第一章 绪论

总结：廖兴宇　15116404939　liaoxingyu@nwpu.edu.cn

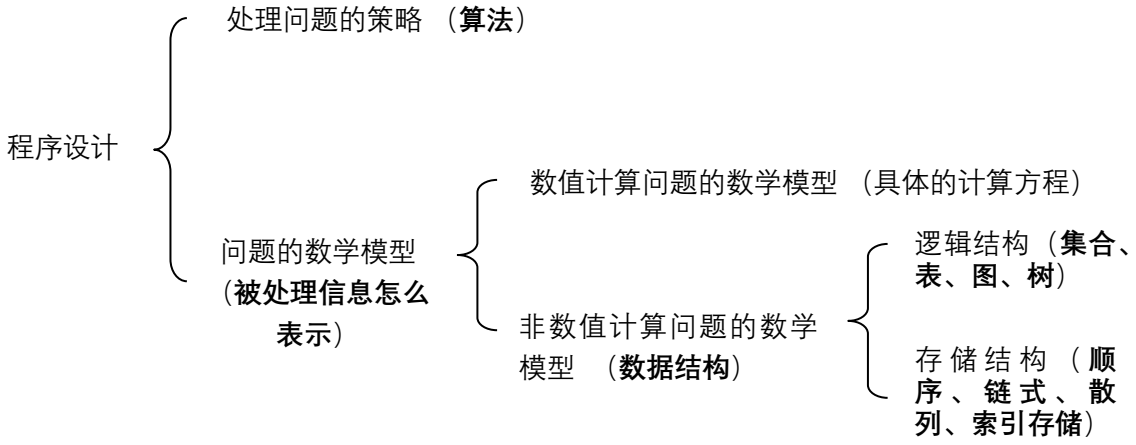
1　为什么要学习《数据结构》课程？其课程地位？

数据结构是一门与程序设计密切相关的课程，美国的一名著名计算机专家、PASCAL语言的创始人 Niklaus E. Wirth （**是一位瑞士计算机科学家，他因在编程语言设计方面的杰出贡献，包括创建了 Pascal 语言，以及在软件工程领域的一系列开创性工作，荣获了1984 年的图灵奖**）在 1976 年出版了一本名为《Algorithm + Data Structures = Programs》（算法+数据结构=程序设计）的专著，在这之后“算法+数据结构=程序设计”这一表述成为了计算机学科的一句名言。

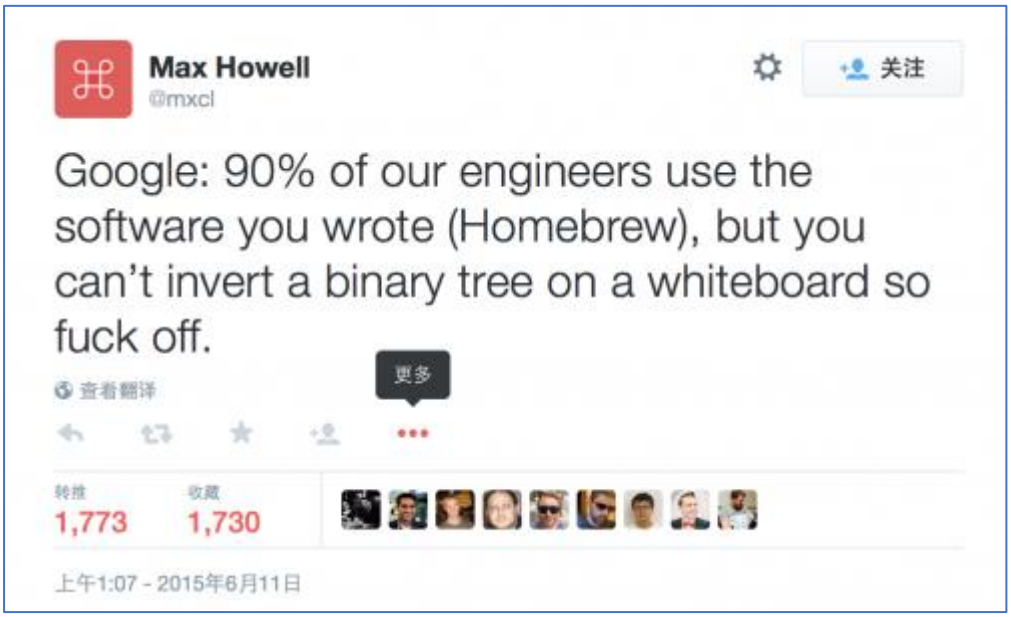
这说明“算法”和“数据结构”是进行程序设计的两大要素，也就是进行程序设计所首先要解决的两个问题。什么是程序设计？**程序设计的任务是：为计算机处理问题（计算机解题）编制一组指令集。所谓的“计算机解题”就是计算机对某类信息进行某一种处理，**这里面包含有两个问题，第一个是：怎么来处理问题，处理问题的策略是什么？**这就是算法要讨论的问题**，第二个是：被处理的信息怎么表示？也就是问题的数学模型是什么？**这就是数据结构要讨论的问题。**

其实任何程序设计问题都包含这两个方面。比如：大家所熟悉的数值计算的程序设计问题。**实例 1**：在建筑设计的结构静力分析计算中，首先需要利用有限元的分析得到一个线性代数方程组，这个方程组就是建筑结构静力分析计算的模型。**实例 2**：要利用计算机来进行全球气象预报，需要求解一组球面坐标系下的一般环流模式方程，这个环流模式方程就是气象预报的数学模型。这两个实例中，计算机主要完成大量的数值计算。所以称之为数值计算的程序设计问题。

然而，当今的计算机往往处理的是非数值计算的问题。比如：信息检索（从海量的数据记录中按照关键字筛选出有用或者感兴趣的数据记录）、模式匹配与识别（语音识别、人脸识别）、商业信息管理系统（进、销、存的记录管理）等。那么这些非数值计算问题中的算法和模型怎么来设计？**非数值计算问题的数学模型的表示和求解方法就是数据结构要研究的内容。**



概括来讲，学习《数据结构》课程的主要意义有：1）数据结构研究的是以何种方式存储、组织数据，以便高效地处理这些数据的一门学科，从而实现非数值计算问题的高效求解。**只有熟悉和精通数据结构的人才能设计和开发出高效的软件系统，数据结构是软件工程师的基本功。所有软件系统的设计都离不开数据结构**。越是大的公司在面试的时候就越是重视数据结构的考察。比如：Homebrew 是一款自由及开放源代码的软件包管理系统，用于简化 macOS 系统上的软件安装过程，其创始人是 Max Howell。Homebrew 被 Google 公司 90%以上的员工所使用，可以说在业界的影响力非常大，但是 Max Howell 却在 Google 公司的面试中被拒，原因是其不能在黑板上写出二叉树的翻转。另外，腾讯，百度，阿里等国内大公司也都非常注重数据结构的考察，大约有 35-60%的题目会与算法、数据结构相关。



2）**数据结构是计算机专业的核心基础课，其处于承前启后的核心课程地位**。数据结构课程既提供编写规范的程序的理论基础和实践指导，又是进一步学习**操作系统、编译原理、软件工程、人工智能、计算机硬件（编码原理、存储装置、存取方法）、计算机网络等**软硬件课程的基础课程。另外，数据结构和算法还是编写软件的核心技术，广泛应用于从系统软件到应用软件、从信息管理系统到移动网络计算、从嵌入式系统到图像图形软件等各种软件开发之中，因此，《数据结构》课程在计算机科学中占有十分重要的地位。熟练掌握数据结构是学好操作系统、软件工程、人工智能等下游课程，以及各种软件开发技术的重要基础。

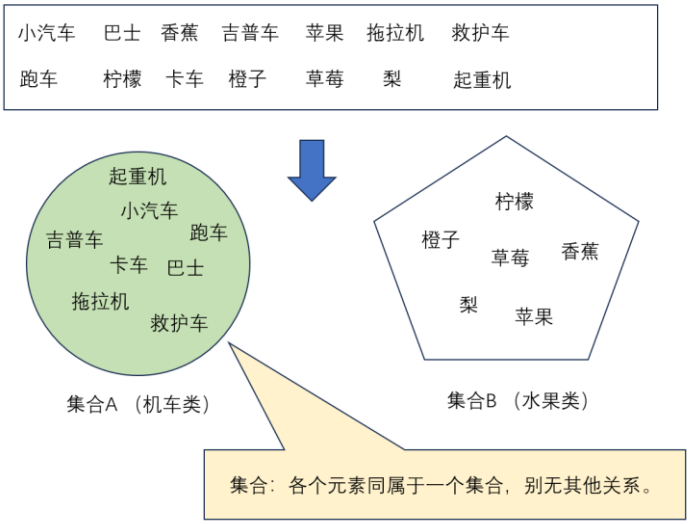
2 非数值计算问题的数学模型（逻辑结构）

描述**非数值计算问题**的数学模型不再是数学方程，而是诸如：集合、表、树、图之类的数据结构。比如，从以下实例可以得出各个问题对应的数据结构。

2.1　集合结构

集合结构：集合结构中各个元素除同属于一个集合外，别无其他关系（集合中任何两个数据元素之间都没有逻辑关系），组织形式松散。

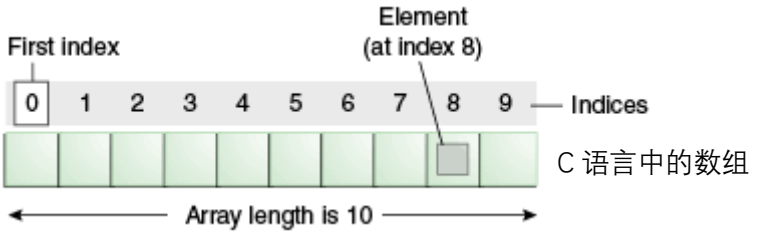
实例 1：将下列名称分为两类。形成的两个类别是典型的集合结构。



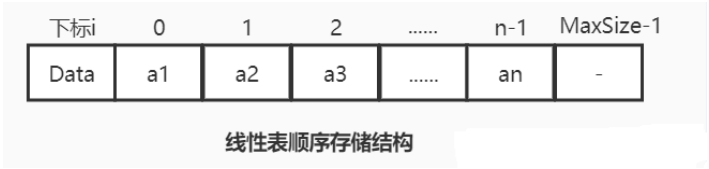
2.2　线性结构（表）

线性结构：线性结构中元素之间存在着“一对一”的线性关系，即除第一个元素以外，其他元素有一个唯一的前驱节点；除最后一个元素以外，都有一个唯一的后继节点。

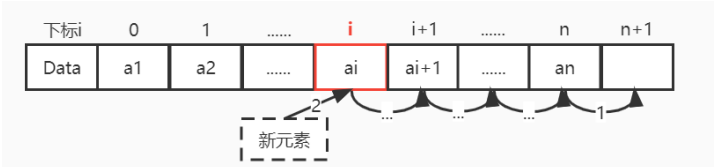
实例 1：C 语言中的数组就是一个典型的线性结构。



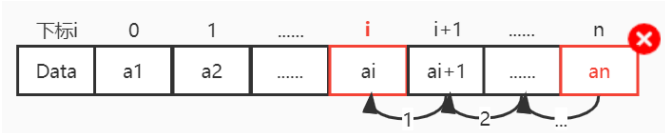
实例 2：在一个数组中分别插入一个元素和删除一个元素？这个问题会涉及到“线性表”这种数据结构的“插入”和“删除”操作。



线性表顺序存储结构



线性表-在下标为i的位置插入元素

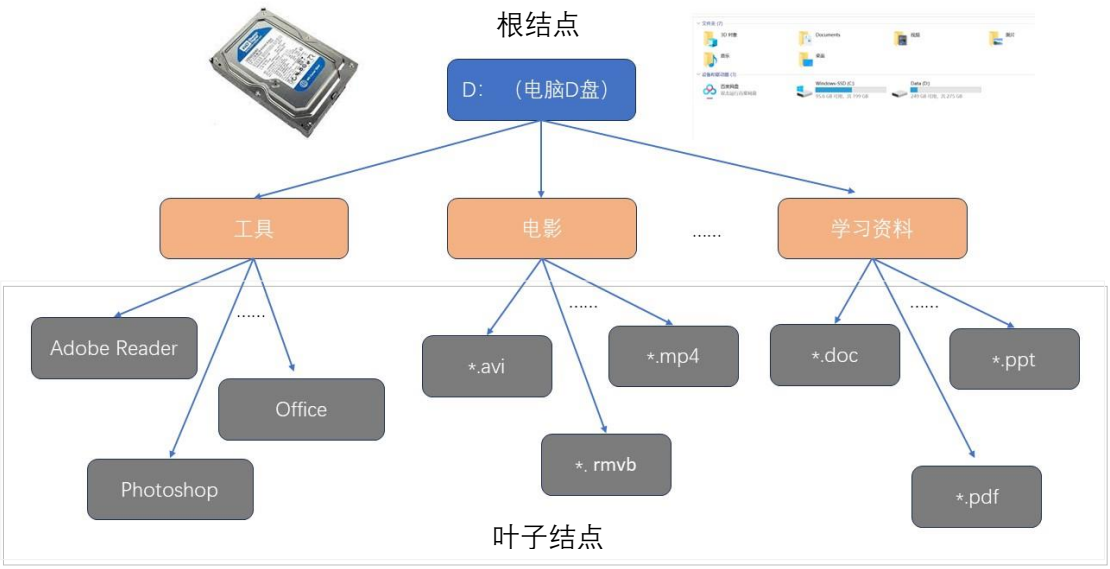


线性表-在下标为i的位置删除元素

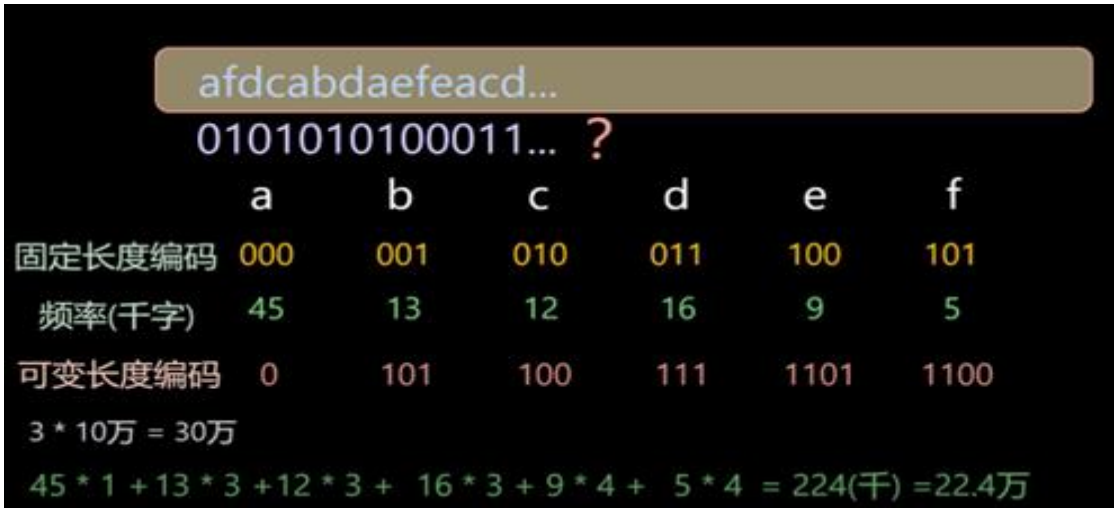
2.3 树型结构

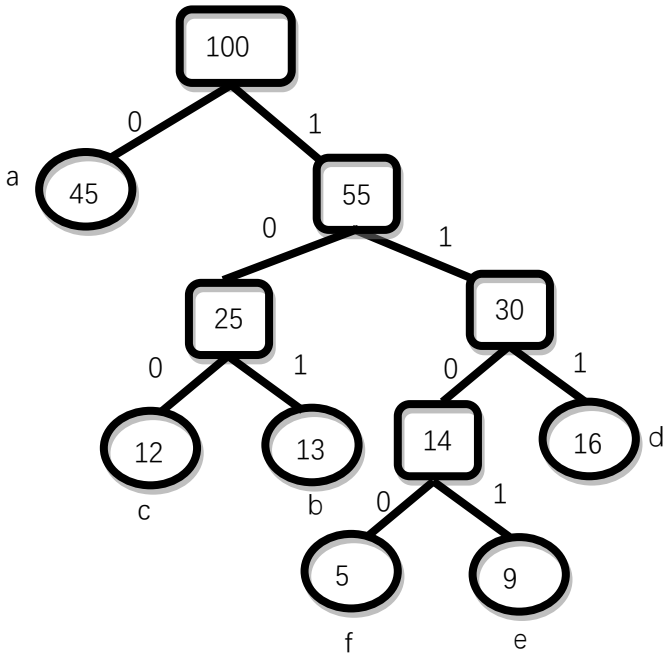
树型结构：树型结构中数据元素之间存在着“一对多”的树形关系（典型的非线性关系），是一类重要的非线性数据结构。 在树形结构中，除根结点没有前驱结点外，其余每个结点有且只有一个前驱结点。 除叶子结点没有后续结点外，其余每个结点的后续节点数可以是一个或多个。

实例 1：个人计算的磁盘系统就是一个典型的树型结构。



实例 2：对长度为 10 万、由 a, b, c, d, e, f 这 6 种字符组成的字符串进行电报编码，要求编码长度最短？这个问题会涉及到“树”这种数据结构。





Huffman 树与 Huffman 编码 （可变编码由上面这棵“树”得来的）

2.4　图型结构

图型结构：是一种复杂的非线性结构，在图型结构中，每个元素都可以有零个或多个前驱，也可以有零个或多个后继，也就是说，图型结构中数据元素之间存在着“多对多”的关系。

实例 1：根据左图中的规则，对同时行驶的结点间画一条连线 来表示它们互相冲突关系。

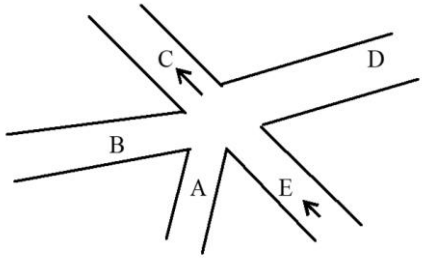


图 1.1　一个交叉路口的模型

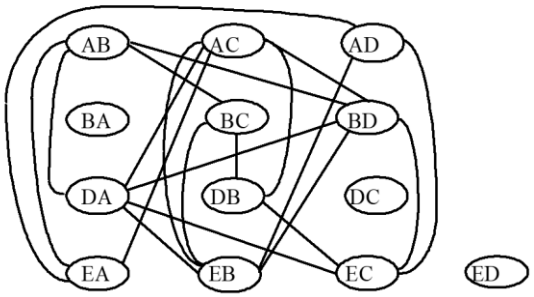
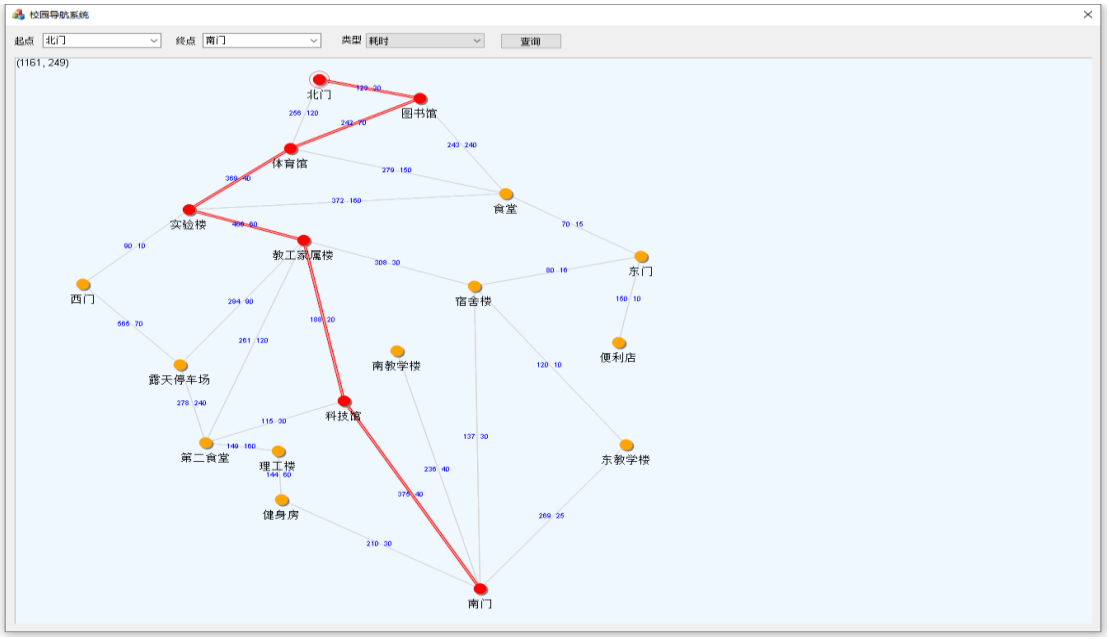


图 1.2　交叉路口的图式模型

根据这个路口的实际情况可以确定 13 个可能通行方向：A→B，A→C，A→D，B→A，B→C，B→D，D→A，D→B，D→C，E→A，E→B，E→C，E→D。可以把 A→B 简写成 AB，用一个结点表示 ，在不能同时行驶的结点间画一条连线 (表示它们互相冲突)，便可以得到如右图所示的表示。这样得到的表示可以称之为“图” 。

实例 2：求两点之间的最短路径？能否设计一套校园导航系统，能够规划出从图中的“北门”到“南门”的最短路径？这个问题会涉及到“图”这种数据结构。



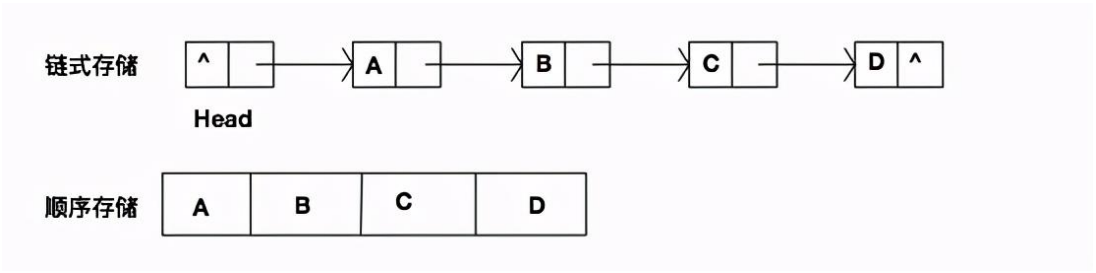
3 非数值计算问题的存储结构

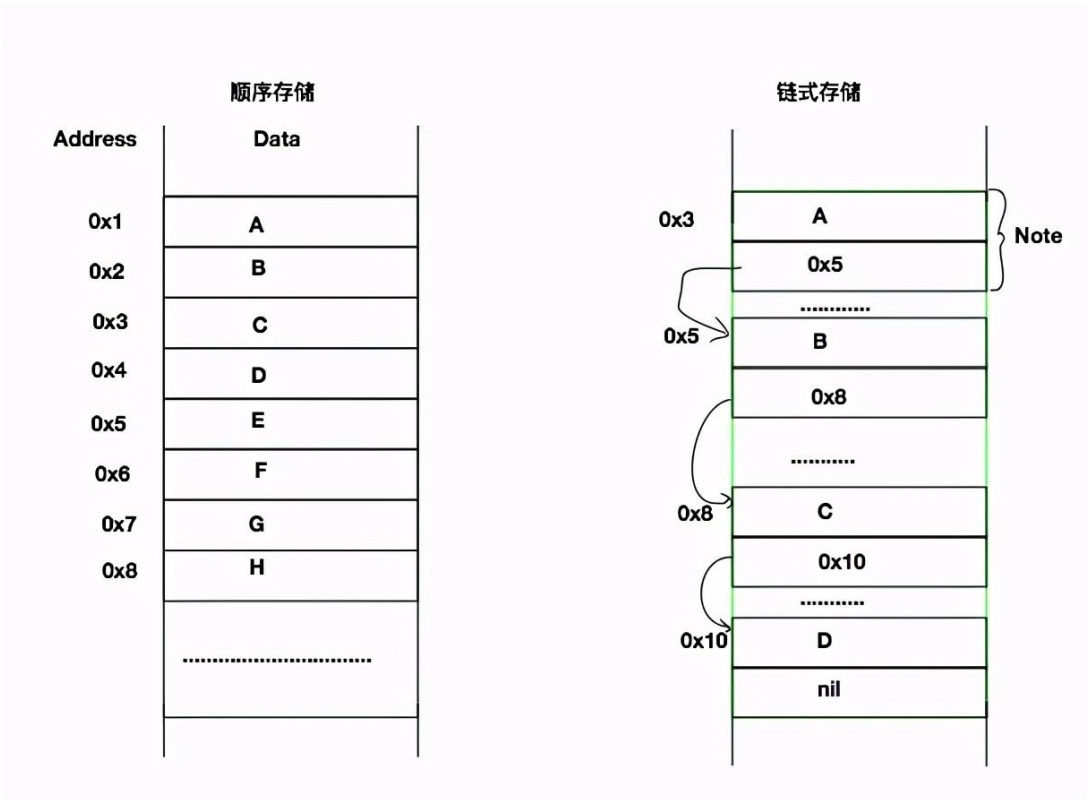
3.1 顺序存储

顺序存储结构是存储结构类型中的一种，该结构是把逻辑上相邻的结点存储在物理位置上相邻的存储单元中，结点之间的逻辑关系由存储单元的邻接关系来体现。

3.2 链式存储

链式储结构的内存地址不一定是连续的，每个数据元素 (结点)的存储包括数据区和指针区两个部分。数据区存放结点本身的数据，指针区存放其后继元素的地址只要知道该线性表的起始地址表中的各个元素就可通过其间的链接关系逐步找到。





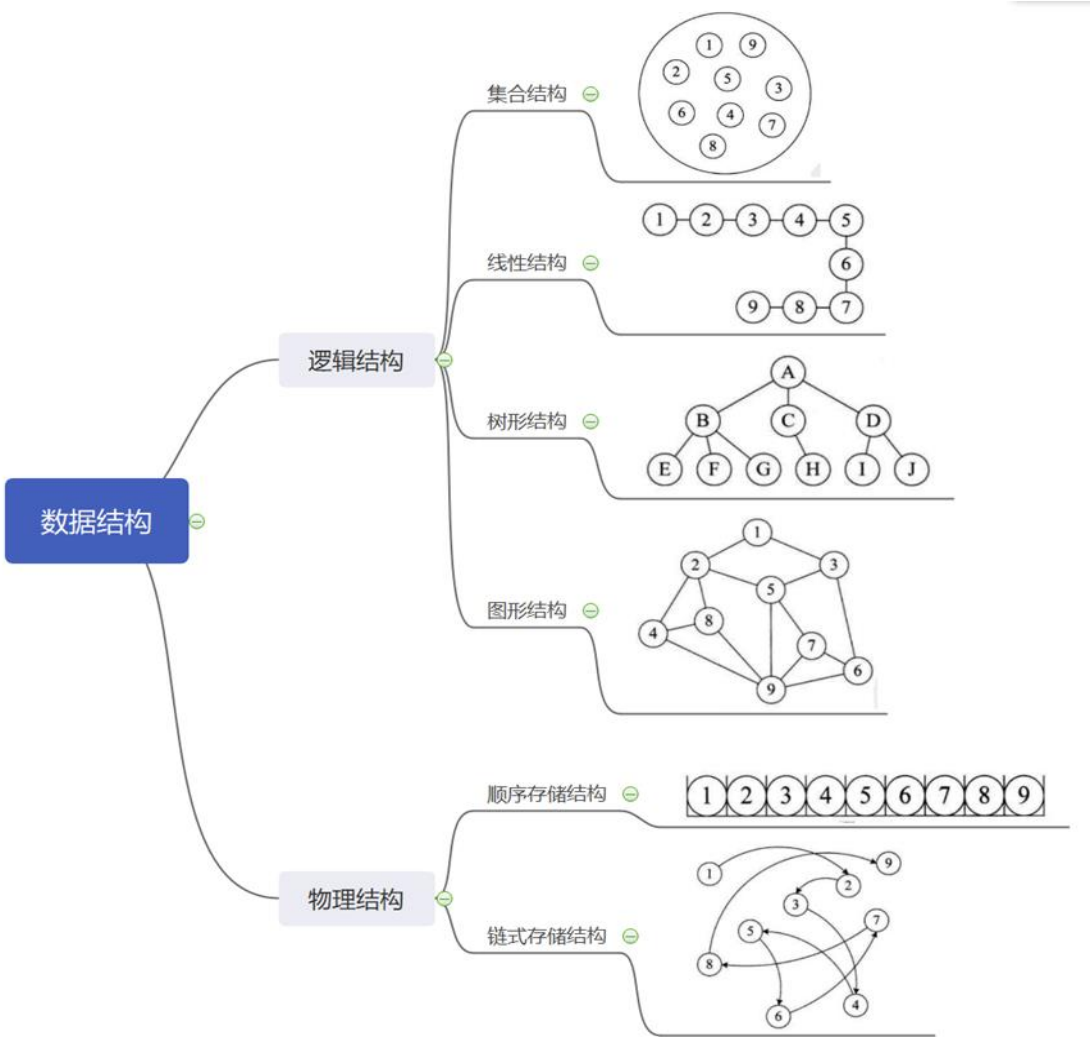
3.3 顺序存储结构和链式存储结构的优缺点

1) **空间上：** 顺序比链式节约空间。是因为链式结构每一个节点都有一个指针存储域。顺序存储密度大（＝1），存储空间利用率高。链表的存储密度 < 1(存储密度 = 结点数据本身所占的存储量/结点结构所占的存储总量)

2) **存储操作上：**顺序支持随机存取，方便操作，顺序表的存储空间是静态分配的,链表的存储空间是动态分配的。

3) **插入和删除上：**链式的要比顺序的方便（因为插入的话顺序表也很方便，问题是顺序表的插入要执行更大的空间复杂度，包括一个从表头索引以及索引后的元素后移，而链表是索引后，插入就完成了）

4) **使用情况：**顺序表适宜于做查找这样的静态操作；链表宜于做插入、删除这样的动态操作。若线性表的长度变化不大，且其主要操作是查找，则采用顺序表；若线性表的长度变化较大，且其主要操作是插入、删除操作，则采用链表。



非数值计算问题的数学模型（数据结构）的逻辑结构和物理结构（存储结构）

4 数据结构中的基本概念和术语

4.1 数据(Data)

在计算机科学中是指所有能输入到计算机中并能被计算机程序处理的符号的总称。

4.2 数据对象（Data Object）

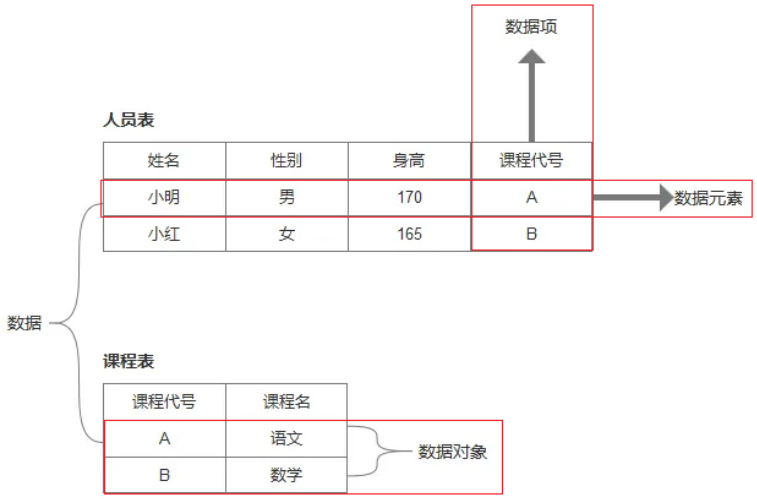
是性质相同的数据元素的集合，是数据的子集。

4.3 数据元素(Data Element)

数据的基本单位（组成数据的、具有一定意义的基本单位），在计算机程序中通常作为一个整体进行考虑和处理，也被称之为记录。

4.4 数据项（Data Item）

一个数据元素可以由若干个数据项(Data Item)组成。数据项是不可分割的最小单位。



4.5 数据结构

是相互之间存在一种或多种特定关系的数据元素的集合。数据结构的形式定义为一个二元组 $Data\ Structure = \{D,S\}$，其中 D 为数据元素的有限集，S 是 D 上关系的有限集。

实例 1：假设一个课程小组由 1 位教师，1~3 名研究生，以及 1~6 名本科生组成。小组成员之间的关系是：教师指导研究生，而每位研究生又指导 1 至 2 名本科生。则可以定义如下的数据结构。

$$Group = \{P, R\}$$

其中：$P = \{T, G_1,\ldots,G_n, S_{11},\ldots,S_{nm}, 1 \le n \le 3, 1 \le m \le 2\}$

\# S_{11}，S_{12}，S_{21}，S_{22}，S_{31}，S_{32}，其中 S_{21} 表示第二为研究生指导的第一名本科生

$R = \{R_1, R_2\}$

$R_1 = \{<T, G_i> \mid 1 \le i \le n, 1 \le n \le 3\}$

$R_2 = \{< G_i, S_{ij} > \mid 1 \le i \le n, 1 \le j \le m, 1 \le n \le 3, 1 \le m \le 2\}$

T 表示教师，G 表示研究生，S 表示本科生。n 为研究生数目，m 为本科生数目。

实例 2：在计算机科学中，复数可取如下定义：复数基本单位是 $i=\sqrt{-1}$，有了这个基本单位，复数空间中的每个数都可以表示为 *a+bi* 的形式。其中，*a* 称之为实部，*b* 称之为虚部。复数定义如下的数据结构。

$$Complex = \{C, R\}$$

其中：C 是包含两个实数的集合$\{c_1, c_2\}$；$R=\{P\}$，而 P 是定义在集合 C 上的一种关系$\{<c_1,c_2>\}$（c_1+c_2i），其中有序对 $<c_1,c_2>$ 表示 c_1 为复数的实部，c_2 为复数的虚部。

4.6 数据类型

数据类型（Data type）是和数据结构密切相关的一个概念。数据类型是一个值的集合和定义在这个值集合上的一组操作的总称。

数据类型可分为两类：1）原子类型；2）结构类型。

数据类型
- **原子类型：** 原子类型的值是不可分割的，例如：C 语言中的基本数据类型（整型，实型、字符型、枚举类型）、指针类型和空类型。
- **结构类型**：结构类型的值是由若干成分按照某种结构组成，因此可以分解，并且它的成分可以是结构的也可以是非结构的。例如：数组的值由若干分量组成，每个分量可以是整数，也可以是数组等。

实例 1：C 语言中整型变量，其值集为某个区间上整数（区间大小依赖于不同的机器），定义在其上的操作有加、减、乘、除和取模等算术运算。

4.7 抽象数据类型

抽象数据类型（Abstract Data Type，简称 ADT）是**数据类型**的延伸，**是指一个数学模型以及定义在该模型上的一组操作**。抽象数据类型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，即不论其内部结构如何变化，只要它的数学特性不变，都不影响其外部的使用。抽象数据类型是一个三元组(D, R, P)，其中 D 为数据对象，R 为 D 上的关系集， P 是对 D 的基本操作集。

```
ADT 抽象数据类型名
{
    数据对象：<数据对象的定义> #D
    数据关系：<数据关系的定义> #R
    基本操作：<基本操作的定义> #P
} ADT 抽象数据类型名
```

抽象数据类型

原子类型： 原子类型变量的值是不可分割的，这类抽象数据类型较少，因为一般情况下，已有的固有数据类型足以满足需求。但是有时也有必要定义新的原子数据类型，例如：位数为 100 的整数。

固定聚合类型：属该类型的变量，其值由确定数目的成分按照某种结构组成。例如：复数是由两个实数按照确定的次序关系确定的。

可变聚合类型： 和固定聚合类型相比，构成可变聚合类型“值”的成分的数目不确定。例如：可定义一个“有序整数序列”的抽象数据类型，其中序列的长度是可变的。

抽象数据类型和**数据类型**实质上是一个概念。例如：各个计算都拥有的“整数”类型是一个抽象数据类型，尽管它们在不同的处理器（广义的处理器，包括计算机的硬件系统和软件系统）上实现的方法可以不同，但是由于其定义的数据特性相同，在用户看来都是相同。因此，“抽象”的意义在于数据类型的数学抽象特性。

通俗来讲，抽象数据类型的定义仅仅取决于它的一组逻辑特性，而其在计算机内部如何表示和实现无论内部的结构怎么变化，只要它的数学特性不变，都不会影响外部的使用。就是说，数据类型和抽象数据类型的相同点是他们都关心逻辑结构。不同点是数据类型既关心逻辑结构又关心物理结构，也就是关注数据结构如何实现，而抽象数据类型只关心抽象特征。

实例 1：

```
ADT Triplet{
   数据对象：D = {e1,e2,e3|e1,e2,e3∈ElemSet(定义了关系运算的某个集合)}
   数据关系：R1 = {<e1,e2>,<e2,e3>}
   基本操作：
      InitTriplet(&T,v1,v2,v3)
      操作结果：构造了三元组 T，元素 e1,e2 和 e3 分别被赋以参数      v1,v2,v3 的值。
      Destroytriplet(&T)
      操作结果：三元组 T 被销毁，  如果初始化失败，则销毁三元组 T。
      Get(T,i,&e)
      初始条件：三元组 T 已经存在，1≤i≤3
      操作结果：用 e 返回 T 的第 i 元的值
      Put(&T,i,e)
      初始条件：三元组 T 已经存在，1≤i≤3
      操作结果：改变 T 的第 i 元的的值为 e
      IsAscending(T)
      初始条件：三元组 T 已经存在
      操作结果：如果 T 的 3 个元素按升序排列，则返回 1，否则返回 0
      IsDescending（T）
```

```
      初始条件：三元组 T 已经存在
      操作结果：如果 T 的 3 个元素按降序排列，则返回 1，否则返回 0
      Max(T,&e)
      初始条件：三元组 T 已经存在
      操作结果：用 e 返回 T 的 3 个元素中的最大值
      Min(T,&e)
      初始条件：三元组 T 已经存在
      操作结果：用 e 返回 T 的 3 个元素中的最小值
}  ADT Triplet
```

实例 2：复数的 ADT

ADT Complex
{
  Data：D = {c1, c2 | c1, c2 ∈ R (R is the set of Real number)}
  数据对象：D = {c1, c2 | c1, c2 ∈ R , R 是实数集}
  Relationship：S = {<c1, c2> (c1 is real part, c2 is imaginary)}
  数据关系：S = {<c1, c2> (c1 是复数的实部, c2 是复数的虚部)}
  Operations：
    Create (&C, x, y)
    构造复数 C，其实部与虚部分别被赋予参数 x 和 y 的值。
    GetReal (C)
    复数 C 已经存在，返回复数 C 的实部值。
    GetImag (C)
    复数 C 已经存在，返回复数 C 的虚部值。
    Add(C1, C2)
    复数 C1 和 C2 已经存在，返回两个复数 C1 与 C2 之和。
    Minus(C1, C2)
    复数 C1 和 C2 已经存在，返回两个复数 C1 与 C2 之差。
    Multiply(C1, C2)
    复数 C1 和 C2 已经存在，返回两个复数 C1 与 C2 之积。
    Divide(C1, C2)
    复数 C1 和 C2 已经存在，返回两个复数 C1 与 C2 之商。
    ...
}ADT Complex

4.8 面向对象技术

面向对象技术强调在软件开发过程中面向客观世界或问题域中的事物，采用人类在认识客观世界的过程中普遍运用的思维方法，直观、自然地描述客观世界中的有关事物。面向对象技术的基本特征主要有抽象性、封装性、继承性和多态性。面向对象的分析方法是利用面向对象的信息建模概念，如实体、关系、属性等，同时运用封装、继承、多态等机制来构造模拟现实系统的方法。

4.9 面向对象与数据结构关系

“数据结构”和“面向对象程序设计”同属于程序设计基础课程，“数据结构”重点描述同类数据元素间的关系及其操作，而“面向对象程序设计”重点描述实体对象的状态与行为。实体对象状态的表达基于数据结构中的元素间的关系描述，数据元素及其相互关系就构成了对实体的的描述，一个数据结构本质就是一个独立的实体对象。

另一方面，根据面向对象的思想，数据结构中的数据元素的类型定义和相关的操作皆被封装在类中，即把基本的数据结构定义成相应的类，对这些类进一步概括抽象，抽取一般特征，形成一种概念上的实体，从而建立一种新的抽象数据结构。

5 算法和算法复杂度分析

5.1 算法的概念

算法是对特定问题求解步骤的一种描述，它是指令的有限序列（它是实现某个计算过程而规定的基本动作的执行序列），其中每一条指令表示一个或多个操作。

5.2 算法的特点

算法具有以下 5 个重要的特点：

1） **有穷性**：一个算法必须总是（对任何合法的输入值）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。
2） **确定性**：算法中的每一条指令必须有确切的含义，读者理解时不会产生二义性。此外，**在任何情况下，算法只能有唯一的一条执行路径**，即对于相同的输入只能得出相同的输出。
3） **可行性**：一个算法是能行的，即**算法中描述的操作都是可以通过已经实现的基本运算执行有限次来实现的**。
4） **输入**：一个算法有**零个或多个输入**，这些输入取自于某个特定的对象的集合。
5） **输出**：一个算法有**一个或多个输出**，这些输出是同输入有着某种特定关系的量。

5.3 算法的设计要求

通常一个“好”的算法应该考虑达到以下的目标：

1）**正确性**：算法应该满足具体问题的需求。a) 程序不含语法错误；b)程序对于几组输入数据能够得出满足规格说明要求的结果;c)程序对于精心选择的典型、苛刻而带刁难性的几组输入数据能够给出满足规格说明要求的结果;d)程序对于一切合法输入数据都能产生满足规格说明要求的结果。通常，d)层是极难达到的，通常以第 c）层意义的正确性作为衡量一个程序是否合格的标准。

2）**可读性**：算法主要是为了人的阅读与交流，其次才是机器执行。可读性好有助于人对算法的理解，晦涩难懂的程序易于隐藏较多错误，难以调试和修改。

3）**健壮性**：当输入非法时，算法也能够适当地做出反应或进行处理，而不是产生莫名其妙的输出结果。

4）**效率与低存储量需求**：通俗讲，效率指的是算法执行的时间。对于同一个问题，如果有多个算法可以解决，那么执行时间最短的算法效率高。存储量需求是指算法执行过程中所需要的最大存储空间。

5.4 算法效率的度量

算法执行时间需要通过依据该算法编制的程序在计算机上运行时所消耗的时间来度量。而度量一个程序的执行时间通常有两种方法：

1）**事后统计**：利用计算机内部的计时器功能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。

缺陷：必须先运行依据算法编制的程序；所得的运行时间统计依赖于计算机的硬件、软件等环境因素，有时容易掩盖算法本身的优劣。

2）**事前分析估计**：一个高级程序语言编写的程序在计算机上运行时所消耗的时间取决于下列因素：

① 依据的算法选用何种策略；
② 问题的规模，例如：求 100 以内还是 1000 以内的素数；
③ 书写程序的语言，对于同一个算法，实现语言的级别越高，执行效率就越低；
④ 编译程序所产生的机器代码的质量；
⑤ 机器执行指令的速度。

同一个算法用不同的语言实现，或者用不同的编译程序进行编译，或者在不同的计算机上运行时，效率均不相同。这表明使用绝对的时间单位衡量算法的效率是不合适的。撇开这些与计算机硬件、软件有关的因素，可以认为一个特定算法“运行工作量”的大小，只依赖于问题的规模（通常用整数量 *n* 来表示），或者说，它是问题规模的函数。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)*，算法的时间量度记作。

$$T（n）= O(f(n))$$

它表示随问题规模 *n* 的增大，算法执行时间的增长率和 *f(n)*的增长率相同，称做算法的**渐近时间复杂度**，简称**时间复杂度**。

被称做问题的**基本操作**的原操作应是其重复执行次数和算法的执行时间成正比的原操作，多数情况下它是最深层次循环内的语句中的原操作，它的执行次数和包含它的语句的频度相同。

5.5 算法复杂度及其举例

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。算法的时间复杂度，也就是算法运行的时间量度，记作：*T(n) = O(f(n))*。它表示随问题规模 *n* 的增大，算法执行时间的增长率和 *f(n)* 的增长率相同，称作算法的渐近时间复杂度，简称为时间复杂度。其中 *f(n)* 是问题规模 *n* 的某个函数。

大写 *O()* 来体现算法时间复杂度的记法，我们称之为大 *O* 记法。一般情况下，随着输入规模 *n* 的增大，*T(n)* 增长最慢的算法为最优算法。

实例 1：

```
int a = 0;
int b = 0;
int c = a + b;
```

分析：时间复杂度为 ***O(1)***。

实例 2：

```
int count = 1;
while(count < n) {
    count = count * 2
}
```

分析：*count = 2, 4, 8, ... , 2^n* 由于每次 *count* 乘以 2 之后，就距离 *n* 更近了一分。 也就是说，有多少个 2 相乘后大于 *n*，则会退出循环。 由 $2^x <= n$ 得到 *x <= logn*。 所以这个循环的时间复杂度为 ***O(logn)***。

实例 3：

```
for (int m = 1; m < n; m++) {
   int i = 1;
   while (i <= n) {
      i = i * 2;
   }
}
```

分析：外层 *for* 循环的执行次数是 *n*, 如实例 2 中的推算，内存 *while* 循环的执行次数为 *logn*, 因此总的时间复杂度为 *O(n*logn)*

实例 4：

```
int i, j, n = 100;
for( i=0; i < n; i++ ){
    for( j=i; j < n; j++ ){
        printf(“I love You!\n”);
    }
}
```

分析：由于当 *i=0* 时，内循环执行了 *n* 次，当 *i=1* 时，内循环则执行 *n-1* 次……当 *i=n-1* 时，内循环执行 1 次，所以总的执行次数应该是：*n*+(*n*-1)+(*n*-2)+…+1 = *n*(*n*+1)/2

n(*n*+1)/2 = $n^2/2+n/2$

用我们推导大 *O* 的攻略，只保留最高项，所以 n/2 这项去掉。第三条，去除与最高项相乘的常数，最终得 ***O(n²)***。

实例 5：二分查找是一种基于分治策略的高效搜索算法。 它利用数据的有序性，每轮缩小一半搜索范围，直至找到目标元素或搜索区间为空为止。 给定一个数组 arr，其元素按从小到大的顺序排列且不重复。

```
private static int binarySearch(int[] arr, int target){
      int left = 0, right = arr.length - 1, mid;
      while (left <= right) {
          mid = (left+ right) / 2;
          if (arr[mid] == target) {
              return mid;
          }
          else if (arr[mid] > target) {
              right = mid - 1;
          }
          else if (arr[mid] < target) {
              left = mid + 1;
          }
      }
      return -1;
}
```

长度为 *n* 的表中查找

第 1 次循环在 *n* / *2* 的范围内查找

第 2 次循环在 （*n* / *2*）/*2* = *n* / 2^2 的范围内查找

……　　……

第 *x* 次循环在 （*n* / 2^x）的范围内查找

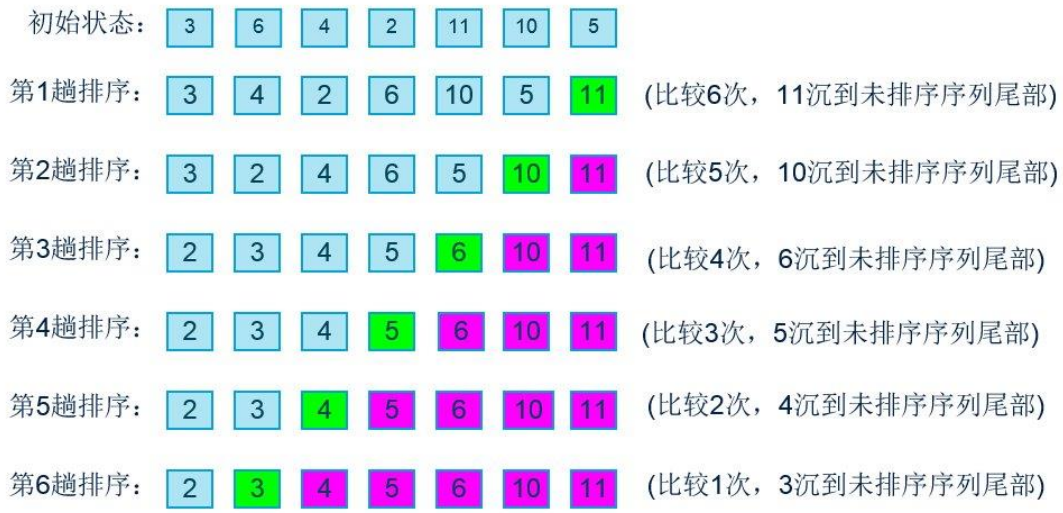
假设最坏的情况下，循环 *x* 次后找到，则：*n* / 2^x =*1* ➡ *x*= $log_2 n$

也就是说循环的基本次数为：$log_2 n$

则时间复杂度为： *O*（$log_2 n$ ）

实例 6：冒泡排序算法是一种基于比较的排序算法，每次冒泡过程，都会有一个数据确定位置。经过 *n* 次冒泡后，就有 *n* 个数据确定了位置。

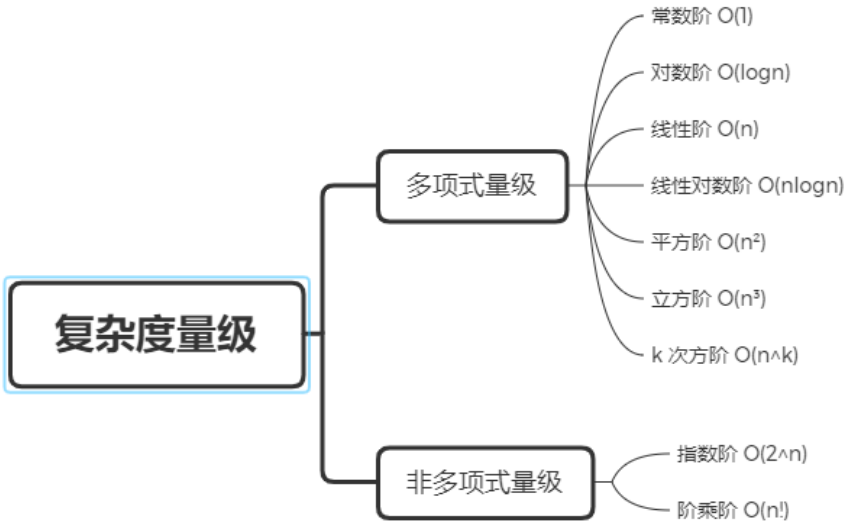
```
public void bubbleSort(int arr[]) {
    for(int i = 0, len = arr.length; i < len - 1; i++) {
        for(int j = 0; j < len - i - 1; j++) {
            if(arr[j + 1] < arr[j])
                swap(arr, j, j + 1);
        }
    }
}
```

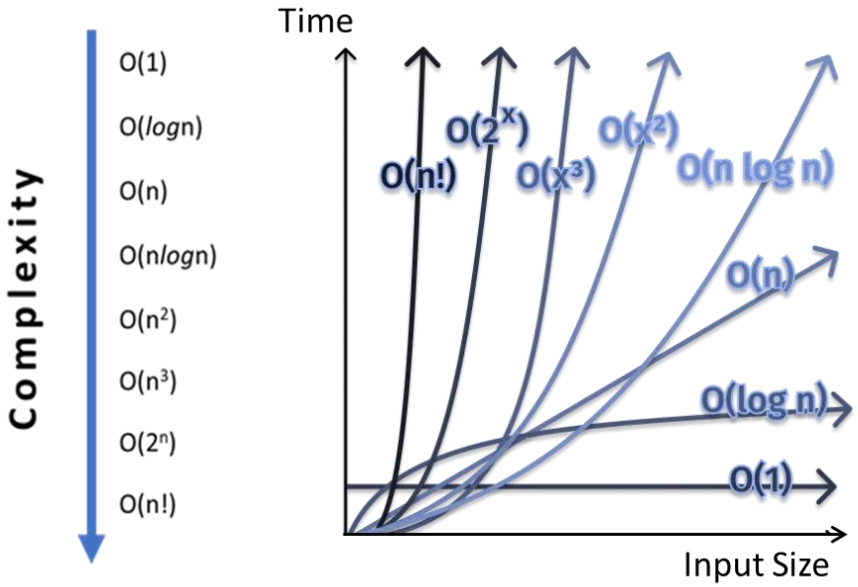


最优的情况下，即待排序列本来就有序时，冒泡排序总的比较次数为 *n-1* 次，也就是说冒泡排序在最好的情况下时间复杂度为 *O(n)*

最差的情况下，即待排序序列为‘逆序’时，冒泡排序总的比较次数为 需要比较 *(n-1) + (n-2) + ...+ 3 + 2 + 1 = n*(n-1)/2* 次比较， 也就是说冒泡排序在最差的情况下时间复杂度为 $O(n^2)$

冒泡排序的最坏时间复杂度为 $O(n^2)$ 冒泡排序的最好时间复杂度为 *O(n)* 综上，因此冒泡排序总的平均时间复杂度为 $O(n^2)$。





$$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$

常數 < 對數 < 線性 < 多項式 < 指數項

5.6　空间复杂度及其举例

空间复杂度 (Space Complexity) 是对一个算法在运行过程中临时占用存储空间大小的量度。算法在运行过程中使用的内存空间主要包括以下几种。**1）输入空间：**用于存储算法的输入数据。**2）暂存空间：**用于存储算法在运行过程中的变量、对象、函数上下文等数据。**3）输出空间：**用于存储算法的输出数据。

一般情况下，空间复杂度的统计范围是**“暂存空间”**加上**“输出空间”**。暂存空间可以进一步划分为三个部分。**1）暂存数据：**用于保存算法运行过程中的各种常量、变量、对象等。**2）栈帧空间：**用于保存调用函数的上下文数据。系统在每次调用函数时都会在栈顶部创建一个栈帧，函数返回后，栈帧空间会被释放。**3）指令空间：**用于保存编译后的程序指令，在实际统计中通常忽略不计。在分析一段程序的空间复杂度时，我们通常统计**暂存数据**、**栈帧空间**和**输出数据**三部分。空间复杂度比较常用的有：*O(1)*、*O(n)*、*O(n²)*，如果申请的是有限个数（常量）的变量，空间复杂度为 *O(1)*。如果申请的是一维数组，队列或者链表等，那么空间复杂度为 *O(n)*。如果申请的是二维数组，那么空间复杂度为 *O(n²)*。

实例 1：

```
int i = 1;
int j = 2;
++i;
j++;
int m = i + j;
```

分析：代码中的 i，j，m 所分配的空间都不随着处理数据量变化，因此它的空间复杂度 *S(n) = O(1)*。

实例 2：

```
int [] m = new int[n]
for (i=1; i<=n; ++i){
   j = i;
   j++;
}
```

分析：空间复杂度为 ***O(n)***。这段代码中，第一行 new 了一个数组出来，这个数据占用的大小为 *n*，这段代码的 2-5 行，虽然有循环，但没有再分配新的空间，因此，这段代码的空间复杂度主要看第一行即可，即 *S(n) = O(n)*。

实例 3：冒泡排序空间复杂度

```
void bubbleSort(int arr[], int size){
    for (int i = 0; i < size - 1; i++){
        for (int j = 0; j < size - i - 1; j++){
            if (arr[j] > arr[j + 1]){
                // 交换相邻元素
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

分析：冒泡排序是一种原地排序算法，实际需要的空间就是每次选出两个数进行比较时申请变量所占的空间，并且这个空间是可以复用的，所以是一个常数，因此记为 O(1)。

一些常用数据结构及其操作的时间复杂度情况统计如下图所示。

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

一些常用排序算法的时间复杂度情况统计如下图所示。

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)