

第一章 绪论

什么是数据结构？

程序 = 数据结构 + 算法

瑞士计算机科学家尼古拉斯·沃斯（Niklaus Wirth）提出的一个著名公式。这个公式揭示了程序设计的核心思想，即程序的本质是数据结构和算法的结合。

第一章 绪论

- 数据结构的基本概念
- 基本的数据结构
 - 线性结构
 - 线性表
 - 栈和队列
 - 数组、串与广义表
 - 非线性结构
 - 树
 - 图
- 基本的数据处理技术
 - 搜索技术（集合与字典）
 - 排序技术

第一章 绪论



第一章 绪论

§1.1 数据结构的概念

早期：数值计算 ——

运算对象是简单的整型、实型或布尔类型数据

$$\begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 0 & 1 \\ 5 & 2 \end{bmatrix} =$$

第一章 绪论

§1.1 数据结构的概念

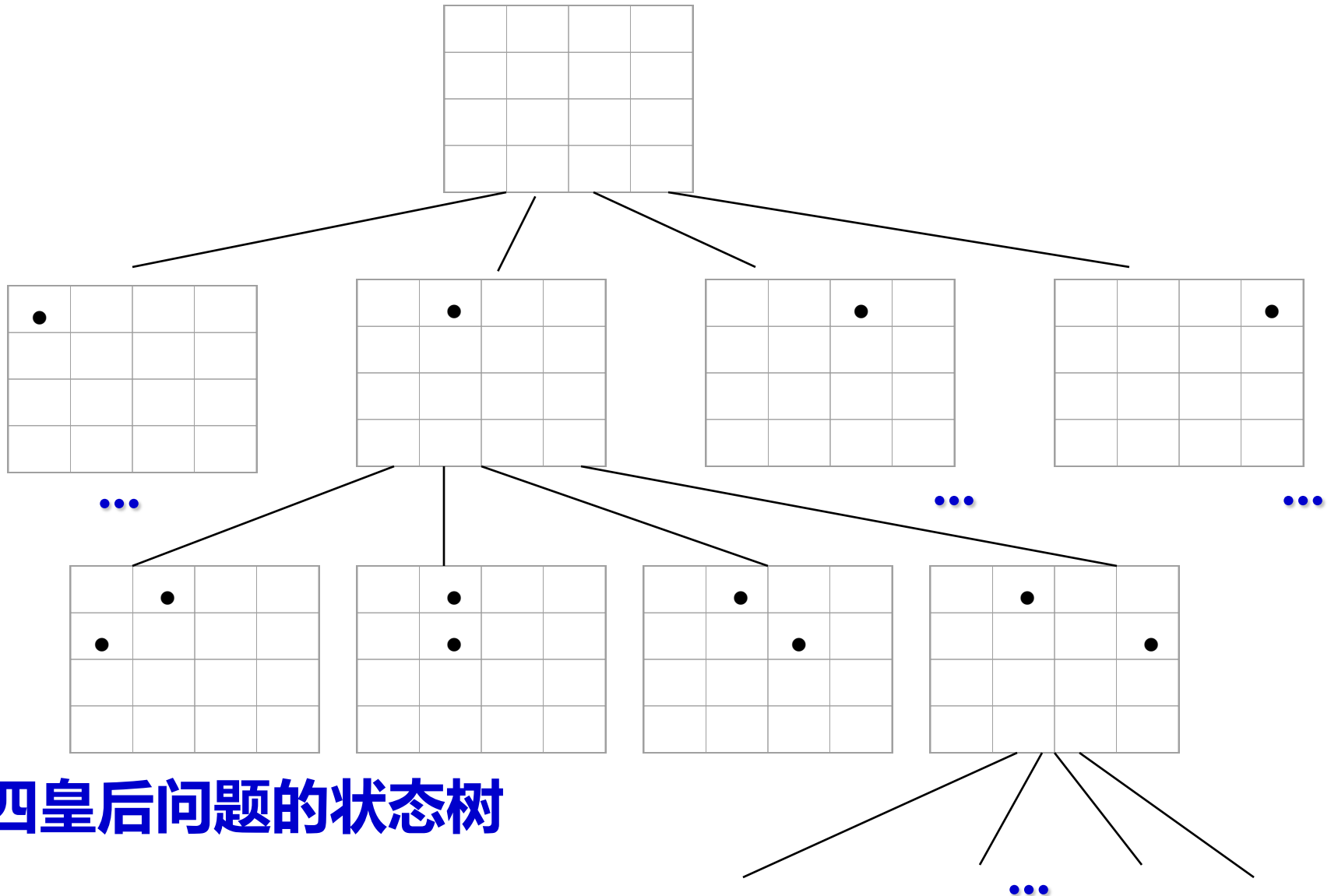
中后期： 非数值计算 ——

处理对象是类型复杂的数据，数据元素之间的相互关系一般无法用数学方程式加以描述

例1: “学生” 表格

	厦 羔	务 鹿	勿爱	荷 俘	泊雀猫肆
1	98131	莱豪协	蚂	m 寂	1979.12
2	98164	修仓雀	蚂	劈 持	1979.07
3	98165	廊确缴	蚂	霜 及	1981.02
4	98182	胰骗棍	檬	浮 赠	1980.10
5	98203	框侈窖	蚂	请 缚	1980.05
6	98224	恭 烫	蚂	是 疑	1981.01
7	98236	硒码宵	檬	神 赠	1980.03
8	98297	啡 柯	蚂	m 寂	1981.01
9	98310	拌渭堪	檬	玖 碌	1981.02
10	98318	鞭 唤	蚂	肛 赠	1979.12

例2：八皇后问题(用四皇后描述)

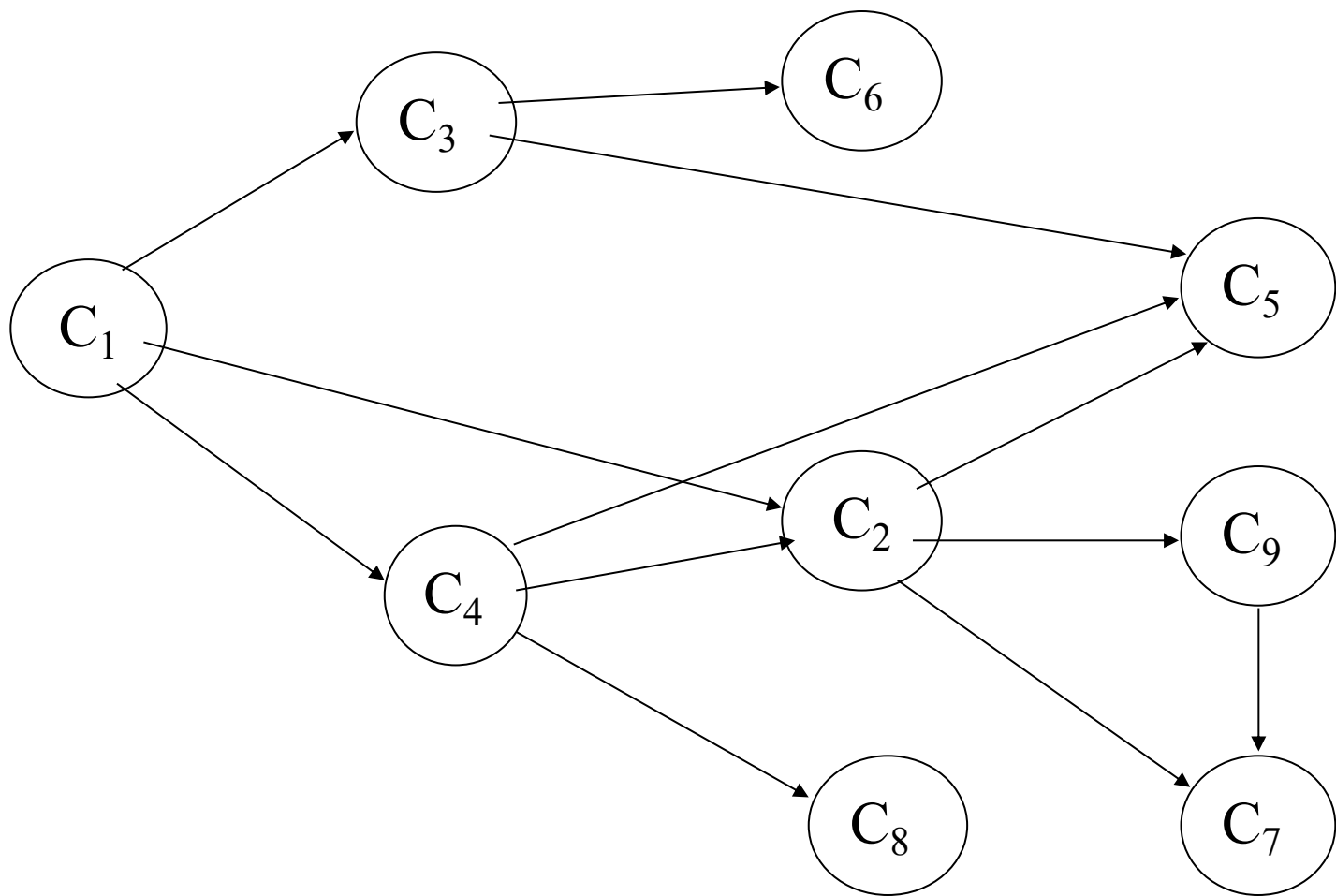


四皇后问题的状态树

例3：教学计划编排问题

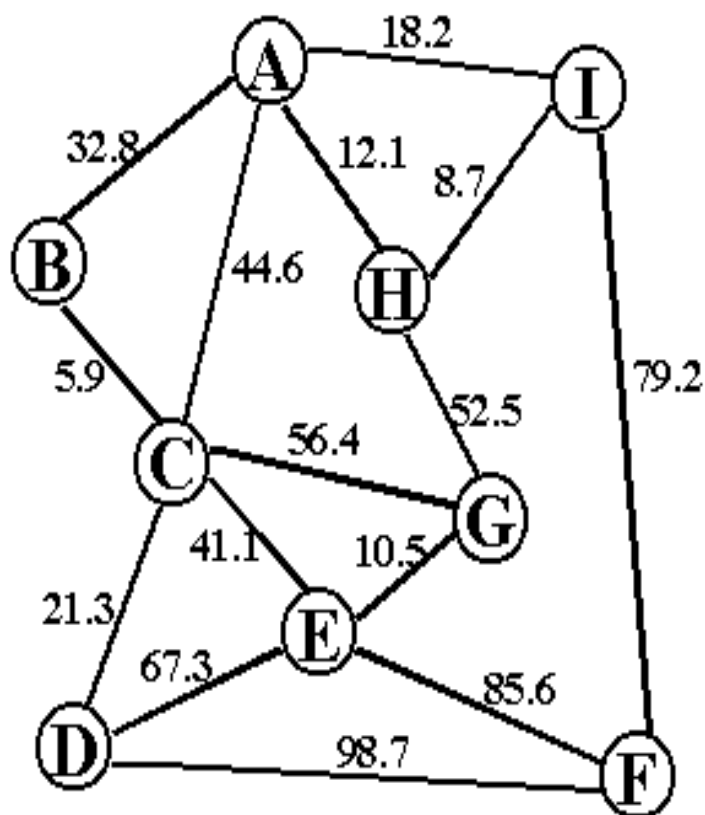
课程编号	课程名称	先修课程
C ₁	计算机导论	无
C ₂	数据结构	C ₁ , C ₄
C ₃	汇编语言	C ₁
C ₄	C程序设计语言	C ₁
C ₅	计算机图形学	C ₂ , C ₃ , C ₄
C ₆	接口技术	C ₃
C ₇	数据库原理	C ₂ , C ₉
C ₈	编译原理	C ₄
C ₉	操作系统	C ₂

(a) 计算机专业的课程设置

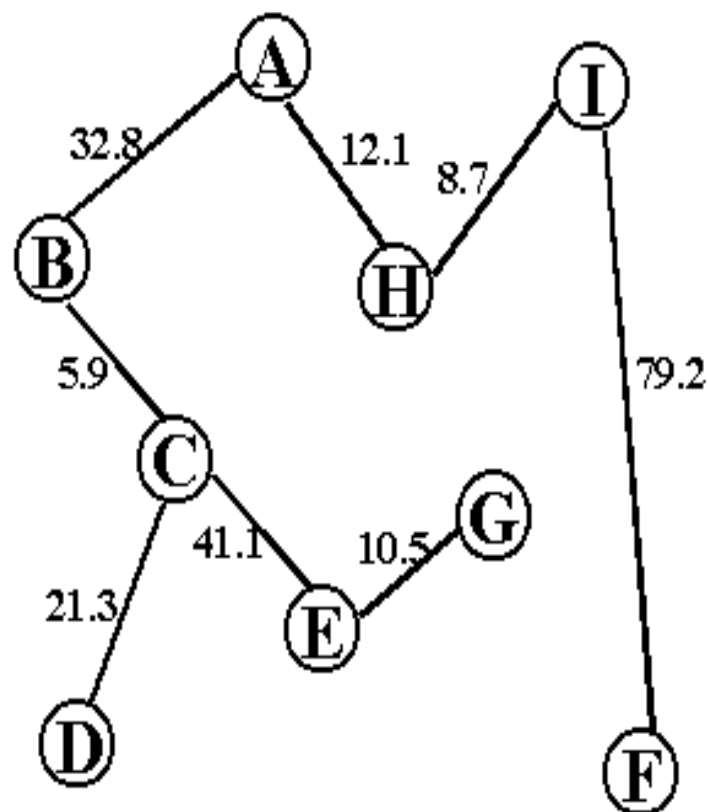


(b) 表示课程之间优先关系的有向图

例4：城市的煤气管道问题



(a) 结点间管道的代价



(b) 最经济的管道铺设

➤ 描述这类非数值计算问题的数学模型不再是数学方程，而是诸如表、树、图之类的数据结构。

➤ 数据结构是一门研究（非数值计算的）程序设计问题中所出现的计算机操作对象以及它们之间的关系和操作的学科。

■ 课程学习前掌握的基本概念：

- 数据
- 数据元素（数据成员）
- 数据对象

- **数据**：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（数值、字符等）的集合。
- **数据元素（数据成员）**：是数据的基本单位。在不同的条件下，数据元素又可称为元素、结点、顶点、记录等。

- **数据对象**：具有相同性质的数据元素（数据成员）的集合。
 - 整数数据对象 $N = \{ 0, \pm 1, \pm 2, \dots \}$
 - 学生数据对象

数据结构的形式定义

数据结构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：

$$\text{Data_Structure} = \{D, R\}$$

其中，**D**是某一数据对象，**R**是该对象中所有数据成员之间的关系的有限集合。

数据结构涉及三个方面：

1. 数据的**逻辑结构**-----从用户视图看，是面向问题的。
2. 数据的**物理结构**（存储结构）-----从具体实现视图看，是面向计算机的。
3. 相关的操作及其实现。

Example:

学生表：逻辑结构-----线性表

物理结构-----数组, 链表

操作-----插入, 删除, 查找

数据结构包括 **“逻辑结构”** 和 **“物理结构”** 两个方面(层次):

- **逻辑结构** 是对数据成员之间的逻辑关系的描述, 它可以用一个数据成员的集合和定义在此集合上的若干关系来表示;
- **物理结构** 是逻辑结构在计算机中的表示和实现, 故又称 **“存储结构”**。

逻辑结构和物理结构的关系

- 数据的**逻辑结构**是从逻辑关系（某种顺序）上观察数据，它是独立于计算机的；可以在理论上、形式上进行研究、推理、运算等各种操作。
- 数据的**存储结构**是逻辑结构在计算机中的实现，是依赖于计算机的；是数据的最终组织形式。
- 任何一个**算法的设计**取决于选定的逻辑结构；而**算法的最终实现**依赖于采用的存储结构。

根据问题来建立逻辑结构

例如: Class = (D, S)

数据集合: $D = \{ a, b_1, \dots, b_n, c_1, \dots, c_n, d_1, \dots, d_n \}$

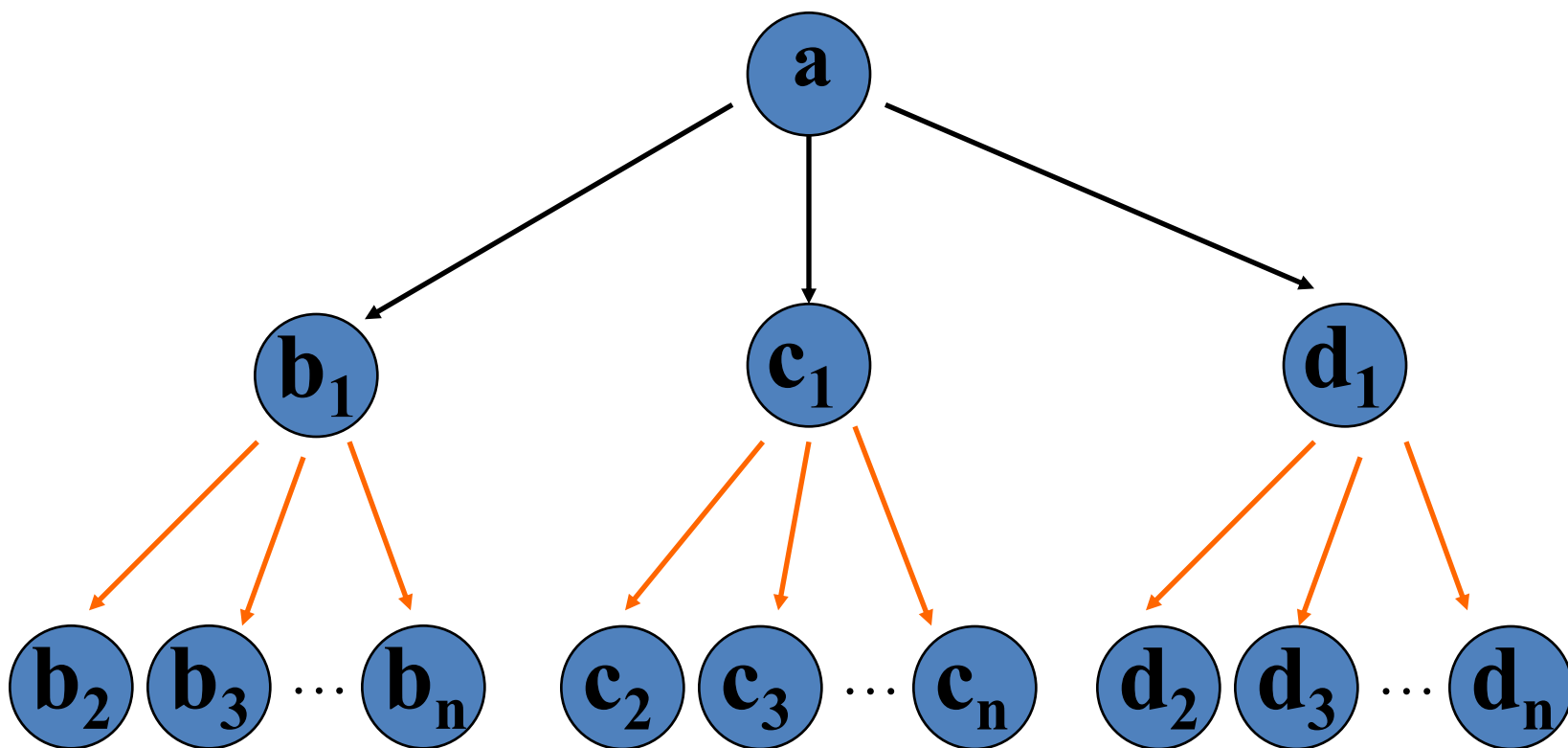
关系集合: $S = \{ R_1, R_2 \}$

$R_1 = \{ \langle a, b_1 \rangle, \langle a, c_1 \rangle, \langle a, d_1 \rangle \}$

//班长-组长

$R_2 = \{ \langle b_1, b_j \rangle, \langle c_1, c_j \rangle, \langle d_1, d_j \rangle \mid j = 2, 3, \dots, n \}$

//组长-组员



班级Class的逻辑结构的图示

存储结构是逻辑结构在存储器中的映象。

数据元素的映象：任何数据元素在计算机中最终都是转化为一个二进制的位串。

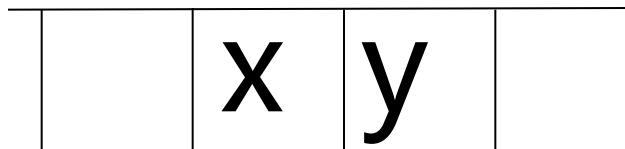
关系的映象：数据元素之间的逻辑关系最终转化为计算机中的存储位置关系或指针链接信息。

关系的映象方法：（关系对 $\langle x, y \rangle$ ）

1. 顺序映象（顺序存储方法）：

以相对的存储位置表示后继关系

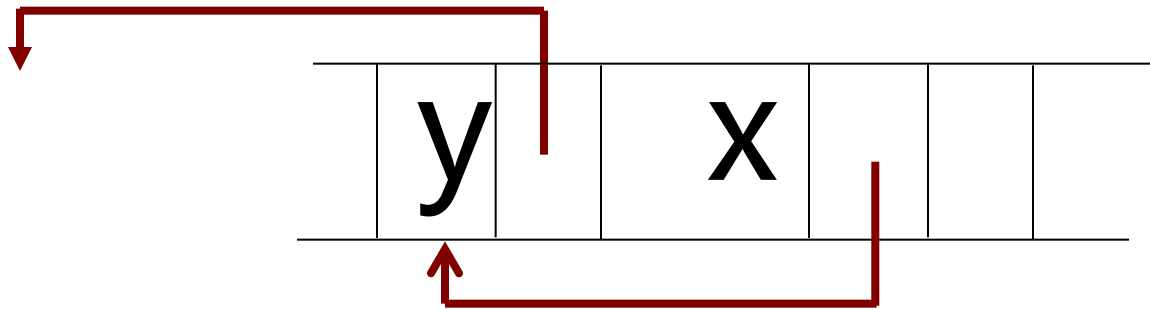
例如：令 y 的存储位置和 x 的存储位置之间差一个常量 C ，而 C 是一个隐含值，整个存储结构中只含数据元素本身的信息



2.链式映象（链接存储方法）：

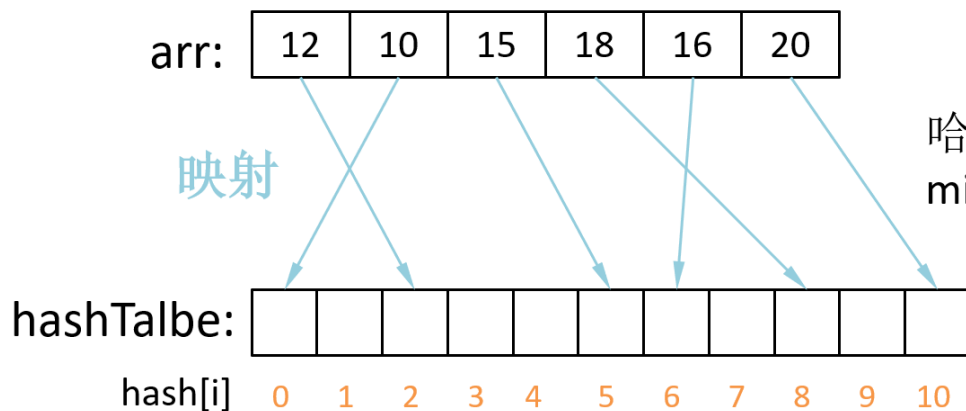
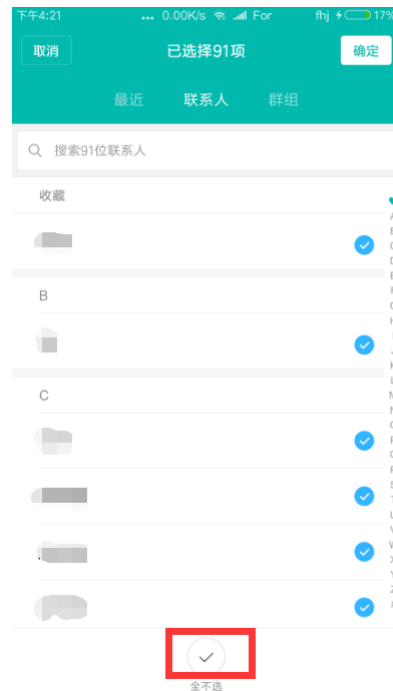
以附加信息(指针)表示后继关系

需要用一個和 x 在一起的**附加信息**
(指针) 指示 y 的存储位置



3.索引存储方法

4.散列存储方法



哈希函数: $\text{arr}[i] - \text{min}$
 $\text{min} = 10$

数据结构的分类

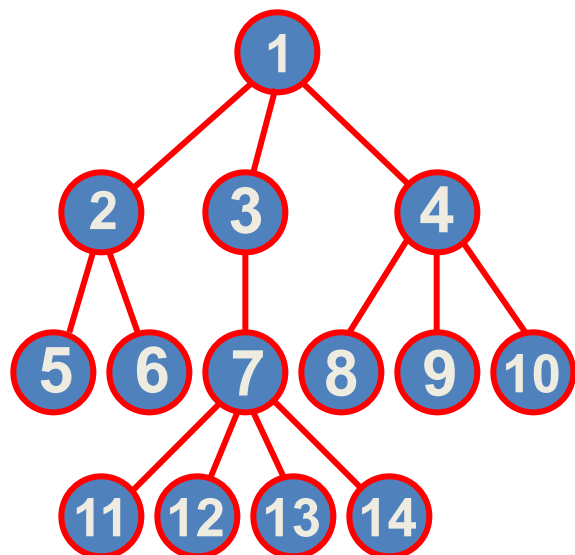
- 线性结构：表、栈、队列
- 非线性结构
 - 层次结构：树，二叉树，堆
 - 网状结构：图
 - 其它：集合

线性结构

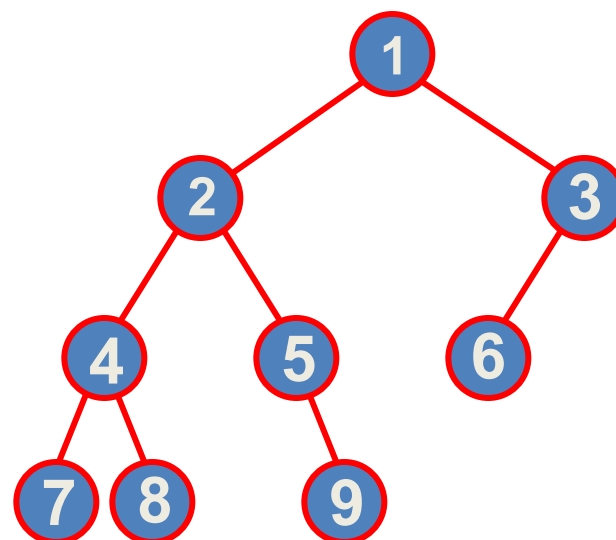


非线性结构——层次结构

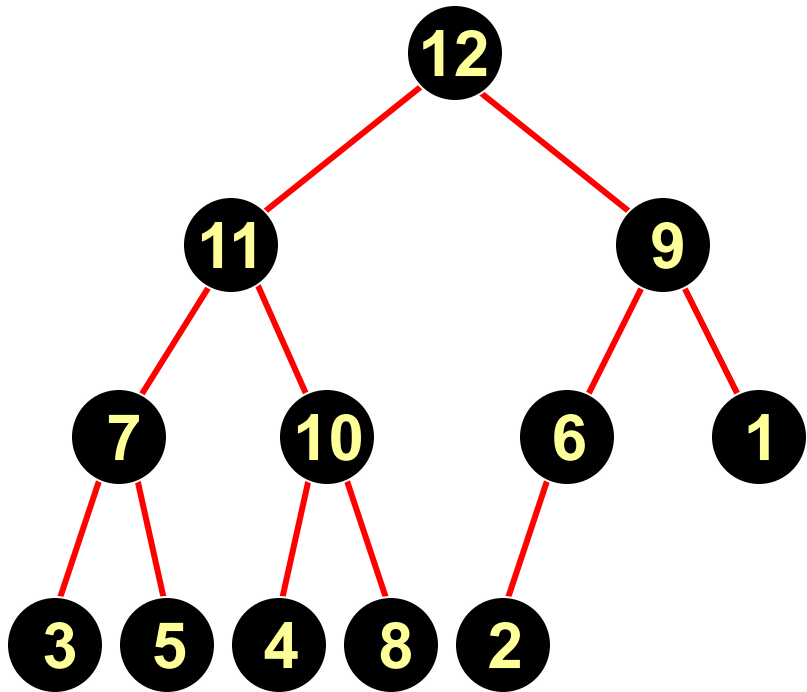
树



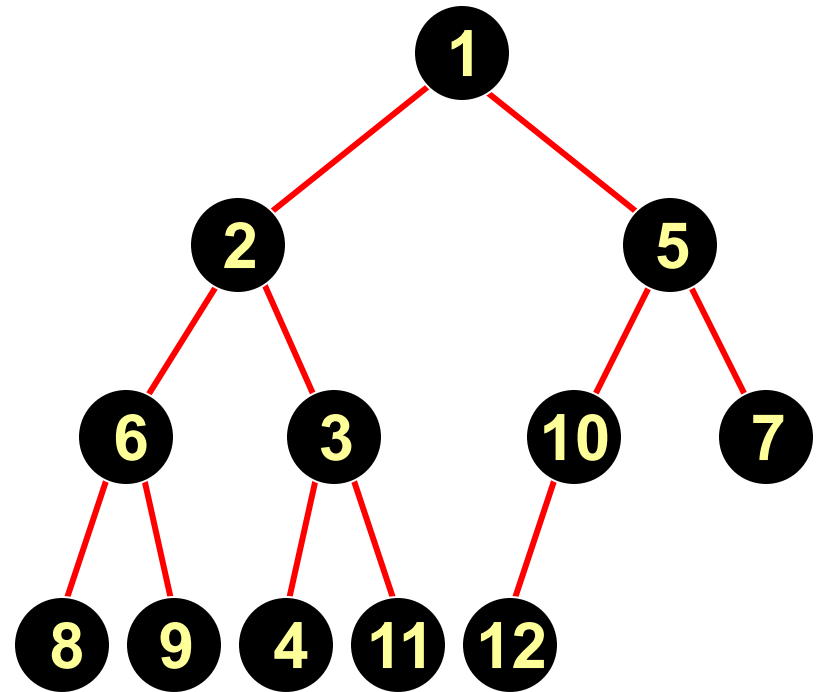
二叉树



堆（特殊的树结构）

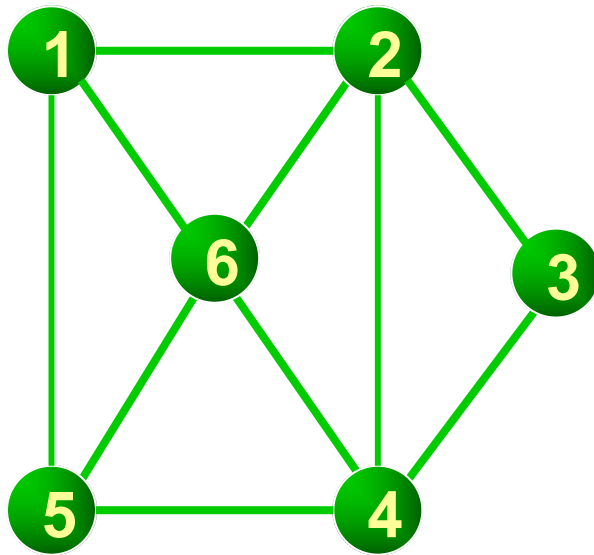


“最大” 堆

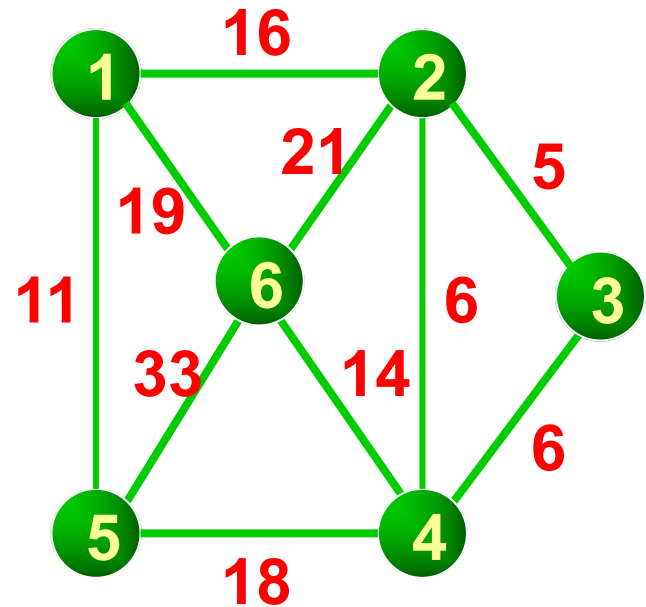


“最小” 堆

非线性结构——群结构



图结构



网络结构

§1.2 数据结构的抽象形式

- 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

- 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.

不同类型的变量，其所能取的值的范围不同，所能进行的操作不同。

例如：整型 (int)

值的范围是：-32768 ~ 32767（16位）

操作是：+，-，*，/，%

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
- 由基本的数据类型组成，并包括一组相关的服务（或称操作）
- 支持了逻辑设计和物理实现的分离，支持封装和信息隐蔽

抽象：抽取反映问题本质的东西，忽略非本质的细节

抽象数据类型的两种视图：

- 设计者的角度：

根据问题来定义抽象数据类型所包含的信息，给出其相关功能的实现，并提供公共界面的接口。

- 用户的角度：

使用公共界面的接口对抽象数据类型进行操作，不需要考虑其物理实现。对于外部用户来说，抽象数据类型应该是一个黑盒子。

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 $x, y \in \textit{NaturalNumber}$;
 $\textit{False}, \textit{True} \in \textit{Boolean}$, $+$ 、 $-$ 、 $<$ 、 $==$ 、 $=$ 等都是可用的服务。

Zero() :

返回自然数0

NaturalNumber

IsZero(x) :

Boolean

Add (x, y) :

NaturalNumber

Subtract (x, y) :

NaturalNumber

Equal (x, y) :

Boolean

Successor (x) :

NaturalNumber

if $(x == 0)$ 返回 *True*

else 返回 *False*

if $(x + y \leq \text{MaxInt})$ 返回 $x + y$

else 返回 *MaxInt*

if $(x < y)$ 返回 0

else 返回 $x - y$

if $(x == y)$ 返回 *True*

else 返回 *False*

if $(x == \text{MaxInt})$ 返回 x

else 返回 $x + 1$

end *NaturalNumber*

§1.3 作为ADT的C++类

- 面向对象的概念

面向对象 = 对象 + 类 + 继承 + 通信

面向对象方法中类的定义充分体现了抽象数据类型的思想

用C++语言描述面向对象程序

- 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

§1.4 算法定义

Nicklaus Wirth

(1984年Turing Award)



算法 + 数据结构 = 程序

(Algorithms + Data Structures = Programs)

处理问题的
策略

问题的数
学模型

为计算机处理
问题编制的一
组指令集

算法的描述

常用的算法的描述方式：

算法描述

自然语言

流程图 特定的表示算法的图形符号

伪语言 包括程序设计语言的三大基本结构及自然语言的一种语言

类语言 类似高级语言的语言，
例如，类**PASCAL**
类**C**语言

一个算法必须满足以下**五个重要特性**:

- **有输入** 有0个或多个输入
- **有输出** 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

```
void selectSort (int a[ ], const int n ) {  
    //按非递减顺序对 $n$ 个关键码  
    // $a[0] \sim a[n-1]$ 排序  
    for ( int  $i = n - 1$ ;  $i > 0$ ;  $i--$  ) {  
        int  $j = \text{MaxKey}$  (a, 0,  $i$ );  
        if ( $j \neq i$ ) swap ( $j$ ,  $i$ );  
    }  
}
```

```
int MaxKey (int a[ ], const int low, const int high) {
```

```
//查找数组a[low] ~ a[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 ( a[max] < a[k] )
```

```
            max = k;
```

```
    return max;
```

```
}
```


§1.5 算法性能分析与度量

算法的性能标准:

- 正确性
- 可使用性
- 可读性
- 效率
- 健壮性
- 简单性

算法的效率包括**时间代价**和**空间代价**，前者指的是**算法执行时间**；后者指的是算法执行过程中**所需的最大存储空间**。两者都与问题的规模有关。

算法效率的衡量方法：

- 后期测试
- 事前估计

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的时间

与运行机器的CPU，存储等有关

算法的事前估计

- 算法的复杂性度量属于事前估计

- 空间复杂度

算法执行时所占用的存储空间

- 时间复杂度

算法执行时所耗费的时间

时间复杂度

- 编译时间

- 运行时间

- 程序步：语法（义）上有意义的一段指令

例如：

注释：程序步数为0

声明语句：程序步数为0

表达式、赋值语句：程序步数为1

循环语句：循环控制语句每次执行的程序步数为1

- 程序步确定方法
 - ◆ 插入计数全局变量 *count*

例 以迭代方式求累加和的函数

```
行  float sum ( float a[ ], const int n )  
1    {  
2        float s = 0.0;  
3        for ( int i = 0; i < n; i++ )  
4            s += a[i];  
5        return s;  
6    }
```

在求累加和程序中加入 $count$ 语句

```
float sum ( float a[ ], const int n )      {  
    float s = 0.0;  
     $count++$ ;           // $count$ 统计执行语句条数  
    for ( int i = 0; i < n; i++ ) {  
         $count+=2$ ;      //针对for语句  
        s += a[i];  
         $count++$ ;      //针对赋值语句  
    }  
     $count+=2$ ;          //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  $count=3 * n + 4$ 
```


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

$x = \text{sum}(R, n);$

本身的程序步数为1;

一次执行对函数 $\text{sum}(R, n)$ 的调用需要的程序步数为 $3*n+4$;

一次执行的程序步数为

$$1+3*n+4 = 3*n+5$$

- **确定程序的准确程序步数十分困难。程序步数本身也不是一个精确的概念，不能确切反映运行时间。**

算法的渐近分析

- 一个算法的“运行时间”通常是随问题规模的增长而增长,它是问题规模(用正整数 n 表示)的函数。
- 统计算法中的关键操作,以关键操作重复执行的次数作为算法的时间度量。
- 算法中关键操作重复执行的次数是规模 n 的某个函数 $T(n)$

例1: $s = 0;$

例2: $s = 0;$ // $a[]$ 中各行进行累加
for (int $i = 0; i < n; i++$)
 $s += a[i];$

例3:
for (int $i = 0; i < n; i++$) { // $x[][]$ 中各行数据累加
 $sum[i] = 0.0;$
 for (int $j=0; j<n; j++$)
 $sum[i] += x[i][j];$
}

各个程序段中的关键操作重复执行的次数?

- 要精确地计算 $T(n)$ 通常较困难
- 引入渐近时间复杂度——在数量级上估计一个算法的执行时间，也能够达到分析算法的目的。（观察算法随 n 增长的趋势）

(大O记号): $T(n) = O(f(n))$

大O表示法：当且仅当存在正整数 c 和 n_0 ，使得 $T(n) \leq c * f(n)$ 对所有的 $n \geq n_0$ 成立，则称该算法的时间增长率在 $O(f(n))$ 中，记为 $T(n) = O(f(n))$ 。

例：一个程序的实际执行时间为

$$T(n) = 2.7n^3 + 3.8n^2 + 5.3 \quad \text{则} \quad T(n) = O(n^3)$$

➤ 使用大 O 记号表示的算法的时间复杂度，称为算法的渐近时间复杂度（Asymptotic Complexity）

➤ 通常用 $O(1)$ 表示常数计算时间。常见的渐近时间复杂度有：

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

大O表示法的几条规则:

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

- 加法规则 针对并列程序段

$$\begin{aligned} T(n, m) &= T1(n) + T2(m) \\ &= O(\max(f(n), g(m))) \end{aligned}$$

- 乘法规则 针对多层嵌套循环

$$\begin{aligned} T(n, m) &= T1(n) * T2(m) \\ &= O(f(n) * g(m)) \end{aligned}$$

- 乘法规则中的常数无关项 $O(c) \rightarrow O(1)$

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

- $O(1)$ 表示常数计算时间

例1：两个并列循环

```
void example (float x[ ][ ], int m, int n, int k) {  
    float sum [ ] = new float[m];;  
    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 (int i = 0; i < m; i++ ) //打印各行数据和  
        cout << “Line ” << i <<  
            “ : ” << sum [i] << endl;  
}
```

渐近时间复杂度为 $O(\max(m*n, m))$

例2：起泡排序

```
template <class Type>
void dataList<Type>::bubbleSort ( ) {
//对表 Element[0] 到 Element[ArraySize-1]
//逐趟进行比较, 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(const 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Element[j-1]与Element[j]
            exchange = 1;          //做 “发生了交换” 标志
        }
    }
}

```

起泡排序的算法时间复杂度分析（最坏情况）：

$$T(n) = \sum_{p=0}^{n-1} (n-p-1) = \frac{n(n-1)}{2}$$

渐近时间复杂度 $O(n^2)$

渐近空间复杂度

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

$S(n)$ 为算法的空间复杂度函数

反映的是当问题规模 n 充分大时，需要的存储空间将如何随之变化。