数据结构

DATA STRUCTURE

——用面向对象方法与C++描述

课程特点

- 介绍如何组织各种数据在计算机中的**存储**、**传递**和**转换**；
- 课程采用**面向对象**的观点讨论数据结构技术；
- 兼有面向过程和面向对象双重特色的**C++语言**作为算法的描述工具；
- 强化**数据结构基本知识**和**面向对象程序设计基本能力**的双基训练。

- **课程要求**
  - ***“听课”+“自学”+“实践”并举***
  - 掌握重要数据结构的**概念**、**使用方法**及**实现**技术；
  - 学会做简单的**算法分析**，包括算法的时间代价和空间代价。

- **如何评价成绩?**

  （拟订，[平行班统一]）

  - 期末闭卷笔试（50%）
  - 期中闭卷笔试（15%）
  - 项目实践
  - 上机实践
  - 作业及出勤情况

  （35%）

成绩为优者不超过30%

前四周上机实践课程内容简介

- 第一周：C++程序设计语言
  - 基础知识
    - 语法、数据类型、指针、函数、传递参数和地址
  - C++开发环境
    - VC的使用技巧
    - 上机操作实例（如何创建工程，如何加入头文件、cpp文件等；怎样编译；怎样调试……）
    - 用具体实例来说明上述概念和使用技巧

- 第二周：面向对象的程序设计OOP
  - 类和对象
  - 构造/析构函数
  - 类的继承
  - 派生类
  - 虚函数与多态
  - 重载
  - 引用类型reference
  - 友元

- **第三周：template介绍**
  - template及其优势
  - 模板的多态性
  - template使用方法
  - 函数模板/类模板
  - 以一些具体实例来介绍template的应用

- **第四周：程序设计风格及设计实践**
  - 编程规范
  - 好程序的定义
  - 命名规则
  - 表达式和语句
  - 函数
  - 如何写Document
  - 注释
  - 什么样的编码是好的编程风格

第一章 绪论

- 数据结构的概念
- 数据结构的抽象形式
- 作为ADT的C++类
- 算法定义
- 算法性能分析与度量
- 本章小结

1.1 数据结构的概念

- 宇宙三要素：物质、能量和信息
- 信息是客观世界在人脑中的反映。
- 数据是信息的载体。
- 数据是怎样在计算机中存储和组织的?
- 举例说明

“学生”表格

	学　号	姓　　名	性别	籍　　贯	出生年月
1	98131	刘澈扬	男	北　　京	1979.12
2	98164	衣春生	男	青　　岛	1979.07
3	98165	卢声凯	男	天　　津	1981.02
4	98182	袁秋慧	女	广　　州	1980.10
5	98224	洪　　伟	男	太　　原	1981.01
6	98236	熊南燕	女	苏　　州	1980.03
7	98297	宫　　力	男	北　　京	1981.01
8	98310	蔡晓莉	女	昆　　明	1981.02
9	98318	陈　　健	男	杭　　州	1979.12

“课程”表格

课程编号	课 程 名	学时
024002	程序设计基础	64
024010	汇编语言	48
024016	计算机原理	64
024020	数据结构	64
024021	微机技术	64
024024	操作系统	48
024026	数据库原理	48

“选课单”包含如下信息

学号	课程编号	成绩	时间

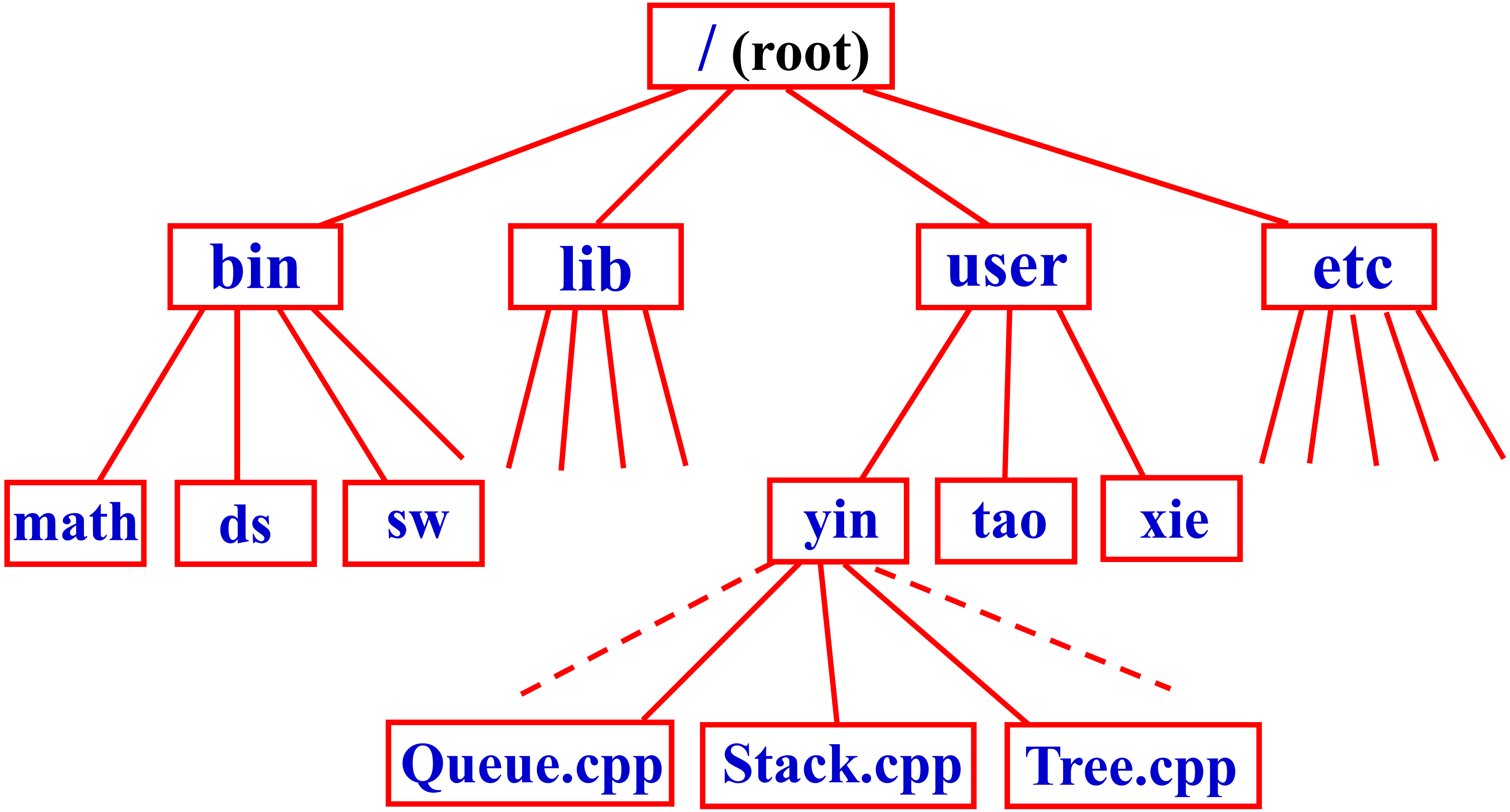
学生选课系统中实体构成的网状关系

学生
(学号,姓名,性别,籍贯)

课程
(课程号,课程名,学分)

选课
(学号,课程号,成绩)

UNIX文件系统的系统结构图



基本概念：

数据 (Data)

- 数据是**信息的载体**，是描述客观事物的数、字符、以及所有能输入到计算机中，被计算机程序识别和处理的符号的集合。
  - 数值性数据 (int)
  - 非数值性数据 (char)

基本概念：

数据元素 (Data Element)

- 数据的基本单位，在计算机程序中常作为一个整体进行考虑和处理。
- 有时一个数据元素可以由若干数据项(Data Item)组成。数据项是具有独立含义的最小标识单位。
- 数据元素又称为元素、结点、记录。

基本概念：

数据对象 (Data Object)

- **数据的子集，具有相同性质的数据成员（数据元素）的集合。**
  - **整数数据对象** $N = \{ 0, \pm 1, \pm 2, \dots \}$
  - **学生数据对象**
- 数据对象中所有成员之间存在某种关系，如学生按学号的排序；按性别的分类等。
- 数据成员及其之间关系，是数据结构研究的主要内容。

什么是数据结构

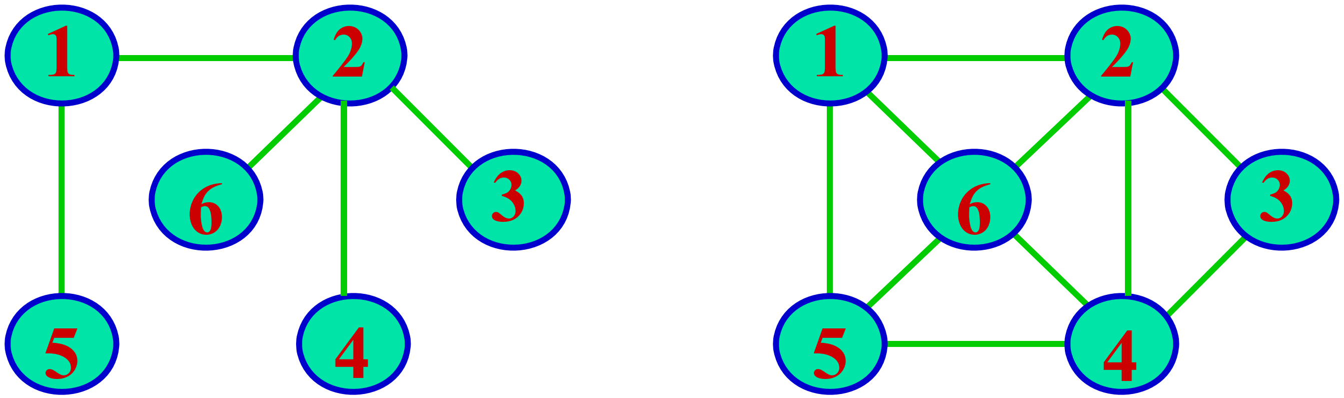
定义：

由某一数据对象及该对象中所有数据成员之间的关系组成。记为：

Data_Structure = {D, R}

其中，D 是某一数据对象，R 是该对象中所有数据成员之间关系的有限集合。

N个网站之间的连通关系



树形关系　　网状关系

数据结构是数据的组织形式

- 数据元素间的逻辑关系，即数据的逻辑结构；
- 数据元素及其关系在计算机存储内的表示，即数据的存储表示；
- 数据的运算，即对数据元素施加的操作。

例如：座位/学生（按班级）

DS第一部分：

数据的逻辑结构

- 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；
- 数据的逻辑结构可以看作是从具体问题抽象出来的数据模型；
- 数据的逻辑结构与数据元素的相对位置无关。

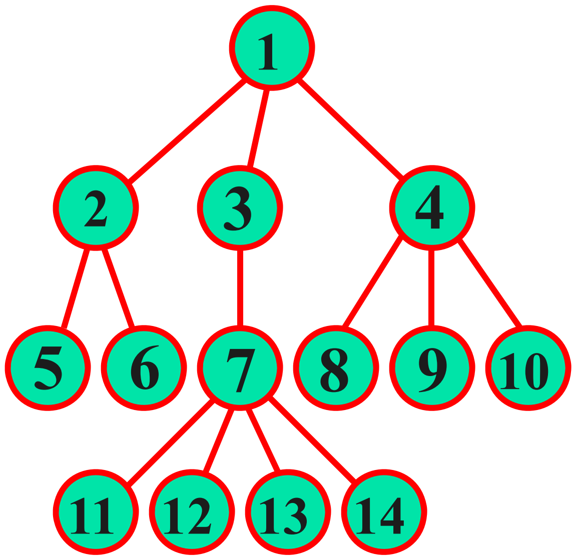
数据的逻辑结构分类

- 线性结构
  - 线性表
- 非线性结构
  - 树
  - 图（或网络）

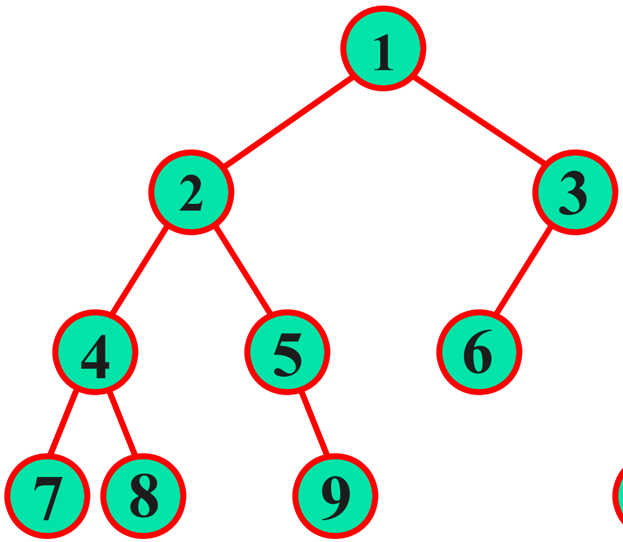
线性结构：

树形结构：

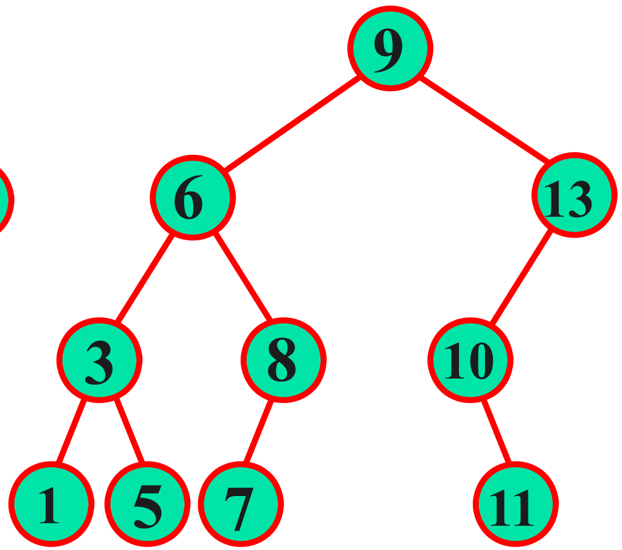
树



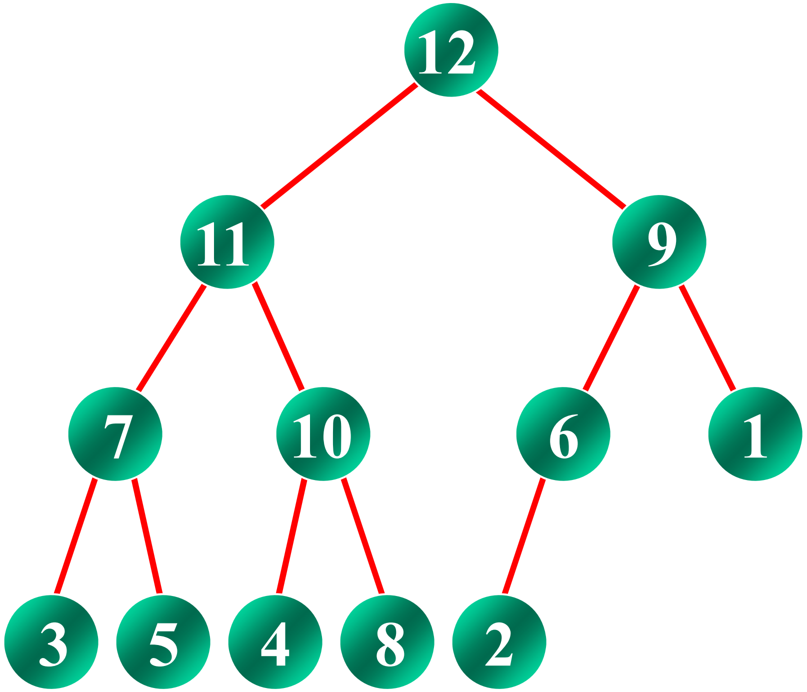
二叉树



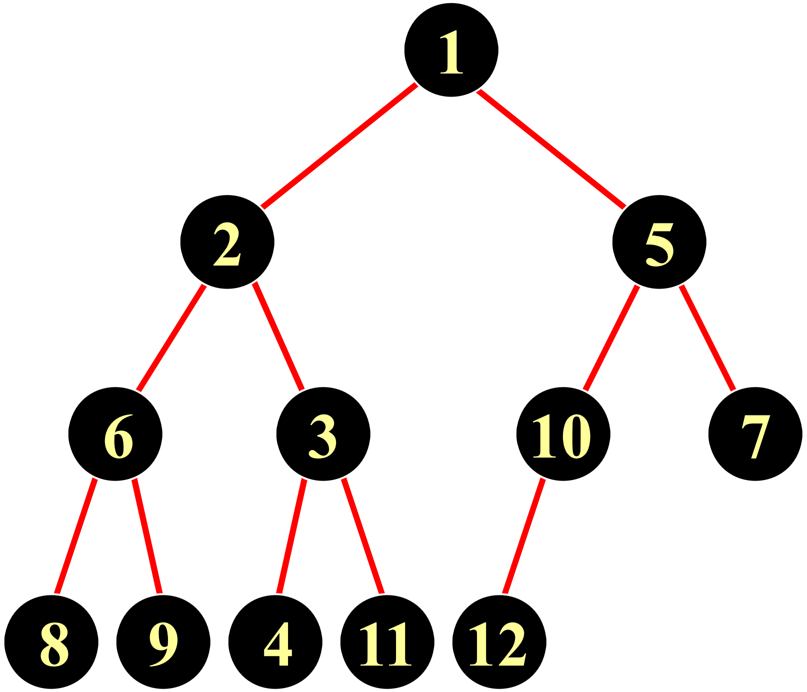
二叉搜索树



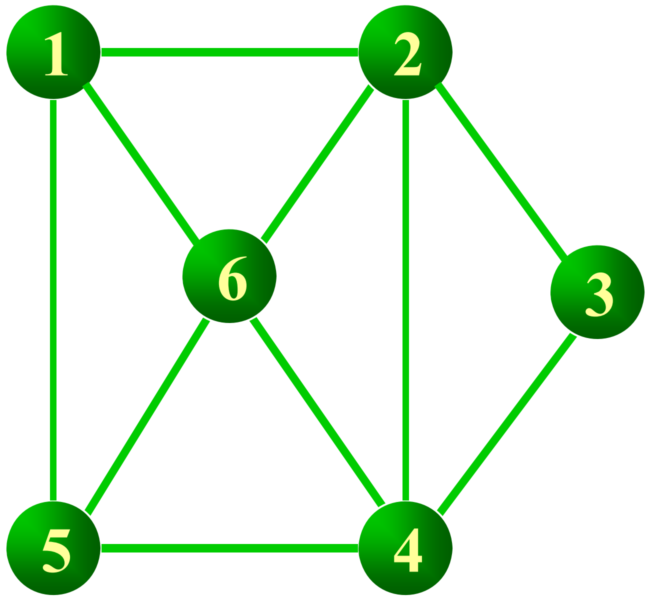
堆结构（树的特例）：



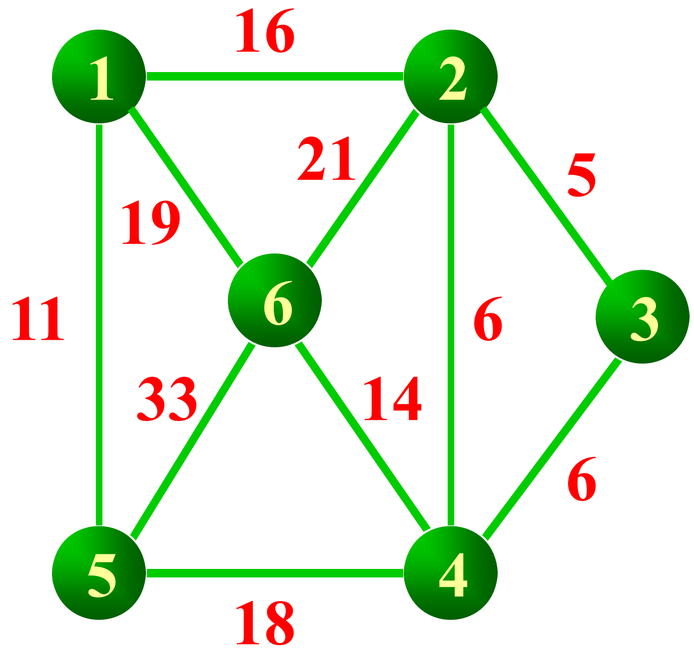
"最大"堆



"最小"堆



图结构



网络结构

DS第二个重要部分：

数据的存储结构

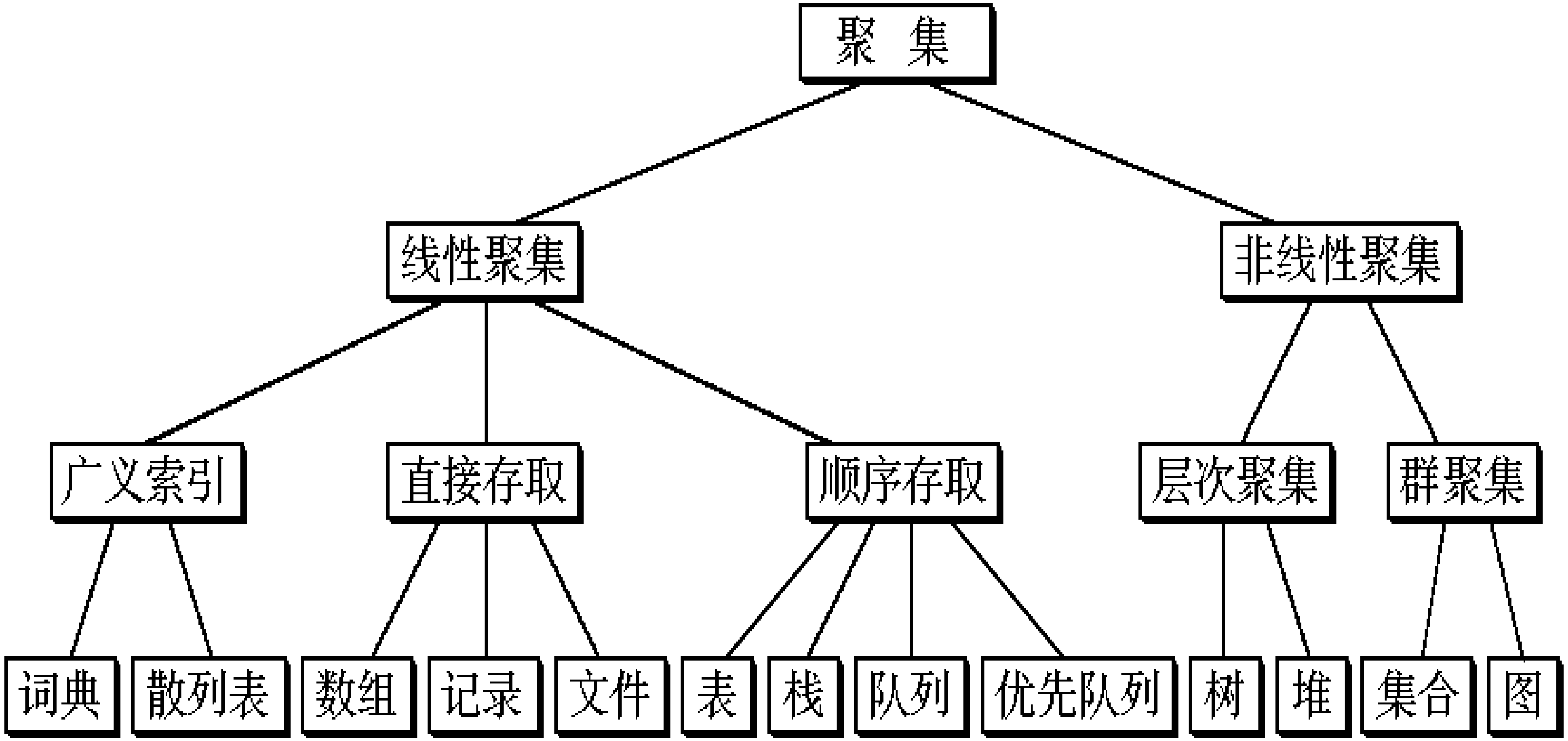
- 数据的存储结构是逻辑结构用计算机语言的实现；
- 数据的存储结构依赖于计算机语言。
  - 顺序存储表示
  - 链接存储表示

  主要用于内存的存储表示

  - 索引存储表示
  - 散列存储表示

  主要用于外存（文件）的存储表示

数据结构的抽象层次



聚 集
线性聚集
非线性聚集
广义索引
直接存取
顺序存取
层次聚集
群聚集
词典
散列表
数组
记录
文件
表
栈
队列
优先队列
树
堆
集合
图

- 线性聚集类
  - 直接存取类 数组、文件
  - 顺序存取类 表、栈、队列、优先队列
  - 广义索引类 线性索引、搜索树
- 非线性聚集类
  - 层次聚集类 树、二叉树、堆
  - 群聚集类 集合、图

数据结构的课程内容体系

方面 层次	数据表示	数据处理
抽象	逻辑结构	基本运算
实现	存储结构	算法
评价	不同数据结构的比较及算法分析	

1.2 数据结构的抽象形式

1.2.1 数据类型

定义： 一组**性质相同的值**的集合，以及定义于这个值集合上的一组操作的总称。

- **C语言中的数据类型**

  char　int　float　double　void

  字符型　整型　浮点型　双精度型　无值

  int取值范围[-32768, 32767]；

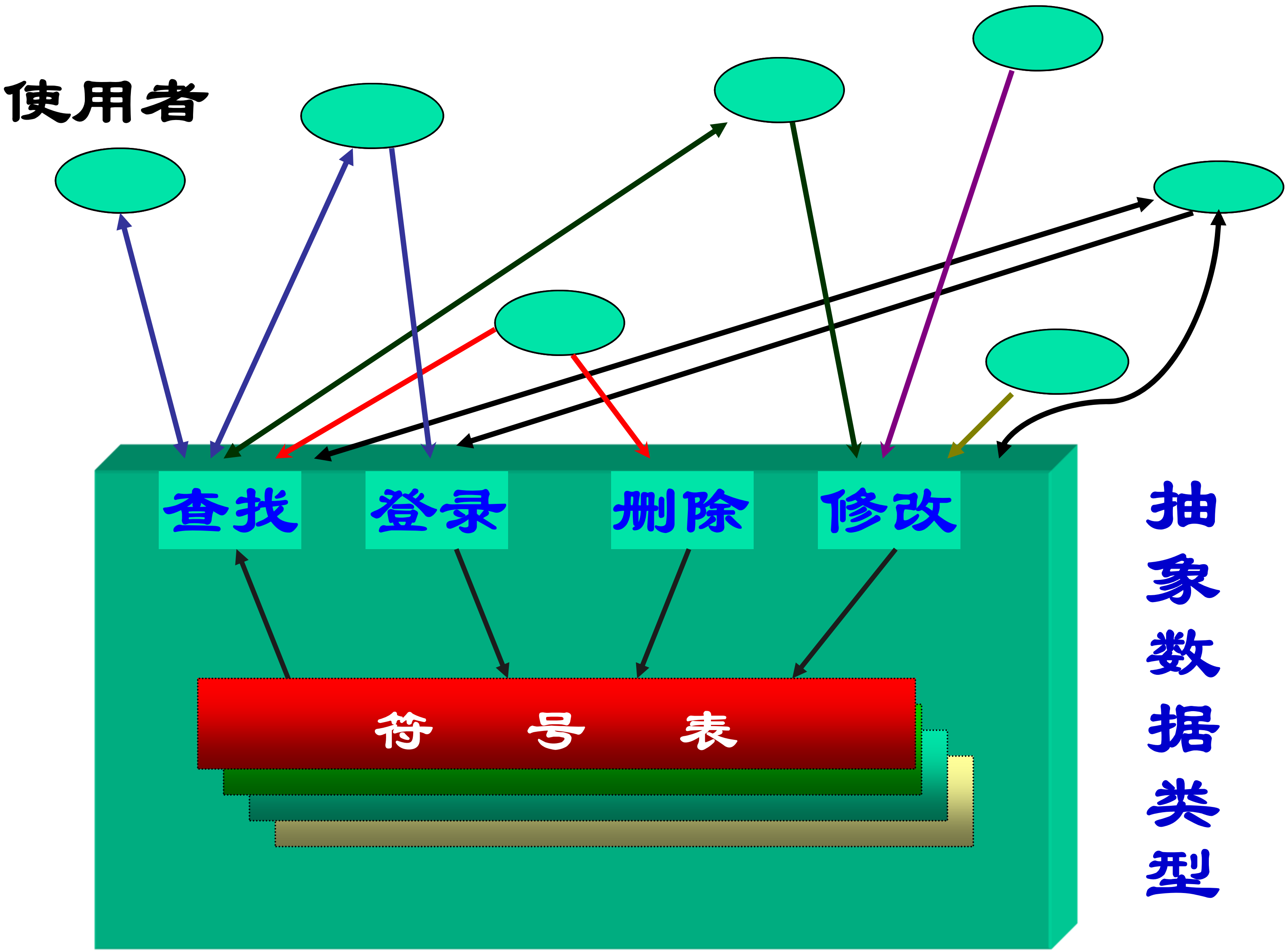
  每个数据类型对应一组操作

  int(6/4)=1　(float)(6.0/4.0)=1.5

- 数据类型由
  - 基本数据类型 或
  - 构造数据类型组成
- 构造数据类型由不同成分类型构成。
- 基本数据类型可以看作是计算机中已实现的数据结构。
- 数据类型就是数据结构，不过是从编程者的角度来使用。

1.2.2 数据抽象与抽象数据类型 (ADTs: Abstract Data Types)

- 由用户定义，用以表示应用问题的**数据模型**。
- **抽象的本质：**抽取反映问题本质的东西，忽略非本质的细节。
- ADT：由**基本的数据类型**组成，并包括**一组相关的服务**（或称操作）。
- **信息隐蔽**和**数据封装**，使用与实现相分离。


使用者
查找
登录
删除
修改
符 号 表
抽象数据类型

自然数的抽象数据类型定义

ADT ***NaturalNumber*** **is**

objects: 一个整数的有序子集合,它开始于0,结束于机器能表示的最大整数(*MaxInt*)。

Function: 对于所有的 $x, y \in NaturalNumber$;

False, *True* ∈ *Boolean*, +、-、<、==、=等都是可用的服务。

Zero( ) : *NaturalNumber* 返回自然数0

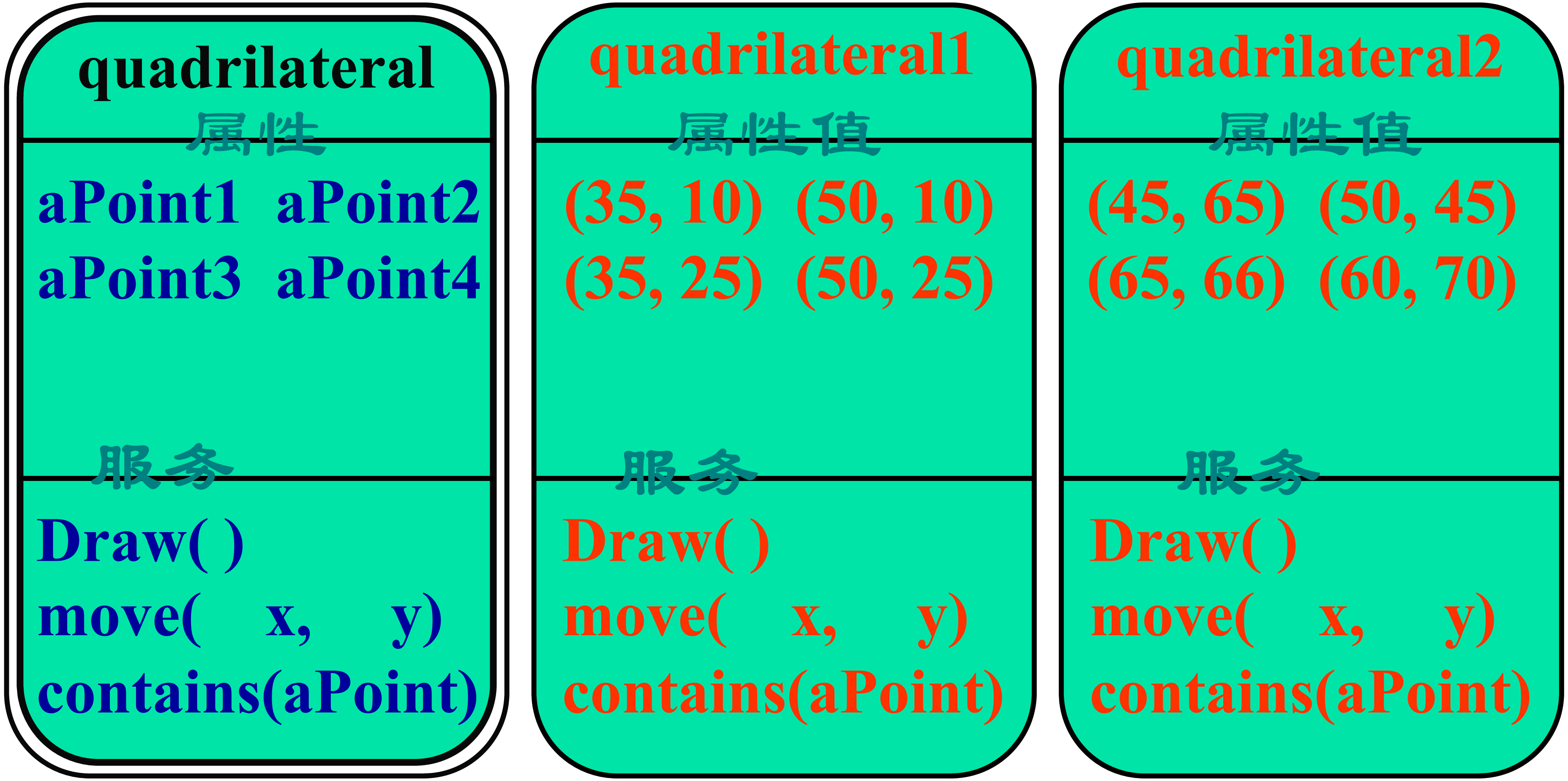
IsZero(x) :	if (x==0) 返回 *True*
Boolean	else 返回 *False*
Add (x, y) :	if (x+y<=*MaxInt*) 返回 x+y
NaturalNumber	else 返回 *MaxInt*
Subtract(x, y) :	if (x<y) 返回 0
NaturalNumber	else 返回 x-y
Equal(x, y) :	if (x==y) 返回 *True*
Boolean	else 返回 *False*
Successor(x) :	if (x==*MaxInt*) 返回 x
NaturalNumber	else 返回 x+1

end *NaturalNumber*

1.3 作为ADT的C++类

- **面向对象的概念**
  - **Codd and Yourdon**
  - **Rational Rose, IBM**
  - **面向对象＝对象＋类＋继承＋通信**
  - **实现信息隐藏和封装**

- 对象
  - 实体、事件、规格说明等。
  - 由一组属性值和在这组值上的一组服务（或称操作）构成。
- 类 (Class)，实例 (Instance)
  - 具有相同属性和服务的对象归于同一类，形成类。
  - 类中的对象为该类的实例。

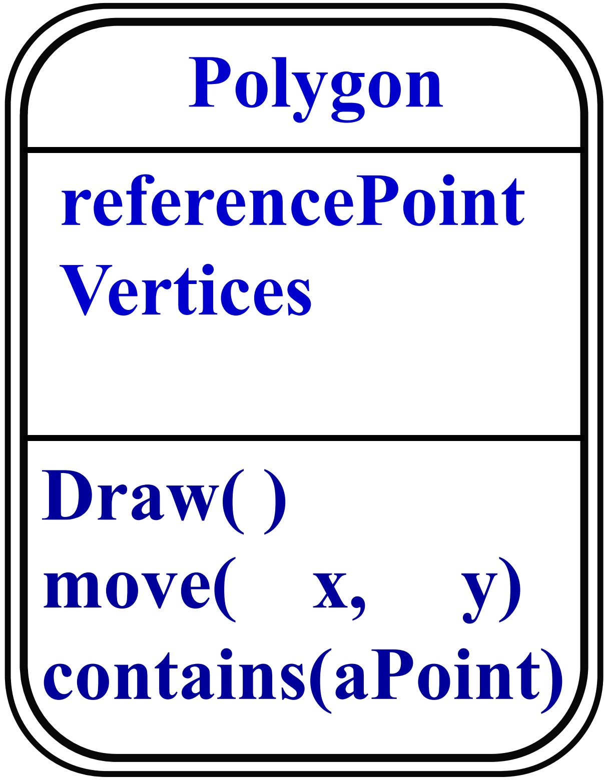

quadrilateral
属性
aPoint1 aPoint2
aPoint3 aPoint4
服务
Draw( )
move(x, y)
contains(aPoint)
quadrilateral1
属性值
(35, 10) (50, 10)
(35, 25) (50, 25)
服务
Draw( )
move(x, y)
contains(aPoint)
quadrilateral2
属性值
(45, 65) (50, 45)
(65, 66) (60, 70)
服务
Draw( )
move(x, y)
contains(aPoint)

四边形类及其对象

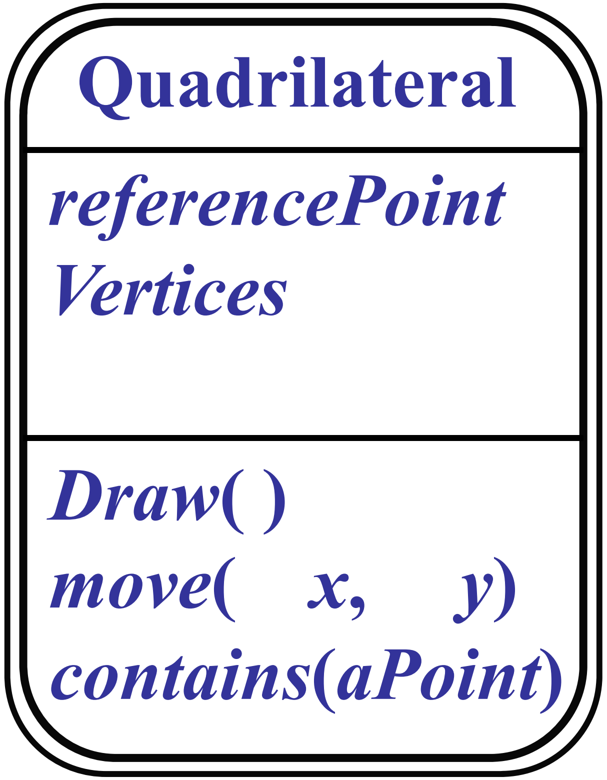
- **继承**
  - 派生类：四边形，三角形，…

    子类　特化类（特殊化类）
  - 基类：多边形

    父类　泛化类（一般化类）
- **通信**
  - 消息传递



Polygon 类



Polygon的子类

Quadrilateral类

用C++描述面向对象程序

- C++的函数特征
- C++的数据声明
- C++的作用域
- C++的类
- C++的对象
- C++的输入/输出
- C++的函数
- C++的参数传递
- C++的函数名重载和操作符重载
- C++的动态存储分配
- 友元(friend)函数
- 内联(inline)函数
- 结构(struct)与类
- 联合(union)与类

详细介绍——上机实践课

模板 (template)

目的：实现软件重用

定义

适合多种数据类型的类定义或算法，在特定环境下通过简单地代换，变成针对具体某种数据类型的类定义或算法。

用模板定义用于排序的数据表类

```
#ifndef DATALIST_H
#define DATALIST_H
#include <iostream.h>
template <class Type> class dataList
{
  private:
    Type *Element;
    int ArraySize;
    void Swap (int m1, int m2);
    int MaxKey (int low, int high);
```

```
public:
    dataList (int size = 10) : ArraySize (size),
        Element (new Type [Size]) { }
    ~dataList ( ) { delete [ ] Element; }
    void Sort ( );
    friend ostream& operator << (ostream
        &outStream, datalist <Type> &outList);
    friend istream& operator >> (istream
        &inStream, datalist <Type> &inList);
};
#endif
```

类中所有操作作为模板函数的实现

```
#ifndef SELECTTM_H
#define SELECTTM_H
#include "datalist.h"
template <class Type> void dataList <Type> ::
Swap (int m1, int m2)
{ //交换由m1, m2为下标的数组元素的值
    Type temp = Element [m1];
    Element [m1] = Element [m2];
    Element [m2] = temp;
}
```

```
template <class Type> int dataList<Type> ::
MaxKey (int low, int high)
{ //查找数组Element[low]到Element[high]
  //中的最大值，函数返回其位置
    int max = low;
    for (int k = low+1; k <= high; k++)
          if ( Element[max] < Element[k] )
               max = k;
    return max;
}
```

```
template <class Type>
ostream & operator << (ostream &OutStream,
dataList <Type> OutList) {
    OutStream << “数组内容：\n”;
    for (int i = 0; i < OutList.ArraySize; i++)
      OutStream << OutList.Element[i] << ‘ ’;
    OutStream << endl;
    OuStream << “数组当前大小：” <<
        OutList.ArraySize << endl;
    return OutStream;
}
```

```
template <class Type>
istream & operator >> (istream &InStream,
dataList <Type> InList) {
//输入对象为InList，输入流对象为InStream
    cout << "录入数组当前大小：";
    Instream >> InList.ArraySize;
    cout << "录入数组元素值：\n";
    for (int i = 0; i < InList.ArraySize; i++) {
        cout << "元素" << i << "：";
        InStream >> InList.Element[i];    }
    return InStream;
}
```

```
template <class Type> void dataList <Type> :: Sort ( )
{ //按非递减顺序对ArraySize个关键码
  //Element[0]到Element[ArraySize-1]排序
   for ( int i = ArraySize-1; i > 0; i-- )
   {
      int j = MaxKey (0, i);
      if ( j != i ) swap (j, i);
   }
}
#endif
```

使用模板的选择排序算法的主函数

```
#include "selecttm.h"
const int SIZE = 10;
int main ( ) {
    dataList <int> TestList (SIZE);
    cin >> TestList;
    cout << TestList << endl;
    TestList.Sort ( );
    cout << TestList << endl;
    return 0;
}
```

模板的详细介绍——上机实践课

1.4 算法定义

- **定义：一个有穷的指令集**，这些指令为解决某一特定任务规定一个运算序列。
- **特性：**
  - **输入Input** 有0个或多个输入；
  - **输出Output** 有一个或多个输出（处理结果）；
  - **确定性** 每步定义都是确切无歧义的；
  - **有穷性** 算法应在执行有穷步后结束；
  - **有效性** 每一条运算应足够基本。

算法Algorithm=程序Program？

- **算法是有穷性** 算法应在执行有穷步后结束。
- 程序可能持续运行，直到系统退出，例如操作系统wait函数。
- 算法是面向问题的。
- 程序是算法的具体语言实现。

算法设计 自顶向下，逐步求精

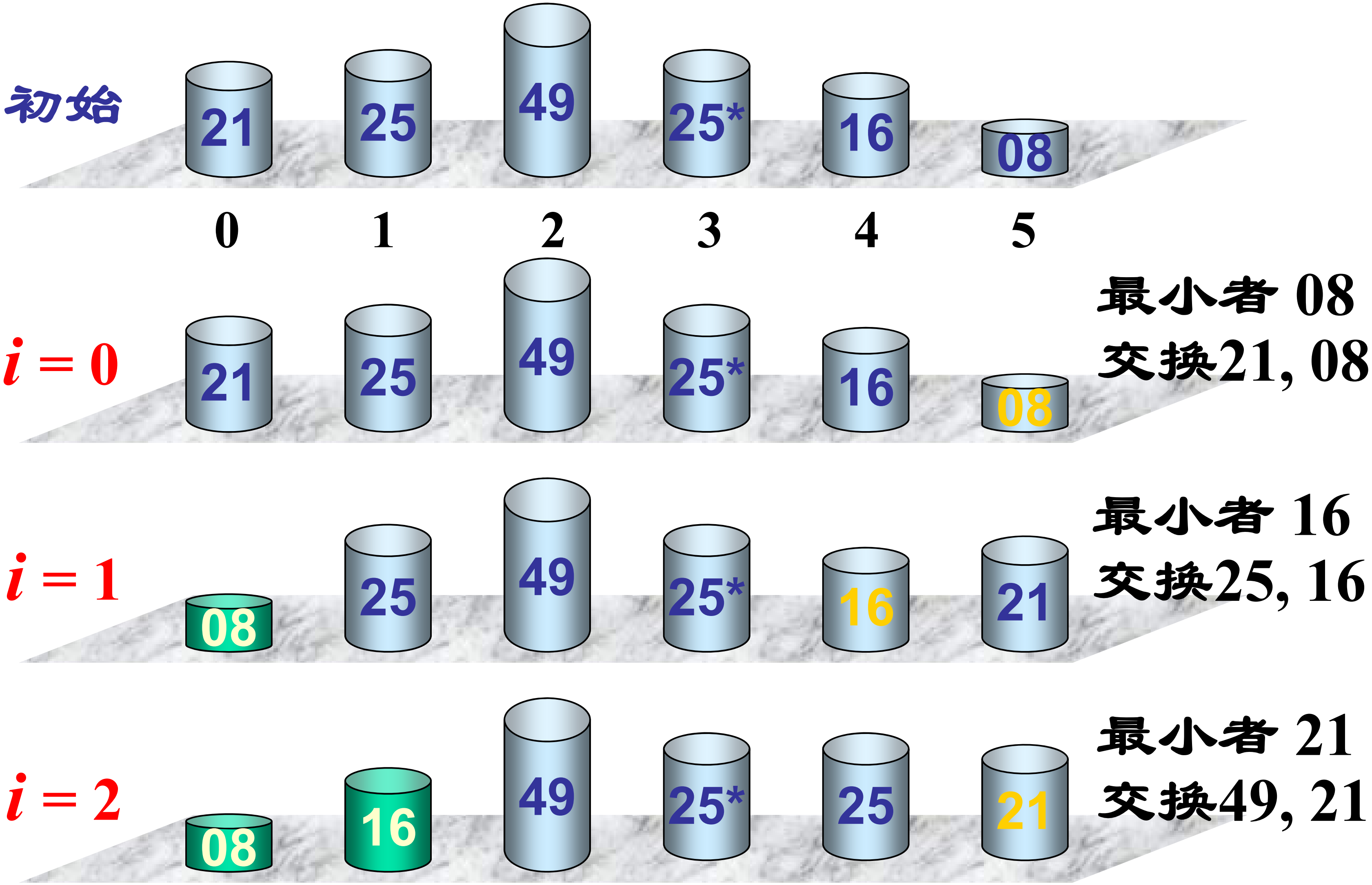
- **事例学习：n个整数的选择排序问题**

基本思想:

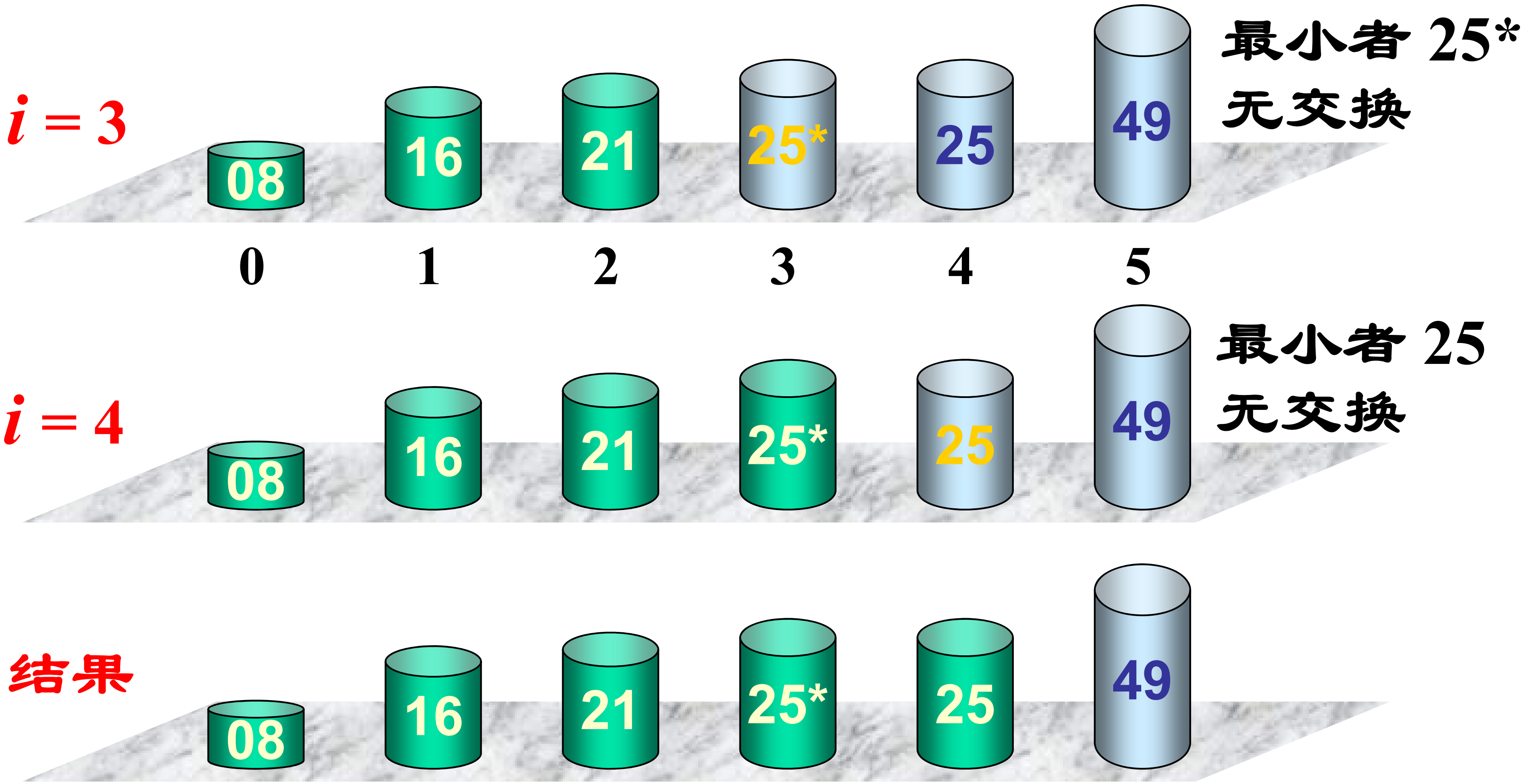
① 在一组对象 V[i] ~ V[n-1] 中选择具有最小排序码的对象；

② 若它不是这组对象中的第一个对象，则将它与这组对象中的第一个对象对调；

③ 在这组对象中剔除这个具有最小排序码的对象，在剩下的对象V[i+1] ~ V[n-1]中重复执行第①、②步，直到剩余对象只有一个为止。


初始
21 25 49 25* 16 08
0 1 2 3 4 5
i = 0
21 25 49 25* 16 08
最小者 08
交换21, 08
i = 1
08 25 49 25* 16 21
最小者 16
交换25, 16
i = 2
08 16 49 25* 25 21
最小者 21
交换49, 21

各趟排序后的结果



i = 3
08
16
21
25*
25
49
最小者 25*
无交换
0
1
2
3
4
5
i = 4
08
16
21
25*
25
49
最小者 25
无交换
结果
08
16
21
25*
25
49

各趟排序后的结果

- **明确问题：递增排序**
- **解决方案：逐个选择最小数据**
- **算法框架：**

```
for ( int i = 0; i < n-1; i++ )
{ //n-1趟从a[i]检查到a[n-1]
   若最小整数在a[k]，交换a[i]与a[k];
}
```

- **细化程序：程序 SelectSort**

```
void selectSort ( int a[ ], const int n )
{ //对n个整数a[0]到a[n-1]按递增顺序排序
    for ( int i = 0; i < n−1; i++ )
    {
        int k = i;
        //从a[i]查到a[n-1]，找最小整数，在a[k]
        for ( int j = i+1; j < n; j++ )
            if ( a[j] < a[k] ) k = j;
        int temp = a[i];  a[i] = a[k];  a[k] = temp;
    }
}
```

1.5 算法性能分析与度量

- 如何度量数据结构和算法的性能好坏?

- <u>算法的性能标准</u>
- <u>算法的后期测试</u>
- <u>算法的事前估计</u>

算法的性能标准

- 正确性
- 可使用性——用户友好性
- 可读性
  - 程序设计风格，印度软件产业，对大项目非常重要，CMM (Capability Maturity Model for Software)
  - 上机实践课详细介绍
- 效率——时间与空间代价
  - 确定问题规模与算法时间和空间的关系
- 健壮性——容错性
- 简单性

算法的后期测试

在算法中的某些部位插装时间函数 time ( ) 测定算法完成某一功能所花费时间。

举例说明

顺序搜索 (Sequenial Search)

```
int seqsearch ( int a[ ], const int n, const int x )
{  //在a[0], …, a[n-1]中搜索x
   int i = 0;
   while ( i < n && a[i] != x )
       i++;
   if ( i == n ) return -1;
   return i;
}
```

插装 time( ) 的计时程序

```
double start, stop;
time (&start);
int k = seqsearch (a, n, x);
time (&stop);
double runTime = stop - start;
cout << " " << n << " " << runTime << endl;
```

缺点:

1、与环境配置有关，不同的PC，不同的系统性能，产生不同的时间差值;

2、需要确定合适的计数方法，处理器自带的计数函数是毫秒级，因此如果少于毫秒的数量级，计数不准。

```
void TimeSearch ( )
{
    int a[1000], n[20];
    for ( int j=1; j<=1000; j++ )
        a[j-1] = j; //a[0]=1, a[1]=2, …初始化a
    for ( j=0; j<10; j++ ) {
        n[j] = 10*j; //n[0]=0, n[1]=10, n[2]=20
        n[j+10] = 100*( j+1 );
        //n[10]=100, n[11]=200, n[12]=300 ...
    }
    cout << "n time" << endl;
```

```
for ( j=0; j<20; j++ ) { //得到计算时间
      long start, stop;
      time (&start); //开始计时
      int k = seqsearch (a, n[j], 0); //不成功查找
      time (&stop); //停止计时
      long runTime = stop - start; //计算运行时间
      cout << " " << n[j] << " " <<
                      runTime << endl; //输出
  }
  cout << "Times are in hundredths of a second."
                << endl;
}
```

程序测试结果输出

n	0	10	20	30	40	50	60	70	80	90
run Time	0	0	0	0	0	0	0	0	0	0

n	100	200	300	400	500	600	700	800	900	1000
run Time	0	0	0	0	0	0	0	0	0	0

假设时间函数*time( )*的测量精度为0.01秒，程序测试结果都是0，表明时间函数的测试精度不够。

改进的计时程序

```
void TimeSearch ( ) {
  int a[1000], n[20];
  const long r[20] = {300000, 300000, 200000, 200000,
 100000, 100000, 100000, 80000, 80000, 50000, 50000,
 25000, 15000, 15000, 10000, 7500, 7000, 6000, 5000,
 5000 };
  for ( int j=1; j<=1000; j++ )
      a[j-1] = j; //初始化a
  for ( j=0; j<10; j++ ) { //为n赋值
      n[j] = 10*j;  n[j+10] = 100*( j+1 );  }
  cout << " n totalTime runTime " << endl;
```

```
for ( j=0; j<20; j++ ) { //得到计算时间
    long start, stop;
    time (&start); //开始计时
    for ( long b=1; b<=r[j]; b++ )
        int k = seqsearch( a, n[j], 0 );
        //不成功查找，执行r[j]次
    time (&stop); //停止计时
    long totalTime = stop - start;
    float runTime =
        (float)(totalTime)/(float)(r[j]);
    //总时间除以重复执行次数
    cout << " " << n[j] << " " << totalTime
         << " " << runTime << endl;
 }
}
```

程序修改后的测试结果输出

n	0	10	20	30	40	50	60	70	80	90
总的运行时间	241	533	582	736	467	565	659	604	681	472
单个运行时间	0.0008	0.0018	0.0029	0.0037	0.0047	0.0056	0.0066	0.0075	0.0085	0.0094
n	100	200	300	400	500	600	700	800	900	1000
总的运行时间	527	505	451	593	494	439	484	467	434	484
单个运行时间	0.0105	0.0202	0.0301	0.0395	0.0494	0.0585	0.0691	0.0778	0.0868	0.0968

测量数据与n基本呈线性关系

- 算法的运行时间依赖于所使用的计算机系统、编译器、可用存储空间大小等。
  - 同样的算法在速度不同的计算机上，执行速度相差非常大。
  - 算法用不同的编译器编译出的目标代码不一样长，完成同样功能所需时间不同。
  - 如果可用存储空间不够，算法需要的运行时间很多；如果空间足够大，则时间明显减少。
- 算法运行时间的测量用于评估算法的正确性和可用性，并不能判断算法的优劣。
  - 通过比较算法的复杂性来评价。
    - 算法复杂性与具体运行环境和编译器无关。

算法的事前估计

- 空间复杂度 (Space Complexity)
- 时间复杂度 (Time Complexity)

- 用来确定问题规模n（比如学生人数）与算法实现时需要的存储空间f(n)，程序步数或者时间开销g(n)的关系。
- 在目前的研究领域，对算法进行分析时，时空复杂性是最重要的基本功，最理想的算法评价标准。

空间复杂度度量

- **存储空间的固定部分**
  程序指令代码的空间，常数、简单变量、定长成分（如数组元素、结构成分、对象的数据成员等）变量所占空间。
- **可变部分**
  尺寸与实例特性有关的成分变量所占空间、引用变量所占空间、递归栈所用空间、通过new和delete命令动态使用空间。

例1：计算表达式

```
float abc(float a, folat b, float, c) {
    return a+b+b*c+(a+b-c)/(a+b)+4.0;  }
```

例2：以迭代方式求累加和的函数

```
float sum ( float a[ ], int n ) {
    float s = 0.0;
    for ( int i = 0; i < n; i++ )
        s += a[i];
    return s;  }
```

例3：以递归方式求累加和的函数

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

时间复杂度度量

- 编译时间
  - 与编译程序有关，与实例性质无关。
- 运行时间
  - 程序步
    - 语法上或语义上有意义的一段指令序列。
    - 执行时间与实例特性无关。
    - 例如，声明语句程序步数为0，表达式程序步数为1。

程序步确定方法

- 插入计数全局变量count；
- 建表，列出各语句的程序步。

例1：以迭代方式求累加和的函数

```
float sum ( float a[ ], int n )
{
    float s = 0.0;
    for ( int i = 0; i < n; i++ )
        s += a[i];
    return s;
}
```

在求累加和程序中加入*count*语句

```
float sum ( float a[ ], int n )
{
    float s = 0.0;
    count++;  //count 统计执行语句条数
    for ( int i = 0; i < n; i++ ) {
        count++;  //针对 for 语句
        s += a[i];
        count++;  //针对赋值语句  }
    count++;  //针对 for 的最后一次
    count++;  //针对 return 语句
    return s;
}
```

执行结束得程序步数? <u>**count = 2*n+3**</u>

程序的简化形式

```
void sum ( float a[ ], int n )
{
    for ( int i = 0; i < n; i++ )
        count += 2;
    count += 3;
}
```

注意：

一个语句本身的程序步数，可能不等于该语句一次执行所具有的程序步数。

例如：赋值语句 x = sum (R, n)

本身的程序步数为 1;

一次执行对函数 sum (R, n) 的调用需要的程序步数为 2*n+3；

一次执行的程序步数为

$$1+2*n+3 = 2*n+4$$

第二种方法：

计算累加和程序程序步数计算工作表格

程 序 语 句	一次执行所需程序步数	执行频度	程序步数
{	0	1	0
float s = 0.0;	1	1	1
for (**int** i=0; i<n; i++)	1	n+1	n+1
s += a[i];	1	n	n
return s;	1	1	1
}	0	1	0
	总程序步数		2n+3

例2：以递归方式求累加和的函数

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

在求累加和程序中加入*count*语句

```
float rsum ( float a[ ], int n )
{
    count++;  //针对if语句
    if (n<=0) {
        count++;  //针对 return语句
        return 0;
    }
    else {
        count+=2;  //针对else与return语句
        return rsum(a, n-1)+a[n-1];
    }
}
```

设count初始值为0，Trsum(n)是程序执行后的count值。

n=0，Trsum(0)=2;

n>0，Trsum(n)=Trsum(n-1)+3;

$$
\begin{aligned}
Trsum(n) &= 3+Trsum(n-1) \\
&= 3+3+Trsum(n-2) = 3*2+Trsum(n-2) \\
&= 3+3+3+Trsum(n-3) = 3*3+Trsum(n-3) \\
&= \ldots\ldots = 3*n+Trsum(0) \\
&= 3n+2
\end{aligned}
$$

计算累加和程序程序步数计算工作表格

程序语句	一次执行所需程序步数	执行频度		程序步数	
		n=0	n>0	n=0	n>0
{	0	1	1	0	0
if (n<=0)	1	1	1	1	1
return 0;	1	1	0	1	0
else return	2+Trsum(n-1)	0	1	0	2+Trsum (n-1)
rsum(a, n-1)+a[n-1];				1	
}	0	1	1	0	0
	总程序步数			2	3+Trsum (n-1)

- 程序步本身就不是一个准确的概念，而是一个抽象的概念。
- 再作一次抽象，从由多种因素构成的时间复杂性中抽取出其主要因素，将常数抽象为**1**，有利于抓住主要矛盾，简化复杂性分析。
- 大**O**表示法

时间复杂度的渐进表示法

例：求两个n阶方阵的乘积 $C = A \times B$

```
void MatrixMultiply ( int A[n][n], int B[n][n],
        int C[n][n] )
{
   for ( int i = 0; i < n; i++ )            … n+1
     for ( int j = 0; j < n; j++ ) {        … n(n+1)
       C[i][j] = 0;                         … n²
       for ( int k = 0; k < n; k++ )        … n²(n+1)
         C[i][j] = C[i][j] + A[i][k] * B[k][j];
                                            … n³  }
}
```

$$2n^3 + 3n^2 + 2n + 1$$

- 算法中所有语句的频度之和是**矩阵阶数n**的函数

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

- 一般地，称 n 是问题的规模。则时间复杂度 T(n) 是问题规模 n 的函数。
- 当n趋于无穷大时，把时间复杂度的**数量级**（阶）称为算法的渐进时间复杂度。

$T(n) = O(n^3)$ ----**大O表示法**

大O表示法——最坏情况

- 当且仅当存在正整数c和n_0，使得$T(n) \le cf(n)$对所有的$n \ge n_0$成立，则称该算法的渐进时间复杂度为$T(n)=O(f(n))$。

- 当实例特性n充分大时，算法的时间复杂度随n变化，在最坏情况下若存在一个增长的上界，即$cf(n)$，则该算法的时间复杂度增长的数量级为$f(n)$，即称该算法的渐进时间复杂度为$T(n)=O(f(n))$。

- **大O表示法的使用**
  - 需要考虑**关键操作的程序步数**。
    - 关键操作大多在循环和递归中；
    - 在多数场合中，程序步骤与执行频度一一对应。
  - 如果给出的是渐进值，可直接考虑关键操作的执行频度，提出其与实例特性n的函数关系$g(n)$。

- **渐进时间复杂度的计算**
  - **单个循环**
    - 循环内的简单语句即为关键操作，该程序段的渐进时间复杂度应是此关键操作的执行频度的大O表示。
  - **几个并列的循环**
    - 分析每个循环的渐进时间复杂度，然后利用大O表示法的加法规则来计算渐进时间复杂度。
  - **多层的嵌套循环**
    - 关键操作应该在最内层循环中，先自外向内层层分析每层循环的时间渐进复杂度，然后利用大O表示法的乘法规则来计算渐进时间复杂度。

- **加法规则** 针对并列程序段

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

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

- 取c、$log_2 n$、n、$nlog_2 n$时间效率比较高；
- 取n^2、n^3时间效率差强人意；
- 取2^n、3^n、$n!$当n稍微大一点，算法的时间代价变为很大，以至于不能计算。

变量计数

```
x = 0;  y = 0;
```
$T1(n) = O(1)$

```
for ( int k = 0; k < n; k ++ )
    x ++;
```
$T2(n) = O(n)$

```
for ( int i = 0; i < n; i++ )
    for ( int j = 0; j < n; j++ )
        y ++;
```
$T3(n) = O(n^2)$

$$T(n) = T1(n)+T2(n)+T3(n) = O(\ \max(\ 1, n, n^2\ )\ )$$
$$= O(n^2)$$

两个并列循环的例子

```
void exam ( float x[ ][ ], int m, int n )
{
    float sum [ ];
    for ( int i = 0; i < m; i++ ) {  //x中各行
        sum[i] = 0.0;  //数据累加
        for ( int j = 0; j < n; j++ )
            sum[i] += x[i][j];  }
    for ( i = 0; i < m; i++ )  //打印各行数据和
        cout << i << " : " <<sum [i] << endl;
}
```

渐进时间复杂度为 $O(max\ (m*n, m))$

- **乘法规则** 针对嵌套程序段

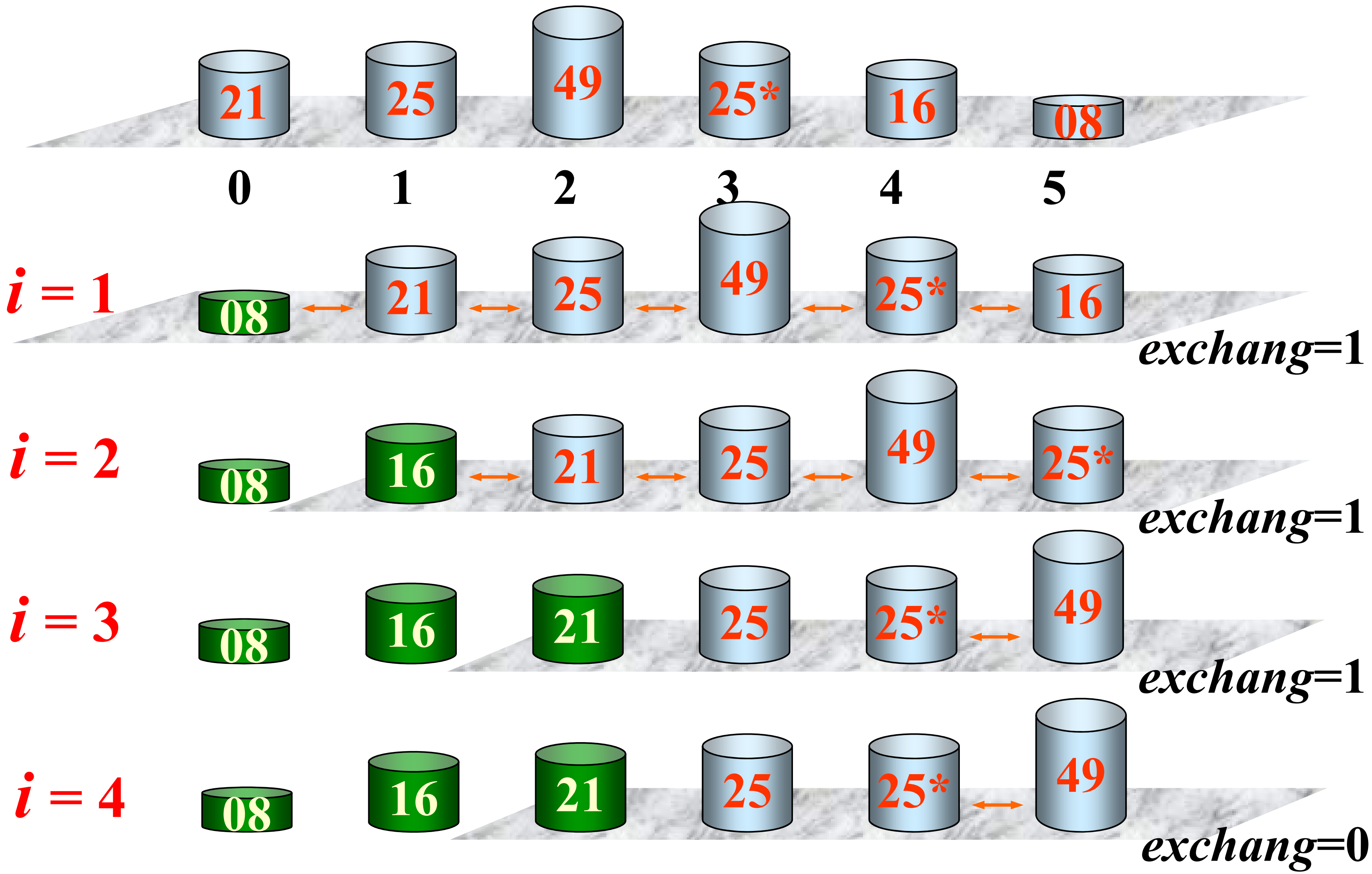
$$T(n, m) = T1(n) * T2(m) = O(f(n) * g(m))$$

如果一个程序的循环中有一个包含有循环的函数调用语句，也可在被调用函数内部寻找关键操作，使用乘法规则来计算渐进时间复杂度。

两个嵌套循环的例子 起泡排序（递增）

基本思想：

设待排序对象序列中的对象个数为 n。最多作 n-1 趟，$i = 1, 2, \ldots, n$-1 。在第 i 趟中从后向前，$j = n$-1, n-2, …, i，顺次两两比较V[j-1].key和V[j].key。如果发生逆序，则交换V[j-1]和V[j]。


21
25
49
25*
16
08
0
1
2
3
4
5
i = 1
08
21
25
49
25*
16
exchang=1
i = 2
08
16
21
25
49
25*
exchang=1
i = 3
08
16
21
25
25*
49
exchang=1
i = 4
08
16
21
25
25*
49
exchang=0

各趟排序后的结果

```
template <class Type> void dataList <Type> ::
bubbleSort ( ) {
//对表逐趟比较, ArraySize 是表当前长度
      int i = 1;  int exchange = 1;
      //当 exchange 为 0 则停止排序
      while ( i < ArraySize && exchange ) {
         BubbleExchange ( i, exchange );
         i++;
      } //一趟比较
}
```

```
template <class Type> void dataList <Type> ::
BubbleExchange(int i, int &exchange ) {
    exchange = 0;  //假定元素未交换
    for ( int j = ArraySize-1; j >= i; j--)
        if ( Element[j-1] > Element[j] ) {
            Swap ( j-1, j );  //发生逆序，交换
            exchange = 1;  //做“发生交换”标志
        }
}
```

渐进时间复杂度

$$O(f(n)*g(n)) = O(n^2)$$

$$\because \sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$$

BubblrSort　　n-1趟

BubbleExchange ( )

n-i次比较

渐进的空间复杂度

- 当实例特性n充分大时，需要的存储空间体积将如何随之变化。
  - 非程序指令、常数、指针等所需要的存储空间；
  - 非输入数据所占用的存储空间；
  - 为解决问题所需要的辅助存储空间。
- 大O表示法
  - 设$S(n)$是算法渐进空间复杂度，在最坏情况下，可以表示为实例特性n的某个函数$f(n)$的数量级：

$$S(n) = O(f(n))$$

两种代价计算方法的比较

- **事前估计——复杂性计算**
  - 更全面地分析程序的执行代价，例如从最坏的情况对程序的代价进行估计，如果知道数据的分布情况还可以对程序执行的平均代价进行估计。
  - 一般这种方法难以获得程序的具体执行时间。
- **事后测试**
  - 可获取程序每次执行所需的时间和空间，这种方法获得的结果是针对某特定的数据和情况下获得。
  - 要获得程序的整体执行效率，需要经过多次反复的测试，并精心设计测试数据。
- 在实际应用中，往往是通过将两者结合的方式对系统的性能进行分析。

本章小结

- 主要讨论DS中的基本概念和性能分析方法。
- 知识点：**Data、Data Object、Data Element、Data Structure、Data Type、ADT、OO、Class**、抽象的层次
- 算法的概念、六个特性及其性能分析方法（时间复杂度和空间复杂度）

随堂练习

例1：分析程序段"i=1; while(i<=n) i=i*2;"的时间复杂度。

例2：有如下计算n!的递归函数Fact(n)，分析其时间复杂度。

```
Fact(int n)
{
    if(n<=1)  return(1);
    else  return (n*Fact(n-1));
}
```

例 1：程序段"i=1; while(i<=n) i=i*2;"的时间复杂度为 $O(\log_2 n)$。

i=i*2，即循环次数 k 满足 $2^k=n$，因此 $k=\log_2 n$。

例 2：有如下计算 n!的递归函数 Fact(n)，分析其时间复杂度。

```
Fact(int n)
{
    if(n<=1) return(1);
    else return (n*Fact(n-1));
```

设 Fact(n)的运行时间函数为 T(n)。该函数中语句 if(n<=1) return(1);的运行时间为 O(1)，递归调用 Fact(n-1)的时间是 T(n-1)，故 else return (n*Fact(n-1));的运行时间为 O(1)+T(n-1)。其中，设两数相乘和赋值操作的运行时间为 O(1)，则对某常数 C、D 有：

$$T(n)=\begin{cases} D & n\le 1 \\ C+T(n-1) & n>1 \end{cases}$$。

现在，来求解该方程。设 n>2，利用上式对 T(n-1)展开，即在上式中用 n-1 替代 n 得到：T(n-1)=C+T(n-2)，并代入 T(n)=C+T(n-1)中，即当 n>2 时有：T(n)=2C+T(n-2)。同理，当 n>3 时有：T(n)=3C+T(n-3)。因此，当 n>i 时有：T(n)=iC+T(n-i)。

最后，当 i=n-1 时有：T(n)=(n-1)C+T(1)=(n-1)C+D。

即 T(n)=O(n)。

课程习题

- 笔做题——**1.4, 1.11**（以作业形式提交）
- 上机题——**1.18**
- 思考题——**1.10, 1.12, 1.13, 1.14**

Any Suggestion or Question

联系方式：

- 张玥杰

Email: yjzhang@fudan.edu.cn