数据结构与算法

Data Structures and Algorithms

主讲人：段友祥

2019年3月

诚信保证

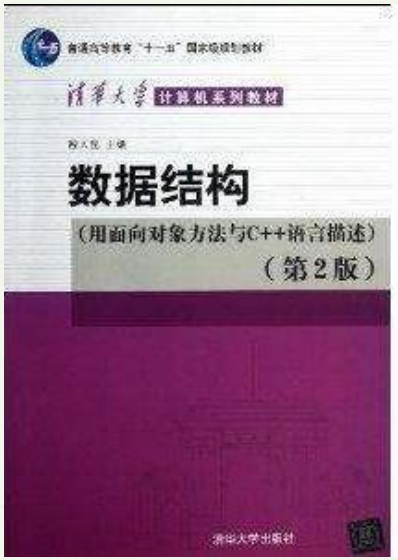
为了培养同学们诚实守信的做人品格，为了保证课程学习的质量，希望同学们做到以下诚信承诺：

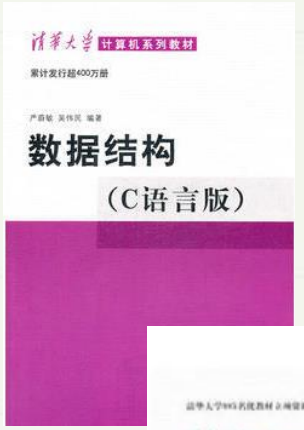
“我郑重承诺：保证在课程学习过程中不弄虚作假（包括上课、作业、实习、考试等等）。”

教材及参考书

- **教材：**

  数据结构（用面向对象方法与C++语言描述）
  - 殷人坤 主编
  - 清华大学出版社

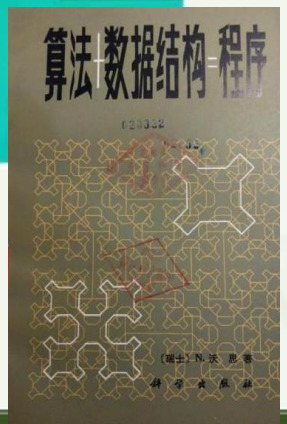




- **参考书：**

(1) 数据结构（C语言版）第2版
  - 严蔚敏，吴伟民编著
  - 清华大学出版社

(2) 数据结构（C++语言版）第3版
  - 邓俊辉 编著
  - 清华大学出版社



(3) 算法＋数据结构＝程序
  - Niklaus Wirth（瑞士）著
  - 科学出版社

★ 更多参考书见课程教学辅助网站

课 程 简 介

- **地位：**数据结构与算法是IT类专业的核心专业基础课程；
- **类型：**必修，理论+实践（50%：50%）；
- **目的：**培养和提高利用计算机解决复杂问题的能力（数据如何更好地存储，如何更有效地处理）——软件开发能力；
- **前导课：**数学（数学分析、高等代数、概率统计等）
  计算机概论， 程序设计 ， 离散数学 等；
- **后续课：**操作系统，数据库原理 ， 编译原理 ， 软件工程等；

★ 更详细的课程信息见课程教学辅助网站

授课内容组织

数据结构与算法基础
(第1章)

→ 为什么学？
如何学？
基本概念（逻辑结构、存储结构、算法、 ADT …）

现实世界数据及特性、操作
（第2-6, 8章）

→ 线性数据结构——线性表
层次数据结构——树
网状数据结构——图

→ 逻辑特性
ADT (模型)定义
ADT (模型)实现

现实世界数据最常见的处理
（第7、9章）

→ 搜索
排序

→ 处理的对象
算法的基本思想
算法的步骤
算法的评价

★ 更详细的内容组织见课程网站

程序设计能力评估测试

- **目的：**

  评估程序设计掌握情况，特别是复杂数据类型的使用，程序的调试、排错等。

- **要求：**

  实事求是地独立完成，真实反映个人的程序设计能力和相关知识掌握的情况，以更好地安排本课程后面的教学和学习过程。在规定时间内提交。

- **测试题目：**

  登录课程辅助网站

  ——课后练习

  ——程序设计能力评估 1-4 题

第一章 绪论

内容提要：

- 数据结构和算法的重要性
- 数据结构的基本概念
- 算法的基础知识
- 抽象数据类型（ADT）

1.1 问题求解

什么是计算机科学？

- 计算机科学是关于计算机（硬件系统）的
- 计算机科学是关于软件（程序）的
- 计算机科学是关于数据（数据表示）的
- 计算机科学是关于算法（数据变换）的
- …

我个人的认识：

—— 研究如何 ***利用*** 计算机 ***求解*** 问题的。

1.1 问题求解

简单说，计算机学科是研究**机器**如何进行信息表示、信息处理、信息传输的科学。

- 信息表示：
  - 存储器
  - 编码和表示
  - 效率
- 信息处理：
  - 处理器
  - 算法
  - 效率
- 信息传输：
  - 信道（介质）
  - 协议、编码
  - 可靠、安全
  - 效率

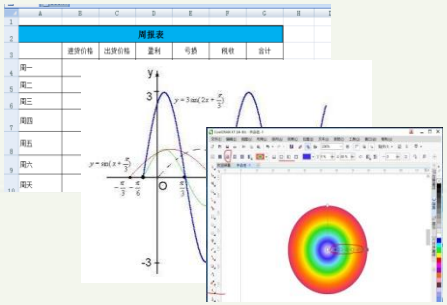
计算机系统=硬件+软件

1.1 问题求解

现实世界的问题

计算机系统

问题的解答



问题的描述包括：

- 要处理的数据
- 期望的结果（隐含着对数据实施的操作–运算）

问题处理软件（程序）：

- 数据在计算机中的存储
- 某种语言描述的算法（程序）

问题处理结果：

- 数字
- 表格
- 图形图像
- ……

1.1 问题求解

- 利用计算机问题求解的基本步骤

- 分析：
- 设计：如何在计算机中存储（表示）问题的数据
  如何处理数据（以期得到正确结果）；
- 编码：
- 测试+维护：

因此，问题的数据在计算机中的表示及处理是利用计算机求解问题的关键！

1.1 问题求解

那么，你的程序里都写的是什么？

- 数据存储的描述，在程序中表现为什么？

  类型和变量

- 数据处理的描述，在程序中表现为什么？

  表达式和流程控制操作（语句）

1.1 问题求解

- **计算机求解问题的本质：**

  把特定领域中的特定问题的求解过程转换为计算机可执行的程序。

- **建模和实现：**

  以计算机为计算模型，对问题进行抽象建模，然后使用工具表示和实现模型。

- **抽象建模：**

  数据结构
  问题求解的基础要素

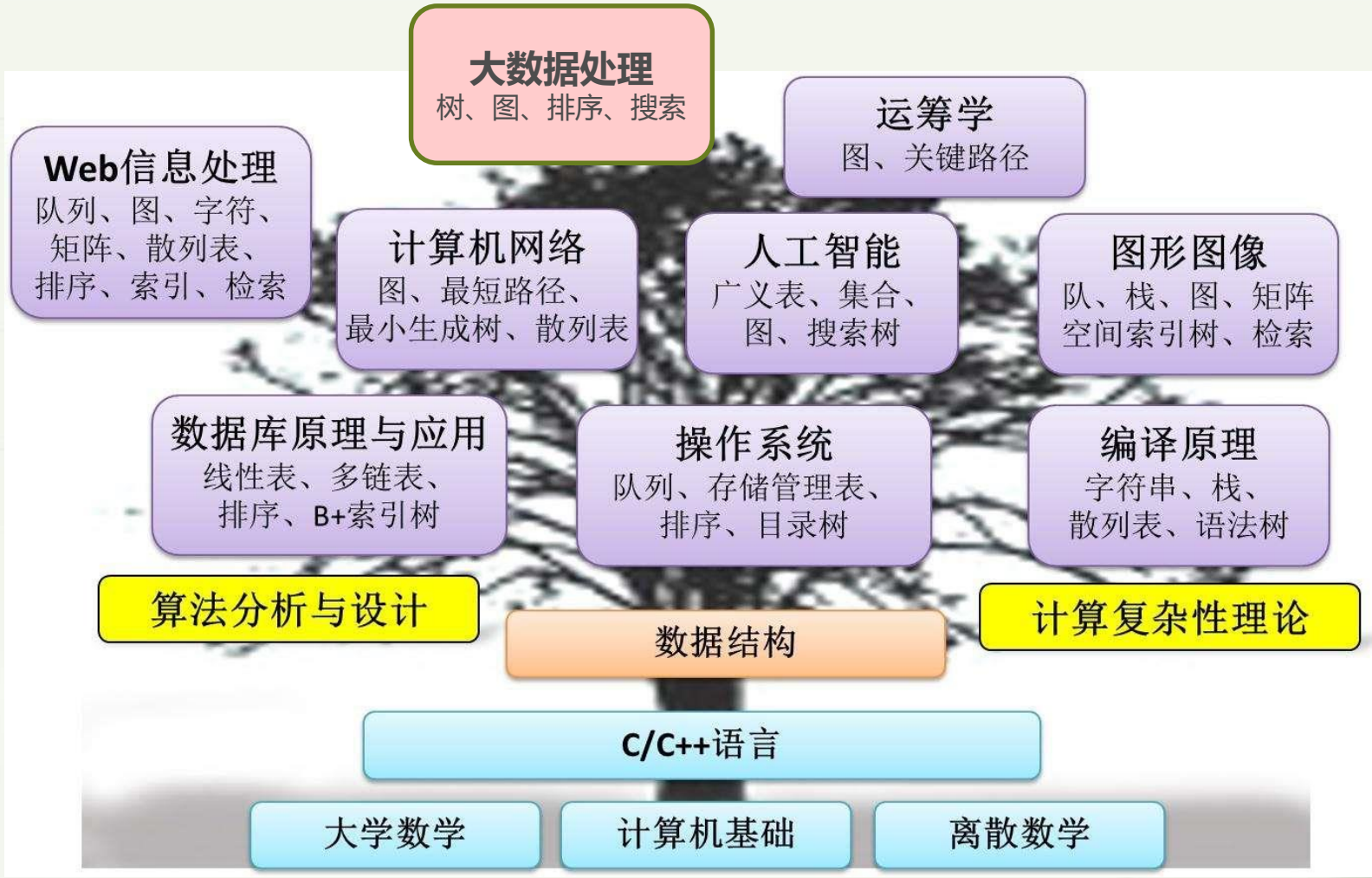
  ＋

  算法设计
  问题求解的关键要素

- **表示和实现：**

  程序设计语言是问题求解过程实现的基本工具。

1.1 问题求解


大数据处理
树、图、排序、搜索
运筹学
图、关键路径
Web信息处理
队列、图、字符、
矩阵、散列表、
排序、索引、检索
计算机网络
图、最短路径、
最小生成树、散列表
人工智能
广义表、集合、
图、搜索树
图形图像
队、栈、图、矩阵
空间索引树、检索
数据库原理与应用
线性表、多链表、
排序、B+索引树
操作系统
队列、存储管理表、
排序、目录树
编译原理
字符串、栈、
散列表、语法树
算法分析与设计
数据结构
计算复杂性理论
C/C++语言
大学数学
计算机基础
离散数学

1.1 问题求解


问题求解
问题
数据操作
对象
子问题
子问题
事物
数据操作
子问题
实体
数据操作
Devices
Locations
Application Groups
Service Sectors
Buildings
Industrial
Commercial/ Institutional
Consumer & Home
Healthcare & Life Science
Transportation
Retail
IT & Networks
Public
Enterprise
Surveillance Equipment
Emergency Services
Specialty
Hospitality
Stores
Non-Vehicular
Vehicles
Transit Systems
Infrastructure
Awareness & Safety
Convenience & Entertainment
Care
In Vitro/Phone
Research
Resource Automation
Connectivity/ Control
Distribution

1.1 问题求解

著名计算机科学家 N. Wirth 提出了经典命题：

数据结构 + 算法 = 程序

Algorithms + Data Structures = Programs

被认为和爱因斯坦的经典命题一样：

$$E=MC^2$$

1.1 问题求解

计算机技术的发展改变了人类的各种活动，促进了社会飞速发展，同时也激起了人的欲望需求，即人类的计算机应用需求越来越大。

这样，人的“贪欲”就对IT技术不断提出新的挑战。因为无论怎样，CPU再快也不会无限快，存储再大再便宜也不会无限大和免费！ 如何应对这些挑战？

- 应对策略之一，发展硬件技术：
  - 容量更大、更便宜、访问速度更快的存储；
  - 速度更快、处理能力更强的CPU；
  - 安全可靠、快速的传输介质；
- 应对策略之二，发展软件技术：
  - 应用的数据如何存储、如何处理（数据结构+算法）
  - 如何构建应用（软件工程）
  - 如何运行应用（系统工程）

1.2 数据结构的产生

随着计算机技术的飞速发展，计算机应用也已经渗透到了社会的各个领域。我们发现有了很大的一些变化：

- ♣ 计算机由最初的单一科学计算到几乎无所不能；
- ♣ 加工处理的对象由数值型变为数值型和非数值型；
- ♣ 处理的数据量由小变为大、巨大（海量存储、计算）；
- ♣ 数据之间的关系由简单变复杂、很复杂；

如何应对？

1.2 数据结构的产生

在硬件条件一定的条件下，我们能不能通过研究问题本身，发现其内在的一些特点和规律，并让计算机利用它们来更好地解决问题？因为，很显然：

将一“大堆杂乱无章”的数据交给计算机处理是很不明智的，结果是加工处理的效率会非常低，有时甚至根本无法进行。

举例：图书管理

1.2 数据结构的产生

信息的表示和组织形式直接影响到数据处理的效率！

于是：

人们开始考虑通过研究、分析问题数据本身的特点，从而利用这些特点提高数据表示和处理的效率。

——*这样就产生了“数据结构”*

需要注意：

数据结构是随着计算机应用的深入而产生的（或者说求解简单问题用不到数据结构）。 ***Why?***

数据结构是计算机高效解决问题的基石！

1.2 数据结构的产生

从现在开始，你想要用计算机求解一个问题，就必须首先想：问题的数据有哪些？它们之间是什么关系？如何表示和存储？如何操作（处理）？

下面看几个例子：

例1. 问题：图书管理，完成书目的管理

数据：？

关系：？

操作：？

1.2 数据结构的产生

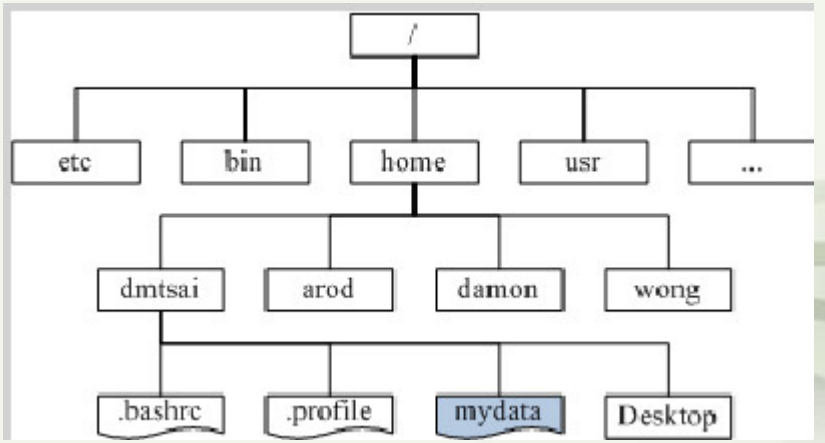
例2. 问题：计算机的文件管理

数据：文件（文件夹）

关系：层次

操作：文件操作

教材P2，例3 UNIX文件系统



1.2 数据结构的产生

例3. 问题：人机对弈，即人与计算机下棋

数据：？

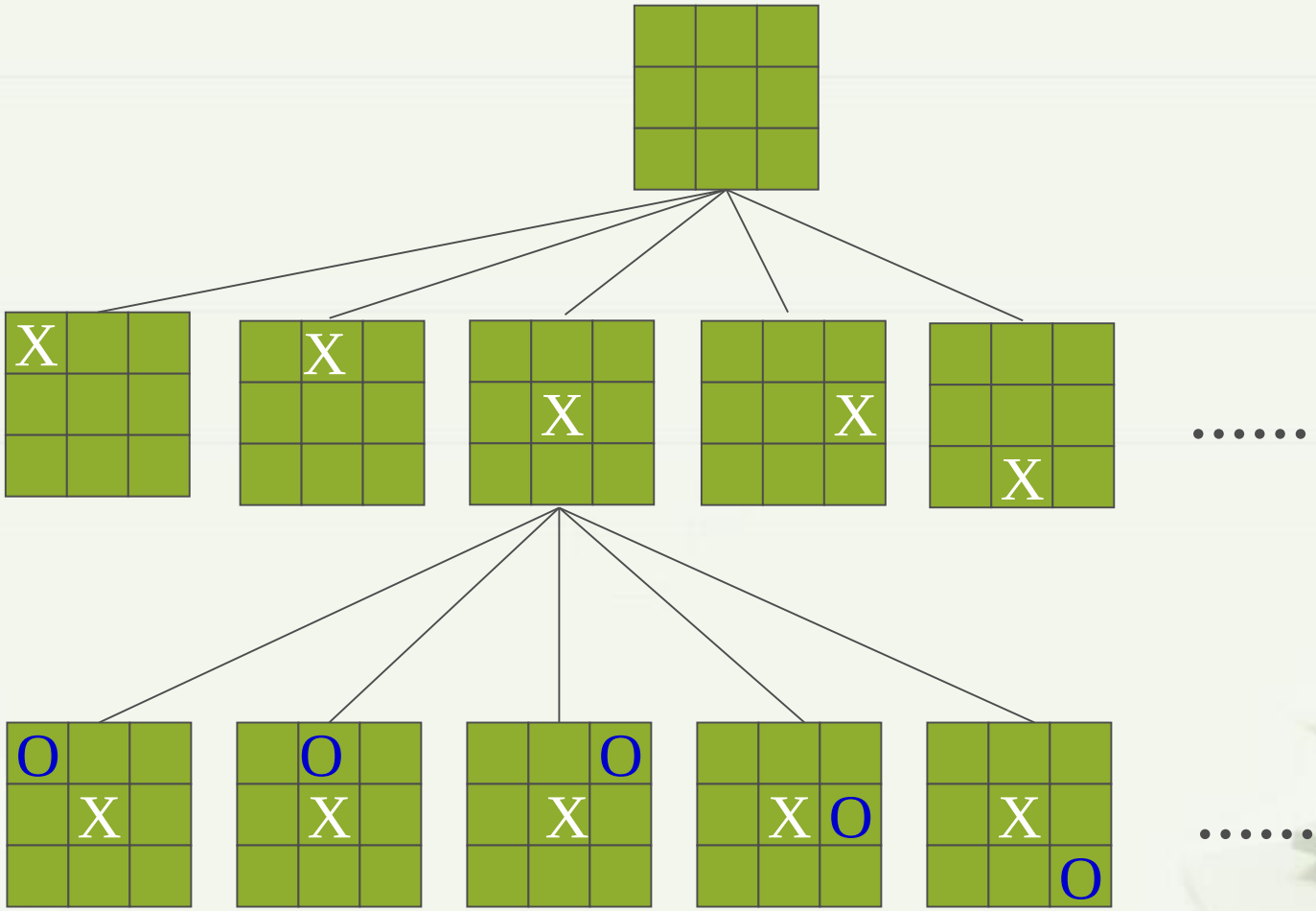
关系：？

操作：？

数据量大还是小？

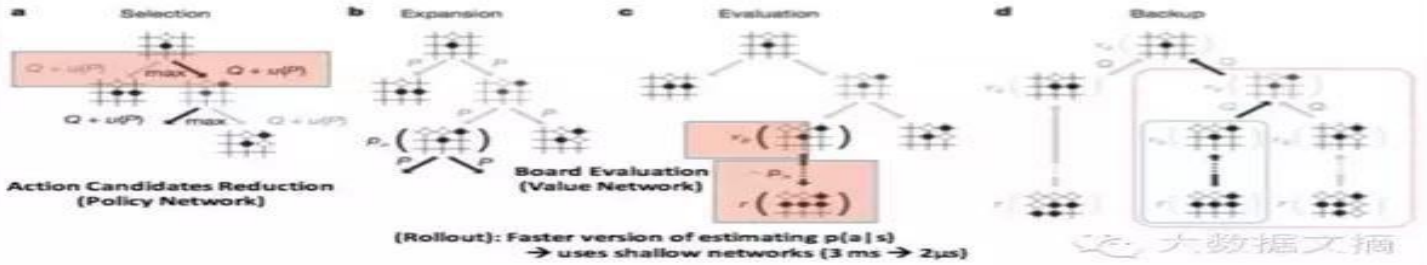
1.2 数据结构的产生

“井”字棋



1.2 数据结构的产生

蒙地卡罗搜索树



深度学习
（Deep Learning）

策略网络
（PolicyNetwork）

预测对手最有可能的落子位置。

价值网络
（Value Network）

计算在目前局势的状况下，每个落子位置的“最后” 胜率.

1.2 数据结构的产生

例4. 好友管理

数据：？

关系：？

操作：？

例5. 道路交通管理

数据：？

关系：？

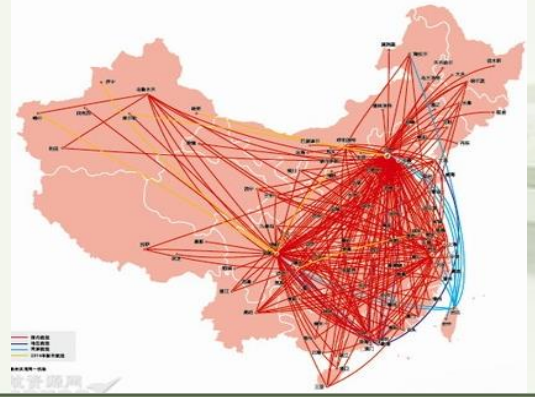
操作：？

例6. 航空航线管理

数据：？

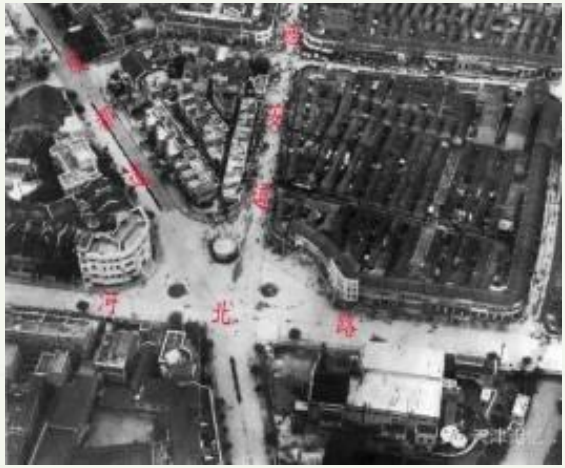
关系：？

操作：？



1.2 数据结构的产生

例7. 多叉路口的交通灯管理，即在多叉路口应设置几种颜色的交通灯，以保证交通畅通。



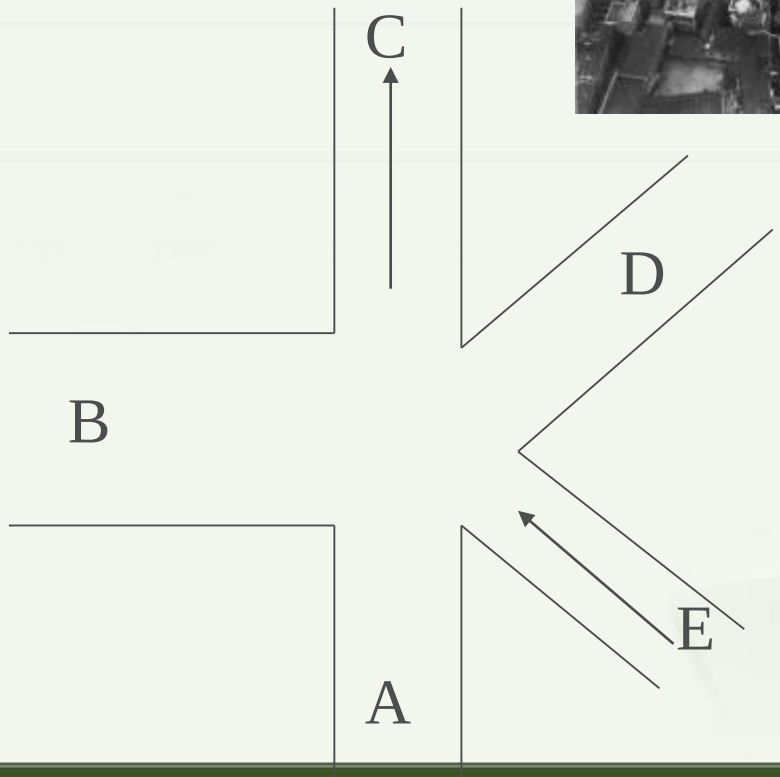
数据：？

关系：？

操作：？

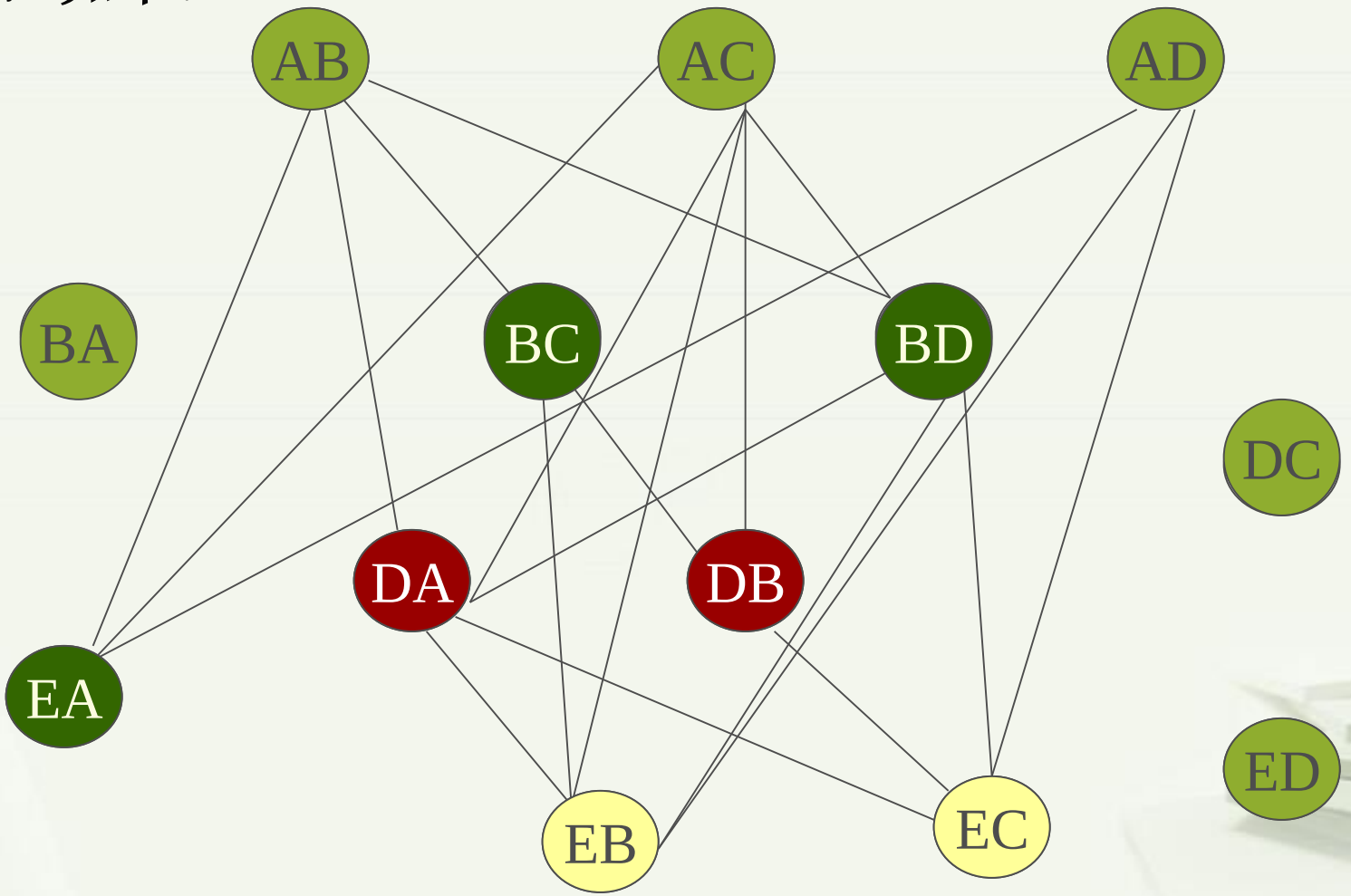
共13条路：

A——>B	B——>A
A——>C	B——>C
A——>D	B——>D
D——>A	E——>A
D——>B	E——>B
D——>C	E——>C
	E——>D



1.2 数据结构的产生

以道路为顶点，道路之间相互矛盾的用连线连接起来，形成一个图，如下：



可以用四种颜色着色，因此需设四种信号灯。

1.2 数据结构的产生

教材P2，例1、例2： 学生选课系统

数据：？

- 学生
- 课程
- 教师

关系：？

- 学生数据之间具有线性关系，按学号或姓名等；
- 课程数据之间具有线性关系，按课程号或课程名
- 学生和课程之间具有多对多的图关系，即一个学生可以选多门课程，一门课程可以被很多学生选；

操作：？

- 初始化：……
- 学生管理：……
- 课程管理：……
- 选课：……
- ……

1.2 数据结构的产生

- 数据结构的发展概况

从时间上看：

50年代末~60年代初	萌芽期	分散于其他课程中
60年代中~70年代初	形成发展期	68年成为独立课程
70年代中~80年代初	完善期	
80年代中~	完善更新期	

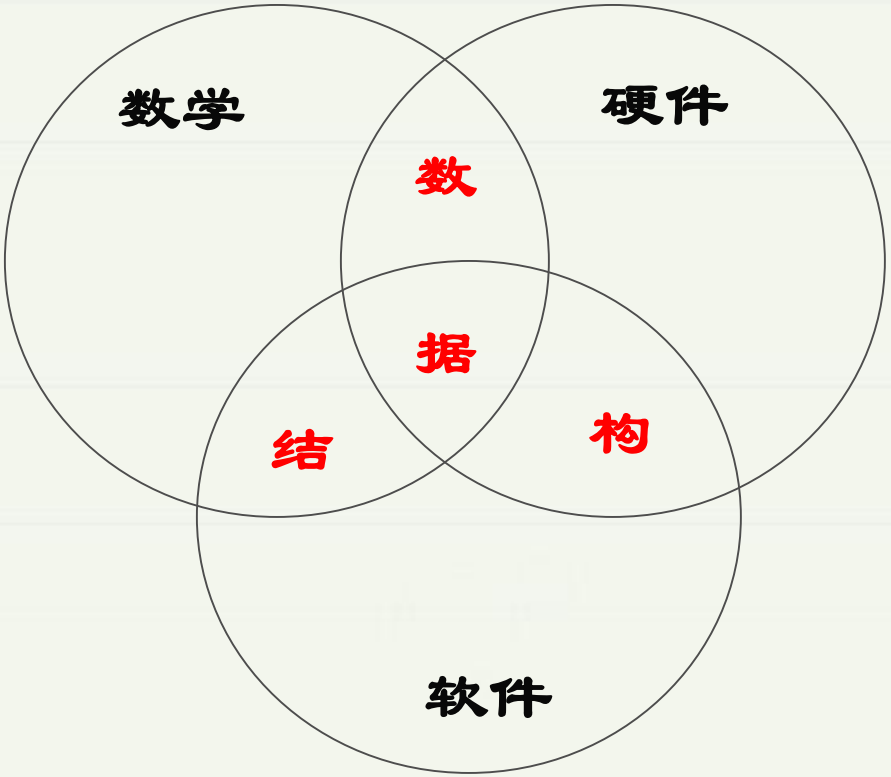
从内容上看：

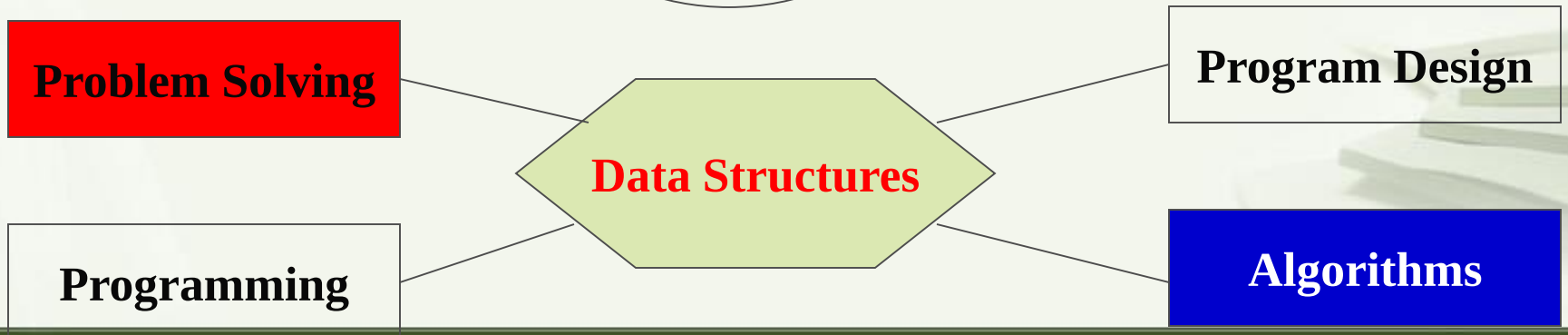
最初：介绍一些表处理系统
稍后：讨论图论，尤其是表和树
再后：讨论离散数学的内容，集合、代数、格
再后：确定了三种基本数据结构及存储结构
再后：加入了分类、检索和文件系统
近几年：涉及到了面向对象等新的内容

在我国，80年代初才列入计算机科学教学计划

1.2 数据结构的产生

- 数据结构的地位





1.3 数据结构的基本概念

[数据 Data] 用于描述客观事物的数值、字符等一切**可以输入到计算机**中，并由**计算机加工处理**的符号集合。
信息（ Information），信息和数据是什么关系？

[数据元素 Data Element] 数据中的一个个个体，是数据的**基本**单位。这是一个相对概念。

[数据项 Data Item] 构成数据元素的成份，是数据不可分割的**最小**单位。

[数据对象 Data Object] 具有相同特性的数据元素的一个集合，是数据的子集。

1.3 数据结构的基本概念

[结构 Structure] 数据元素之间的关系(Relation)

[数据结构 Data Structure] 相互之间存在一种或多种特定关系的数据元素的集合，即带结构的数据元素的集合。

数据结构=数据+结构　　记作：

Data_Structure=(D,R)

其中：**Data_Structure是数据结构的名称**

D是数据元素的有限集合（一般为一个数据对象）

R是D上关系的有限集

注：这里说的数据元素之间的关系是指元素之间本身固有的逻辑关系，与计算机无关。

因此，又称为：数据的逻辑结构、抽象数据结构

1.3 数据结构的基本概念

数据元素之间的逻辑关系分为：

（1）线性关系

（2）非线性关系：层次关系、网状关系

相应地，数据结构也分为：

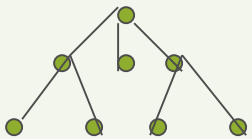
（1）线性结构(linear structure)

线性表

（2）非线性结构(nonlinear structure)

层次结构(hierarchical structure)：树

群结构(group structure)：集合，图



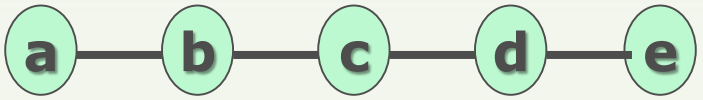
1.3 数据结构的基本概念

根据数据结构的定义，任何一个数据结构都可以利用集合和关系形式化描述出来。

例1. $DS_1=(D_1, R_1)$

$D_1=\{a, b, c, d, e\}$；

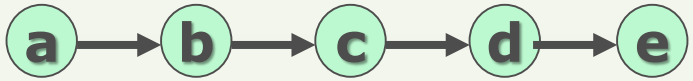
$R_1=\{(a, b), (b, c), (c, d), (d, e)\}$



例2. $DS_2=(D_2, R_2)$

$D_2=\{a, b, c, d, e\}$；

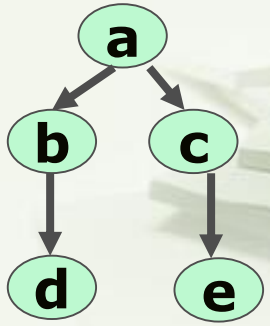
$R_2=\{<a, b>, <b, c>, <c, d><d, e>\}$



例3. $DS_3=(D_3, R_3)$

$D_3=\{a, b, c, d, e\}$；

$R_3=\{<a, b>, <a, c>, <b, d>, <c, e>\}$



1.3 数据结构的基本概念

教材上例子：学生选课系统

Student= (D_s, R_s)

D_s= {学生$_i$,$1 \le i \le n$ };

R_s= {<学生$_{i-1}$,学生$_i$> $2 \le i \le n$}

Course= (D_c, R_c)

D_c= {课程$_i$,$1 \le i \le m$ };

R_c= {<课程$_{i-1}$,课程$_i$> $2 \le i \le m$}

S_C= (D, R)

$D= D_s \cup D_c$;

R_c= {<学生$_i$,课程$_j$> ,$1 \le i \le n$, $1 \le j \le m$}

自己找一些一些身边问题的例子，深刻理解这几个概念！

1.3 数据结构的基本概念

研究数据结构的目的是干什么？

[数据的存储结构] 数据结构在计算机中的表示（映象），即数据结构在计算机中的组织形式。又称为数据的物理结构。

数据结构的物理结构是指逻辑结构的存储镜像(image)。数据结构 *DS* 的物理结构 *P* 对应于从 *DS* 的数据元素到存储区*M*（维护着逻辑结构*S*）的一个映射：

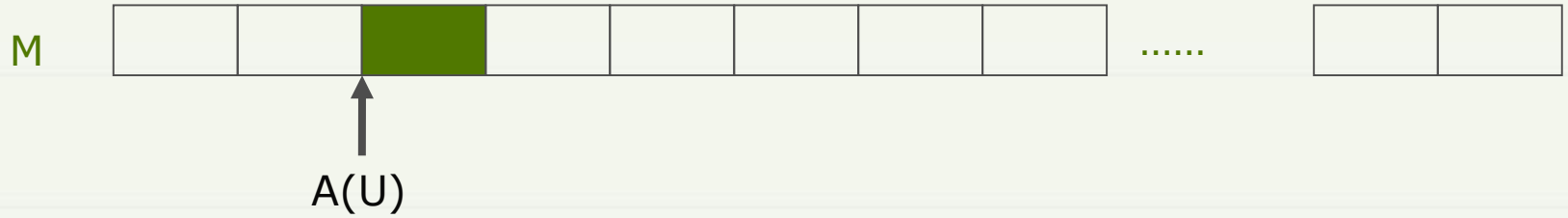
$$P:(D, S) \dashrightarrow M$$

特别要注意：

我们研究数据结构的目的是要利用数据之间的关系（结构），因此，在存储时既要存储元素本身，还要存储（表示）关系！！

1.3 数据结构的基本概念

存储器抽象模型： 一个存储器 M 是一系列固定大小的存储单元，每个单元 U 有一个唯一的地址 $A(U)$，该地址被连续地编码。每个单元 U 有一个唯一的后继单元 U'=succ(U)。



P 的四种基本映射模型：

顺序（sequential）
链接（linked）
索引（indexed）
散列（hashing）。

1.3 数据结构的基本概念

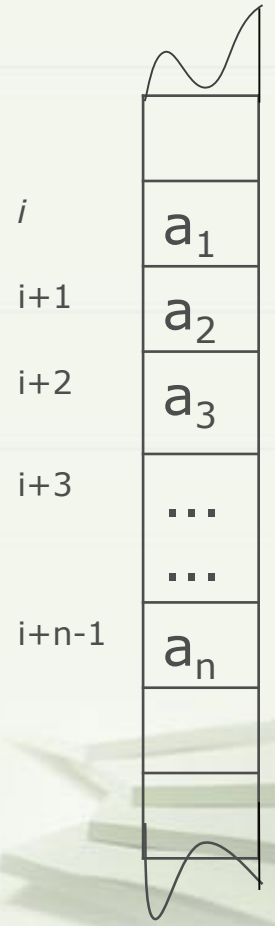
- 顺序映射：占用连续地址空间，元素依次存放。用物理上的相邻映射出逻辑上的关系（结构）；

优点：占用空间少（没有显式存储关系）；

空间连续；

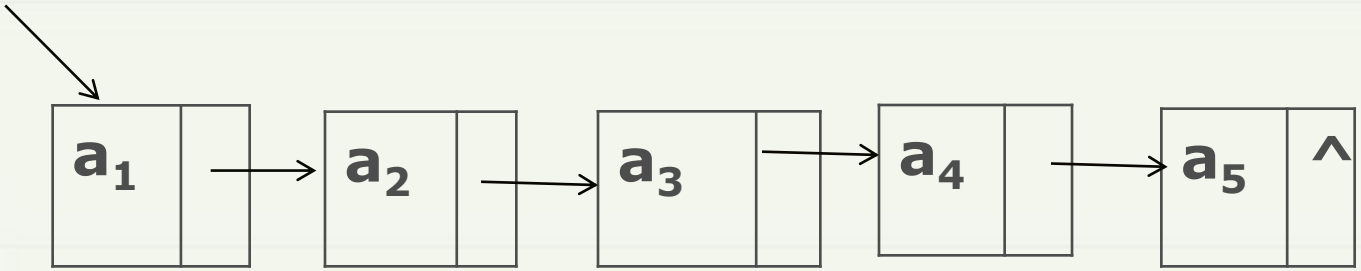
缺点：表示关系的能力弱。

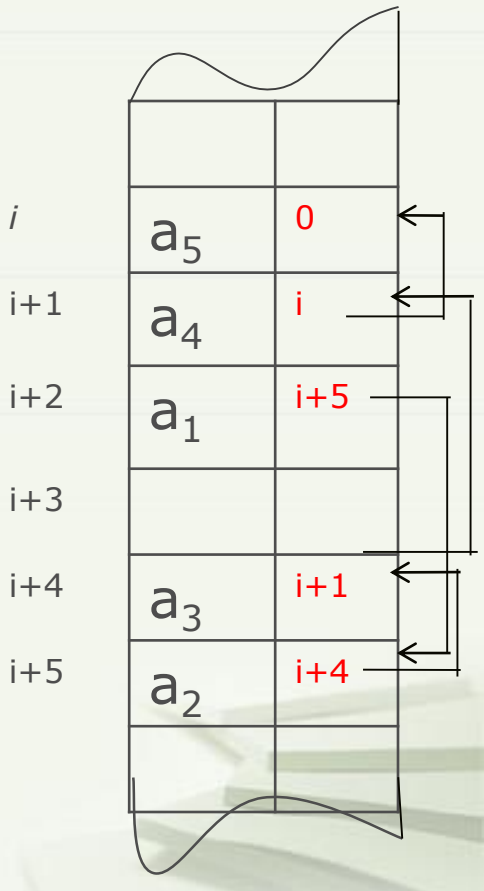
逻辑上关系发生改变时，必须物理上要调整；



1.3 数据结构的基本概念

- **链式映射：** 占用空间任意，元素任意存放。在存放元素的同时，还存放与其有关系的元素的地址（指针），即**通过指针映射出逻辑上的关系；**





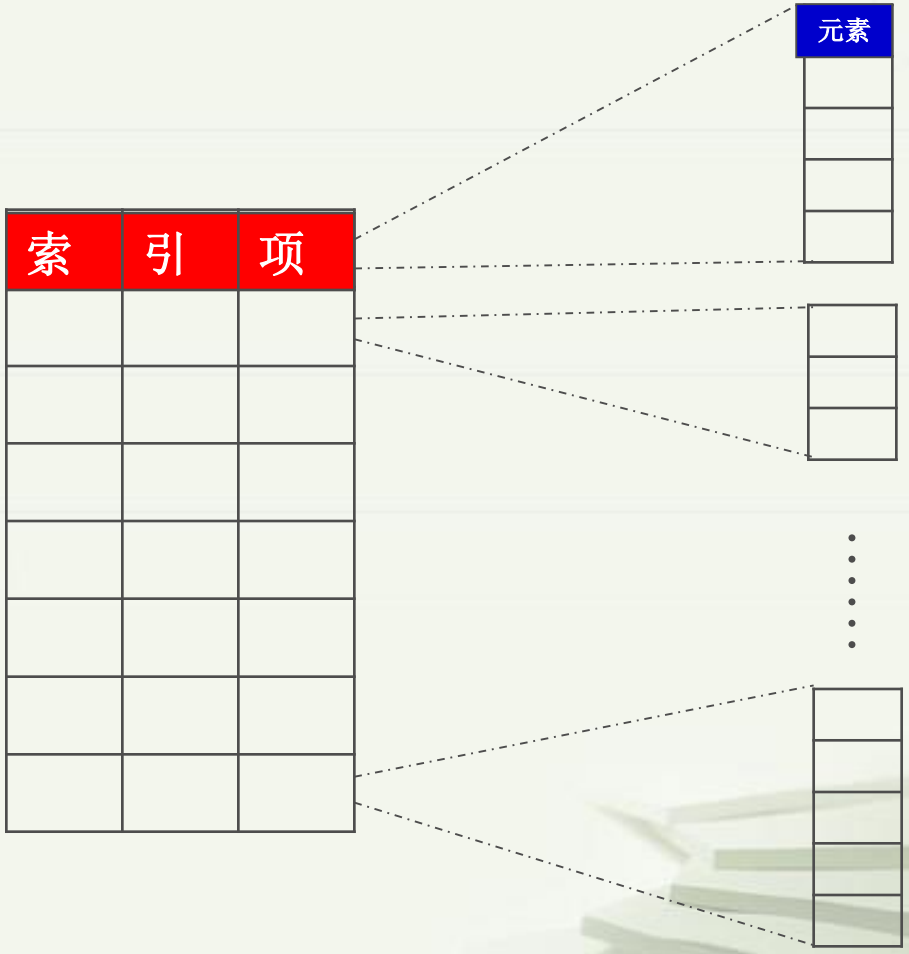
✓ 优点：空间任意；
显式地存储关系；
表示关系的能力强；

✓ 缺点：占用空间较多；

1.3 数据结构的基本概念

- 索引（indexed）映射：

如何映射关系？



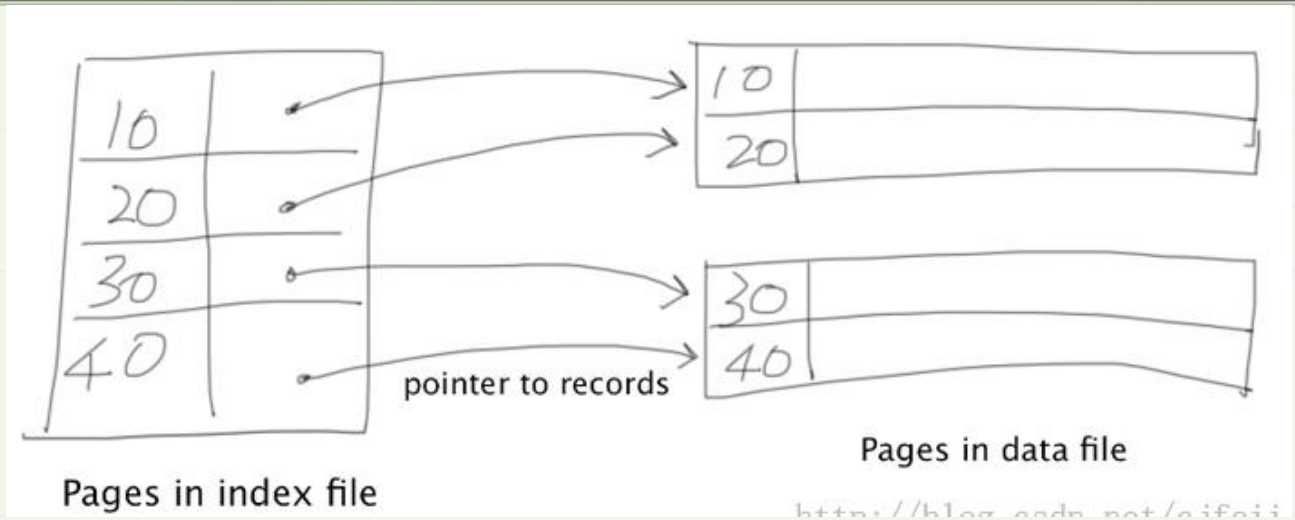
- 稠密索引(Dense Index)
- 稀疏索引(Spare Index)

  - 多级索引

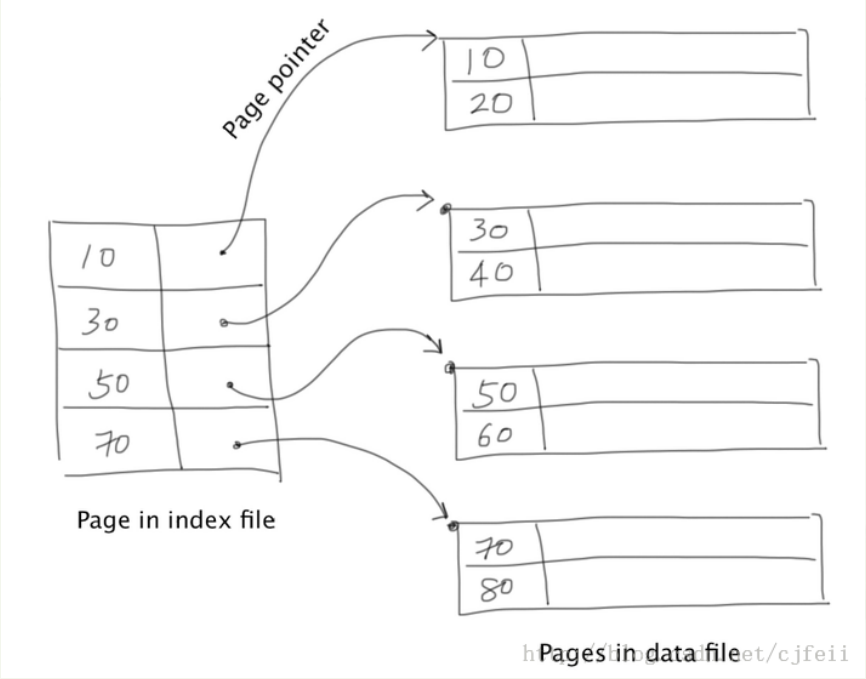
  - 索引的建立与维护

1.3 数据结构的基本概念

Dense Index：

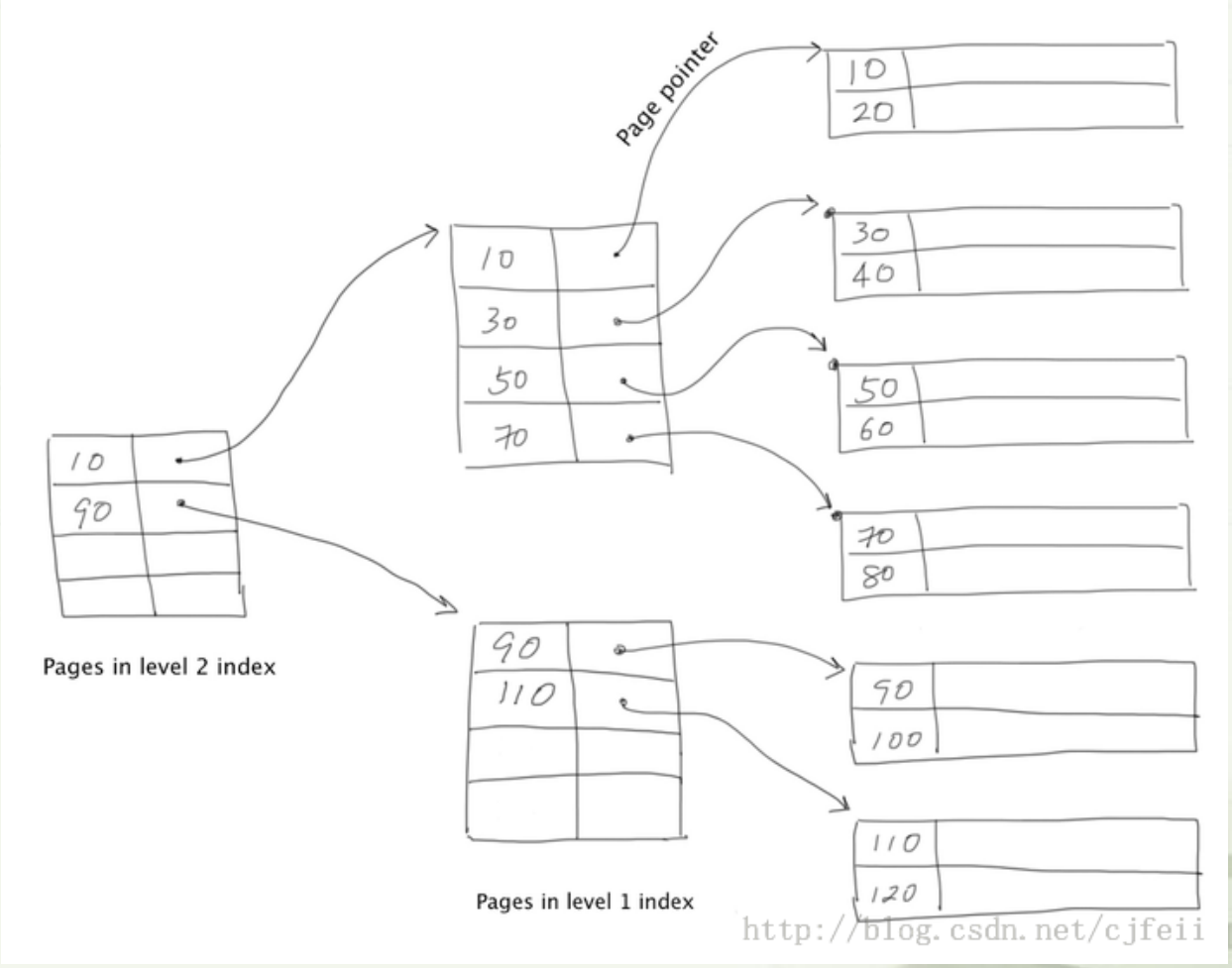


Spare Index：

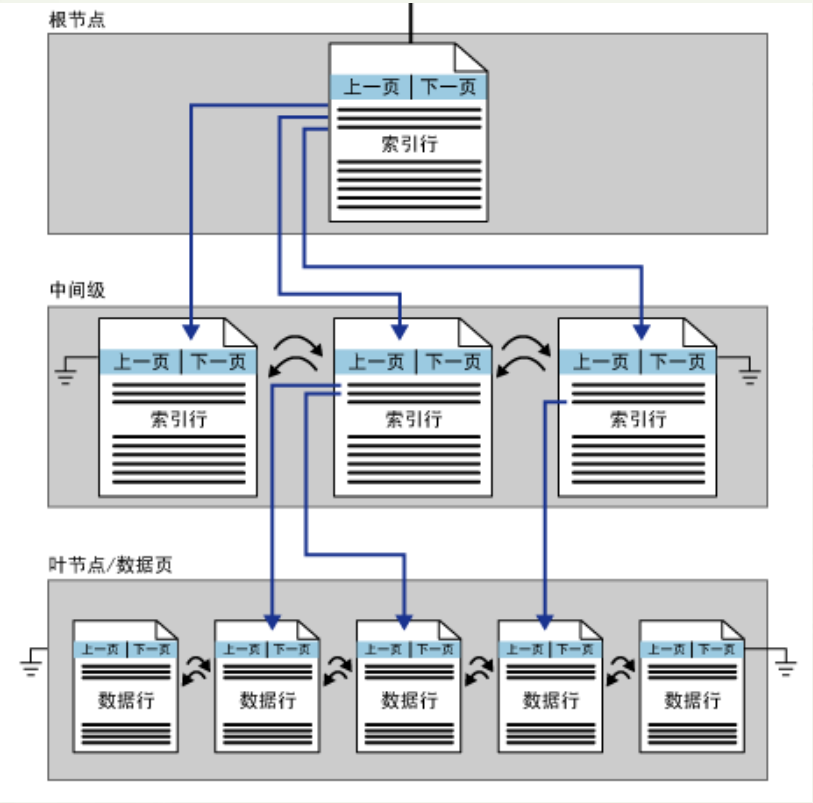


1.3 数据结构的基本概念

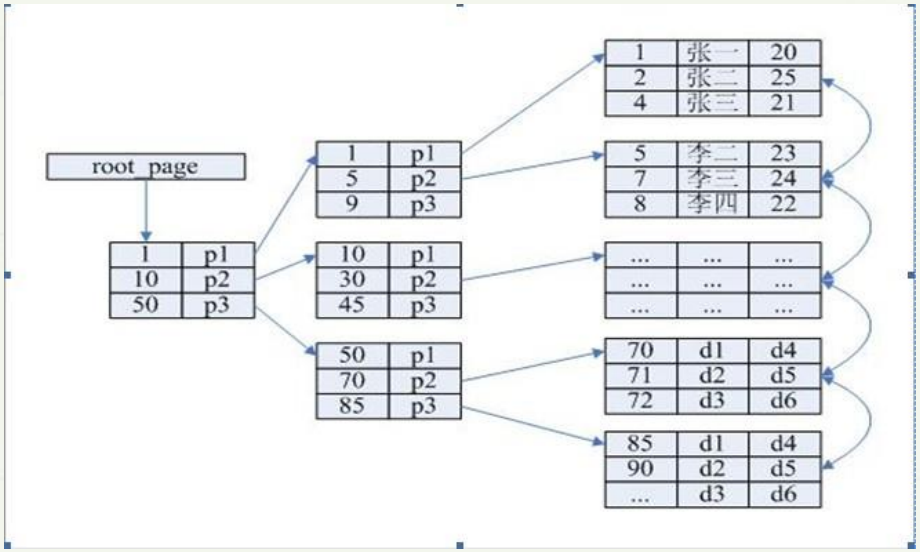
multilevel index：

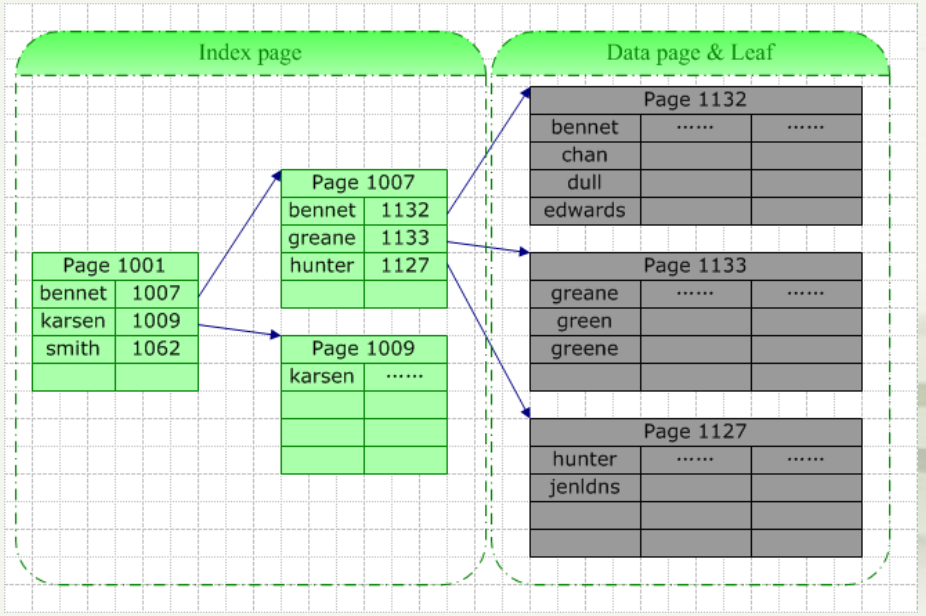


1.3 数据结构的基本概念



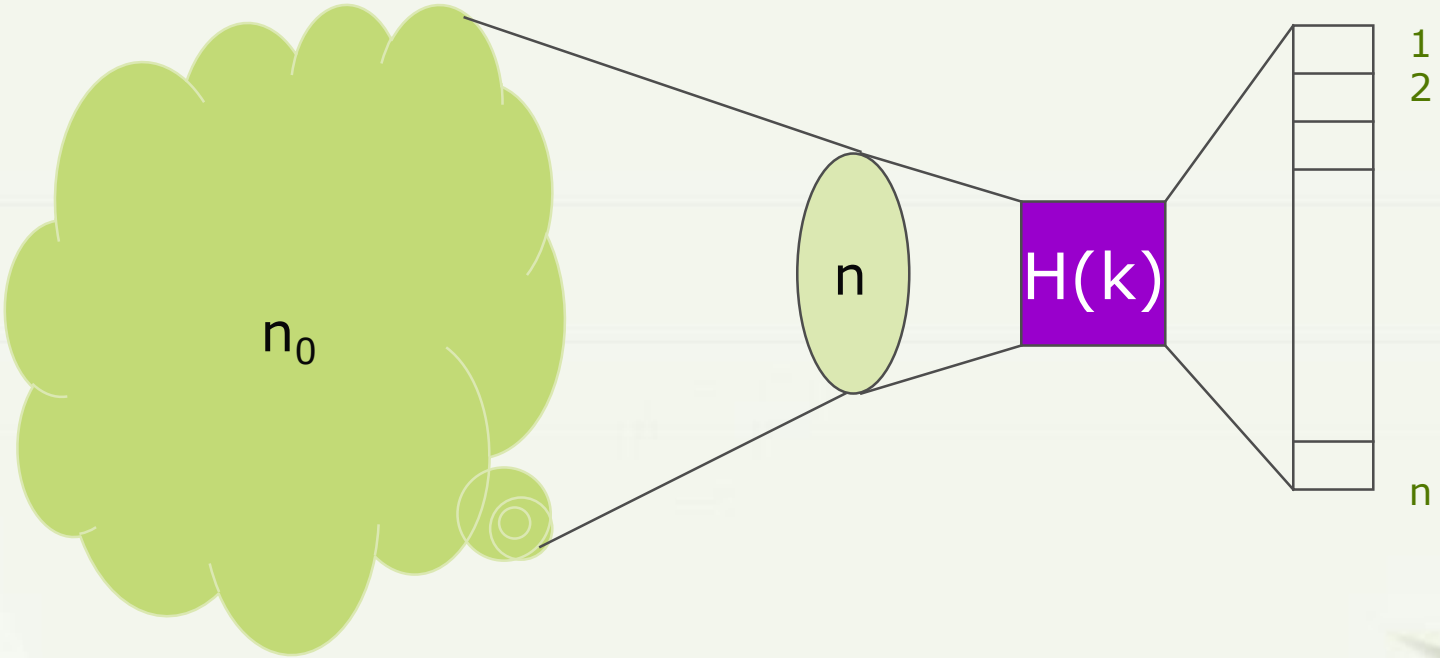
SQL Server 表和索引存储结构





1.3 数据结构的基本概念

- 散列（hashing）映射：



如何映射关系？

关键：

- HASH函数
- 解决冲突

1.3 数据结构的基本概念

因此，我们至少可以得到4×3=12种可能的数据结构的物理数据结构：

逻辑结构
Lists
Trees
Graphs

存储结构
Sequential
Linked
Indexed
Hashing

同一逻辑结构采用不同的关系映射方式，可以得到不同的存储结构。选择何种存储结构来存储相应的逻辑结构，具体问题具体分析，主要考虑运算方便及算法的时间和空间要求。——课程目标之一！

1.3 数据结构的基本概念

- 数据结构和操作

研究数据结构的目的是为了更好地加工和处理。要处理必须先存储起来，所以必须讨论存储结构，但存储起来还不是最终目的，还要对存储的数据结构实施一些操作（或运算），因此数据结构和操作是分不开的。

因此，数据结构主要有三个方面的内容：数据的逻辑结构；数据的物理存储结构；对数据的操作，即算法。

1.4 算法的基础知识

- 算法的定义

计算(calculate)：指运用事先规定的规则，将一组数值变换为另一(所需的)数值的过程。一般要有一个计算模型。

算法(Algorithm)：简单说，就是解决问题的一种方法或过程，由一系列计算步骤构成（目的是将问题的输入变换为输出）。即，它是一个定义良好的计算过程，它以一个或一组值作为输入，并产生一个或一组值作为输出。

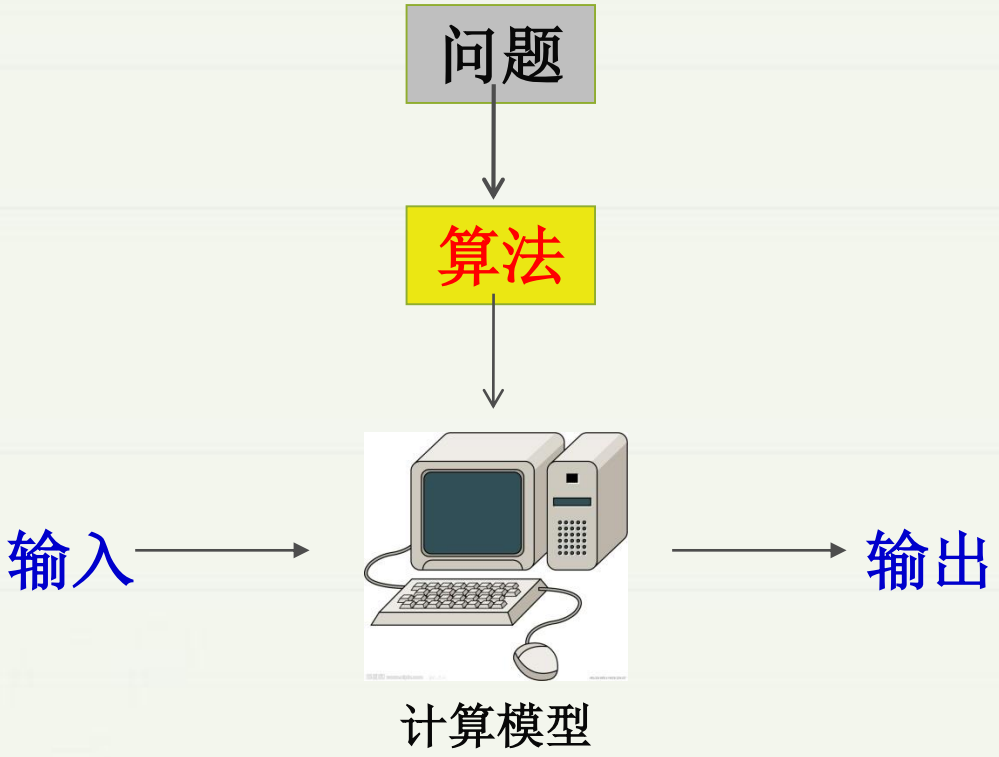
- Algorithm: a method or a process followed to solve a problem.
- A mapping of input to output（Or Transformation from input to output）
- A problem can have many algorithms.

1.4 算法的基础知识

- 算法的定义

$y = f(x)$

x 是输入
y 是输出



算法与问题：

如果问题的一个算法能应用于该问题的任何一个实例，得到该问题的正确解，称这个算法解决了这个问题。

1.4 算法的基础知识

- 算法的特性

（1）有穷性/终止性：有限步内必须停止；

（2）确定性：每一步都是严格定义和确定的动作；

（3）能行性：每一个动作都能够被精确地机械执行；

（4）输入：满足给定约束条件的输入（可以没有）；

（5）输出：满足给定约束条件的结果（必须有）；

算法与程序：

一个算法用某种程序设计语言写出来就是程序，但程序描述的不一定都是算法。

1.4 算法的基础知识

- 算法的设计

算法设计是求解问题的关键，也是软件的核心。这是人的脑力创新的一种表现形式。算法设计是一种艺术。但人们长期设计算法已总结出了一些算法设计策略：

最朴素的策略：蛮力

高级策略：

. 分治策略
. 贪心策略
. 动态规划策略
. 回溯策略
. 智能策略
. 概率策略

……

1.4 算法的基础知识

- 算法的设计

（1）蛮力策略

蛮力策略基本思想：

该策略直接基于问题的描述和所涉及的概念定义去求解问题。是一种简单直接地解决问题的方法，也是一种最容易应用的算法设计策略。

“粗笨的气力”，“Just do it !”

蛮力策略特点：

- 适应范围广；易理解；
- 效率一般不高（适合小规模）

1.4 算法的基础知识

- 算法的设计

（1）蛮力策略

典型的蛮力策略算法：

■ 顺序查找、简单选择排序、求最大最小值、矩阵乘法等。

■ 基本遍历和搜索：无知搜索（盲目搜索或穷举搜索）

◇ 深度优先搜索

◇ 广度优先搜索

有知搜索：利用某些关于问题或者问题解的知识，克服无知搜索的盲目性，有效地指导搜索过程，使之尽快到达答案状态。

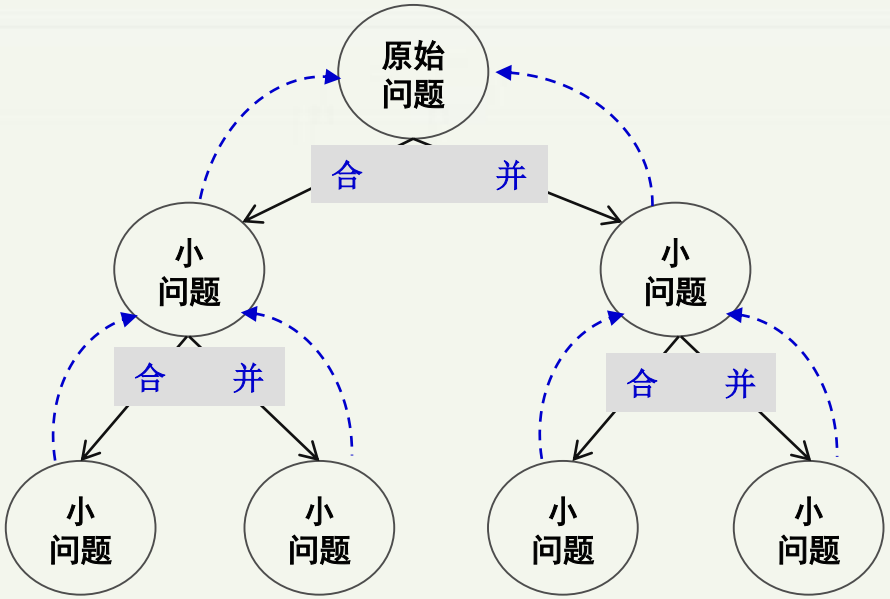
1.4 算法的基础知识

- 算法的设计

（2）高级策略之：分治

分治策略的基本思想：　　“小问题总是容易解决的！”

即：将一个难以直接解决的大问题，分割成一些规模较小的相同子问题，以便各个击破，分而治之。



1.4 算法的基础知识

- 算法的设计

（2）高级策略之：分治

具体地：为了解决一个大（复杂）问题，可以：

(a) 分解：把它分解成两个或多个更小的问题；—划分步

(b) 解决：分别解决每个小问题；—治理步

(c) 合并：把各个子问题的解采用某种方式组合起来，得到原问题的解；—组合步

注意：

- 若分解后的子问题仍然很大，对子问题可以继续应用分而治之的策略。
- 简单的分解和合并，并不会产生高效算法！

1.4 算法的基础知识

- 算法的设计

（2）高级策略之：分治

分治策略算法的基本框架：

```
Divide_and_Conquer(P)
{
    if (|P|<=n0)    // |p|=n表示问题的规模，n0为一临界值
       return(ADHOC(P) ); //解决小问题的基本算法
    else divide P into smaller subinstance P1,P2,...Pk;  // 分之
    for (i=1;i<=k;i++)
        yi=Divide_and_Conquer(Pi);   // 治之
    t=MERGE(y1,y2,...yk);   // 组合而得到解
    return(t)
}
```

1.4 算法的基础知识

- 算法的设计

（2）高级策略之：分治

分治策略的关键：

分解：将大问题分解为小问题，最常见的是二分，“均衡”

合并：由子问题的解得到复杂问题的解。具体问题具体分析。

分治与递归：

分治和递归是一对孪生姐妹。

- **n!=n*(n-1)!**
- **hanoi问题**
- **求n个数的和问题**

1.4 算法的基础知识

- 算法的设计

（2）高级策略之：分治

典型分治策略解决的问题及算法：

- 查找问题之：二分（折半）查找
- 排序问题之：归并排序
- 排序问题之：快速排序
- 矩阵乘法之：Strassen算法（7小矩阵乘法）

♣课外实习：有兴趣的同学可以自己设计开发下面的游戏：

- 汉诺塔问题
- 残缺棋盘问题

1.4 算法的基础知识

- 算法的设计

（3）高级策略之：贪心　　一般用于求解最优化问题

最优化问题的定义：

最优化问题（optimization problem），即求最优解问题。每个最优化问题都包含一组限制条件（constraint）和一个优化函数（optimization function）。

符合限制条件的问题求解方案称为可行解（ feasible solution）；满足约束条件的解不是唯一的！

使优化函数取得最佳值的可行解称为最优解（optimal solution）；

1.4 算法的基础知识

- 算法的设计

（3）高级策略之：贪心

例，船的装载问题

问题描述： 有一条大船，最大载重为C，有n个重量不同，但大小相同的装货物的箱子，重量分别为$w_1,w_2,...w_n$。现要将箱子装船，要求在船上装入最多的箱子，且船不能超载。

限制条件：$\sum_{i=1}^{n} x_i w_i \le C$　　$x_i = \begin{cases} 1 & \text{装} \\ 0 & \text{不装} \end{cases}$

优化函数：$\sum_{i=1}^{n} x_i$

可以用蛮力策略，但是……

思考：背包问题是典型的最优化问题，也是很多问题的抽象模型。请写出该问题的限制条件和优化函数。

1.4 算法的基础知识

- 算法的设计

（3）高级策略之：贪心

人们在研究、探索更快、更直接的求最佳解的方法中，发现了贪婪（贪心）策略。

贪心策略是通过“分步决策”的方法来求解问题的，它在求解问题的每一步上依据“某种准则”做出当前看来最好的决策，产生n-元组解的各个分量。

1.4 算法的基础知识

- 算法的设计

（3）高级策略之：贪心

贪心策略的基本思想：

首先选定一种最优量度标准（贪心准则），作为选择当前分量值的依据。然后依据该标准，在每一步上做出当前最好的选择，最终每一步的最优决策正好构成问题的最优解。 即，局部贪心得到全局最优！

最优量度标准，又称为贪心准则(Greedy criterion)，也称贪心选择性质。这种度量标准考虑的局部最优性！

可以看出，贪心并不能保证最终能得到整体最优解（为什么）？只有能得到最优解的贪心策略才是正确的！

1.4 算法的基础知识

- 算法的设计

（3）高级策略之：贪心

贪心策略算法的基本框架：

```
Greedy(A, n)     // A[1:n]代表那些输入

   solution={};     //解向量初始化为空集
      for i from 1 to n do    //多步决策，每次选择解向量的一个分量
          x=Select(A);     //遵循最优量度标准选择分量
          //判定加入新分量后的部分解是否可行
          if Feasible(solution, x) then
               solution=Union(solution, x);    //形成新的部分解
          endif
      endfor
      return(solution)  // 返回生成的最优解
   end Greedy
```

1.4 算法的基础知识

- 算法的设计

（3）高级策略之：贪心

贪心策略的关键：

◇ 问题满足最优子结构性质；
◇ 确定贪心准则；
◇ 证明局部最优选择可以导致全局最优结果；

典型用贪心策略解决的问题及算法：

- 最少个数找零。贪心准则：面值最大的优先
- 装载（背包问题等）。贪心准则：轻的优先
- 活动安排问题。贪心准则：相容且结束早的优先
- 最短路径问题的Dijkstra算法。
- 哈夫曼编码问题的最优二叉树算法。
- 最小连通代价问题的最小生成树算法，Prim 算法， Kuskal 算法

1.4 算法的基础知识

- 算法的设计

（4）高级策略之：动态规划

动态规划策略的基本思想：

通常也一般用于求解最优化问题，与贪心策略类似，也是通过子问题的最优解求得整个问题的最优解。但是是从全局出发，通过问题分析，挖掘子问题最优解和整个问题最优解的关系，建立动态规划方程。

1.4 算法的基础知识

- 算法的设计

（4）高级策略之：动态规划

动态规划策略的关键：

- ◇ 问题具有最优子结构性质；
- ◇ 建立动态规划方程——设计
  (即，刻画出子问题最优解和原问题最优解的关系)
- ◇ 求解动态规划方程——求解

动态规划策略算法的基本框架：

动态规划策略设计的算法就是求解动态规划方程的过程。通常有两种形式：

自顶向下的递归；（+备忘录）

自底向上的递推；

1.4 算法的基础知识

- 算法的设计

（4）高级策略之：动态规划

典型动态规划策略求解的问题

- 矩阵连乘问题
- 最大子和问题
- 最长公共子序列问题
- AOE的关键路径问题

……

1.4 算法的基础知识

- 算法的设计

（5）高级策略之：回溯

回溯法有“通用解题法”之称，用它可以系统地搜索问题的所有解。

回溯策略的基本思想：

定义问题的解空间树，然后按深度优先策略，从根结点出发搜索解空间树。当搜索至解空间树的任意一点时，先判断该结点是否包含问题的解：*如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；*否则，进入该子树，继续按深度优先策略搜索。

1.4 算法的基础知识

- 算法的设计

（5）高级策略之：回溯

回溯策略的关键：

- ◇ 问题具有约束集完备性；
- ◇ 确定解空间树；
- ◇ 深度优先搜索+回溯（杀死、剪枝）；

剪枝函数：

其一：约束函数，扩展顶点处剪去不满足约束的子树；

其二：限界函数，剪去不能得到最优解的子树；

1.4 算法的基础知识

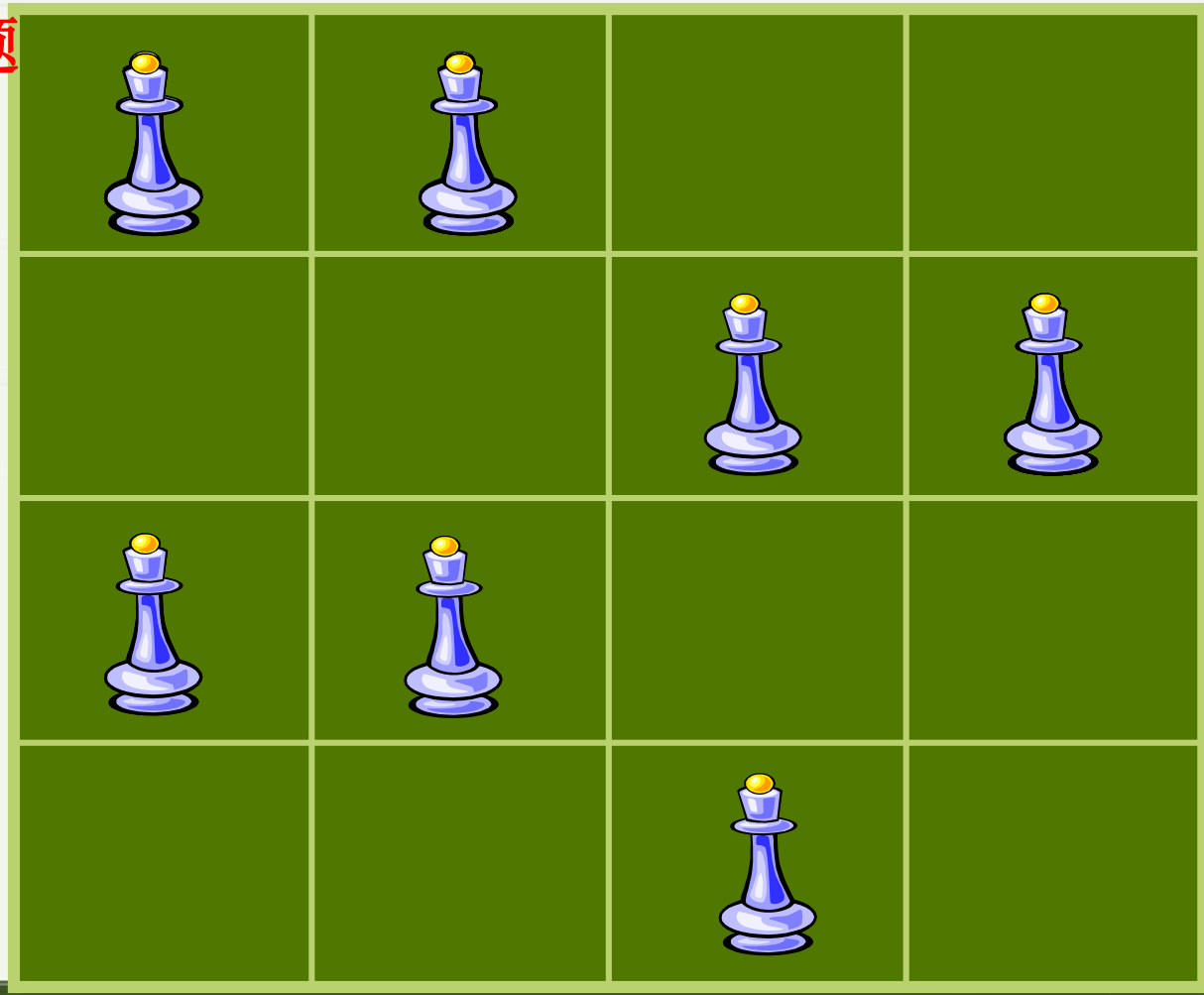
- 算法的设计

（5）高级策略之：回溯

典型回溯策略求解的问题

- N后问题
- 迷宫问题
- 数独问题
- 子集和问题

回溯策略举例：
四后问题



1.4 算法的基础知识

- 算法的描述

（1）自然语言描述
特点：容易，但有时罗嗦、有二义性；

（2）图示（流程图、N-S图、PAD图等）
特点：直观清晰，但不宜实现；

（3）算法语言（伪代码）
特点：严谨、简洁，易程序实现；

（4）程序设计语言
特点：可以直接运行，但太严格；

1.4 算法的基础知识

- 算法的评价标准

✓ 正确性(Correctness)

✓ 易读性(Readability)

✓ 健壮性(Robustness)

✓ 有效性(Efficiency)：时间、空间

简单性；一般性；有效性；

1.4 算法的基础知识

- 算法的性能分析

计算资源包括运行时间和存储空间，而且是有限资源。而人追求应用更广、完成任务更大、速度更快、效果更好……的欲望是无止境的，因此资源有效利用很重要!

解决同一问题的各种不同方案（核心就是数据结构和算法）的资源效率有时差别很大，这种差距的影响往往比硬件方面的差距还要大。

[算法分析 Algorithm Analysis] 估量一个算法效率的方法，包括运行时间效率和空间效率——算法的复杂性

1.4 算法的基础知识

一个例子：

计算机A，每秒能执行10亿条指令；机器语言编程；用插入排序算法来对n个数进行排序，需要$2n^2$条指令。

计算机B，每秒执行1000万条指令；高级语言编程；用归并排序算法来对n个数进行排序，需要$50n\lg n$条指令。

现在有100万个数要进行排序，两台计算机所用的时间分别为：

计算机A花时间：$\dfrac{2\bullet(10^6)^2\text{条指令}}{10^9\text{条指令/秒}}=2000\text{秒}$

计算机B花时间：$\dfrac{50\bullet10^6\lg10^6\text{条指令}}{10^7\text{条指令/秒}}\approx100\text{秒}$

可以看出，尽管计算机B速度慢，而且是效率低的编译器，但是其排序速度比计算机A快了近20倍！而且随着数据规模的变大，这种差距更大！（例如当数据达到1000万，插入排序要2.3天，归并排序只要20分钟）。

1.4 算法的基础知识

- 算法性能的分析方法

经验测试（后期测试）：选择样本数据、运行环境运行算法计算出空间、时间。

先验估计（事前估计）：根据算法的逻辑特征（基本数据量、操作）来估算。

优点：精确
缺点：可比性差，效率低

优点：可比性强
缺点：不精确，仅仅是估计

那么，如何撇开计算机本身来估算一个算法的复杂性呢？

1.4 算法的基础知识

- 算法的空间性能分析——空间复杂度

（1） 算法的空间性能的影响因素

- 指令空间（由机器决定）；
- 数据空间（常量、变量占用空间）；
- 环境栈空间；

数据空间—用来存储算法所有常量和变量的值。 分成两部分：

存储简单量

存储复合变量

1.4 算法的基础知识

- **算法的空间性能分析**——空间复杂度

（2）度量方法：

单个常量、变量：由机器和编译器规定的类型存储决定

数组变量：所占空间等于数组大小乘以单个数组元素所占的空间。

结构变量：所占空间等于各个成员所占空间的累加；

例如：

double a[100]; 所需空间为100×8＝800

int matrix[r][c]; 所需空间为 2×r×c

1.4 算法的基础知识

- **算法的时间性能分析**——时间复杂度

（1）算法的时间性能的影响因素：

- 机器的运行速度（执行代码的速度）；
- 书写程序的语言；
- 编译产生代码的质量；
- 算法的策略；
- 问题的规模；

1.4 算法的基础知识

- 算法的时间性能分析——时间复杂度

（2）算法的时间性能度量：

令T(P)表示算法P需要的时间，则有：

$$T(P) = c + t_P(\text{实例特征})$$

$t_P(n)=c_aADD(n)+c_sSUB(n)+c_mMUL(n)+c_dDIV(n)+\cdots$

其中，c_a，c_s，c_m，c_d表示一个加、减、乘、除操作所需的时间，函数ADD，SUB，MUL，DIV分别表示P中所使用的加、减、乘、除等基本操作的次数；

即T(P)是算法中一些基本操作的执行次数的累加和。
它一般是关于规模(n)的函数　　教材P29-32(略)

1.4 算法的基础知识

- 算法的时间性能分析——时间复杂度

（3）渐近时间复杂性：

统计出算法的基本操作次数和T(P) 后，怎么下结论呢？

例如：T(P1)=1000n+28

T(P2)=n^2

问：哪个算法更好？

既然我们的目的是为了比较，就应该将注意力集中在被比较的对象之间最主要的差别。例如，

数学知识告诉我们：如果两个算法的基本操作次数分别是3n+2和3n+20，则这两个算法的时间复杂性不会有太大的差别！

1.4 算法的基础知识

- 算法的时间性能分析——时间复杂度

这样，就人们找到了一个关注点， 算法性能的增长率！

算法的增长率(growth rate)：是指当问题规模增长时，算法代价的增长速率。具体就体现在两个函数的变化趋势上。

T(n)=f(n)——时间代价

S(n)=g(n)——空间代价

简单说，就是看看这个算法的资源消耗的增长有多快，增长快的（阶高），其资源消耗就大，复杂性就高！

——这就是复杂性渐近态的思想

1.4 算法的基础知识

- 算法的时间性能分析——时间复杂度

例如： $T_1(n)=3n^2+4n\log n$

$T_2(n)=2n^2$

在n很大后，两个函数变化趋势接近一致。可以说$n\to\infty$时，$T_1(n)$渐近于$T_2(n)$，可以用简单的$T_2(n)$来表示算法的复杂性。当然，更简单的函数是n^2。 显然，应该用最简单的函数来代表增长率一样的这一类算法的复杂性。

因此，渐近复杂性分析只关心度量函数的阶就够了，因为按照渐近的思想，度量函数中的常数因子对算法复杂性的影响不敏感。

可以看出，渐近复杂性分析主要专注于问题的“规模”、“基本的操作”和增长率。

1.4 算法的基础知识

- **算法的时间性能分析**——时间复杂度

复杂性的渐近态：

对算法，精确计算其增长率有时很难，如果能粗略比较出哪个更快或更慢就可以了。于是在关注增长率的基础上提出渐近估计的概念。即规模变大时，趋势渐近于什么。即“和谁差不多一样快”

渐近复杂性分析（Asymptotic Algorithm Analysis）

确切说：渐近分析是指当输入规模很大，或者说大到一定程度时对算法的研究和分析。（从微积分意义上说是达到极限时）。 这样得到的就是算法的渐近复杂性。

1.4 算法的基础知识

渐近时间复杂度的本质：

从增长率可以看出：算法不同，算法的时间复杂性不同（增长率不同），当计算机速度提高时，你得到的解决问题规模的**增益**是不同的！

例如，当计算机的运行速度是提高为原来的**10**倍时，不同的算法在解决问题的规模上得到的收益是有很大不同的。

f(n)	n	n’	变化	n/n’
10n	1000	10000	n’=10n	10
20n	500	5000	n’=10n	10
5nlogn	250	1842	3.16n<n’<10n	7.37
$2n^2$	70	223	n’= $\sqrt{10}$ n	3.16
2^n	13	20	n’=n+7	-

注：
n是原来规模；
n’是现在规模

1.4 算法的基础知识

最好、最坏、平均时间复杂性：

算法性能还与输入样本的分布有关，这样，同样的规模，如果输入样本分布不同，其算法性能可能有不同的表现，于是就有了：

最好时间复杂性：

最坏时间复杂性：

平均时间复杂性：

问题的时间复杂性怎么定义？

解决一个问题的所有算法中，时间复杂性最好的那个算法的时间复杂性定义为问题的时间复杂性。（即代表着问题目前求解的最好策略！）

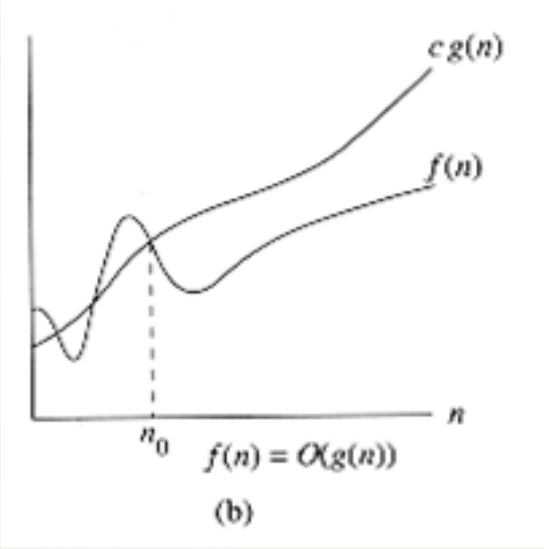
1.4 算法的基础知识

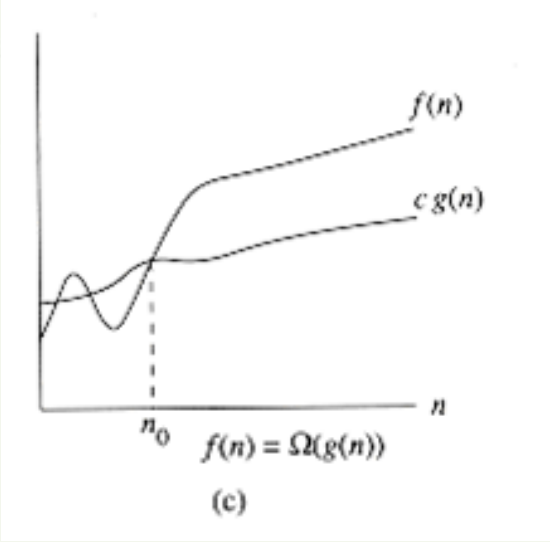
渐近时间复杂性的数学符号(数学定义 略！)：

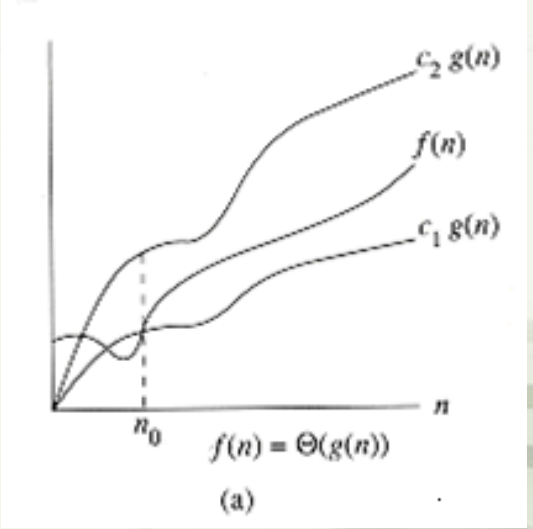
渐近上界：O ——增长率的最小上界（限）——最坏

渐近下界：Ω ——增长率的最大下界（限）——最好

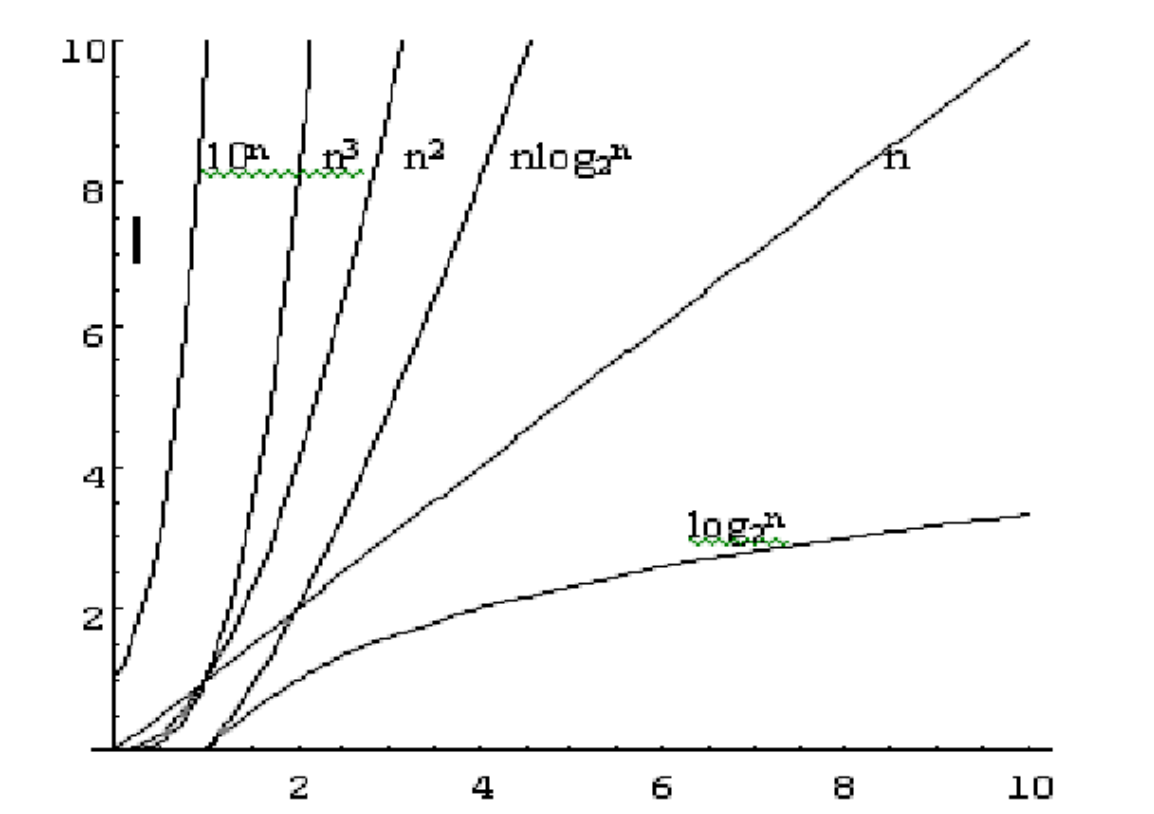
渐近确界：Θ ——增长率的上、下界相同







1.4 算法的基础知识



f(n)函数曲线变化速度的比较

$$c < \log_2 n < n < n\log_2 n < n^2 < n^3 < 2^n < 3^n < n!$$

1.4 算法的基础知识

算法分析的一般步骤：

（1）确定问题规模参数；

（2） 确定关键（基本）操作、存储；

（3）依据算法流程，建立各个量之间的“和”关系模型；

（4）利用数学方法求出累计和函数T(n),S(n)；

（5）给出渐近表示。

1.4 算法的基础知识

- 一般算法的时间复杂性分析：

1. 决定哪个（哪些）参数表示输入规模；
2. 确定基本操作。赋值、读、写等一般看作基本操作；
3. 划分三种控制结构；
4. 对各种控制结构，有
   （1）顺序操作序列，可以累加或取最大值；
   （2）分支的条件判断一般看作基本操作，分支结构的时间定义为条件成立和不成立时的和；
   （3）循环结构的时间为循环体的时间乘循环次数；
5. 函数调用时要考虑函数的执行时间。
6. 建立其基本操作执行次数的“求和”表达式。
7. 利用数学知识求和得到其T(n)
8. 确定阶，得到渐近表示：一般给出O表示。

1.4 算法的基础知识

- 一般算法的时间复杂性分析：

例1. 分析下面程序段表示的算法的时间复杂性：

```
i=1; k=0;
while(i<n)
  { k=k+10*i;  i++; }
```

T(n)=1+1+n+2(n-1)=3n

T(n)=O(n)

```
i=1; k=0;
do
{ k=k+10*i;  i++;
}while(i<n);
```

T(n)=1+1+n+2(n-1)=3n

T(n)=O(n)

1.4 算法的基础知识

- 一般算法的时间复杂性分析：

```
for(i=1;i<=n; i ++)
   for(j=1;j<=n; j ++)
      { ++x; s+=x; }
```

$T(n)=n+1+n(n+1)+2(n*n)=3n^2+2n+1$

$T(n)=O(n^2)$

```
for(i=1;i<=n; i ++)
   for(j=1;j<=i; j ++)
      ++x;
```

$$T(n)=n+1+\sum_{i=1}^{n}(i+1)+\sum_{i=1}^{n}i$$

$T(n)=O(n^2)$

1.4 算法的基础知识

- 一般算法的时间复杂性分析：

```
i=1;
while(i<=n)
   i=i*2;
```

$T(n)=1+\log_2 n+1+\log_2 n=2\log_2 n+2$

$T(n)=O(\log_2 n)$

```
s=0;
for(i=0;i<=n;i++)
  for(j=0;j<=i;j++)
     for(k=0;k<=j;k++)
        s++;
```

$$T(n)=1+n+2+\sum_{i=0}^{n}(i+1)+\sum_{i=0}^{n}\sum_{j=0}^{i}(j+1)+\sum_{i=0}^{n}\sum_{j=0}^{i}j$$

$T(n)=O(n^3)$

1.4 算法的基础知识

- 一般算法的时间复杂性分析：

```
x=91; y=100;
while(y>0)
  if(x>100)
    { x=x-10; y--; }
  else x++;
```

T(n)=1+ 1+ 1001+1000

T(n)=O(1)

1.4 算法的基础知识

- 一般算法的时间复杂性分析：

例2. 有算法如下：

```
MaxElement(A[0..n-1])
  {
     //功能：？
     //输入：数组A[0..n-1]
     //输出：？
     maxval=a[0];
     for(i=1;i<=n-1;i++)
        if(a[i]>maxval)
           maxval=a[i];
     return maxval;
  }
```

T(n)=O(n)

1.4 算法的基础知识

- 一般算法的时间复杂性分析：

例3. 有算法如下：

```
UniqueElements(A[0..n-1])
  {
    //功能：？
    //输入：数组A[0..n-1]
    //输出：？
     for(i=0;i<=n-2;i++)
       for(j=i+1;j<=n-1;j++)
          if(a[i]==a[j]) return False;
      return True;
  }
```

$T(n)=O(n^2)$

$$T(n)=n+\sum_{i=0}^{n-2}(n-i)+\sum_{i=0}^{n-2}(n-i-1)+1$$

1.4 算法的基础知识

- 一般算法的时间复杂性分析：

例4. 有算法如下(p34 程序1.26)

```
void example(float x[][n],int m)
{  float sum[m];int i,j;
   for(i=0;i<m;i++)
   { sum[i]=0;
     for(j=0;j<n;j++)
        sum[i]=sum[i]+x[i][j];
   }
  for(i=0;i<m;i++)
    cout<<“Line”<<i<<“:”<<sum[i]<<endl;
}
```

T(m,n)=O(m×n)

1.4 算法的基础知识

- 一般算法的时间复杂性分析：

例5. 分析顺序查找算法的复杂性(p36 程序1.27)

```
int SequenceSearch(int a[],int n,int key)
{ //若找到，则返回元素的下标，否则返回-1；
   for(int i=0;i<n;i++)
      if(a[i]==key) return i;
   return -1;
}
```

$T_{min}(n)=O(1)$

$T_{max}(n)=O(n)$

$T_{ave}(n)=O(n)$

```
int SequenceSearch (int a[],int n,int key)
{ a[0]=x;
   i=n;
   while(a[i]!=x)
      i=i-1;
   return i
}
```

1.4 算法的基础知识

- 递归算法的时间复杂性分析：

递归算法是比较特殊的一类算法，其时间复杂性分析过程一般有下面几步：

1. 根据递归算法 确定问题的相关参数；
2. 分析递归算法，建立递推关系，给出边界条件，写出时间的递归方程；
3. 求解递归方程；
4. 给出递归方程解的渐近表示；

其中，求解递归方程式是关键，也是比较难的一步！

1.4 算法的基础知识

- 递归算法的时间复杂性分析：

例6. Hanoi塔求解问题

问题描述，略！

算法：

```
void Hanoi(int n,char c1,char c2,char c3)
{ if(n==1) printf("%c->%c\n",c1,c2);
  else { Hanoi(n-1,c1,c3,c2);
         printf("%c->%c\n",c1,c2);
         Hanoi(n-1,c3,c2,c1);
       }
}
```

算法分析：

$$T(n)=\begin{cases} c & n=1 \\ 2*T(n-1)+c & n>1 \end{cases}$$

$T(n)=2\times 2\times 2\times \ldots=2^n-1=O(2^n)$

1.4 算法的基础知识

- 递归算法的时间复杂性分析：

例7. 二分查找

```
int B_S(int l,int r,int a[],int x)
{ int m;
  if(r<l) return -1;
  else {
        m=(l+r)/2;
        if(a[m]==x) return m;
        else{ if(x>a[m]) l=m+1;
              else r=m-1;
              B_S(l,r,a,x); }
      }
}
```

算法：

算法分析：

$$T(n)=\begin{cases} c & n=0 \\ T(\frac{n}{2})+c & n>0 \end{cases}$$

$T(n)=c+c+c+\ldots=\log_2 n\times c$

$=O(\log_2 n)$

1.4 算法的基础知识

例8. 有5个算法如下，你可以得出什么分析结论？

算法	T(n)	时间复杂度
A_1	1000n	O(n)
A_2	100nlogn	O(nlogn)
A_3	$10n^2$	$O(n^2)$
A_4	n^3	$O(n^3)$
A_5	2^n	$O(2^n)$

结论：

$2 \le n \le 9$	A_5最好
$10 \le n \le 58$	A_3最好
$59 \le n \le 1024$	A_2最好
$n > 1024$	A_1最好

1.5 抽象数据类型

- 抽象思维方法

舍去复杂系统中非本质的细节，只把其中某些本质的、能反映系统重要宏观特性的东西提炼出来，构成系统的模型，并且深入研究这些特性。

在思考一个复杂问题的解决时，首先应考虑能否将它抽象简化，否则很难理解或者实现它！

例如：对房子的抽象

再如：有一堆鸡蛋，进行了编号，对它们进行如下操作：

- 找出最重的；
- 取走某一个；
- 全部取走；

1.5 抽象数据类型

- 数据类型

“数据类型”是大家非常熟悉的一个概念，那么，什么是数据类型？

[数据类型 Data Type] 一个数据值的集合和定义在这个值集上的一组操作的总称。

注意： 数据类型是一个非常重要的概念，要正确理解它！！

（1） 高级语言中的数据类型实际上包括：数据的逻辑结构、数据的存储结构及所定义的操作的实现。

（2）高级语言中的数据类型按值的不同特性分为：

原子类型（如整型、实型、字符型、布尔型）

结构类型（如数组、结构体等）

1.5 抽象数据类型

（3）数据类型并不局限于高级语言，它实际上是一个广义的概念。

例如：“教师”就是一个数据类型，它有值“教龄”，有操做“教书”等；如果具体说小学教师、大学教师，可以看作是一个具体的类型（好像有了存储结构）；

（4）撇开计算机，现实中的任何一个问题都可以定义为一个数据类型——称为抽象数据类型

[抽象数据类型 Abstract Data Type，ADT] 一个数学模型及定义在这个模型上的一组操作（或运算）的总称。

1.5 抽象数据类型

- 抽象数据类型和数据结构

抽象数据类型 = 数学模型+操作
= 数据结构+操作
= 数据+结构+操作

它是一种描述用户和数据之间接口的抽象模型，亦即它给出了一种用户自定义的数据类型。

一个ADT可以用三元组表示：（D，S，P）

其中：D是数据对象；

S是D上的关系集；

P是对D的基本操作集。

1.5 抽象数据类型

- 抽象数据类型的定义及描述

ADT 抽象数据类型的名称

（数学模型 数据结构 { 数据元素(D)；结构(S)）

操作(P)

END ADT

1.5 抽象数据类型

- 用面向对象的规范描述ADT

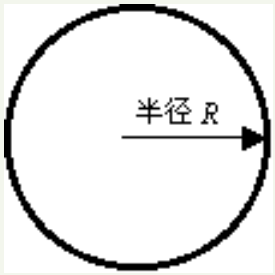
```
ADT   ADT名称  is
      Data
         描述数据的结构
      Operations
         构造函数
         Initial values：用来初始化对象的数据
         Process：初始化对象
         操作1
            Input,Preconditions,process,output,postconditions
         操作2
            ……
         操作n
end ADT ADT名称
```

1.5 抽象数据类型

- **ADT定义举例**

例1. 计算圆的周长和面积

圆是平面上与圆心等距离的所有点的集合。从**图形显示角度看**，圆的抽象数据类型包括圆心和半径；而从**计量角度看**，它所需要的抽象数据类型只需半径即可。如果从计量角度来给出圆的抽象数据类型，那么它的数据取值范围应为半径的取值范围，即为非负实数，而它的操作形式为确定圆的半径（赋值）、求圆的面积、求圆的周长等。



1.5 抽象数据类型

问题描述：给定圆的半径，求其周长和面积

ADT定义：

ADT *circle*
data : 0≤r ,实数
structure: NULL
operations: area //计算面积 s= π r²
circumference //计算周长 l=2 π r
END ADT

？*思考：* 对几何“圆”，如何定义它的**ADT**?

1.5 抽象数据类型

例2. 复数的运算

问题描述： 在高级语言中，没有复数类型，但是我们可以借助已有的数据类型解决复数类型的问题，如复数运算。

ADT定义：

ADT *complex*

data : $z=c_1+c_2i$ *//c_1,c_2均为实数*

structure: NULL

operations: create(z,x,y) *//生成一复数*

add(z_1,z_2,s) *//复数加法*

subtract(z_1,z_2,difference) *//复数减法*

multiply(z_1,z_2,product) *//复数乘法*

get_realpart(z) *//求实部*

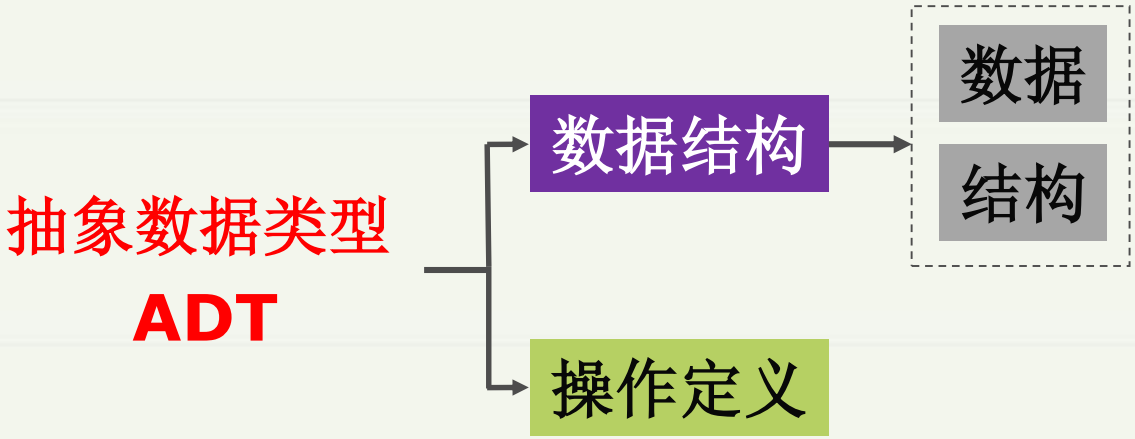
get_imagpart(z) *//求虚部*

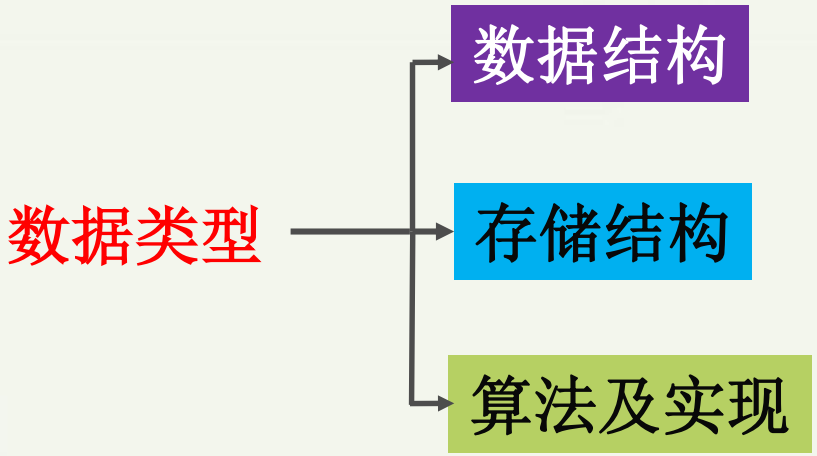
printc(z) *//输出一复数*

END ADT

1.5 抽象数据类型

- 抽象数据类型和数据类型





1.5 抽象数据类型

- 抽象数据类型和数据类型

一个数据类型的实现一般分为三个阶段：

1. ADT阶段，又称为定义阶段
2. 虚拟数据类型阶段，又称为表示阶段
3. 物理数据类型阶段，又称物理实现阶段

那么，高级语言中的数据类型是哪个阶段？

你过去编程解决问题是哪个阶段？

1.5 抽象数据类型

数据类型

ADT定义：

- 数据结构
- 操作（抽象）

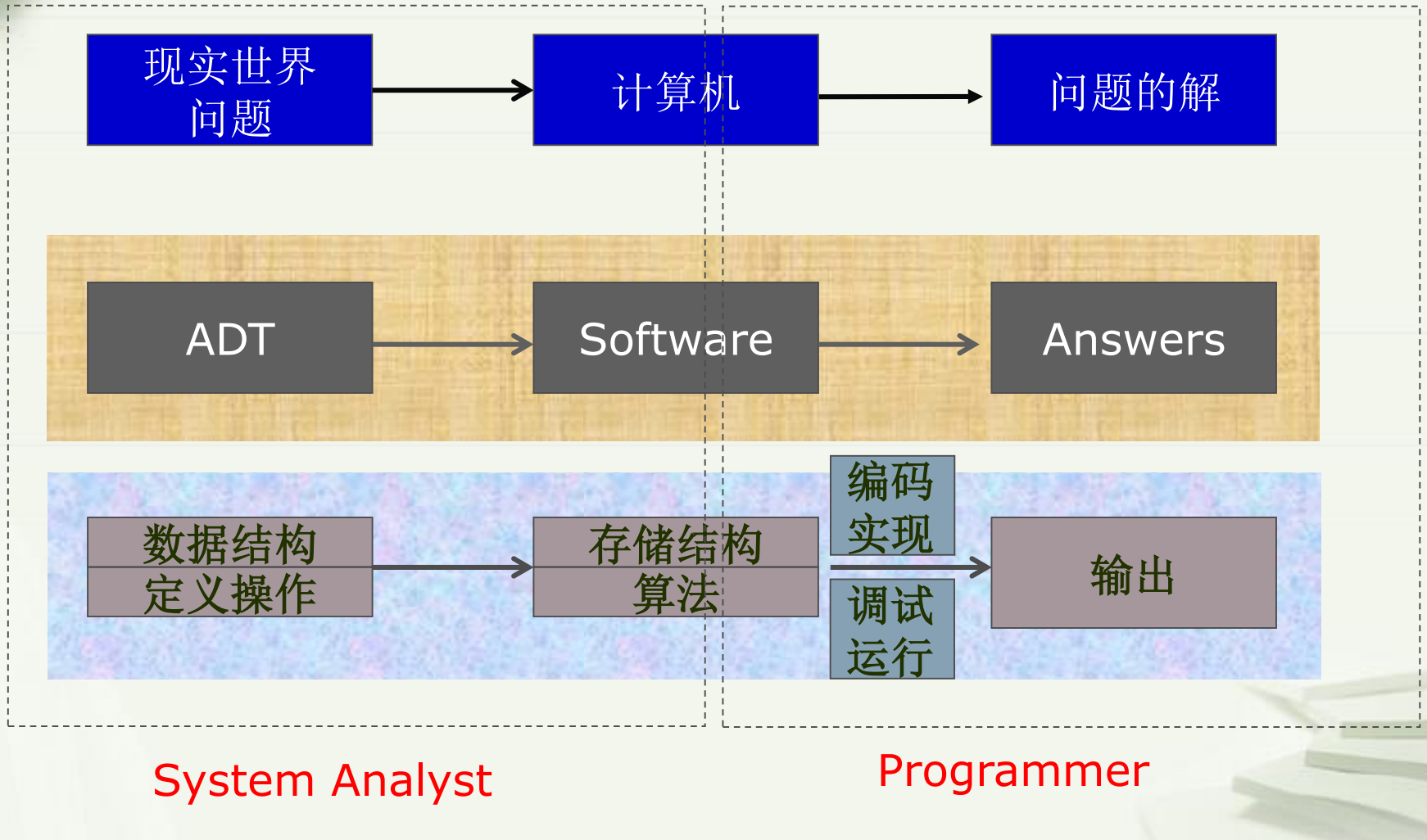
数据：逻辑形式

ADT实现：

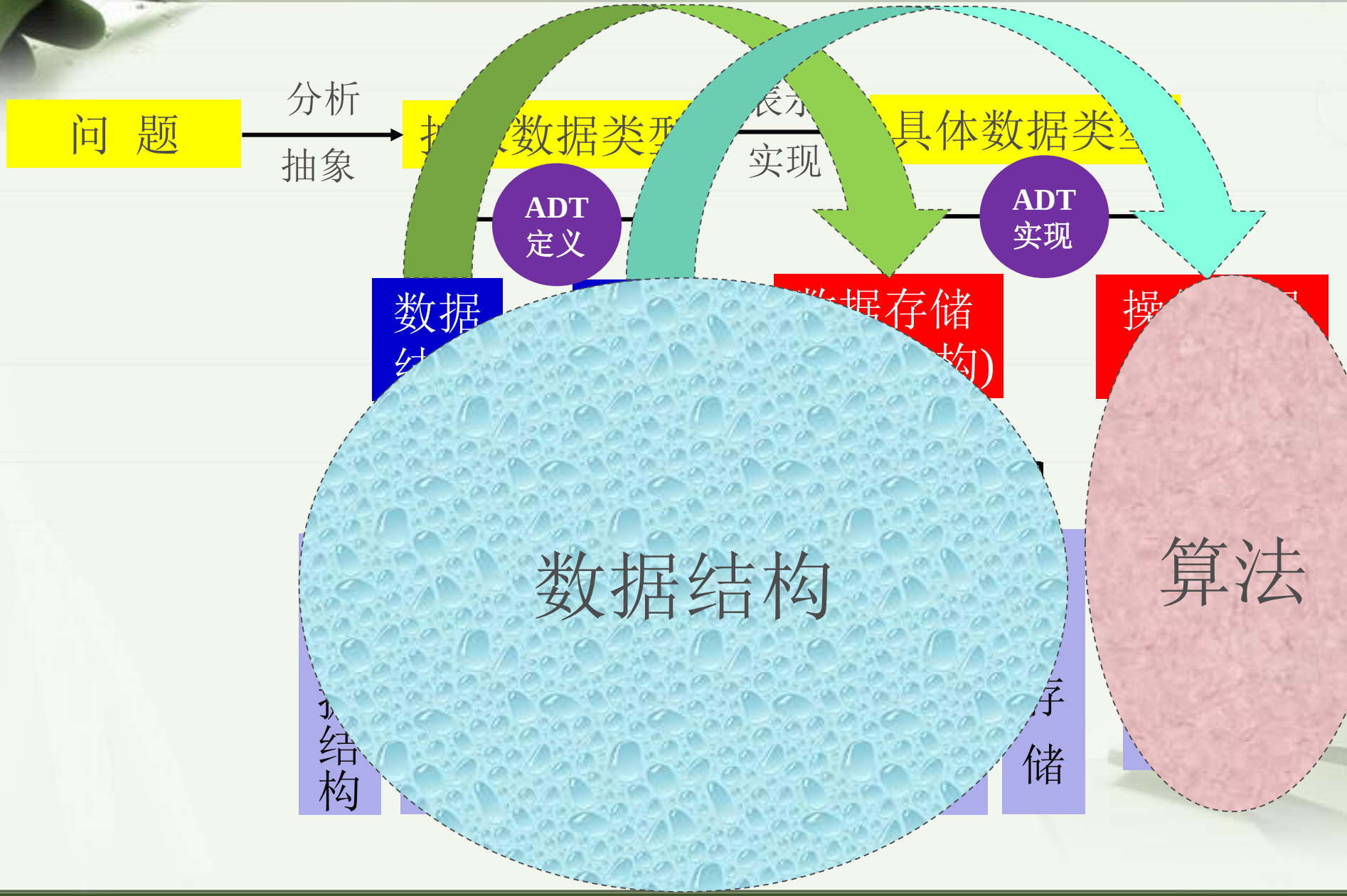
- 物理结构
- 算法 （具体）

数据：物理形式

1.5 抽象数据类型


现实世界
问题
计算机
问题的解
ADT
Software
Answers
数据结构
定义操作
存储结构
算法
编码
实现
调试
运行
输出
System Analyst
Programmer

1.5 抽象数据类型



1.6 ADT与C++类

抽象数据类型、面向对象、建模等概念在本质上是一致的！

▶ 抽象数据类型可以看作是描述问题的模型，它独立于具体实现。其优点是将数据和操作封装在一起，使得用户程序只能通过在ADT里定义的某些操作来访问其中的数据，从而实现了信息隐藏。

▶ 在C + + 中，可以用类（包括模板类）的说明来表示ADT，用类的实现来实现ADT。因此，C + + 中实现的类相当于是数据结构及其在存储结构上实现的对数据的操作。

▶ ADT和类的概念实际上反映了程序或软件设计的两层抽象：ADT相当于是在**概念层（或称为抽象层）**上描述问题，而类相当于是在**实现层**上描述问题。而对象或类的实例则可以看作是**应用层**。

关于 ADT与C++类，请同学们自学
（教材 1.3节）

本章小结

重点和难点：

1. 数据结构的重要性：研究问题本身的数据特性，并在计算机中存储，以实现问题的高效求解。

——目的是问题高效求解！

- 数据结构是基础要素
- 算法是关键要素
- 程序设计语言是工具

2. 基本概念：数据结构、存储结构、操作，及其之间的关系

- 数据结构的种类：
- 存储结构的种类：
- 存储如何映射数据结构

本章小结

重点和难点：

3. 算法的定义：由计算模型把输入转换为输出的步骤

4. 算法特性(要素):
   - 有限性
   - 确定性
   - 可行性
   - 输入和输出

5. 算法常见的设计策略：哪些？
   - 基本思想
   - 关键
   - 适用条件
   - 特点

本章小结

重点和难点：

6. 算法的评价标准：

正确性、易读性、健壮性、有效性

7. 算法复杂性分析：分析什么？如何分析？

- 事前还是事后？
- 空间性能如何评价？
- 时间性能如何评价？
- 渐近复杂性的由来

►经验测试　►事前评估

☐ 确定基准
☐ 累计求和T(n)
☐ 考察增长率
☐ 渐近表示O

本章小结

重点和难点：

- 渐近时间复杂性的本质

  解决同一问题的不同复杂性的算法，在计算机性能提高时，得到的增益是不同的。
  在相同时间内，不同算法可解问题的规模增加幅度不同；或者说，在问题规模增大时，时间的增幅有很大不同。

- 能基本掌握算法的分析方法

  ■确定基准——基本存储、基本操作
  ■求累加和
  ■渐近表示

本章小结

重点和难点：

8. 抽象数据类型、数据类型的概念

- 与数据结构、存储结构、操作的关系
- 抽象与建模
- 如何定义一个抽象数据类型
- 如何实现一个抽象数据类型

特点：

①封装与隐藏：

将数据和操作封装在一起，内部结构和实现细节对外屏蔽，实现信息隐藏

②抽象：

用户只能通过ADT里定义的接口和说明来访问和操作数据。

本章小结

重点和难点：

- **概念层（抽象）——ADT定义**
- **实现层——ADT实现**
- **应用层——调用ADT解决实际问题**

◆ **抽象数据类型实现与程序设计**

通过程序设计语言中的类型来实现数据存储，函数实现操作

- C
  - 抽象数据类型
  - 数据对象　　类型变量（多为结构体）
  - 基本操作　　函数
- C++，Java等
  - 抽象数据类型 类class
  - 数据对象　　数据成员
  - 基本操作　　成员函数(方法)


END