第一章 绪论

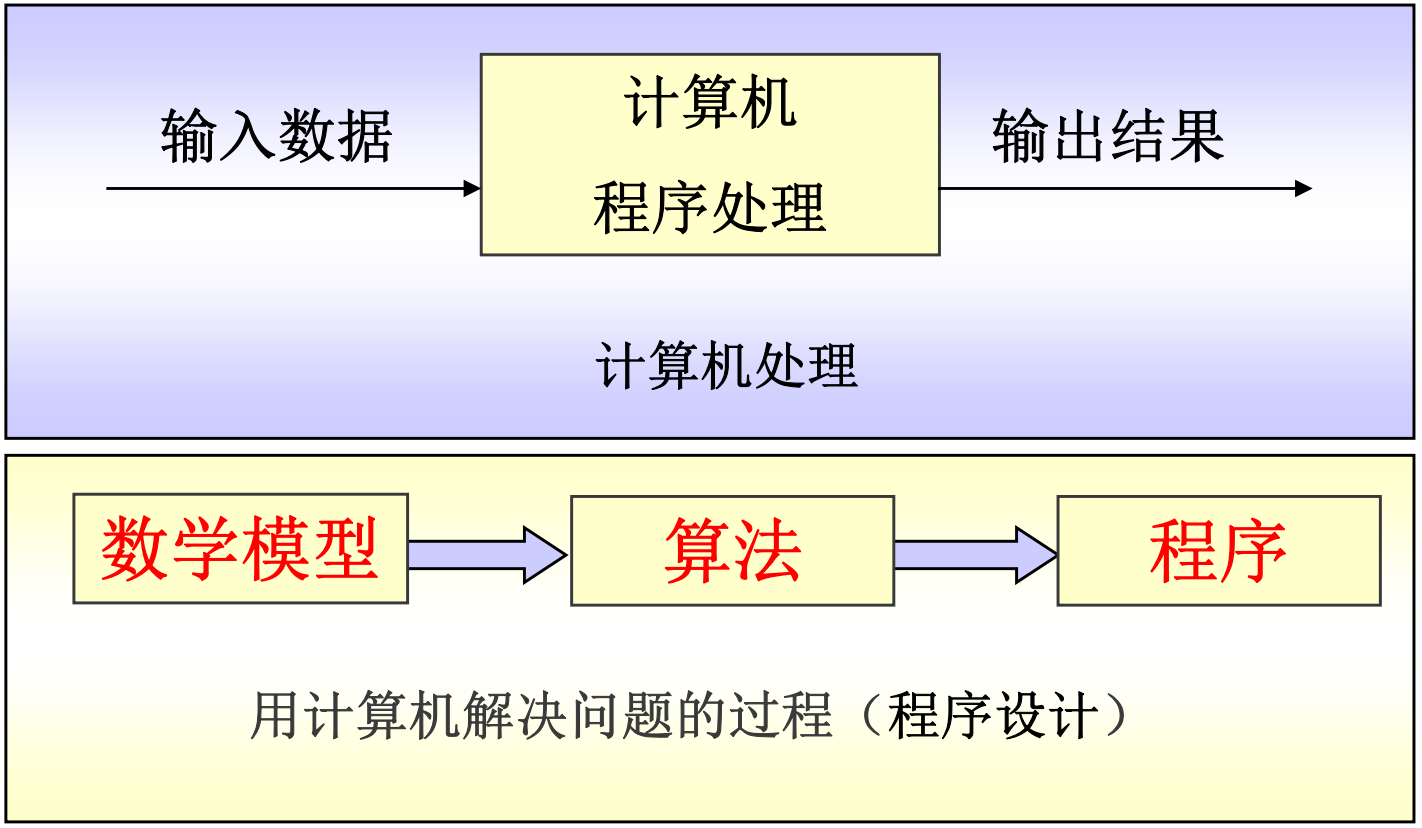
史树敏

bjssm@bit.edu.cn

中教#1006

List of Contents

1.1 数据结构的定义



- 程序设计: 为计算机处理问题编制一组指令集
- 算　　法: 处理问题的策略
- 数据结构: 数据的组织与操作

1.1 数据结构的定义

- Niklaus Wirth：
  - ¶ Programs = Algorithm + Data Structures

- **Data Structures =**
  Data Set + Relations + Operations

- 数据集+ 关系 + 操作

数据结构研究的主要内容

- 计算机处理的问题

  - 计算机处理数据的种类和能力
    - 数字 (整数，实数)
    - 字符、文字、图形、图象、声音
  - 计算机应用
    - 数值领域 → 数值计算问题
    - 非数值领域 → 非数值计算问题

数值计算问题举例

- 数值计算的程序设计问题
  - 例：物体从100米高的塔顶落到地面的时间——二次方程
  - 例：结构静力分析计算——线性代数方程组
- **建模**
  - **涉及对象**：高度h，时间t，重力加速度g （g=9.8）
  - **对象之间的关系**：$h = \frac{1}{2}gt^2$
- **设计求解问题的方法**：t=sqrt(2h/g)
- **编程**

```
main ( )
{       float t, h , g ; g=9.8;
        scanf (”%f”, &h);
        t = aqrt(2*h/g);
        printf (”The falling time is %f\n”, t);
}
```

非数值计算的程序设计问题1

- 例1.0：求一组整数(假设5个)中的最大值
- **建模**
  - **涉及对象**： 5个整数
  - **对象之间的关系**：大小关系
- **设计求解问题的方法**
  - 基本操作是“比较两个数的大小”
  - 首先将第一个数记为当前最大值，然后依次比较其余4个整数，如果该某个整数大于当前最大值，就更新当前最大值。
- **编程**

非数值计算的程序设计问题1

- 例1.0：求一组整数(5个)中的最大值

```
main ( )
{       int d[5], i, max;
        for( i=0; i<5; i++)
                scanf (”%d”, &d[i]);
        max = d[0];
        for( i=1; i<5; i++)
                if ( max < d[i]) max = d[i];
        printf (”The max number is %f\n”, max);
}
```

非数值计算的程序设计问题1

- 例1.1：求一组整数 (M个)中的最大值
- **建模**：
  - **涉及对象**： M个整数
  - **对象之间的关系**：大小关系
- **设计求解问题的方法**：
  - 基本操作是“比较两个数的大小”
  - 首先将第一个数记为当前最大值，然后依次比较其余m-1个整数，如果该某个整数大于当前最大值，就更新当前最大值。
- **编程**：

非数值计算的程序设计问题1

- 例1.1：求一组整数（M个)中的最大值

```
main ( )
{       int *d, M, i, max;
        scanf (”%d”, &M);
        d = (int *) malloc(sizeof(int) * M);
        for( i=0; i<M; i++)  scanf (”%d”, &d[i]);
        max = d[0];
       for( i=1; i<M; i++) if ( max < d[i]) max = d[i];
            printf (”The max number is %f\n”, max);
        free(d);  }
```

非数值计算的程序设计问题2

- 例2　已知研究生的选课情况，试设计安排课程的考试日程的程序。 要求在尽可能短的时间内完成考试。

A	B	C	D	E	F
算法分析	形式语言	计算机图形学	模式识别	网络技术	人工智能

	杨润生	石　磊	魏庆涛	马耀先	齐砚生
选修1	A	C	C	D	B
选修2	B	D	E	F	F
选修3	E		F	A	

非数值计算问题举例2: 建模

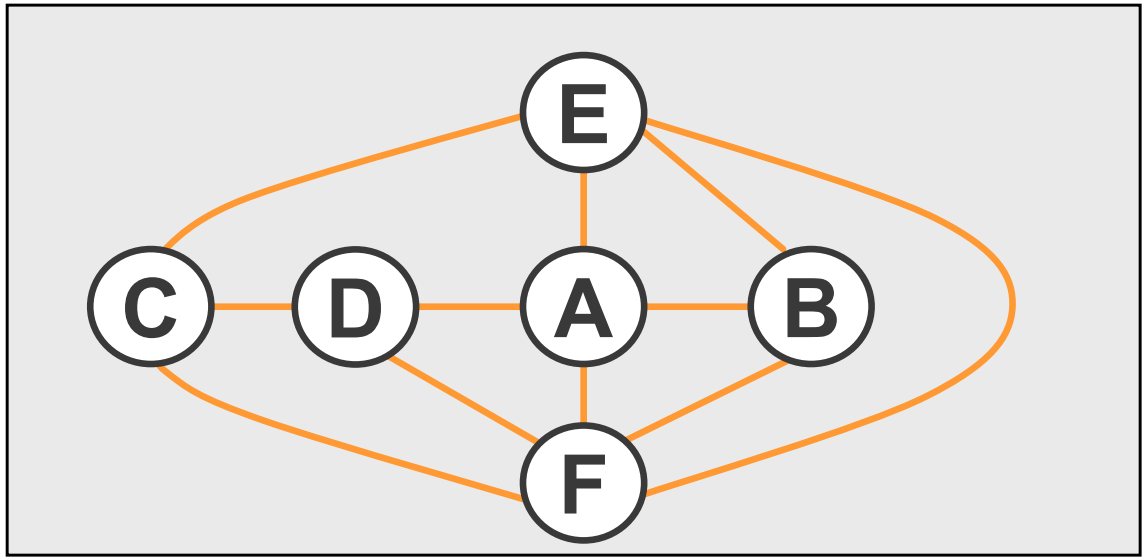
- **建立模型**
  - ¶ **涉及对象**：课程
  - ¶ **约束关系**：同一学生选修的课程不能安排在同一时间考试
  - ¶ **模型**：图——表达课程之间的约束关系

A	B	C	D	E	F
算法分析	形式语言	计算机图形学	模式识别	网络技术	人工智能

	杨润生	石 磊	魏庆涛	马耀先	齐砚生
选修1	A	C	C	D	B
选修2	B	D	E	F	F
选修3	E		F	A	

非数值计算问题举例2: 建模

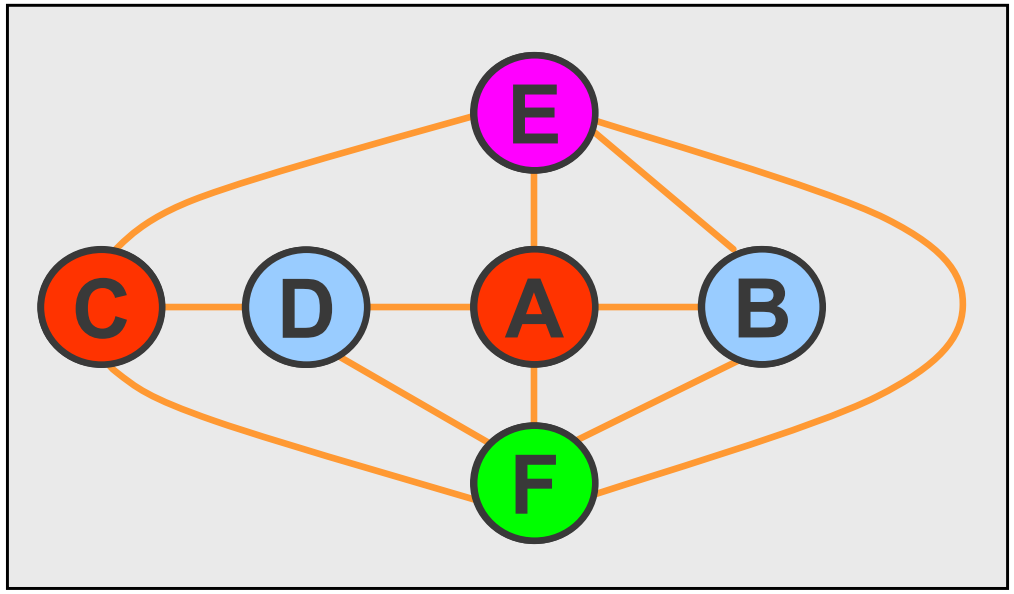
- **图**：顶点：表示课程；
- **边**：同一学生选修的课程用边连接，表示课程间的约束关系



	杨润生	石 磊	魏庆涛	马耀先	齐砚生
选修1	A	C	C	D	B
选修2	B	D	E	F	F
选修3	E		F	A	

非数值计算问题举例2: 求解（着色法）

- 每种颜色代表一个考试时间（用尽量少的颜色为顶点着色）；
- 着色原则：相邻顶点着不同颜色;不相邻顶点着相同颜色；
- 着相同颜色的顶点（课程）安排在同一时间考试；



不冲突课程组			
{ A, C}	{B, D}	{E}	{F}

考试日程：
第1天： A, C
第2天： B, D
第3天： E
第4天： F

非数值计算问题举例2: 求解（着色法）

- 求解考试日程的流程

1) *i*=1; V ={ 图中所有顶点的集合 }（*i*表示第*i* 天）

2) 若 V 非空 DO

2-1) 置 ***NEW*** 为空集合；

2-2) 在 V 中取一点，找出**所有**与之“不相邻”顶点；

2-3) 将这些顶点加入 ***NEW***，从 V 中去掉这些顶点；

（第 *i* 天考试课程为 ***NEW*** 中顶点所对应的课程）

2-4) 输出NEW中顶点所对应的课程；

2-5) *i* = *i* +1；

3)若 V 空，结束

非数值计算问题举例3

- 非数值计算的程序设计问题
- 例3：八数码问题
  - ¶ **涉及对象：** 棋盘→棋盘的格局
  - ¶ **对象关系：** 移动方格操作
  - ¶ **基本操作：空格** (上移，下移，左移，右移)

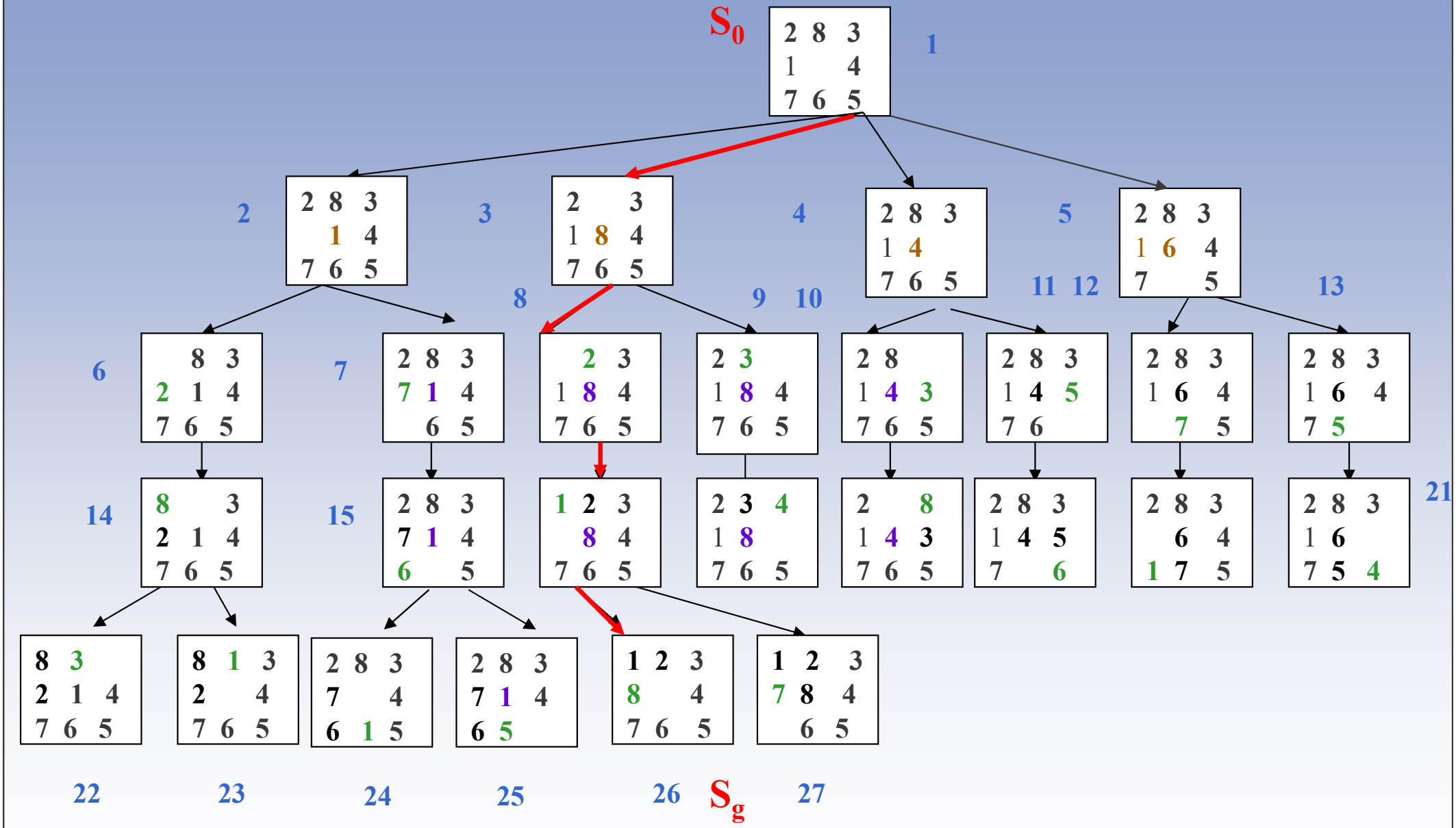
S_0

2	8	3
1		4
7	6	5

S_g

1	2	3
8		4
7	6	5

八数码问题



Route: $S_0 \rightarrow 3 \rightarrow 8 \rightarrow 16 \rightarrow 26(Sg)$

数值问题与非数值问题的程序设计比较

数值问题

对象：塔高度h，

下落时间t

——用数值表示

模型（对象之间的关系）

二元一次方程

——用方程表示

问题求解方法

计算方法

编程：

非数值问题

对象：课程

——用课程名表示

——*不能用数值表示*

模型（对象之间的关系）

课程间有“冲突”关系

——*不能用方程表示*

问题求解方法

……

编程：

数据结构研究的主要内容

- 概括地说
  - 数据结构是一门研究“**非数值计算的程序设计问题中计算机操作对象以及它们之间的关系和操作**”的学科。
- 具体地说
  - 数据结构主要研究数据之间有哪些结构关系，如何表示，如何存储，如何处理。

1.2 基本概念和术语

- 1.2.1 数据与数据结构
- 1.2.2 数据类型
- 1.2.3 抽象数据类型

1.2.1 数据与数据结构

- **数据（data）：**
  - 所有能被输入到或产生在计算机中，且能被计算机处理的符号的集合。
  - 是对计算机处理的对象的一个统称，被看作为信息的载体。
- **数据元素（data element）：**
  - 是数据结构中讨论的基本单位
  - 例如：描述一个学生的数据元素可以是下表结构
- **数据项（data item）：**
  - 数据不可分割的最小标识单位。
  - 一个数据元素可由若干数据项组成。

学号	姓名	出生日期	入学日期	班级	专业

数据在系统开发中的视图

- 数据在系统开发中的讨论范畴分为

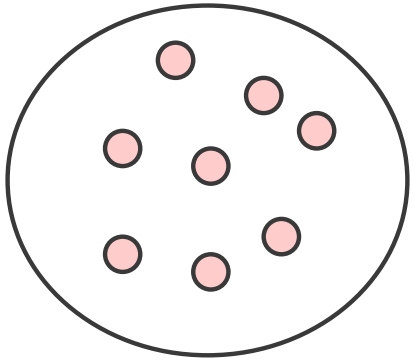
  数据结构 + 数据内容 + 数据流

- **数据结构**：指某一数据元素集合中数据元素之间的关系。
- **数据内容**：指这些数据元素的具体涵义和内容。
- **数 据 流**：指这些数据元素在系统处理过程中是如何传递和变换的。

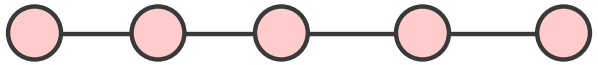
- 因此，讨论数据结构时，主要不是讨论数据元素的内容和如何处理。

1.2.1 数据与数据结构

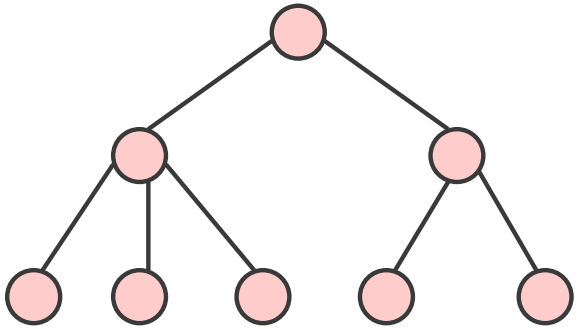
- 数据结构：带有关系和运算的数据集合



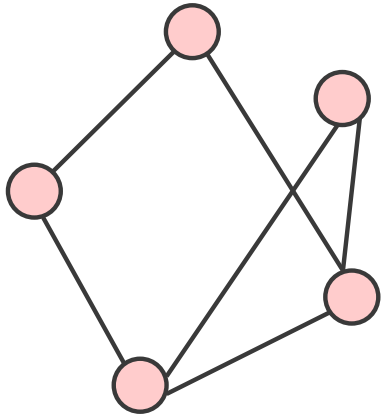
a. 集合关系



b. 线性关系



c. 树型关系



d. 图型关系

数据结构

- 数据结构：带有关系和运算的数据集合
- 数据之间结构关系：是具体关系的抽象。例1：

学生间学号顺序关系是一种线性结构关系

线性结构关系是对学生间学号顺序关系的一种抽象表示

学号	姓名	出生日期	入学日期	班级	专业
cs001	张扬	02122001	01092019	cs20141	计算机
cs002	李明	07112001	01092019	cs20141	计算机
cs003	杨华	01052001	01092019	cs20142	计算机
cs004	贾茹	15042001	01092019	cs20142	计算机

数据结构

- 数据结构：带有关系和运算的数据集合
- 例1：定义学生数据结构中的运算（基本操作）：
  - 查询学生信息
  - 插入学生信息
  - 修改学生信息
  - 删除学生信息

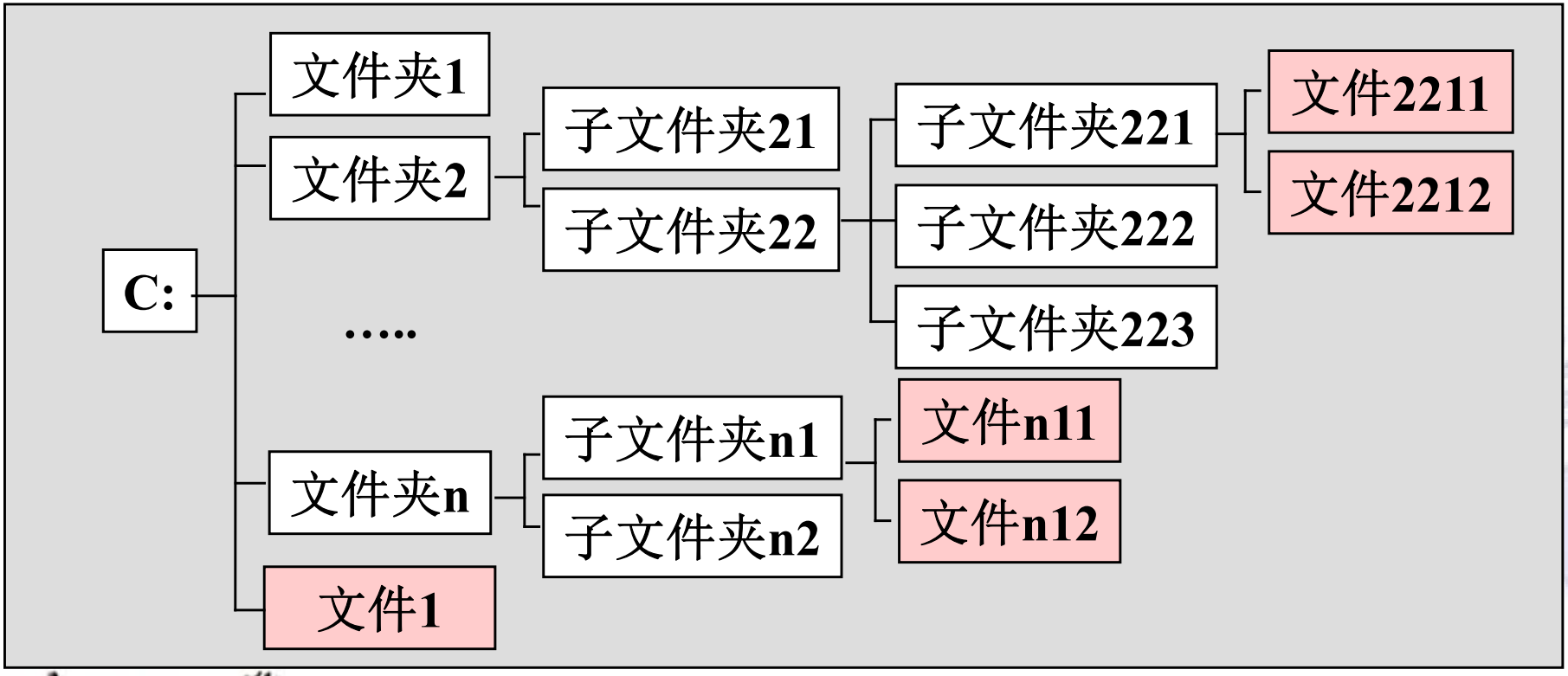
学号	姓名	出生日期	入学日期	班级	专业
cs001	张扬	02122001	01092019	cs20141	计算机
cs002	李明	07112001	01092019	cs20141	计算机
cs003	杨华	01052001	01092019	cs20142	计算机
cs004	贾茹	15042001	01092019	cs20142	计算机

数据结构

- 例2：计算机文件系统

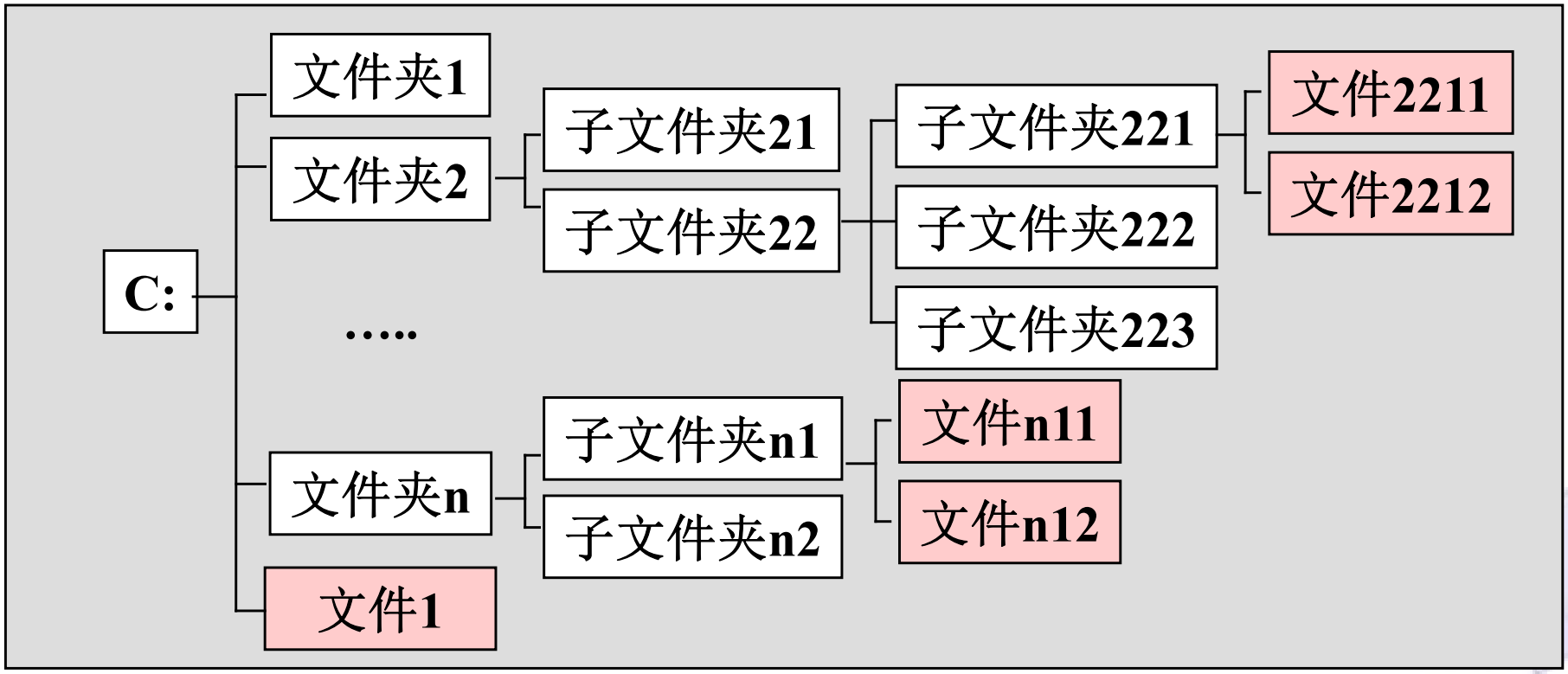
文件夹或文件间的关系是一种树型结构关系

树型结构关系是对文件系统关系的一种抽象表示

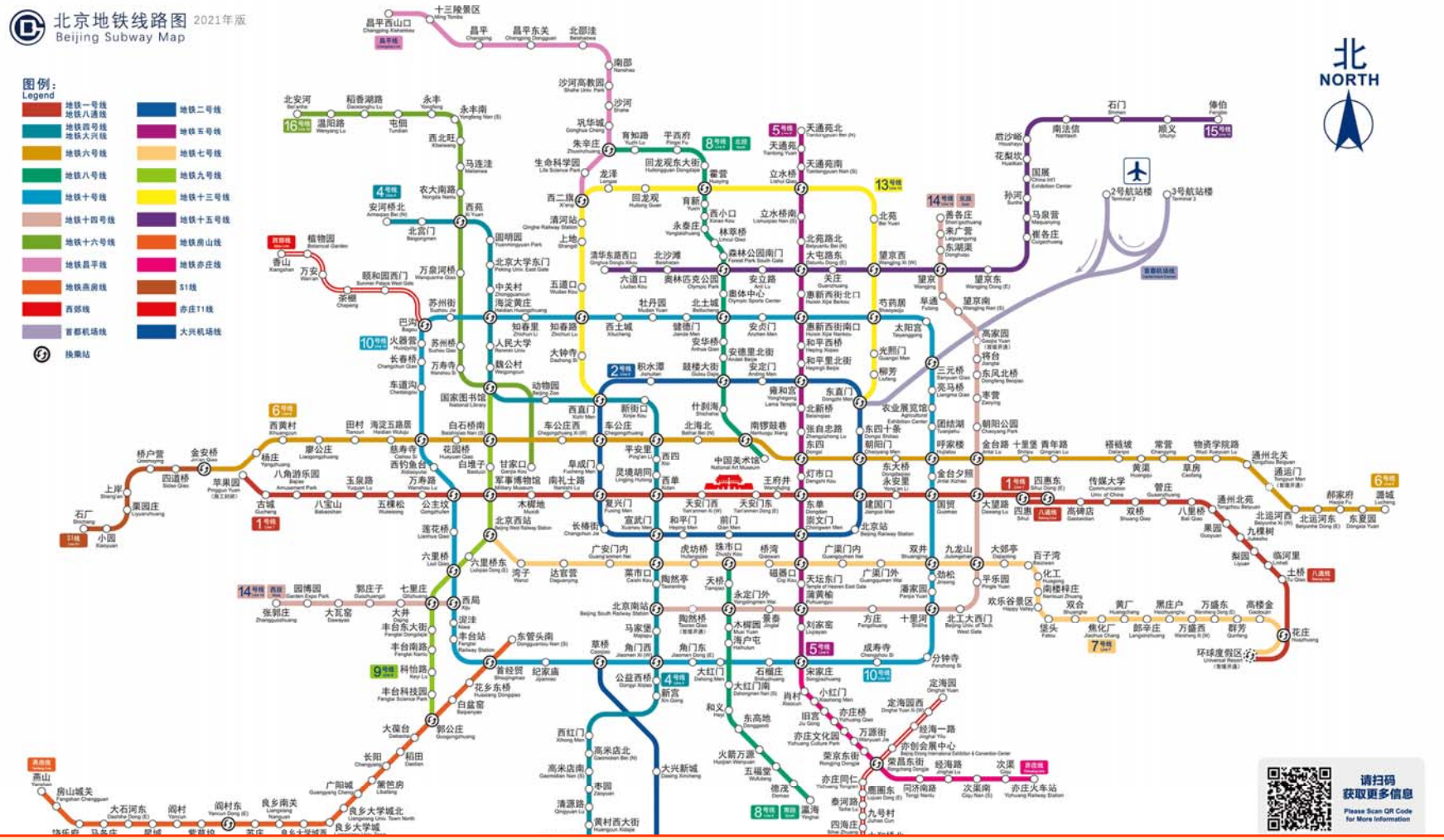


数据结构

- 例2：定义计算机文件系统中的操作
  - 查找、新建、删除文件夹
  - 查找、新建、删除文件



数据结构 例3：地铁线路图



地铁站之间的连接关系是一种图型结构关系

图型结构关系是对地铁站之间的连接关系的一种抽象表示

数据结构

- 定义地铁图中的操作：
  - 查询地铁站
  - 关闭地铁站...

不同的“关系”构成不同的“数据结构”。

这些关系称为数据的逻辑结构。

数据结构是相互之间存在着某种逻辑关系的

数据元素集合及定义在该集合上的操作集合

数据结构的表示

- 图示表示

  图示表示是由顶点和边构成的图，其中顶点表示数据，边表示数据之间的结构关系。

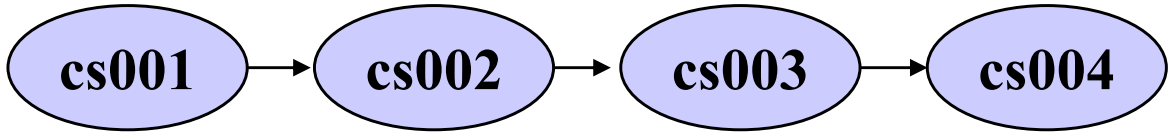
- 二元组表示：
- 数据结构是一个二元组：

  Data_Structures = (D, S)

  - **D 是数据元素的有限集，**
  - **S 是 D上关系的有限集。**

数据结构的表示

- 线性关系的表示



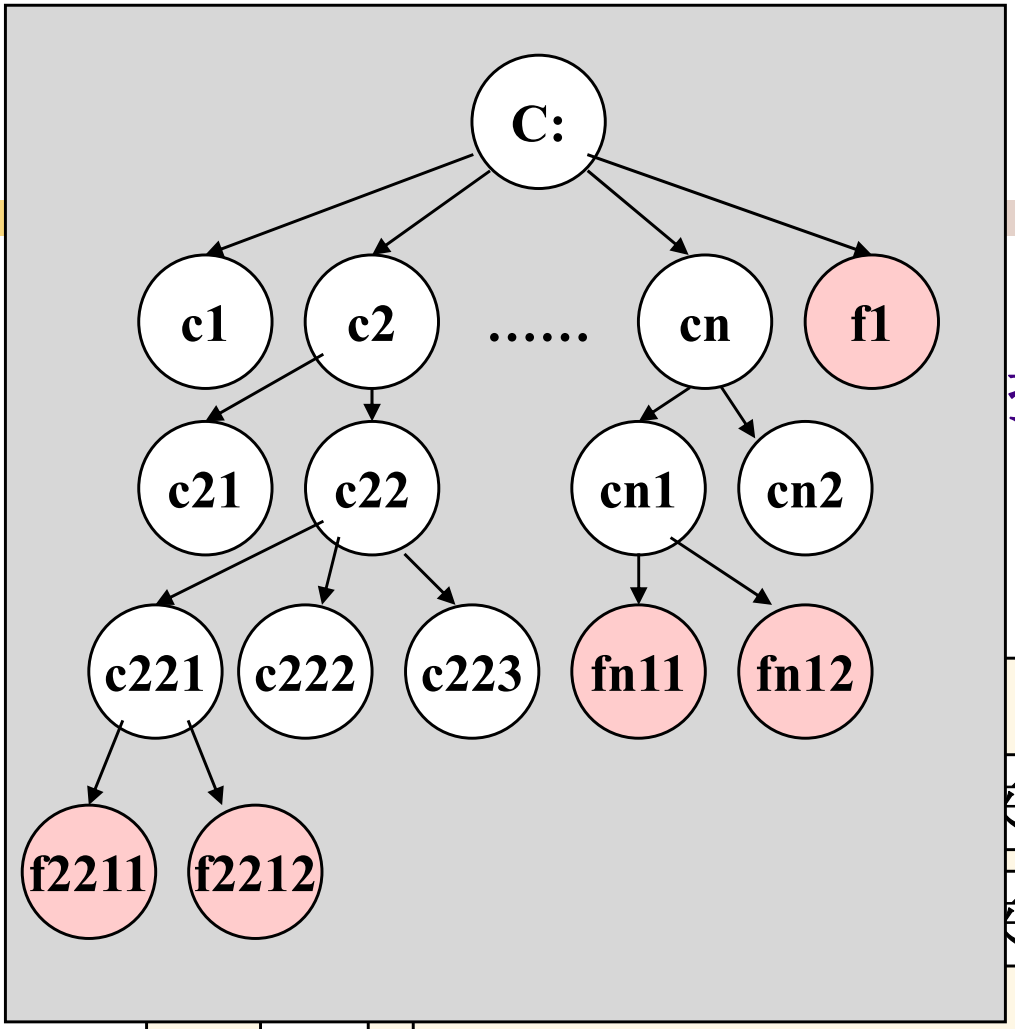
- 集合关系的表示

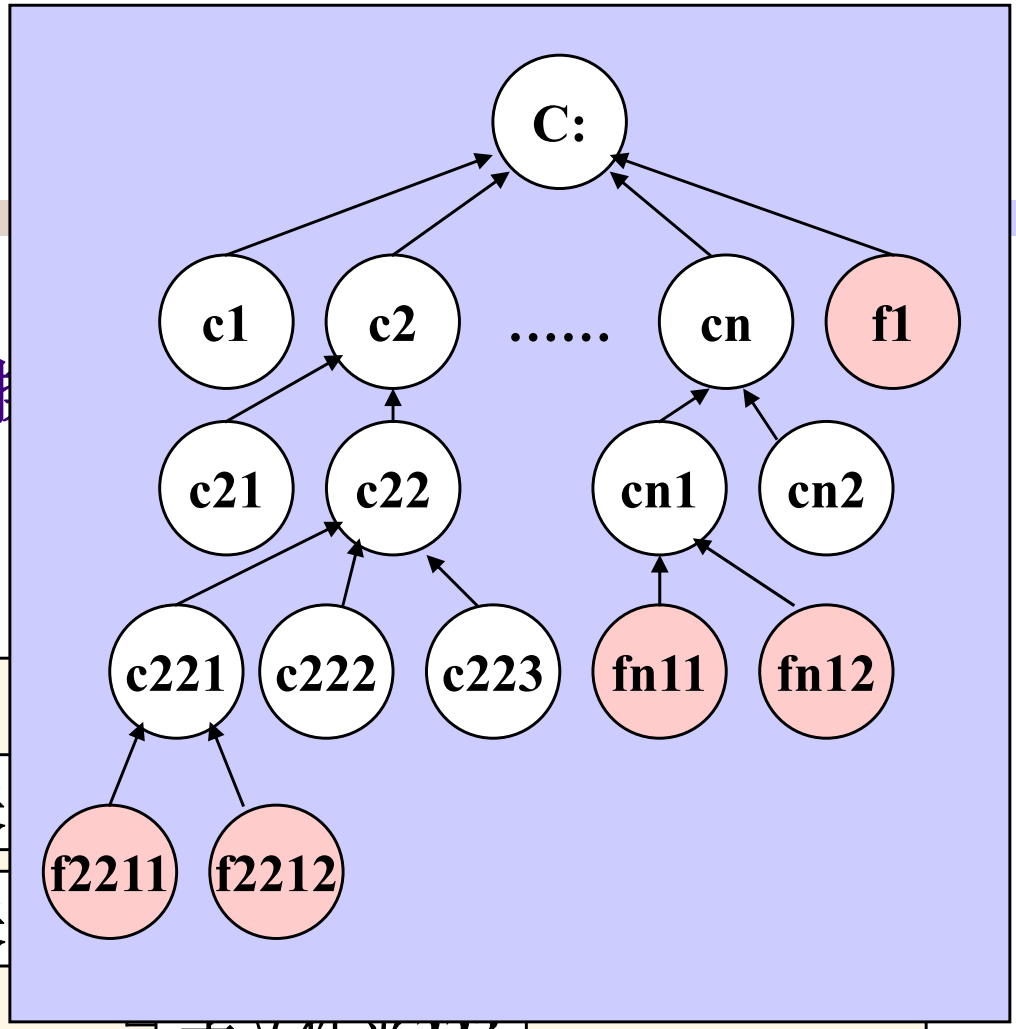
D = { cs001，cs002，cs003，cs004}

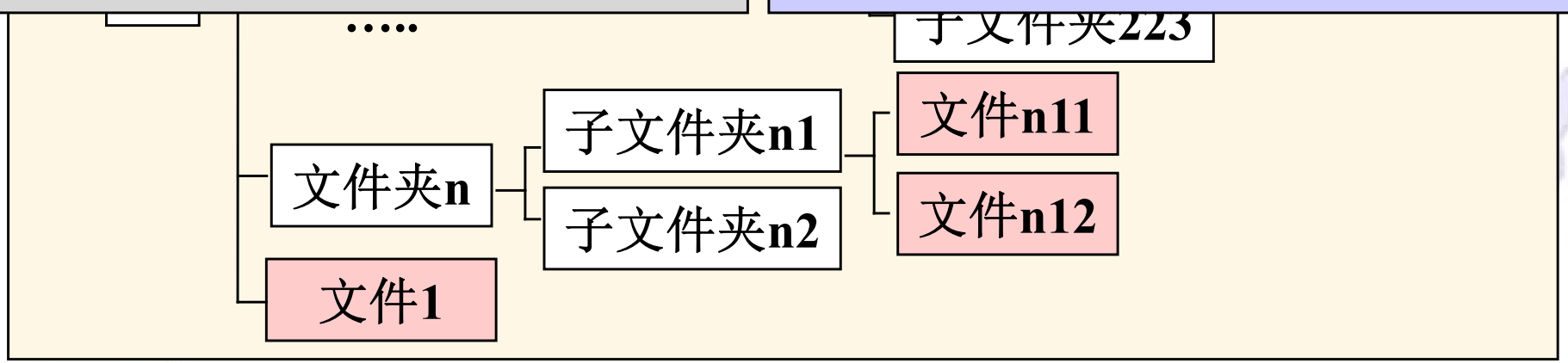
S = { R }

R = { <cs001,cs002>, <cs002,cs003>, <cs003,cs004>}

学号	姓名	出生日期	入学日期	班级	专业
cs001	张扬	02122001	01092019	cs20141	计算机
cs002	李明	07112001	01092019	cs20141	计算机
cs003	杨华	01052001	01092019	cs20142	计算机
cs004	贾茹	15042001	01092019	cs20142	计算机


C:
c1
c2
……
cn
f1
c21
c22
cn1
cn2
c221
c222
c223
fn11
fn12
f2211
f2212


C:
c1
c2
……
cn
f1
c21
c22
cn1
cn2
c221
c222
c223
fn11
fn12
f2211
f2212


……
子文件夹223
文件夹n
子文件夹n1
子文件夹n2
文件n11
文件n12
文件1

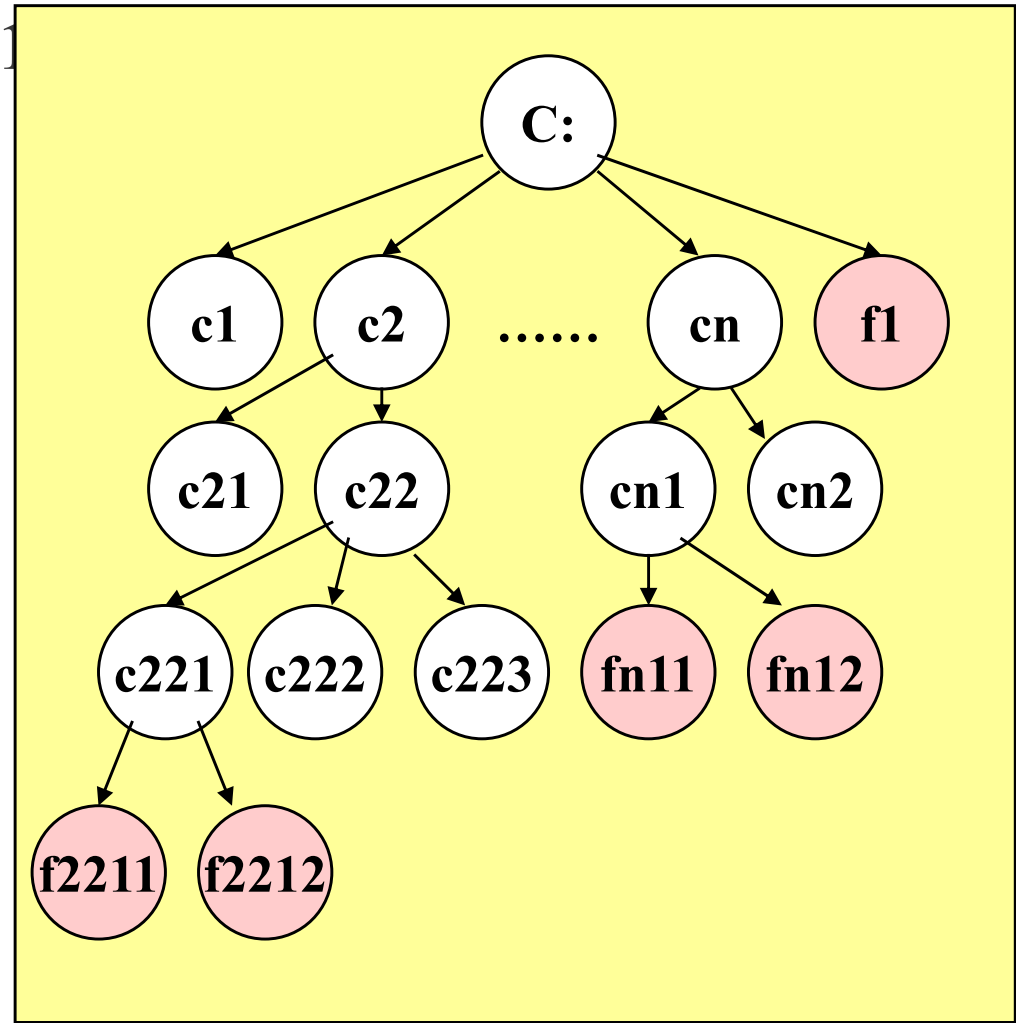
数据结构的表示

- 树型关系的表示

D = { C， c1, c2,…, cn, f1, c21, c22, cn1, cn2, c2221, c222,c222, c223, fn11, fn12, f221

S = { R }

R = { <C,c1>, <C,c2>, …, <C,cn>, <C,f1>, …… }



数据的逻辑结构

- 数据的逻辑结构抛弃了数据元素及其关系的实现细节，只反映了系统的需求（从用户使用层面上看），而不考虑如何实现。
  - 数据的逻辑结构从逻辑关系上描述数据，与数据的存储无关；
  - 数据的逻辑结构与数据元素的内容无关；
  - 数据的逻辑结构与其所包含的数据元素个数无关；
  - 数据的逻辑结构与其数据元素所处的位置无关。

数据的存储结构

- 数据的存储结构

  —— 逻辑结构在存储器中的映象

  算法的设计更多地取决于所选定的逻辑结构，
  算法的实现则更依赖于采用的存储结构。

- 存储结构包含两个方面：
  - “数据元素”的映象
  - “关系”的映象

数据的存储结构

- 数据的存储结构是逻辑结构用计算机语言的实现，通常**存储结构有三个任务**：
  - ¶ 内容存储：存储各数据元素的内容或值，每个元素占据独立的存储空间。
  - ¶ 关系存储：直接或间接地，显式或隐式地存储各数据元素之间的逻辑关系。
  - ¶ 附加存储：为使施加于数据结构上的运算得以实现而附加的存储空间。

数据的存储结构

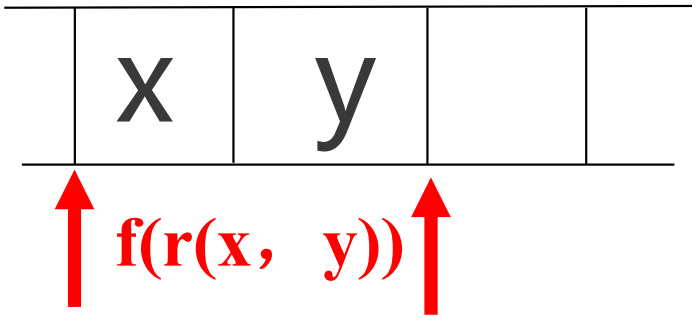
- 存储结构分类
  - 顺序存储：用一个连续的空间来存储数据元素。常用一维或二维数组。
  - 链式存储：不必事先分配空间，在运行时动态分配存储空间，链入到系统中。
- 常用于内存中的算法或程序。
  - 索引方式：用于对索引文件的实现。
  - 散列方式：用于直接存取文件的实现。
- 常用于外存中的算法或程序。

数据的存储结构—数据元素的映象

- 用二进制位(bit)的位串表示数据元素
  - $(321)_{10} = (101000001)_2$
  - A $= (001000001)_2$
  - 图像
  - 图形
  - 声音

数据的存储结构—关系的映象

- 即如何表示两个元素之间的关系<x, y>
- 方法一：顺序映象（顺序存储结构）
  - 用数据元素在存储器中的相对位置来表示数据元素之间的逻辑关系
- 如:令 y 的存储位置和 x 的存储位置间差一个常量 C



y 的存储位置可以根据 x 的存储位置， y和x的关系，直接进行计算：

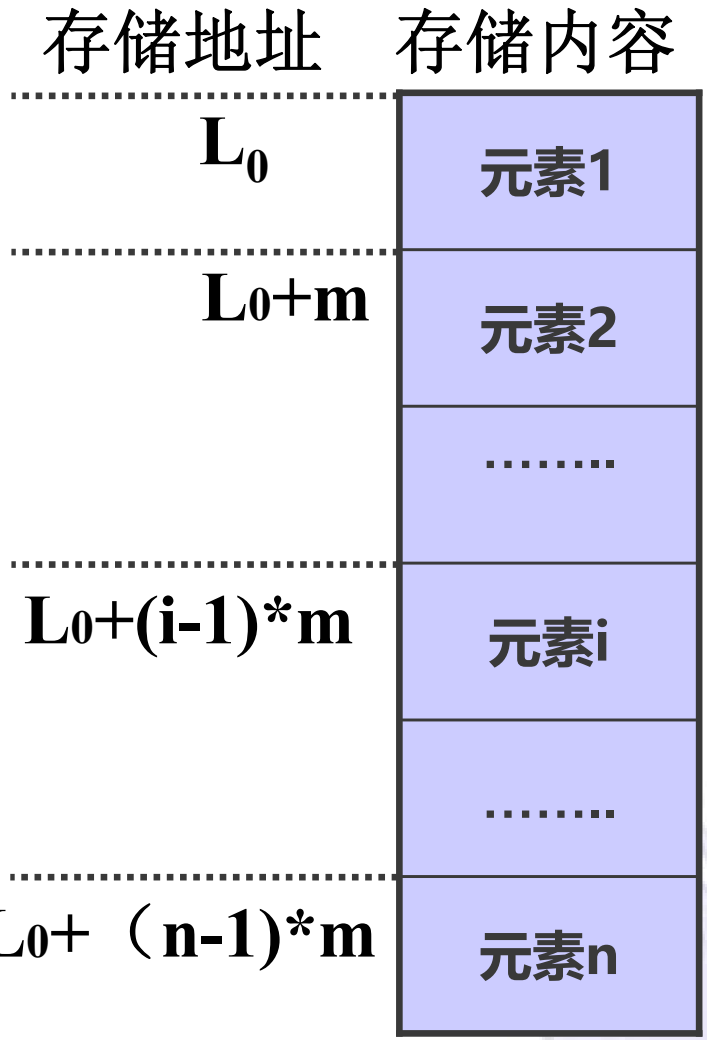
$$Loc(y) = Loc(x) + f(r(x，y))$$

数据的存储结构—关系的映象

◆顺序存储结构

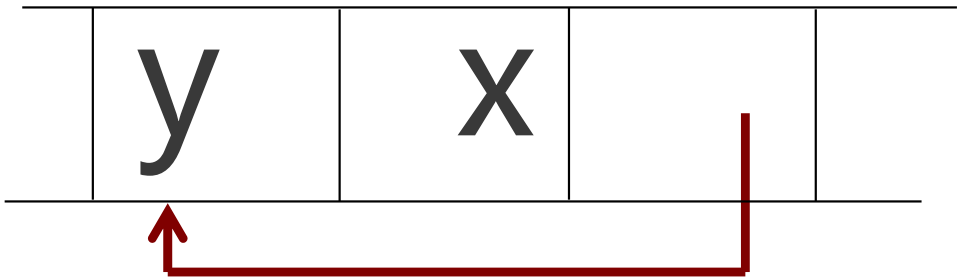
Loc(元素i) =

$L_0 + (i-1) * m$



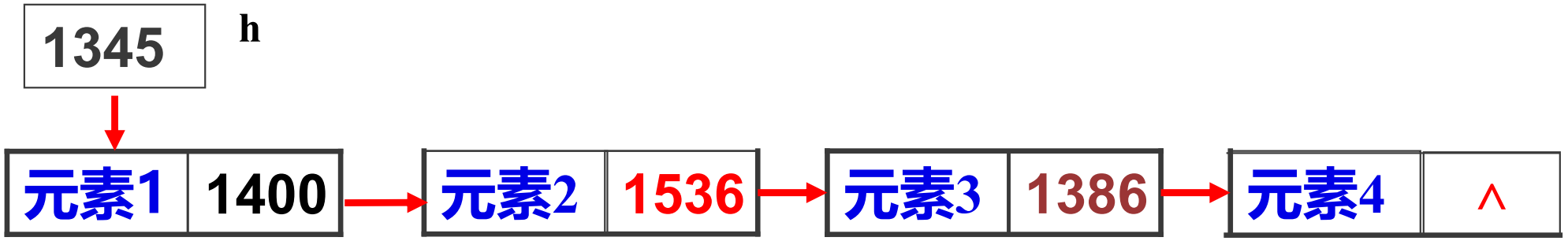
数据的存储结构—关系的映象

- 方法二：链式映象（链式存储结构）
  - 以附加信息(指针)表示数据关系
  - 需要用一个和 x 在一起的附加信息指示 y 的存储位置



数据的存储结构—关系的映象

◆链式存储结构



存储地址	存储内容	指针
1345	元素1	1400
1386	元素4	∧
…….	……..	…….
1400	元素2	1536
…….	……..	…….
1536	元素3	1386

数据的存储结构—关系的映象

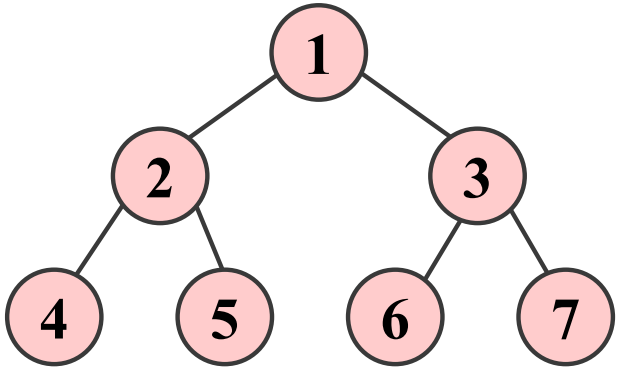
◆链式存储结构

借助指示元素存储地址的指针来表示数据元素之间的逻辑关系。

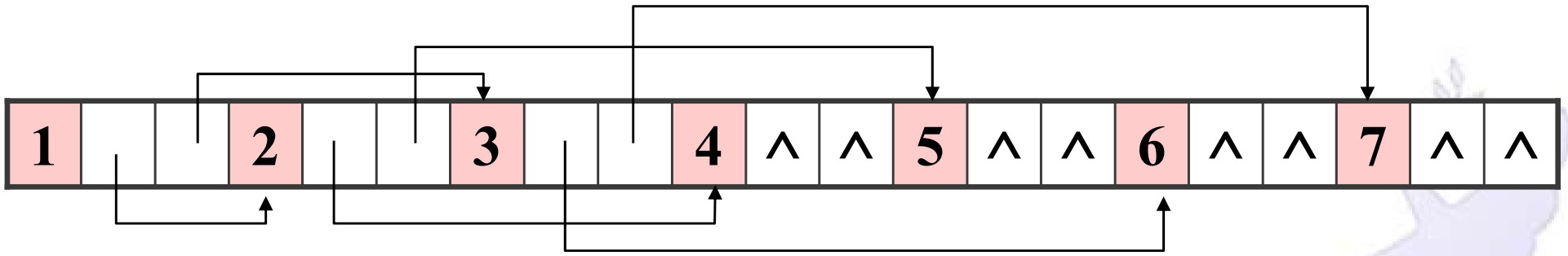
- ¶ 在结点的存储结构中附加指针字段，两个结点的逻辑后继关系用指针的指向来表达。
- ¶ 任意的逻辑关系，均可使用这种指针地址来表达。一般将数据结点分为两部分：
  - ▸ 存放数据，称为数据字段
  - ▸ 存放指针，称为指针字段

数据的存储结构—关系的映象

- 树型结构的链式存储结构



树型结构



数据的存储结构

- 不同的编程环境中，存储结构可有不同的描述方法。
- 当用高级程序设计语言进行编程时，通常可用高级编程语言中提供的数据类型加以描述。

- 例如：以三个带有次序关系的整数表示一个**长整数**时，可利用 C 语言中提供的整数数组类型。

定义长整数为:

typedef int *Long_int [3]*

1.2.2 数据类型

- 数据类型：在一种程序设计语言中，变量所具有的数据种类。
- 例1： FORTRAN语言中
  - 变量的数据类型有整型、实型和复数型 …
- 例2：C语言中数据类型：基本类型和构造类型
  - 基本类型：整型、浮点型、双精度、字符型、…
  - 构造类型：数组、指针、结构、联合、枚举型、自定义

1.2.2 数据类型

- 不同类型的变量，其取值范围不同，所能进行的操作不同
- 数据类型：是一个值的集合和定义在此集合上的一组操作的总称。
- 如何处理不同语言中的数据类型呢？

1.2.3 抽象数据类型

- 抽象数据类型（Abstract Data Type 简称ADT）

ADT是对数据结构的一种更准确的抽象描述，它忽略了数据结构的具体实现步骤，将更多的注意力放在数据的基本操作上，通过基本操作描述数据的逻辑关系。

抽象数据类型的定义格式

ADT 抽象数据类型名 {

数据对象：〈数据对象的定义〉

数据关系：〈数据关系的定义〉

基本操作：〈基本操作的定义〉

} ADT 抽象数据类型名

基本操作名（参数表）

初始条件：〈初始条件描述〉

操作结果：〈操作结果描述〉

抽象数据类型的定义格式

基本操作名（参数表）

初始条件：〈初始条件描述〉

操作结果：〈操作结果描述〉

- 参数表
  - ¶ 赋值参数 只为操作提供输入值。
  - ¶ 引用参数 以&打头，除可提供输入值，还将返回操作结果。
- 初始条件 描述了操作执行之前数据结构和参数应满足的条件，若不满足，则操作失败，并返回相应出错信息。
- 操作结果 说明了操作正常完成之后，数据结构的变化状况和应返回的结果。

练习

在下列结论中只有一个是正确的，它是：

A. 递归函数中的形式参数是自动变量

B. 递归函数中的形式参数是外部变量

C. 递归函数中的形式参数是静态变量

D. 递归函数中的形参可根据需要自己定义存储类型

§9-2 指针与函数（review）

三、指向函数的指针

知识点补充（★）

- **代码段**（**二进制代码**，代码段内容为只读，可以被多个进程共享，一个子进程和他的父进程是共享这个代码段的）
- **数据段**（存储:程序中被初始化的变量，其中包括**全局变量和初始化的静态变量**）
- **未初始化数据段**(顾名思义,存储**没有初始化的静态变量**，俗称bss段)
- **堆heap**（存储: 程序运行中**动态分配的变量**）
- **栈stack**（存储:**函数调用时保存**函数的返回值、参数和函数内部的一些局部变量）
- **命令行参数和环境变量**

ADT举例

- 例如: 抽象数据类型“复数”的定义:

ADT Complex {

数据对象: D＝{e1,e2 | e1,e2 ∈RealSet }

数据关系: R1＝{<e1, e2> | e1是复数的实数部分
| e2 是复数的虚数部分 }

基本操作:

AssignComplex(&Z, v1, v2)

操作结果：构造复数 Z，其实部和虚部分别被赋以参数 v1 和 v2 的值

DestroyComplex(&Z)

操作结果：复数Z被销毁。

ADT举例（CONT.）

GetReal(Z, &realPart)

初始条件：复数已存在。

操作结果：用**realPart**返回复数**Z**的实部值。

GetImag(Z, &ImagPart)

初始条件：复数已存在。

操作结果：用**ImagPart**返回复数**Z**的虚部值。

Add(z1, z2, &sum)

初始条件：**z1, z2**是复数。

操作结果：用**sum**返回两个复数**z1, z2** 的和值

} ADT Complex

ADT 的特征

- **特征一：数据抽象**
  - 用ADT描述程序处理的实体时，强调的是其本质的特征、其所能完成的功能以及它和外部用户的接口（即外界使用它的方法）。
- **特征二：数据封装**
  - 将实体的外部特性和其内部实现细节分离，并且对外部用户隐藏其内部实现细节。

ADT 的实现

- ADT需要通过固有数据类型(高级编程语言中已实现的数据类型)来实现
- 例如对上述复数ADT的实现：

```
//存储结构的定义
typedef  struct {
    float realpart;
    float imagpart;
} complex;
```

```
// -----基本操作的函数原型说明
void Assign( complex &Z, float realval, float imagval );
// 构造复数 Z,其实部和虚部分别被赋以参数
// realval 和 imagval 的值
float GetReal( complex Z );  // 返回复数 Z 的实部值
float Getimag( complex Z );  // 返回复数 Z 的虚部值
void add( complex z1, complex z2, complex  &sum );
 // 以 sum 返回两个复数 z1, z2 的和
```

ADT 的实现（CONT.）

```
//// -----基本操作的实现
void add( complex z1, complex z2, complex  &sum ) {
 // 以 sum 返回两个复数 z1, z2 的和
  sum.realpart = z1.realpart + z2.realpart;
  sum.imagpart = z1.imagpart + 2.imagpart;
}
```

预定义常量和类型

- 函数结果状态代码
  - #define TRUE 1
  - #define FALSE 0
  - #define OK 1
  - #define ERROR 0
  - #define INFEASIBLE -1
  - #define OVERFLOW -2
- Status 函数返回类型
  - typedef int Status;

注：课程中使用类C语言语法,请认真看1.2节

3 算法的分析和度量

- 算法及其特征
- 算法设计的原则
- 算法效率的衡量方法和准则

算法

- 算法：是为了解决某类问题而规定的一个有限长的操作序列。
- 问题求解：将问题递归性的分解为一个或者多个小问题，再综合子问题的解决方案。

```
void mult(int a[ ], int b[ ], int& c[ ] ) {
// 以二维数组存储矩阵元素，c 为 a 和 b 的乘积
  for(i=1; i<=n; ++i)
    for(j=1;j<=n;++j)
    {
          c[i][j]=0;
          for(k=1;k<=n;++k)
              c[i][j]+=a[i][k]*b[k][j];
    }
```

描述算法的工具

- 自然语言：易理解，但不精确，易有二义性
- 约定的符号描述：
  - 流程图（直观清晰并且比较精确，但是不容易实现）
  - 伪代码（较严谨且简洁，易用程序实现）
- 计算机高级语言描述：
  - 最终形式：**C、Pascal、C++、Java…**

算法及其特征

- **输入（Inputs）**有 0 个或多个输入
- **输出（Outputs）**有一个或多个输出(处理结果)
- **有穷性（ finiteness ）**
  - 在执行有穷步骤之后一定能结束
  - 每个步骤都能在有限时间内完成
- **确定性（ determinative ）**
  - 每一条指令必须有确切的含义, 不存在二义性。
  - 只有一个入口和确定性的出口(任何情况下, 执行路径唯一)
- **可行性（ feasibility ）**
  - 算法是可行的。
  - 算法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。（可计算问题，计算理论）

算法设计的原则

- 评价一个好的算法有以下几个标准：

1. **正确性(Correctness)**
   算法应满足具体问题的需求。每一个算法，要考虑前置条件和后置条件。
2. **可读性(Readability)**
   算法应该好读。以有利于阅读者对程序的理解。(命名规范、层次结构、注释。。。)
3. **健壮性(Robustness)**
   算法应具有容错处理。当输入非法数据时，算法应对其作出反应，而不是产生莫名其妙的输出结果
4. **效率与存储量需求**
   效率指的是算法执行的时间；存储量需求指算法执行过程中所需要的最大存储空间。一般，这两者与问题规模有关

算法的评价标准

- 对算法是否“正确”的理解可以有以下四个层次：
  - a. 程序中不含语法错误；
  - b. 程序对于几组输入数据能够得出满足要求的结果；
  - c. 程序对于精心选择的、典型、苛刻且带有刁难性的几组输入数据能够得出满足要求的结果；
  - d. 程序对于一切合法的输入数据都能得出满足要求的结果；
- 通常以第 c 层意义的正确性作为衡量一个算法是否合格的标准。

算法设计的基本方法（算法策略）

- 穷举法：逐个检查所有可能的解，直到找到解
- 迭代法：逐步逼近可能的解
- 贪心法：分步完成，局部最优得到整体最优
- 回溯法：彻底搜索，深度优先，试探所得，剪枝
- 分支界限法：彻底搜索，广度优先，试探求得
- 分治法：问题规模缩小，分而治之，如折半查找
- 动态规划法：问题分解（缩小规模），得到各个分解结果，再自底往上求最后结果]

- A*算法
- 遗传算法、进化计算
- 蚁群算法、粒子群算法、支持向量机、……

算法的分析和度量

- 算法效率的衡量方法和准则
- 算法的存储空间需求

算法效率的衡量方法和准则

- 和算法执行时间相关的因素：
  - 1. 计算机执行指令的速度
  - 2. 编译程序产生的机器代码的质量
  - 3. 编写程序的语言
  - 4. 问题的规模
  - 5. 算法选用的策略（折半 Vs 彻底扫描）

不能用诸如微秒、纳秒等真实时间计量

- 硬件架构：大型机 vs 微机
- 机器速度：Cray vs PC
- 编程语言：C vs LISP
- 数据规模：100 vs 10000

算法效率的衡量方法和准则

算法效率的度量

- 程序中所用算法运行时所要花费的时间代价
- 程序中使用的数据结构占有的空间代价

算法的时间复杂度：算法的时间效率

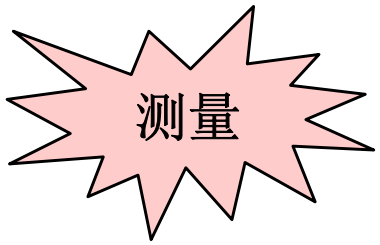
算法的空间复杂度：算法的空间效率

- 由于算法的复杂性与其所求解的问题规模直接有关，因此通常将问题规模 n 作为一个参照量，求算法的时空开销(分别设为T和S)与 n 的函数关系 f和s。

$$T(n) = f(n) \qquad S(n) = s(n)$$

算法效率的衡量方法和准则

- 通常有两种衡量算法效率的方法：
  - **方法一：事后统计法**（通过上机运行，测试算法花费的时间。算法的平均执行时间，算法的最大执行时间等）
    - 缺点1：必须编写、执行程序
    - 缺点2：其它因素掩盖算法本质
    - 优点：效率举证



  - **方法二：事前分析估算法**
    - 算法分析感兴趣的不是具体的资源占用量，而是与具体平台和具体输入实例无关，且随输入规模增长的值可预测。

事前分析估算法

- **算法时间复杂度**
  - ¶ 与问题规模之间的关系：用一定“规模”的数据作为输入时程序运行所需的“基本操作”数来描述**时间效率**。
    - ▸ 数据规模：n，
    - ▸ 运行时间：t 。
  - ¶ 说明：完成一个“基本操作”所需的时间应该与具体的被操作的数无关

```
mult(int a[ ], int b[ ], int& c[ ] )
{
 // 以二维数组存储矩阵元素，
        c 为 a 和 b 的乘积
  for(i=1; i<=n; ++i)
    for(j=1;j<=n;++j)
    {
        c[i][j]=0;
        for(k=1;k<=n;++k)
        c[i][j]+=a[i][k]*b[k][j];
    }
```

算法效率的衡量方法和准则

- 算法时间复杂度与问题的规模之间的关系
  - **线性关系：** $t_1 = cn$，若$n_2 = 2n$
    $\rightarrow t_2 = 2t_1$;
  - **平方关系：** $t_1 = n^2$，若$n_2 = 2n$
    $\rightarrow t_2 = (2n)^2 = 4t_1$;
  - **立方关系：** $t_1 = n^3$，若$n_2 = 2n$
    $\rightarrow t_2 = (2n)^3 = 8t_1$;
  - **对数关系：** $t_1 = \log_2 n$。若$n_2 = 2n$
    $\rightarrow t_2 = \log_2(2n) = \log_2 n + 1 = t_1 + 1$

$$f(n) = n^2 + 100n + \log_{10} n + 1000$$

算法效率的衡量方法和准则

算法的渐进分析 (asymptotic analysis)

- 简称算法分析，就是估计当数据规模 n逐步增大时，资源开销f(n)的增长趋势
  - 当n增大到一定值以后，资源开销的计算公式中影响最大的就是n的幂次最高的项，其他的常数项和低幂次项都是可以忽略的
- 一般函数关系都相当复杂，计算时只考虑显著影响函数的量级，即结果为原函数的一个近似值。
- 对资源开销的一种不精确估计，提供对于算法资源开销进行评估的简单化模型。

算法效率的衡量方法和准则

算法渐进分析：大 O 表示法

- 假如随着问题规模 **n** 的增长，算法的执行时间**T(n)**的增长率与 **f(n)** 的增长率相同，则记作：

$$T(n) = O(f(n))$$

称 **O(f(n))** 为算法的**渐近时间复杂度 (Asymptotic time complexity)**，简称为**算法的时间复杂度**。

算法效率的衡量方法和准则

算法渐进分析：大 O 表示法

◆ 假设 g 和 f 为从自然数到非负实数集的两个函数

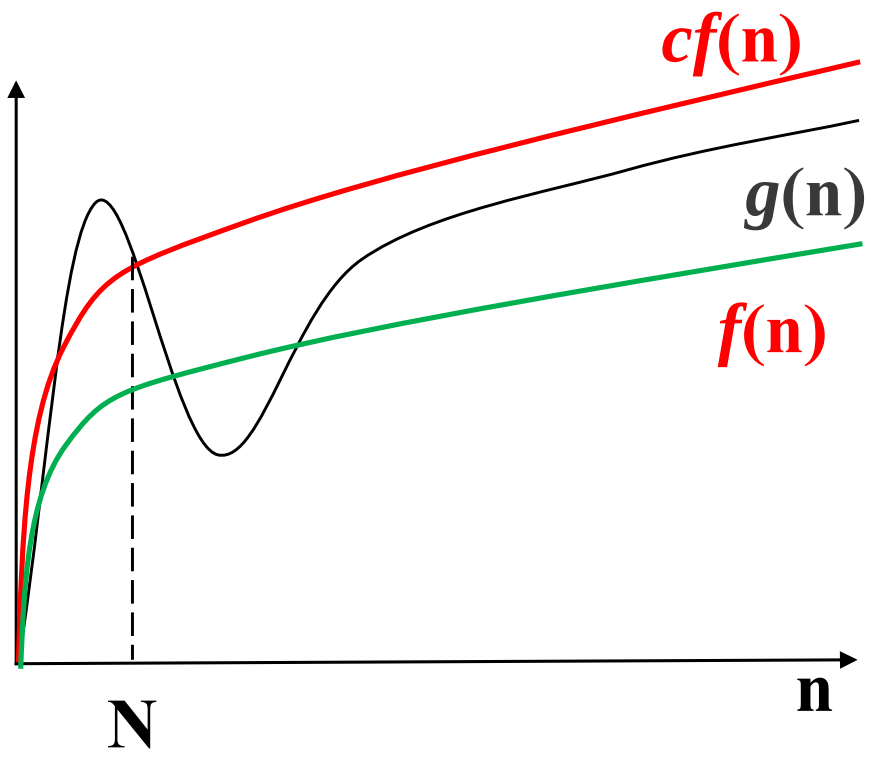
定义1：如果存在正数c和N，使得对任意的n≥N，都有

$$g(n) \le cf(n),$$

则称$g(n)$在集合$O(f(n))$中，或简称$g(n)$是$O(f(n))$的。

算法效率的衡量方法和准则

算法渐进分析：大 O 表示法



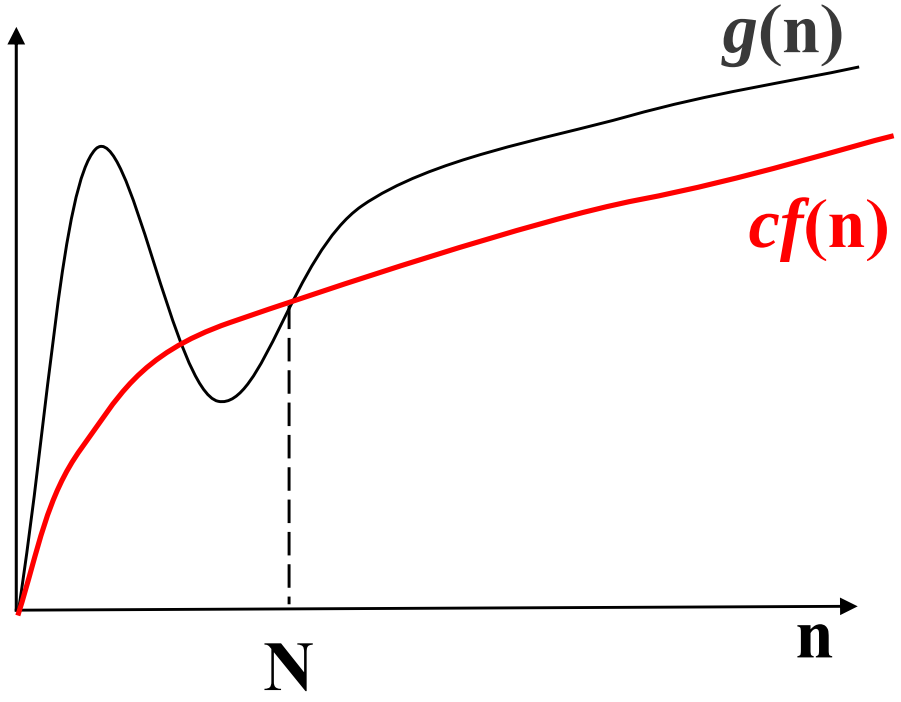
- 定义说明了函数 g和 f 之间的关系：函数*f*(n) 是函数 *g*(n) 取值的上限（**upper bound**），或说函数*g*的增长最终至多趋同于*f*的增长。
- 因此，大O表示法提供了一种表达函数增长率上限的方法

- 一个函数增长率的上限可能不止一个。

g(n)是O(*f*(n))的，*f*(n)是*g*(n)的紧密上限

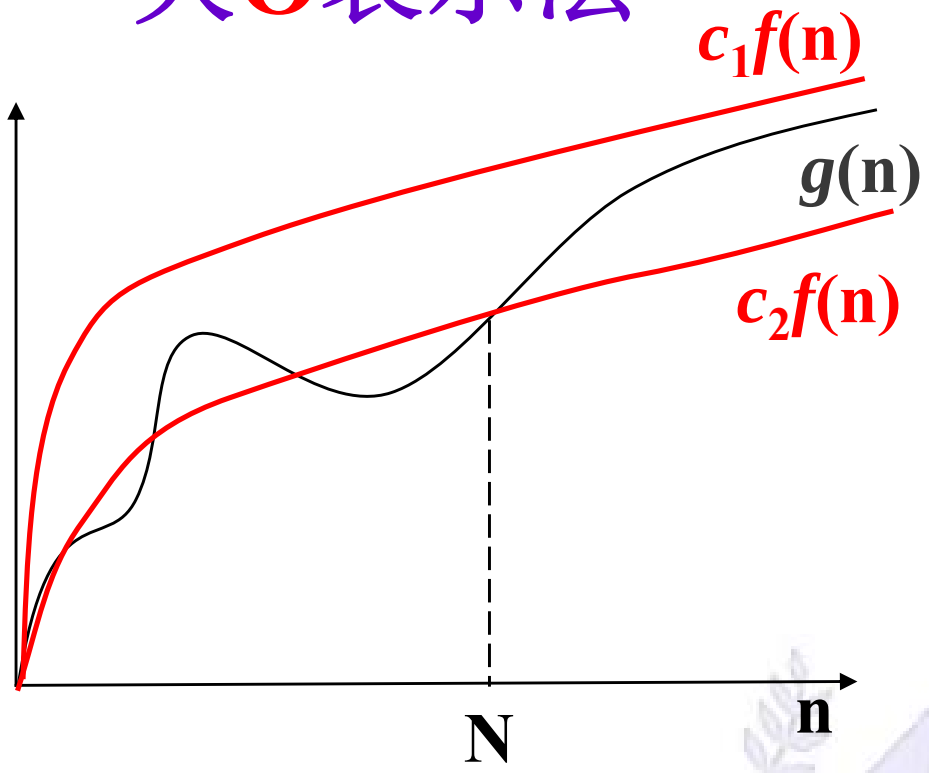
算法效率的衡量方法和准则

大Ω表示法



$g(n)$是$\Omega(f(n))$的
$f(n)$是$g(n)$的紧密下限

大Θ表示法



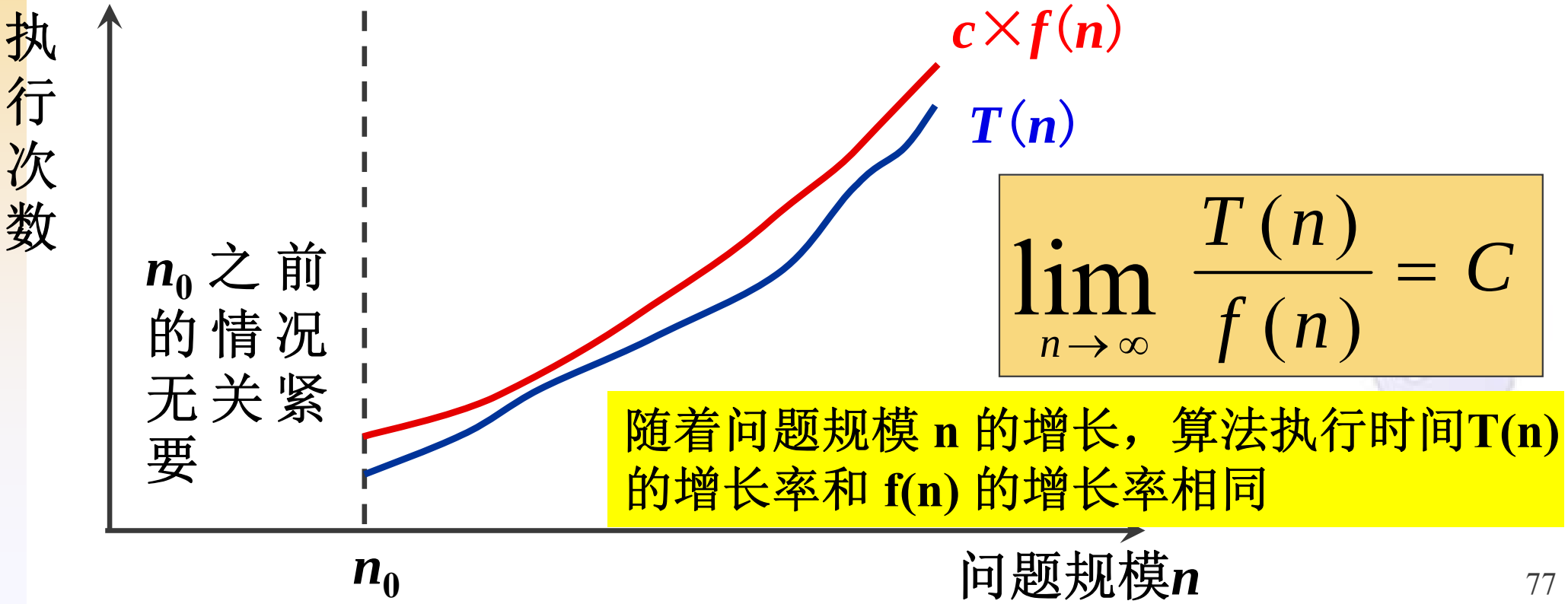
$g(n)$是$\Theta(f(n))$的
$f(n)$是$g(n)$的紧确界

算法效率的衡量方法和准则

算法渐进分析：大 O 表示法

若存在两个正的常数c和$n0$，对于任意$n \geq n0$，都有$T(n) \leq c \times f(n)$，则称 $T(n)=O(f(n))$。

当问题规模充分大时在渐近意义下的阶



$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = C$$

随着问题规模 n 的增长，算法执行时间T(n)的增长率和 f(n) 的增长率相同

算法效率的衡量方法和准则

大 O 表示法的加法原则

- 如果两个并列的程序段的渐进时间复杂度分别为：

  $T1(n)=O(f(n))$，$T2(m)=O(g(m))$

  则将两个程序段连在一起整个程序的时间复杂度为：

  $$T(n, m) = T1(n)+T2(m) = O(\max(f(n), g(m)))$$

算法效率的衡量方法和准则

大 O 表示法的乘法原则

- 如果嵌套程序段的渐进时间复杂度分别为：

  $T1(n)=O(f(n))$，$T2(m)=O(g(m))$

  则将两个程序段连在一起整个程序的时间复杂度为：

  $$T(n, m)=T1(n)\times T2(m)=O(f(n)\times g(m))$$

算法效率的衡量方法和准则

例：已知运行时间为$t = 20\times\log_2 n + n + 100n\log_2 n + n^2$

问$T(n) = O(f(n)) = ?$

n足够大时，

$20\times\log_2 n < 20\times n$

$< 100\times n\log_2 n$

$< 100\times n^2$

所以$T(n) = O(n^2)$

当 n 足够大时忽略所有低次幂和最高次幂的系数。

算法效率的衡量方法和准则

♦ 大O表示法的运算规则

- ¶ 单位时间
  - ▸ 简单布尔或算术运算
  - ▸ 简单I/O
  - ▸ 函数返回
- ¶ 加法规则：$f_1(n)+f_2(n)=O(\max(f_1(n), f_2(n)))$
  - ▸ 顺序结构，if 结构，switch结构
- ¶ 乘法规则：$f_1(n)\times f_2(n)=O(f_1(n)\times f_2(n))$
  - ▸ for, while, do-while结构

算法效率的衡量方法和准则

如何估计时间复杂度?

- 算法由控制语句和原操作（预定义数据类型的操作）组成
- 算法的执行时间=

  Σ原操作(i)的执行次数×原操作(i)的执行时间
- **算法的执行时间 与 原操作执行次数之和 成正比**

算法效率的衡量方法和准则

- 例1、矩阵乘法
- 基本操作在算法中重复执行的次数作为算法运行时间的衡量准则
- 频度：是指该语句重复执行的次数
- 总次数为: n^2+n^3.
- 时间复杂度为$T(n)=O(n^3)$

```
void mult(int a[], int b[], int&
c[] ) {
 // 以二维数组存储矩阵元素，
    c 为 a 和 b 的乘积
  for(i=1; i<=n; ++i)
    for(j=1;j<=n;++j)
    {
A×n²   c[i][j]=0;
        for(k=1;k<=n;++k)
B×n³    c[i][j]+=a[i][k]*b[k][j];
    }
```

A: 赋值操作单位时间
B: 乘法+赋值单位时间

如何估计算法的时间复杂度？

- 例 2 {++x;}
  - 基本操作： ++ x
  - 语句频度为 1
  - 时间复杂度为O(1)，即常量阶

- 例 3 for(i=1;i<=n;++i)
  { ++x; s+=x; }
  - 基本操作：++x; s+=x;
  - 语句频度为：2n
  - 其时间复杂度为O(n)，即时间复杂度为线性阶。

算法效率的衡量方法和准则

- 例 4
```
for(i=0;i<n; i++)
    for(j=0;j<n;j++)
        {++x;s+=x;}
```
  - 语句频度为：$2n^2$
  - 其时间复杂度为：$O(n^2)$，即时间复杂度为平方阶。

- 例 5
```
for(i=0;i<n; i++)
    for(j=1; j<=i-1; j++)
        {++x; a[i, j]=x;}
```
  - 语句频度为：$2(1+2+3+\ldots+n-2) = (n-1)(n-2)$
  - 其时间复杂度为：$O(n^2)$，即时间复杂度为平方阶。

算法效率的衡量方法和准则

- 例6 int t=1;

  while(t<n) t = t*2;

  - 语句频度为：logn
  - 其时间复杂度为: O(logn), 即时间复杂度为对数阶。

- 例7 int s=1; int m = 2^n;

  while(s<m) s++;

  - 语句频度为：2^n
  - 其时间复杂度为: $O(2^n)$, 即时间复杂度为指数阶。

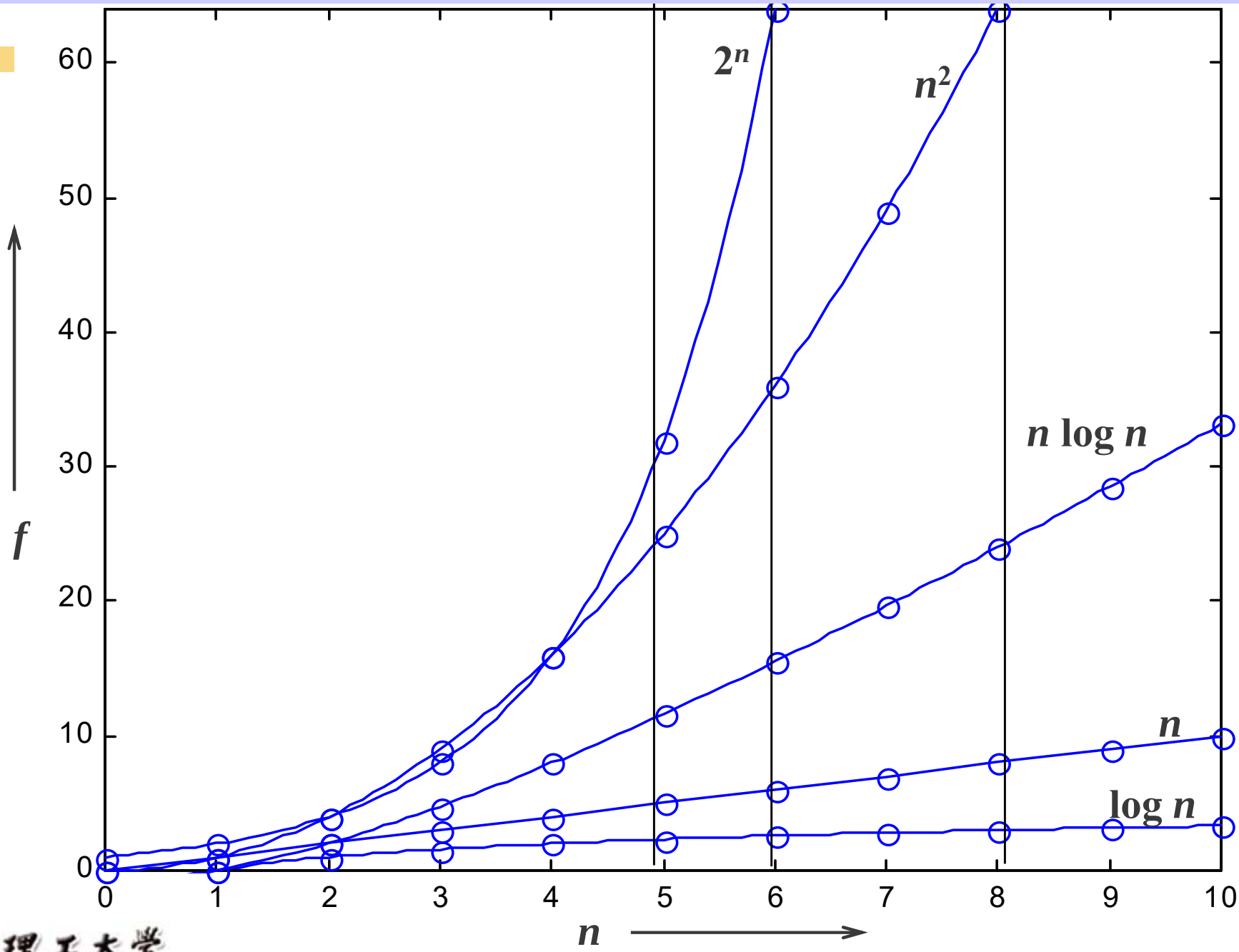
算法效率的衡量方法和准则

- 多项式时间的算法最常用的六种。其关系为：
  - $O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(n^3)$
- 指数时间的关系为：
  - $O(2^n)<O(n!)<O(n^n)$
- 当n取得很大时，指数时间算法和多项式时间算法在所需时间上非常悬殊。
  - $c < \log_2 n < n < n\log_2 n < n^2 < n^3 < 2^n < 3^n < n!$

n	$\log_2 n$	$n\log_2 n$	n^2	n^3	2^n	$n!$
4	2	8	16	64	16	24
8	3	24	64	512	256	80320
10	3.32	33.2	100	1000	1024	3628800
16	4	64	256	4096	65536	$2.1*10^{13}$
32	5	160	1024	32768	$4.3*10^9$	$2.6*10^{35}$
128	7	896	16384	2097152	$3.4*10^{38}$	∞
1024	10	10240	1048576	$1.07*10^9$	∞	∞
10000	13.29	132877	10^8	10^{12}	∞	∞

算法复杂性的不同数量级的变化

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



算法效率的衡量方法和准则

- 递归算法的执行次数：
  - 计算正数数组a中前n个元素之和的递归算

```
float rsum（float a[], int n）//递归程序{
        if (n<=0) return 0;
      else return rsum(a, n-1) + a[n-1]);
}
```

算法效率的衡量方法和准则

- 时间复杂度计算：

$$T(n)=\begin{cases}1, & n\le 0\\ 2+T(n-1), & n>0\end{cases}$$

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

$$=2+2+T(n-2)=2*2+T(n-2)$$

$$=2*2+2+T(n-3)=2*3+T(n-3)=\ldots=$$

$$=2n+T(0)=2n+1$$

- 时间复杂度为O(n)

```
float rsum（float a[], int n）//递归程序{
    if (n<=0) return 0;
    else return rsum(a, n-1) + a[n-1]);
}
```

算法效率的衡量方法和准则

- 算法的时间复杂度级别

例：假设 CPU 每秒处理 10^6 条指令，对于输入规模为 10^8 的问题，时间代价 $T(n) = 2n^2$：

操作次数：$T(n) = T(10^8) = 2\times(10^8)^2 = 2\times10^{16}$

运行时间：$2\times10^{16}/10^6 = 2\times10^{10}$秒

每天86,400秒，因此需要231,480 天(634年)

若，时间代价为 $n\log n$：

操作：$T(n)=T(10^8) = 10^8\times\log 10^8 = 2.66\times10^9$

运行：$2.66\times10^9/10^6 = 2.66\times10^3$秒=44.33分

算法效率的衡量方法和准则

- 算法的时间复杂度级别

例：设 CPU 每秒处理 10^6 个指令，则每小时能够解决的最大问题规模：

$$T(n) / 10^6 \leq 3600$$

- 对 $T(n) = 2n^2$

  即 $2n^2 \leq 3600 \times 10^6$

  $$n \leq 42,426$$

- 对 $T(n) = n\log n$

  即 $n\log n \leq 3600 \times 10^6$

  $$n \leq 133,000,000$$

算法效率的衡量方法和准则

- 算法中基本操作重复执行的次数，会随问题的输入数据集不同而不同。
- 最好情况：n次
- 最坏情况：
  1+2+3+…+n-1
  =n(n-1)/2
- 平均时间复杂度为：
  $O(n^2)$

```
Void bubble-sort (int a[]，int n)
{ //起泡排序，从小到大排列
    for( i=n-1,change=TURE;
            i>1 && change;- -i)
    {
        change=false;
        for( j=0;j<i; ++j)
            if ( a[j]>a[j+1]) {
                a[j] ←→a[j+1];
                change=TURE}
    }
}//bubble-sort
```

算法效率的衡量方法和准则

♦最好情况、最坏情况、平均情况

如果问题规模相同，时间代价与输入数据分布有关，则需要分析最好情况、最坏情况和平均情况。

✓平均情况：已知输入数据是如何分布的，通常假设等概率分布

✓最差情况：分析执行时间的上限，实时系统，

算法效率的衡量方法和准则

◆ 算法的存储空间需求

- ¶ 固定空间
  输入数据、结果数据、程序代码所需的空间
- ¶ 可变空间(辅助空间)
  中间结果、递归调用所需的空间

算法效率的衡量方法和准则

◆ 算法所需的辅助空间度量

假如随着问题规模 n 的增长，算法运行所需辅助空间 S(n) 的增长率与 g(n) 的增长率相同，则记作：

$$S(n)=O(g(n))$$，

称O(g(n)) 为算法的空间复杂度。

算法效率的衡量方法和准则

- 若输入数据所占空间只取决于问题本身，和算法无关，则只需要分析除输入和程序之外的辅助变量所占额外空间。

- 若所需额外空间相对于输入数据量来说是常数，则称此算法为原地工作。

- 若所需存储量依赖于特定的输入，则通常按最坏情况考虑。

算法效率的衡量方法和准则

```
void bubble_sort (int a[ ], int n )
{  //起泡排序
    for (i = n-1,change=TRUE; i >0&&change; - -i )
    {   change= FALSE;
        for (j = 0; j<= i; ++j )
            if (a [j] >a[ j+1])
            {  temp=a [j]; a[j]= a[ j+ 1]; a[j+1]=temp;
               change=TRUE;
            }
    }
}
```

S(n) = O(1)

算法效率的衡量方法和准则

空间复杂度：求 整数 n 的阶乘

```
int f(unsigned int n){
    if (n==0 || n==1)
        return 1;
    return n*f(n-1);
}
```

S(n) = O(n)

算法分析思考题

例题1. 算法的时间复杂度与（　）有关。

A. 问题规模　　B. 计算机硬件的运行速度

C. 源程序的长度　　D. 编译后执行程序的质量

解答：A。

分析：

算法的具体执行时间与计算机硬件的运行速度、编译产生的目标程序的质量有关，但这属于事后测量。算法的时间复杂度的度量属于事前估计，与问题的规模有关。

算法分析思考题

例题2. 某算法的时间复杂度是$O(n^2)$，表明该算法（　）。

A. 问题规模是n^2　　B. 问题规模与n^2成正比

C. 执行时间等于n^2　　D. 执行时间与n^2成正比

解答：D

分析：

算法的的时间复杂度是$O(n^2)$，这是设定问题规模为 n 的分析结果，所以A、B 都不对；它也不表明执行时间等于n^2，它只表明算法的执行时间$T(n)\leq c\times n^2$（c为比例常数）。有的算法，如$n\times n$矩阵的转置，时间复杂度为$O(n^2)$，不表明问题规模是n^2。

算法分析思考题

例题3. 有实现同一功能的两个算法A_1和 A_2，其中：

A_1 的渐进时间复杂度是$T_1(n) = O(2^n)$，

A_2 的渐进时间复杂度是$T_2(n) = O(n^2)$。仅就时间复杂度而言，具体分析这两个算法哪个好。

解答：比较算法好坏需比较两个函数2^n和n^2。

当$n = 1$时，$2^1 > 1^2$，算法A_2好于A_1

当$n = 2$时，$2^2 = 2^2$，算法A_1与A_2相当

当$n = 3$时，$2^3 < 3^2$，算法A_1好于A_2

当$n = 4$时，$2^4 > 4^2$，算法A_2好于A_1

当$n > 4$时，$2^n > n^2$，算法A_2好于A_1

当$n \to \infty$时，算法A_2在时间上显然优于A_1。

算法分析思考题

例题4　设n是描述问题规模的非负整数，下面程序片段的时间复杂度是

```
x = 2;
while ( x < n/2 )
    x = 2*x;
```

A. $O(\log_2 n)$　　B. $O(n)$　　C. $O(n\log_2 n)$　　D. $O(n^2)$

解答：A

分析：确定基本操作，即 while 循环中的语句 x = 2*x。因为每次循环 x 都成倍增长，设 $x = 2^k < n/2$，$2k+1 < n$，则 $k < \log 2n - 1$，实际 while 循环内的语句执行了 log2n - 2次。

算法分析思考题

例题5 下面说法中错误的是

① 算法原地工作的含义是指不需要任何额外辅助空间

② 在相同问题规模 n 下，时间复杂度为$O(n)$的算法总是优于时间复杂度为$O(2^n)$的算法

③ 所谓时间复杂度是指在最坏情形下估算算法执行时间的一个上界

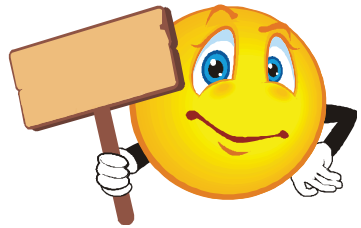
④ 同一个算法，实现语言的级别越高，执行效率越低

A. ①　　B. ①②　　C. ①④　　D. ③

解答：A。

算法原地工作的含义指空间复杂度 $O(1)$

本章学习要点



1. **熟悉**各名词、术语的含义，掌握基本概念。

2. **理解**ADT的意义。

3. **理解**算法五个要素的确切含义。

4. **掌握**计算语句频度和估算算法时间复杂度的方法。

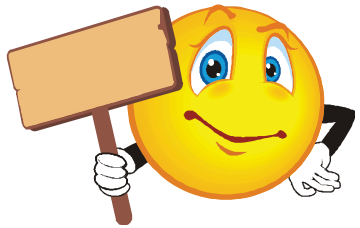
- 务必**提前复习C语言中的数组、指针、结构、内存申请和释放等内容。**

绪论—必须掌握的基本概念

数据结构：带有结构和操作的数据元素集合

- 结构
  - 数据的逻辑结构
    - 线性结构
      - 线性表
      - 栈
      - 队列
    - 非线性结构
      - 树形结构
      - 图形结构
  - 数据的存储结构
    - 顺序存储
    - 链式存储
    - 索引存储
    - 散列存储
- 操作
  - 数据操作的定义（功能）
  - 数据操作的实现（在计算机上）

数据结构学习要求



1.理解各章**基本概念**

2.**存储结构：**

- 1）掌握基本存储结构：表、栈、队列、二叉树（顺序结构、链式结构（动静态）存储信息、含义及C 语言描述）
- 2）理解复杂存储结构（图、树），理解图和树上的应用：如: 最小生成树、最短路径等

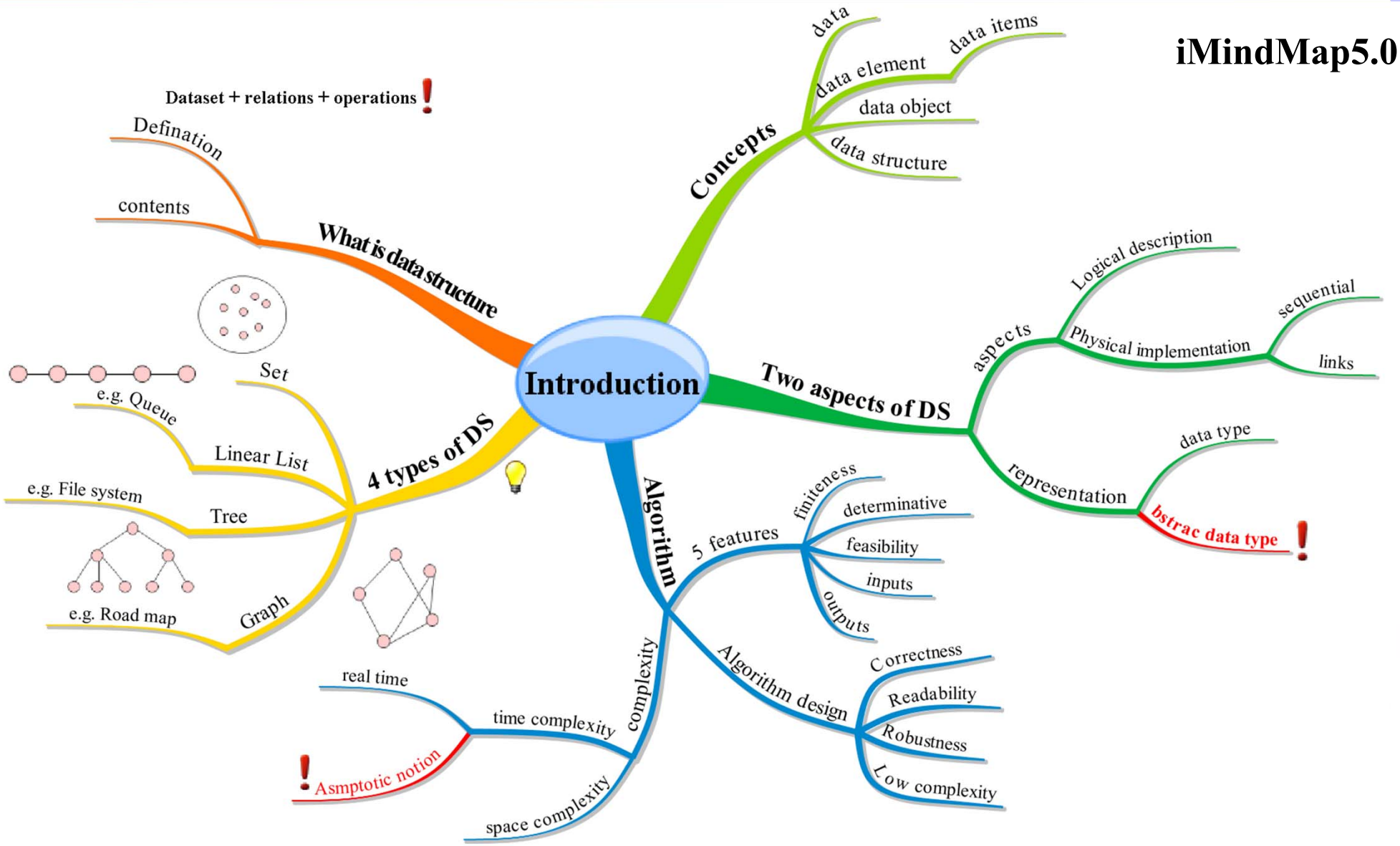
3.**算法:**

¶ 1) 掌握基本算法（构造、销毁、插入、删除、遍历、查找、排序等) 的主要步骤/基本思想、主要操作的实现(C语言描述)，并能应用到对应问题中，能推广到相似问题中；

¶ 2）复杂算法：掌握方法；

¶ 3）能够计算基本算法的时间复杂度和空间复杂度

Review


iMindMap5.0
Introduction
What is data structure
Dataset + relations + operations
Defination
contents
Concepts
data
data element
data items
data object
data structure
4 types of DS
Set
e.g. Queue
Linear List
e.g. File system
Tree
e.g. Road map
Graph
Two aspects of DS
aspects
Logical description
Physical implementation
sequential
links
representation
data type
bstrac data type
Algorithm
5 features
finiteness
determinative
feasibility
inputs
outputs
complexity
real time
time complexity
Asmptotic notion
space complexity
Algorithm design
Correctness
Readability
Robustness
Low complexity

END of Chapter I

史树敏
计算机学院