但在实际开发过程中也会涉及到算法相关的问题，而算法和数据结构密切相关，数据结构又是计算机专业的基础课程，因此我需要补补课。

这本书通篇风趣幽默，没错，我目前看的书基本都是既基础又轻松的，这样可以加强我阅读的专注度和看完的决心。

看的过程中发现对很多概念和理念有了新的理解，当初读书时候很多专业名词只是死记硬背，在工作后有了一些项目经验，再回过头看对很多东西有一种豁然开朗的感觉。

算法和数据结构是相辅相成的，两者相互交叉，通过阅读《算法图解》、《大话数据结构》，能够对算法为什么那样写有更深的理解，对数据结构的把握和运用也能更深入。

我觉得，前端的深入学习和研究最终离不开后端相关的知识，只有深入了解计算机底层的设计和原理，多考虑实现过程中有哪些可行的方法和其优缺点，才能帮助我们更好地写代码，完成预期的需求。