数 据 结 构 // Data Structures

《**Algorithms and Data Structures**》、《**Algorithms: Design and Analysis**》
-- An introduction to fundamental data types, algorithms, and data structures.

- **Algorithm**: method for solving a problem.
- **Data structure**: method to store information.

topic	data structures and algorithms
data types	stack, queue, binary tree, union-find, priority queue
strings	KMP, tries
sorting	quicksort, mergesort, heapsort, radix sorts
searching	BST, red-black BST, B-tree, hash table
graphs	BFS, DFS, Prim, Kruskal, Dijkstra

- The algorithm's impact is **broad and far-reaching**.
- Two **applications of algorithms**:
  1. Internet routing (Suppose you need send an email to a friend)
  2. Sequence alignment (computational biology and elsewhere) (you might consider sequences of DNA base pairs)

数 据 结 构 // Data Structures

《**Algorithms and Data Structures**》、《**Algorithms: Design and Analysis**》
-- An introduction to fundamental data types, algorithms, and data structures.

[*教材*]

严蔚敏, 吴伟民: *数据结构(C语言版)*, 清华大学出版社, 1997(2007、2012)
严蔚敏, 吴伟民: *数据结构习题集*, 清华大学出版社, 1997

[*参考书*]

1. N.Wirth, *Algorithm + Data Structures = Programs*, by Prentice-Hall, Englewood Cliffs,N.J. 1976
2. D. E. Knuth, *The Art of Computer Programming*, by Addison-Wesley Company, Inc. 1973 Vol.1/*Fundamental Algorithms*(2nd Ed.)
   Vol.3/*Sorting and Searching*
3. Robert Sedgewick and Kevin Wayne, *Algorithms*(4th Ed.), by Addison-Wesley 2011
4. Thomas H. Cormen etc., *Introduction to Algorithms*(3rd Ed.), MIT Press, 2009
5. 程杰, *大话数据结构*, 清华大学出版社, 2011

liuyh@nju.edu.cn

Introduce to Niklaus Wirth

- **1934**年生于瑞士温图特尔, **1958**年在苏黎世联邦理工学院获得学士学位, **1963**年在美国加州大学伯克利分校**(UC Berkeley)**获得博士学位. **1963**年到**1967**年, 美国斯坦福大学助理教授, **1968**年至今瑞士苏黎世联邦理工学院教授.
- 由于发明多种影响深远的程序设计语言, 并提出“结构化程序设计”这一革命性概念获得**1984**年的“图灵奖”**(A. M. Turing Award, ACM** 于**1966**年设立**)**.
- **Pascal**之父, 追求简单. 几十年来, 人们力求去发明非冯·诺依曼**(Von Neumann)**结构的计算机, 但鲜有成功, **Wirth**认为就是因为该结构实在是太简单了. 简单就是好.

Introduce to Donald Ervin Knuth

- 1938年生于美国威斯康星州, 1963年(25岁)获得Caltech(加州理工学院)数学博士学位, 36岁获得图灵奖(1974年), 是该奖历史上 最年轻的获奖者.
- 1968年, 因Addison-Wesley出版社约稿, 他的著作《计算机程序设计艺术》(TAOCP)第一卷正式出版(据说比尔·盖茨曾花了几个月的时间读完这一卷).
  Knuth说:要是看不懂, 就别做程序员. 同年, 从Caltech副教授去Stanford当教授.
- 1973年, 这本书出到了第三卷. 按计划该套共7本, 但100多万套的发行量令图灵奖颁奖委员震惊. Knuth获此殊荣之后, 宣布从此歇笔: 因为排版工具太差, 破坏了这套书的美. 这让外界十分震惊.
- 其后的10年, 他用Pascal写出了科学/数学公式排版系统TeX, TeX 作为一个软件产品, 也令人叹为观止. 它的版本号不是自然数列, 也不是年份, 而是 从3开始, 不断逼近圆周率(目前最新版本是3.1415926); 他还设立了奖金: 谁发现TeX的一个错误, 就付他2.56美元(256美分刚好是16进制的1美元), 第二个错误5.12美元, 第三个10.24美元…以此类推. 结果直到今天, 他也没有为此付出多少钱. 据说, 几位获奖者 将有他签名的支票视为珍宝, 并不兑现.
- 2008年, 在TAOCP的前三卷面市35年之后, 第四卷终于面世, 他已是白发古稀老人.

Ch1 绪 论

1.1 几个相关的基本概念

数据(data): 能被计算机识别、存储、加工处理的用来描述客观事物的物理符号.

数据元素(data element): 数据的基本单位, 也称结点(node)或记录(record / entry).

它是数据结构处理的基本对象, 在算法/程序中通常把它作为一个整体来考虑和操作.

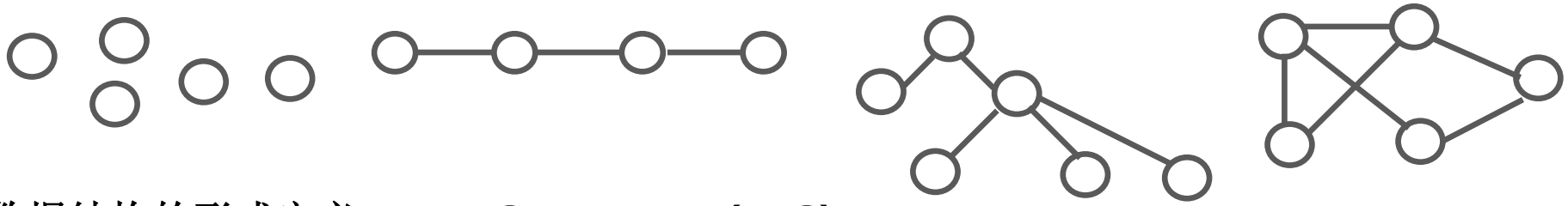
数据项(data item): 有独立含义的数据最小单位, 也称域(field). 一个数据元素可由若干个数据项构成.

数据对象(data object): 具有相同性质的数据元素的集合.

数据结构: 数据对象中数据元素之间存在的一种或多种特定"关系".(个体定义)

根据数据元素之间关系的不同特性, 有四种基本数据结构:

- 集合: 数据元素间除“同属于一个集合”外, 无其它关系
- 线性结构: 一对一, 如线性表
- 树形结构: 一个对多个, 如树
- 图状结构: 多个对多个, 如图



数据结构的形式定义 Data _Structure = (D, S)

其中: D是数据元素的有限集,

S是D上关系的集合.

1.1 几个相关的基本概念

用数学符号表示的客观事物自身所具有的结构特点的数学模型

数据的逻辑结构: 独立于计算机, 只反映数据元素及其之间的逻辑关系.←抽象

数据的存储(物理)结构: 数据的逻辑结构在计算机存储器中的表示/映象(其描述方法依赖于某种程序设计语言). 其中, 两种常见存储结构分为:

顺序存储结构: 除存储数据元素自身信息外, 借助元素在存储器中的相对位置来表示数据元素间的逻辑关系. 一般用程序设计语言中的一维数组表示 →

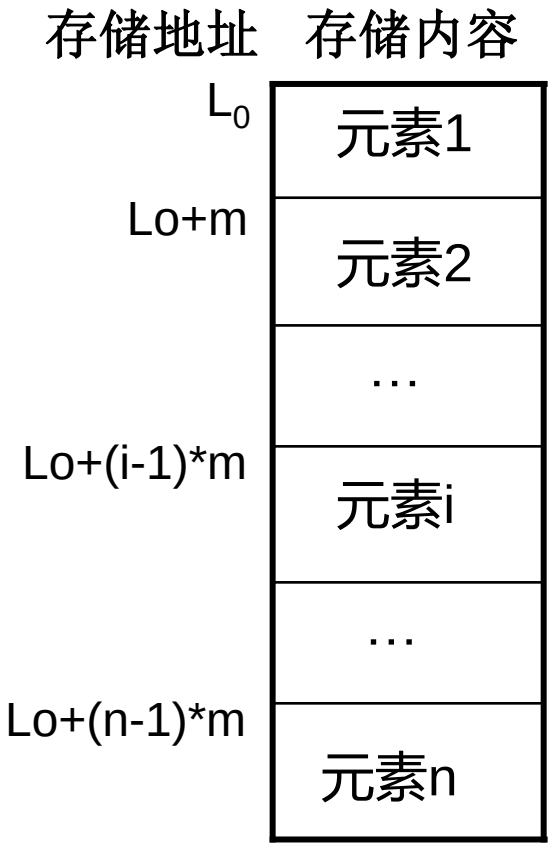
链式存储结构: 除存储数据元素自身信息外,借助指示元素存储地址的指针表示数据元素间的逻辑关系. 一般用程序设计语言中的指针表示 →

此外, 其他的存储结构还包括*索引存储结构*、*散列存储结构*等表示方法.

1.1 几个相关的基本概念

顺序存储结构 数据元素之间的逻辑关系通过地址得到直接反映—连续存储

假设n个数据元素之间为线性关系, 如下图所示:



L_0 – 基地址

m – 每个元素占用的单元数

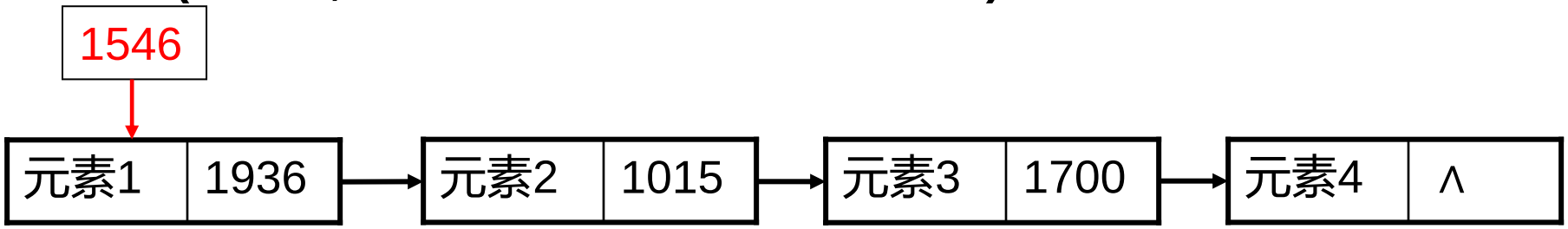
Loc(元素i) = L_0+(i-1)*m

Back

1.1 几个相关的基本概念

链式存储结构 数据元素之间的逻辑关系通过指针得到间接反映—离散存储

head(头指针, 即第1个数据元素的首字节地址)



存储地址	存储内容	指针(存储地址)
1015	元素3	1700
1546	元素1	1936
…	…	…
1700	元素4	∧
…	…	…
1936	元素2	1015

Back

1.1 几个相关的基本概念

数据类型(data type): 数据值的集合和在这个值集上的一组操作/运算.

抽象数据类型(abstract data types, ADT): 基于逻辑关系的数学模型以及定义在这个模型上的一组操作/运算. 其定义形式为(例子在以后章节中体现):

ADT 抽象数据类型名 { //A User-Defined Data Type
数据元素：<数据对象的定义>
数据关系：<数据元素关系的定义>
操作/运算：<基本操作运算的定义> }
其中：基本操作运算名<参数表>
初始条件：<初始条件描述>
操作结果：<操作结果描述>

} Yes to what to do / No to how to do

①ADT相对于数据类型来说其范围有了扩充, 即不局限于计算机内置的数据类型, 如int, char.
②ADT的定义仅决定于它的一组逻辑特征, 与其在计算机内的表示和实现无关, 即只要其数学特征不变, 不管其存储结构如何变化都不影响其外部使用. More about ADTs … (The OO's view)

List ADT(一个例子: 线性表的抽象数据类型 / A **list** contains elements of same type)
get() – Return an element from the list at any given position.
insert() – Insert an element at any position of the list.
remove() – Remove the first occurrence of any element from a non-empty list. OR **removeAt()**
size() – Return the number of elements in the list.
isEmpty() – Return true if the list is empty, otherwise return false. **isFull()**

How to write client programs for useful ADTs?
eg. above operations (APIs) can be performed on a list.

1.1 几个相关的基本概念

Object-oriented programming:
- **Create your own data types (sets of values and ops on them)**
- **Use them in your programs (manipulate *objects*)**

An object holds a data type value.
Variable names refer to objects.

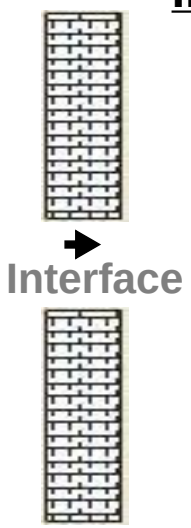
ADTs (Abstract Data Types)

- **Client uses class as abstraction**
  - **Invokes public operations only, Internal implementation not relevant!**
- **Client can't and shouldn't muck with internals**
  - **Class data should private**
- **Imagine a "wall" between client and implementer**
  - **Wall prevents either from getting involved in other's business**
  - **Interface is the "chink" in the wall**
- **Consider Lexicon**

Client

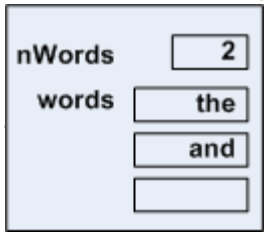
Declares, init object;
Doesn't know internal structure of object;
Manipulates object through public member functions.

```
Lexicon lex;
lex.addWord("cat");
lex.containsWord("pig");
lex.nWords++;
lex.words[0] = "dog";
```



Implementer

Knows internal structure;
Has access to private data;
Manipulates object in implementing member functions.




```
words[nWords++] = str;
```

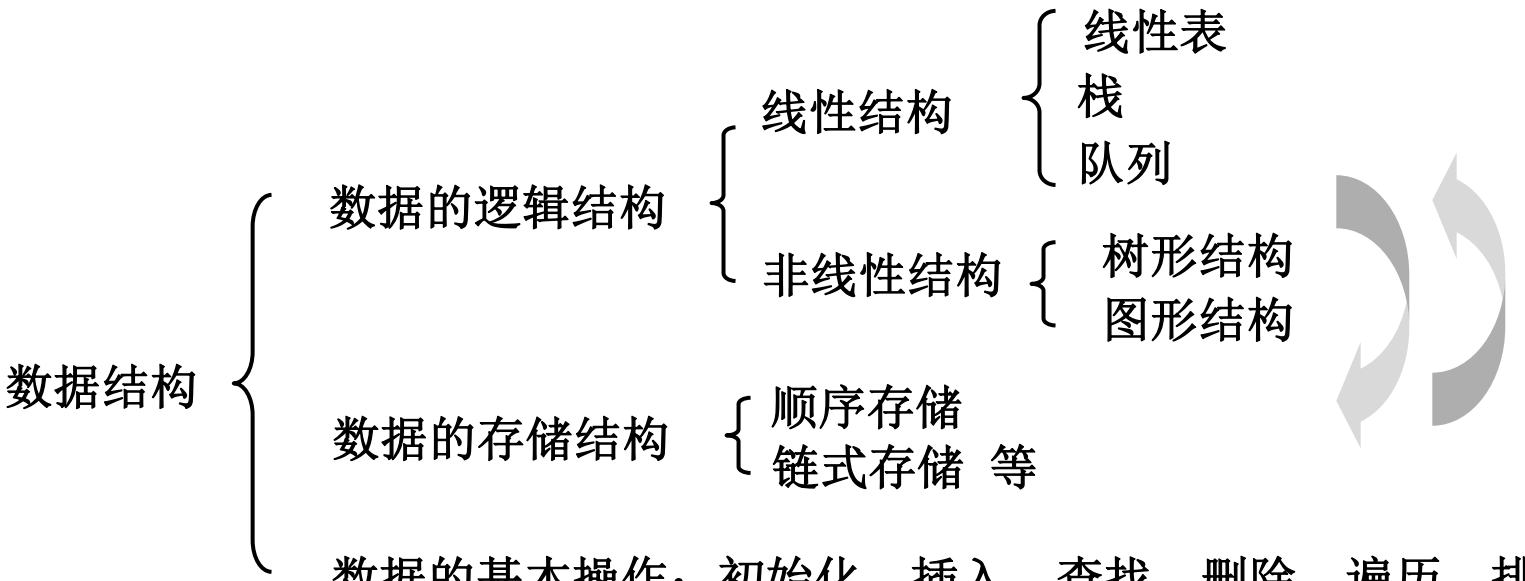
1.1 几个相关的基本概念

Why ADTs?

- **Abstraction**
  Client insulated from details, works at higher-level
- **Encapsulation**
  Internals private to ADT, not accessible by client
- **Independence**
  Separate tasks for each side (once agreed on interface)
- **Flexibility**
  ADT implementation can be changed without affecting client

1.1 几个相关的基本概念

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、其间的关系及对之进行基本操作的学科. (学科定义) ⇨ 无法用数学方程加以描述



1.2 算法与算法描述工具

算法(algorithm): An algorithm is a method for solving a problem that is suitable for implementation as a computer program.

vs. 程序(program): 计算机指令的某种组合, 控制计算机的工作流程, 完成一定的逻辑功能, 以实现某个任务. 程序是算法, 算法不一定是程序.

算法特征:
- 有穷性: 一个算法总是在执行有限步后结束, 且每步均在有限时间内完成
- 确定性: 算法中的每条指令必须有确切的含义, 不能产生二义性
- 可行性: 算法中的运算均可通过已经实现的基本运算的执行来完成
- 输入: 一个算法有零个或一个或多个输入
- 输出: 一个算法应有一个或多个输出, 且与输入有特定关系

算法描述工具:

(1)通过自然语言描述

例如: 求两个正整数M与N的最大公因子, 即能够同时整除M和N的最大正整数.

(1) M除以N, 将余数送中间变量R(0≤R<N);
(2) 测试余数R是否等于零?
  a) 若R等于零, 求得的最大公因子为当前N的值, 算法结束.
  b) 若R不等于零, 将N送入M, 将R送入N, 重复算法的(1)和(2).

(2)采用程序流程图的形式来描述

(3)采用某种具体的语言来描述

(4)通过pseudo-code的方法，如伪c/c++、Java等

Review of the type definition with *typedef* and pointers

1.2 算法与算法描述工具

算法设计要求：

- 正确性(correctness) ➔to satisfy specifications
- 健壮性/鲁棒性(robustness) vs. reliability
- 效率(efficiency) ➔ time & space ➔ 算法优化
- 可读性(readability) /modularity/maintainability ➔ 符合程序设计方法学的要求

```
//基本思想, 包括precondition & postcondition
[返回值类型] 函数名(形式参数说明)
{
  …;
  语句序列;
   // ①commenting the key code;
   // ②naming conventions;
   // ③using whitespace and indentation;
   // ④making the functions short and manageable;

   i += 1; // increment i by one; ×

}
```

1.2 算法与算法描述工具 -- 算法优化的几个例子

算法优化, 提高性能 -- 用单循环语句输出九九乘法表

```
for (int i = 1, j = 1; j <= 9; i++) {
    printf("%d*%d=%2d ", i, j, i*j);
    if (i == j) { i=0; j++; printf("\n"); } }  //O(n^2) -> O(n)
```

算法优化, 提高性能 -- 百钱买百鸡问题

鸡翁一, 值钱五, 鸡母一, 值钱三, 鸡雏三, 值钱一. 百钱买百鸡, 问翁、母、雏各几何?

这是一个经典的不定方程问题, 有些类似于动态规划中的"多阶段决策".

```
for (x = 1; x < 20; x++)
    for (y = 1; y < 33; y++) {
        z = 100 - x – y;
        if ((z % 3 == 0) && (x*5 + y*3 + z/3 == 100)) printf(x, y, z); } //O(n^2)
```

x+y+z=100 ① 5x+3y+z/3=100 ②
②x3-①得: 7x+4y=100 即 y=25-(7/4)x ③
又因0<y<100的自然数, 则可令 x=4k ④
将④代入③得: y=25-7k ⑤
将④⑤代入①得: z=75+3k ⑥
要满足0<x, y, z<100的话，k的取值范围只能是1, 2, 3.

```
for (k = 1; k <= 3; k++) { x = 4*k; y = 25 – 7*k; z = 75 + 3*k; printf(x, y, z); } //O(n)
```

算法优化, 提高性能 -- 不借用辅助变量交换两个数值变量的值

```
Swap(i, j) { tmp = i; i = j; j = tmp; }
Or no tmp variable
{ i = i+j; j = i-j; i = i-j; }  //不需占用额外存储空间
```

1.3 算法分析技术初步 -- asymptotic analysis

在一个算法中该语句重复执行的次数 (frequency count)

时间复杂性(Time Complexity)

算法中基本操作重复执行的次数依据算法中最大语句频度来估算,它是问题规模n的某个函数f(n), 算法的时间量度记作 T(n) = O(f(n)).

input size

O是数学符号, 其数学含义: 若f(n)和g(n)是定义在正整数集合上的两个函数, 则 f(n) = O(g(n)) [read f(n) is big O(g(n))] 表示存在正的常数c和n_0, 使得当$n \geq n_0$时均满足 $0 \leq f(n) \leq cg(n)$. O体现的是渐近性(asymptotic / growth curve)而非精确性.

如: 设有函数 f(n) = 4n+200, 则f(n) = O(n)[$\because f(n) \leq cn$, 这里c=5, 且$n \geq 200$]

或: 当$n \to \infty$时, 有$f(n)/g(n)$ = 常数$c \neq 0$(即f(n)的函数值增长速度与g(n)的增长速度同阶), 则称f(n)和g(n)同阶或同一数量级, 记作: f(n) = O(g(n)).

如: $\lim_{n\to\infty}(f(n)/g(n)) = \lim_{n\to\infty}((2n^3+3n^2+4n+1)/n^3) = 2$, 则$f(n) = O(n^3)$

一般说来, 一个函数的增长速度与该函数的最高次阶同阶.

空间复杂性(Space Complexity)

当问题的规模以某种单位由1增至n时, 解决该问题的算法实现所占用的空间也以某种单位由1增至f(n), 则称该算法的空间代价是f(n), 记作 S(n) = O(f(n)).

- *Auxiliary Space* is the extra space or temporary space used by an algorithm.

Big-O notations most commonly used in analyzing algorithms:

$O(1)$, $O(\log_2 n)$, $O(n)$, $O(n\log_2 n)$, $O(n^2)$, $O(n^3)$, $O(n^k)$, $O(2^n)$

1.3 算法分析技术初步 -- asymptotic analysis

几个例子：① //set of non-loop and non-recursive statements. eg. Swap() function

```
{ temp = i; j = i; j = temp; }
```

又如: //Here c is a constant

```
for ( i = 1; i <= c; i++ ) {
    //some O(1) expressions
}
```

不随问题规模n的变化而变化 ➔ O(1)

② 循环变量被加/被减一个常数

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

$$\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j} 1 = \sum_{i=1}^{n}\sum_{j=1}^{i} j = \sum_{i=1}^{n} i(i+1)/2 =$$

$$[n(n+1)(2n+1)/6+n(n+1)/2]/2$$

➔ $T(n) = O(n^3/6+低次项) = O(n^3)$

又如:

```
{ i = 0; j = 1;
  while ( i < n ) { //some O(1) task
    i = i+j;
    j++; } }
```

循环控制变量i每次迭代分别增加1、2、3、4 …直到i>=n. 经x次循环后$i=x(x+1)/2$, $\therefore x(x+1)/2 < n$ ➔ $O(n^{1/2})$

1.3 算法分析技术初步 -- asymptotic analysis

几个例子：③ 多个连续循环(consecutive loops)语句

```
{ for ( i = 1; i <= n; i++ )
    for ( j = i; j <= n; j++ )
        if ( a[i].data < a[j].data )  Swap(a[i], a[j]); //降序排序
  for ( i = n; i >= 1; i -= 1 ) printf (a[i]); }  //升序输出
```

时间、空间复杂性: $O(n^2)$、$O(1)$

➔ $\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = \sum_{i=1}^{n}(n-i+1) = n(n+1)-n(n+1)/2 = n(n+1)/2$

④ 循环变量被乘以/除以一个常数

```
for ( i = 1; i <= n; i *= 2 ) {
   //some O(1) expressions
}
```

```
for ( i = n; i > 0; i /= 2 ) {
   //some O(1) expressions
}
```

$\sum_{i=0}^{k} 2^k = \sum(1+2+4+\ldots+2^k)$ ➔ $2^k = n$, $k = \log_2 n$ ➔ $O(\log_2 n)$

1.3 算法分析技术初步 -- asymptotic analysis

几个例子：⑤ 在 **a[0..n-1]** 中查找给定的**K**值:

```
{ i = n-1;
  while ((i >= 0) && (a[i] != K))  i--;
  return i; }
```

1. **The best-case: a**中最后一个元素的值等于**K, 即a[n-1] = K**
   则算法中语句最大执行频度**f(n)**是常数**1**
2. **The worst-case: a**中没有与**K**值相等的元素
   则语句 **i --** 的执行频度**f(n) = n**

Exercises (After class):

1. 简述数据结构的主要研究内容, 并说明数据的逻辑结构和数据的存储结构之间的关系.

2. 设有三个函数f, g, h分别为: $f(n) = 100n^3+n^2+100$, $g(n) = 25n^3+5000n^2$,

$h(n) = n^{1.5}+5000n\log_2 n$. 判断下列关系式是否成立:

①$f(n)=O(g(n))$ ②$g(n)=O(f(n))$ ③$h(n)=O(n^{1.5})$ ④$h(n)=O(n\log_2 n)$

3. 设n为正整数, 求下列程序片断的时间复杂性:

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

②
```
i = 1; j = 0;
while ( i + j <= n ) {
    if ( i > j ) j++;
    else i++;
}
```

③
```
x = n;  // n > 1
while ( x >= ( y + 1 ) * ( y + 1 ) )
    y++;
```

④
```
x = 91; y = 100;
while ( y > 0 )
    if ( x > 100 ) { x = x – 10; y--; }
    else x++;
```

Type definition with *typedef*

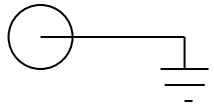
typedef data_type new_name;

// where **new_name** is the new name given to the type **data_type**.

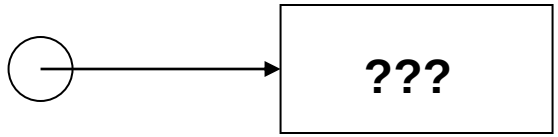
For examples:

```
typedef  int INTEGER;
INTEGER i, j;  // here the variables i and j are defining as the type int

typedef  int INTEGER;
typedef  struct node {
  INTEGER count;
  struct node *next; } Node;
Node *p;   // here the variable p is defining as a pointer points to
           // a variable which be defined as the type struct node

#define Row 20
#define Col   20
typedef  boolean Matrix[Row][Col];
// 二维数组Matrix中的每个分量均是布尔类型
Matrix  A;  // A是一个布尔型的二维数组变量
```

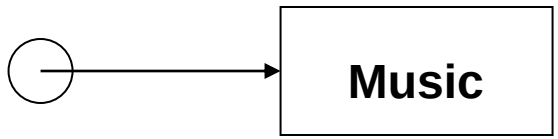
Pointers



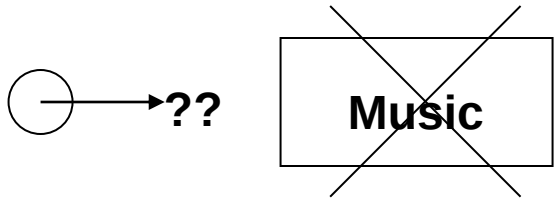
p = Null;



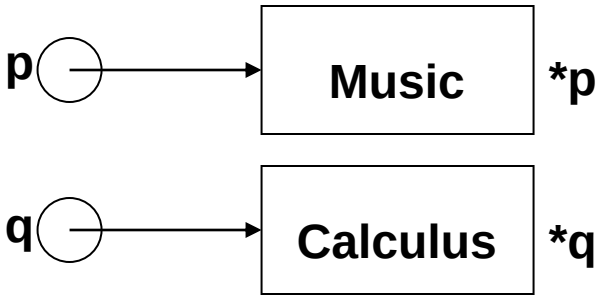
p = (Node *)malloc(sizeof(Node));



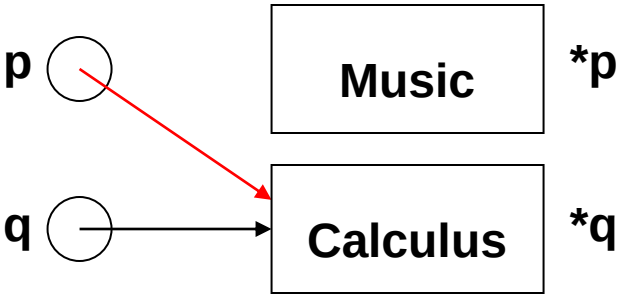
*p = ‘Music’;



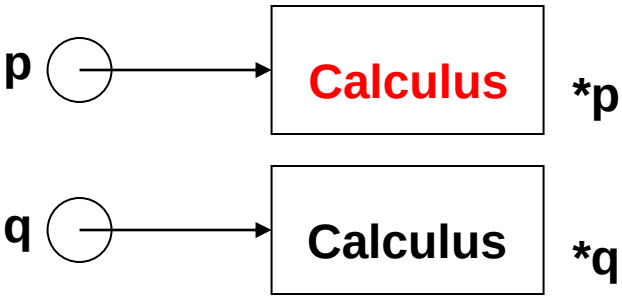
Free(p);



p = q



*p = *q



C++包含一个新增的与指针有关的特征--- reference (引用)

引用就是担当对象另一个名字的隐含指针，用取地址操作符&表示。也就是说，引用的作用如同一个变量名，它给被引用变量一个可替换的名字。引用实质上是通过指针实现的。

例如：
```
int a = 1;
int &r = a;  //定义引用变量r
r++;   //等价于a++
r = 4; //即a的值为4; 引用r是对a的操作, 而不是a的拷贝
```

引用的一个重要用途是允许用户使用引用调用传递函数参数(注: C++的缺省值是单向的值传递)。从概念上C++的引用参数和C的指针参数都实现了地址传递，但在函数里对这些参数操作的描述方式不同。

```
/*C example */
void increment(int *value)
main() {
   int i = 10;
   increment(&i); }
void increment(int *value) {
   (*value)++; }
```

```
// C++ example
void increment(int &value)
main() {
   int i = 10;
   increment(i); }
void increment(int &value) {
   value++; }
```

Theorem 1 $1 + 2 + \cdots + n = n(n+1)/2$

$1^2 + 2^2 + \cdots + n^2 = n(n+1)(2n+1)/6$

Theorem 2

$$\sum_{k=0}^{m-1} 2^k = 1 + 2 + 4 + \cdots + 2^{m-1} = 2^m - 1$$

$$\sum_{k=1}^{m} k \times 2^{k-1} = 1\times1 + 2\times2 + 3\times4 + \cdots + m\times2^{m-1} = (m-1)\times2^m + 1$$

Theorem 3 $\ln(x+1) = x - x^2/2 + x^3/3 - x^4/4 + \cdots$

$e^x = 1 + x + x^2/2! + x^3/3! + \cdots$

Theorem 4 $C(n, k) = \dfrac{n!}{k!(n-k)!}$

Notation

The floor of a real number x (denoted by $\lfloor x \rfloor$) is to be the largest integer less than or equal ($\leq$) to x, and the ceiling of x (denoted by $\lceil x \rceil$) is to be the smallest integer greater than or equal ($\geq$) to x. (Page124)

For example, supposed x=5/2=2.5, then $\lfloor x \rfloor = 2$, $\lceil x \rceil = 3$;

if x=2, $\lfloor x \rfloor = \lceil x \rceil = 2$ 结论: 若x是整数,则有 $\lfloor x \rfloor = \lceil x \rceil$, 否则有 $\lceil x \rceil = \lfloor x \rfloor + 1$