

数据结构

夏天

xiat@ruc.edu.cn

中国人民大学信息资源管理学院

概论

- 计算机程序设计和数据处理的理论与技术基础
- 核心内容：
 - * 线性表、栈和队列、字符串、树与森林、图、排序、查找
- 目标
 - * 掌握数据结构的特点、存储方法和基本运算
 - * 初步掌握算法的时间和空间分析技术
 - * 能够针对不同数据对象的特性，选择适当的数据结构和存储结构以及相应的算法

课程信息

- 教室安排 : 信息楼 403; 上课时间 : 周四 16:00 – 17:30
- 教学方式
 - * 课堂教学与演示 (听讲、看书、练习)
 - * 上机练习
- 平时成绩 :
 - * 课程作业 : 50%
 - * 研讨交流与发言情况 : 30%
 - * 课堂提问 : 10%
 - * 考勤 : 10%
- 最终成绩 :
 - * 随堂闭卷考试 (50%) + 平时成绩 (50%)
- 要求独立完成作业, 切勿抄袭!
- 助教 : 张贤哲, 可在群中联系

学习建议

- 教材
 - * 严蔚敏，吴伟民. 《数据结构 (C 语言版)》，清华大学出版社. (国内经典教材，写作严谨，广泛使用。)
 - * 算法导论
 - * 大话数据结构
- 在线课程
- 编程语言
 - * 选择一种作为实践语言：Java、Python (根据同学们反馈选择)
 - * 能够看懂不同编程语言的代码
 - * 伪代码 (Pseudocode)
 - 一种非正式，类似于英语结构，用于描述模块结构图的语言。

1997 年的人机大战



1997 年 5 月 11 日，国际象棋世界冠军卡斯帕罗夫与 IBM 公司的超级电脑深蓝（Deep Blue）对弈。



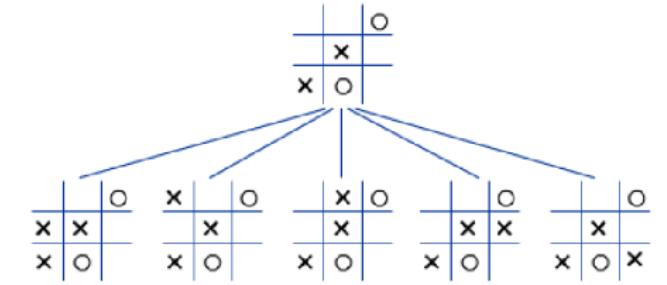
棋迷在纽约通过电视观战。当日，卡斯帕罗夫在纽约再次负于深蓝，从而在当年的“人机大战”中以一胜二负三和的战绩败北。

- ① “深蓝”重量达 1.4 吨，有 32 个 CPU，每个 CPU 有 8 块专门为进行国际象棋对弈设计的处理器，平均运算速度为每秒 200 万步。总计 256 块处理器集成在 IBM 研制的 RS6000/SP 并行计算系统中，从而拥有每秒超过 2 亿步的惊人速度。
- ② IBM 研制小组向“深蓝”输入了 100 年来所有国际特级大师开局和残局的下法。美国特级大师本杰明将他对象棋的理解编成程序教给“深蓝”。虽不会思考，但它无穷无尽的计算能力在很大程度上弥补了这一缺陷。



① 模型：棋盘、棋子的表示

② 算法：对弈的规则和策略



③ 对弈的本质即在该空间里进行有效的搜索



2016 年 3 月 9 日，Google 旗下的 AlphaGo 电脑击败韩国九段棋手李世石。

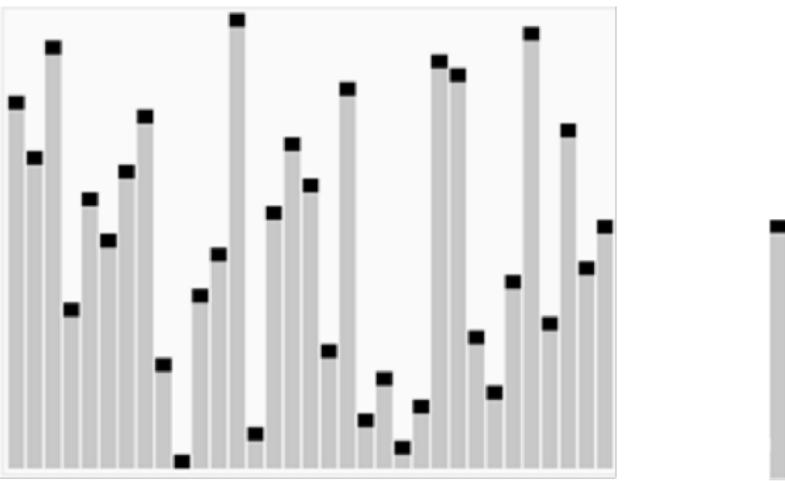


2017 年 5 月 27 日，世界排名第一的人类棋手柯洁负于 AlphaGo，人机大战 2.0 定格在了 0:3。

- 对于擅长于计算的计算机来说，围棋的难度在很大程度上来自于 19×19 路棋盘背后所蕴含的巨大的无法穷尽的变化（3361 种下法），这是基于“穷尽法”的“深蓝”无法在围棋上战胜人类的原因。
- AlphaGo 取得如此成绩，关键是深度学习和类神经网络技术。
- AlphaGo 将棋盘看作是一个 19×19 像素构成的图片，利用类似于卷积神经网络的技术预测下一步走法。

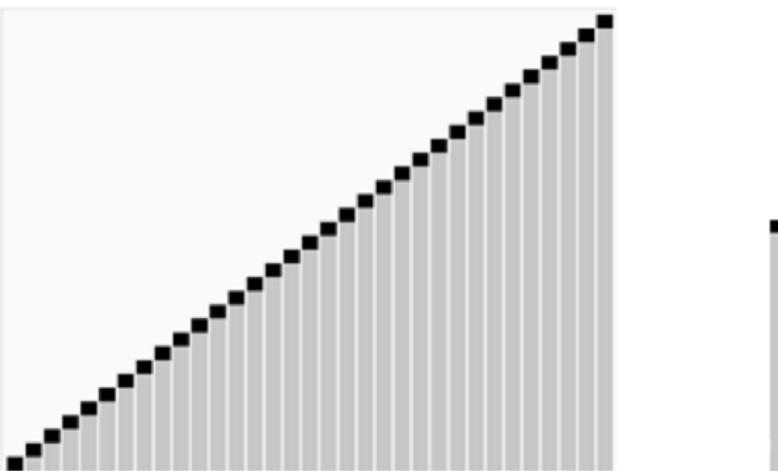
依赖于高效的搜索

- 如何快速的搜索到右方的图例？



依赖于高效的搜索

- 如何快速的搜索到右方的图例？



数据有序有利于查找。需要对数据进行高效的组织/排序。

Algorithms + Data Structures = Programs

- 算法
 - * 处理问题的策略/操作步骤
- 数据结构
 - * 静态数据表示的数学模型 (以及必须的操作)
- 程序
 - * 为计算机处理问题编制的一组指令集

数据结构：研究描述现实世界实体的数学模型及其上的操作在计算机中的表示和实现。

FAQ

- ① 数据结构是又一门编程课吗？
- ② 不会 C 语言怎么办？
- ③ 用别的语言学习数据结构可以吗？
- ④ 怎样学好数据结构？

基本概念和术语

- 基本概念和术语
- 抽象数据类型的表示与实现
- 算法和算法分析

基本概念

- 1、数据 (Data)

- * 对客观事物的符号表示，在计算机科学中特指所有能被输入到计算机中并能被计算机程序处理的符号的总称。数据是程序加工的“原料”。
 - * 例：一个文字处理程序的处理对象是字符串。

- 2、数据元素 (Data Element)

- * 数据的基本单位，在计算机程序中通常作为一个整体被考虑和处理。
 - * 例：一本书，一条学生记录。
 - * 数据元素可由若干数据项 (Data Item) 组成。

基本概念

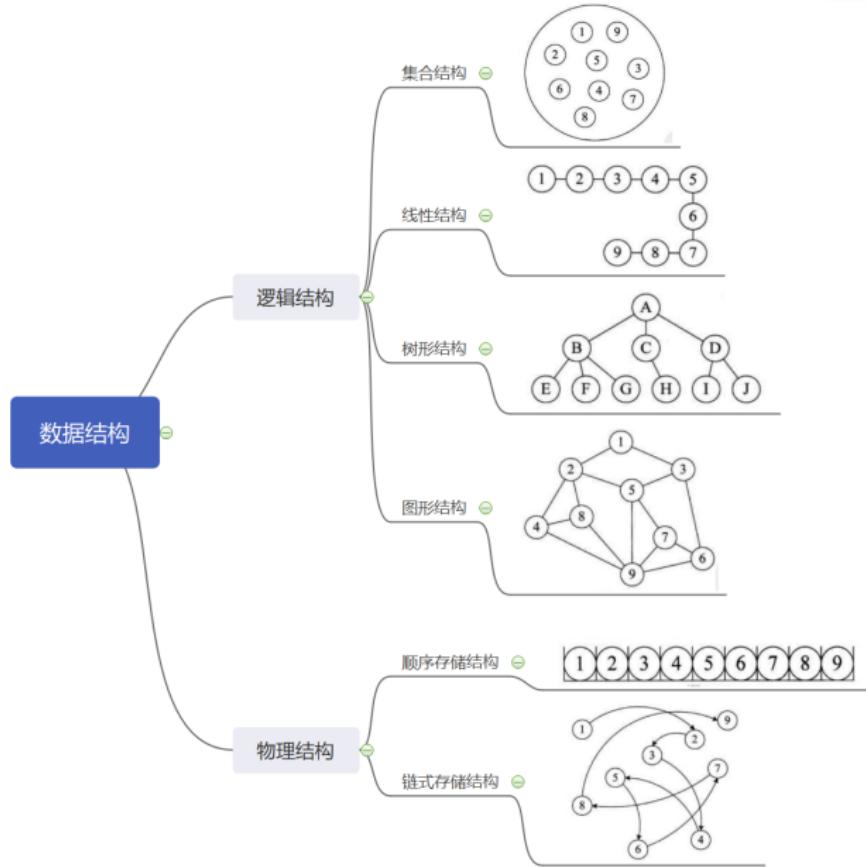
- 3、数据对象 (Data Object) : 性质相同的数据元素的集合 , 是数据的一个子集。
 - * 例 1: 整数数据对象, $N = \{0, 1, 2, \dots\}$
 - * 例 2: 字符数据对象, $C = \{'A', 'B', \dots\}$
- 4、数据类型 (Data Type) : 一个值的集合和定义在这个值集上的一组操作的总称。
 - * 原子类型
 - 不可分解的数据类型
 - 如 : 整型、字符型、指针型、空类型...
 - * 结构类型
 - 可分解为若干成分
 - 如 : 数组由分量组成 , 分量可以是整型 , 也可以是数组.

数据结构 (狭义上) 是相互之间存在一种或多种特定关系的数据元素的集合。

- 逻辑结构
 - * 集合、线性结构、树形结构、图状或网状结构
- 物理结构：数据结构在计算机中的表示/映像
 - * 数据元素的表示是结点 (node)，即在计算机内用若干位组合起来表示一个数据元素。
 - * 数据关系的表示有顺序映像和非顺序映像两种，由此得到顺序存储结构和链式存储结构。

物理结构

- 顺序映像 (顺序存储结构)
 - * 逻辑上相邻的数据元素存储在物理位置上相邻的存储单元中比如在高级语言中的“一维数组”。
- 非顺序映像 (链式存储结构)
 - * 数据元素可以存储在计算机内任意位置上，它们的逻辑关系用指针来链接。比如在高级语言中的“指针”。



抽象数据类型的表示和实现

- 抽象数据类型 (Abstract Data Type) 是指一个数学模型以及定义在该模型上的一组操作。
 - * 例如，矩阵 + (求转置、加、乘、求逆、求特征值) 构成一个矩阵的抽象数据类型
- “抽象”在于数据类型的数学抽象特性，与具体表示和实现无关。

抽象数据类型的示例

ADT Complex {

数据对象: $D = \{e_1, e_2 \mid e_1, e_2 \text{ in RealSet}\}$

数据关系: $R_1 = \{\langle e_1, e_2 \rangle \mid e_1 \text{ 是复数的实数部分, } e_2 \text{ 是复数的虚数部分}\}$

基本操作:

InitComplex(&Z, v1, v2)

操作结果: 构造复数 Z, 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.

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

算法和算法分析

- 基本概念和术语
- 抽象数据类型的表示与实现
- 算法和算法分析

Alkhwarizmi 与 Algorithm

- Alkhwarizmi (约 780~约 850), 数学家，代数与算术的整理者。
阿拉伯文 Alkhwarizmi 原意是来自 (al-) 花刺子模 (Khwarizmi)。
- Alkhwarizmi 在当时的学问中心巴格达，服务于宫廷。他写了一本有关代数的书，这本书转成欧文，书名逐渐简化为 algebra (代数)。
- 后来 Alkhwarizmi 又引进了印度数字发展算术，后经 Fibonacci (1170~1250 年) 引入欧洲。欧洲人把 Alkhwarizmi 这个字拉丁化，称用十进位印度阿拉伯数字来进行有规则可寻的计算的算术为 Algorithm。后来算术转用其它的字 (如 arithmetic)，而 algorithm 则成为计算机科学的行话。

算法 (Algorithm)

- 算法是对特定问题求解步骤的一种描述，是指令的有限序列，其中每一条指令表示一个或多个操作。算法可以看做是从输入到输出的一个映射。
 - * Input → Output
- 算法的描述
 - * 自然语言
 - * 程序流程图
 - * 伪码：它忽略高级程序设计语言中一些严格的语法规则与描述细节，因此比程序设计语言更容易描述和被人理解，而比自然语言更接近程序设计语言。它虽然不能直接执行但很容易被转换成高级语言。
 - * 编程语言

试写出 [9,2,5,1,6,7,10] 经过下列程序处理后的序列。

//冒泡法

```
void bubble (int a[], int n) {  
    for (i=n-1; i>=1; i--) {  
        for (j=0; j<i; j++)  
            if (a[j]>a[j+1]) {  
                swap(&a[j], &a[j+1]);  
            }  
    }  
}
```

- $i = n - 1 \dots$
- $i = n - 2 \dots$

算法设计的要求

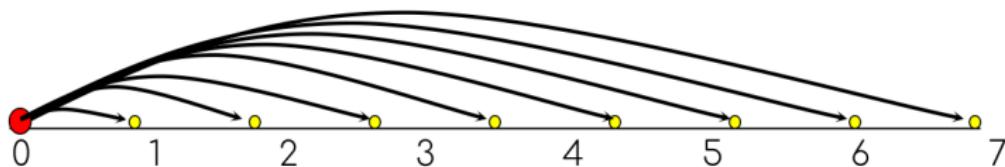
- 无语法错误
 - 对于选择的、典型、苛刻甚至带有刁难性的几组输入能够得出满足要求的结果
 - 对于一切合法输入都给出满意结果
 - 对非法输入有恰当的反映，而不是中断程序、或给出错误的输出
-
- 正确性
可读性
健壮性
高效率与低存储量
- 算法应易于（人的）理解、易于发现错误及进行调试。
 - 效率指的是算法执行时间；存储量指的是算法执行过程中所需的最大存储空间。两者都与问题的规模有关。

算法效率

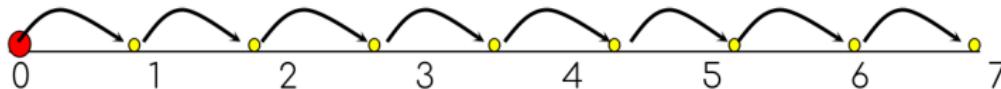
- 算法执行时间需通过依据该算法编制的程序在计算机上运行时所消耗的时间来度量。
- 相关影响因素包括：
 - * 机器硬件配置直接影响计算机执行指令的速度
 - * 编写程序的语言：越高级，效率越低
 - * 算法选用的策略
 - * 问题的规模：规模越大，耗时越久
 - * 编译程序产生的机器代码的质量

快递员配送包裹

- 假定快递员以恒定速度行驶，比较下面的方案
 - * 方案 1：配送 n 个包裹的总时间为 $n^2 + n$



- * 方案 2：配送 n 个包裹的总时间为 $2n$



各函数的增长率

在问题规模增长时，算法执行时间必定也会增长。我们关心的是这个执行次数以什么样的数量级（增长率）增长。

n	$\log_2 n$	$n \cdot \log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	2	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1,024	4,294,967,296

观察 $g(n)$ 和 $f(n)$ 的相对增长趋势

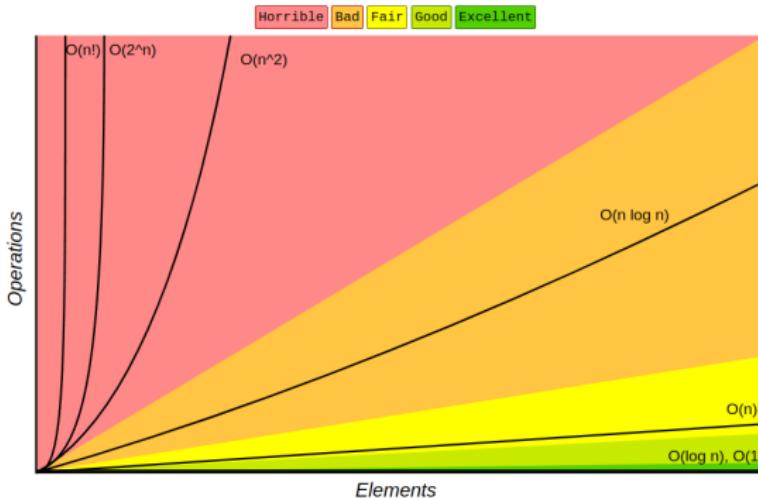
n	$g(n) = n^2$	$f(n) = n^2 + 4n + 20$
10	300	360
50	2,500	2,720
100	10,000	10,420
1,000	1,000,000	1,004,020
10,000	100,000,000	100,040,020

- 当 n 变得很大时， n^2 成为主导项，通过 $g(n)$ 的变化情况就可以预测 $f(n)$ 的变化。

$$O(c_1 n^k + c_2 n^{k-1} + \dots + c_k) = O(n^k)$$

- 例如， $T(n) = 3n^2 + 4n + 20$ ，记为 $O(n^2)$

Big-O Complexity Chart



From better to worse:

$O(1)$	constant time
$O(\log n)$	log time
$O(n)$	linear time
$O(n \log n)$	log linear time
$O(n^2)$	quadratic time
$O(n^3)$	cubic time
$O(2^n)$	exponential time

From: <https://www.bigocheatsheet.com>

大 O 表示法

- 算法的 (渐近) 时间复杂度记作 : $T(n) = O(f(n))$
 - * 表示随问题规模 n 的增大 , 算法执行时间的增长率和 $f(n)$ 的增长率相同。
- 通常是从算法中选取一种最基本的元操作 , 以该操作重复执行的次数 (频度) 来度量算法效率

例：矩阵相乘 (C 代码示例)

```
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] = c[i][j] + a[i][k] * b[k][j]; // 该语句可看作元操作  
    }
```

整个算法的执行时间与元操作重复执行的次数 n^3 成正比，记作

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

例：矩阵相乘 (Python 代码示例)

$n = 3$

$a = [[1,2,3],[4,5,6], [7,8,9]]$

$b = [[1,2,3],[4,5,6], [7,8,9]]$

```
c = [[0]*n for _ in range(n)]
```

```
for i in range(n):
```

```
    for j in range(n):
```

$c[i][j] = 0$

```
    for k in range(n):
```

$c[i][j] = c[i][j] + a[i][k] * b[k][j]$

整个算法的执行时间与元操作重复执行的次数 n^3 成正比，记作

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

请观察 [10,9,7,6] 和 [6,7,9,10] 在处理上的不同。

```
void bubble(int a[], int n) {  
    for (i=n-1; i>=1; i--) {  
        for (j=0; j<i; j++)  
            if (a[j]>a[j+1]) {  
                swap(&a[j],&a[j+1]);  
            }  
    }  
}
```

请观察 $[10,9,7,6]$ 和 $[6,7,9,10]$ 在处理上的不同。

```
void bubble(int a[], int n) {  
    for (i=n-1; i>=1; i--) {  
        for (j=0; j<i; j++)  
            if (a[j]>a[j+1]) {  
                swap(&a[j],&a[j+1]);  
            }  
    }  
}
```

- 若初始文件是正序的，一趟扫描即可完成排序。
- 若初始文件是反序的，需要进行 $n - 1$ 趟排序。
- 计算平均复杂度，或最坏情况下的时间复杂度.

- 最好情况的时间复杂度（初始文件是正序的）
[6,7,9,10] — 一趟扫描即可完成排序，共计 $n - 1$ 次比较（0 次交换）。
冒泡排序最好情况的时间复杂度为 $O(n)$
- 最坏情况的时间复杂度（初始文件是反序的）
[10,9,7,6] — 需要进行 $n-1$ 趟扫描，每趟进行 i 次比较 + 交换
($1 \leq i \leq n - 1$)，共计 $\frac{n(n - 1)}{2}$ 次比较 + 交换。
冒泡排序的最坏情况的时间复杂度为 $O(n^2)$

算法的存储空间需求 (内存)

- 算法需要为：输入数据、程序、辅助变量提供存储空间。
- 算法的空间复杂度： $S(n) = O(g(n))$
- 随着问题规模增大，算法运行所需存储量的增长率与 $g(n)$ 的增长率相同。
- 若所需存储量依赖于特定的输入，则通常按最坏情况考虑。
- 相对于时间复杂度而言，空间复杂度很多时候不需进行分析。

小结

- ① 数据结构研究非数值计算的程序设计问题中计算机的操作对象以及它们之间的关系和操作。
- ② 数据、数据元素、数据对象、数据结构的概念
- ③ 常见的基本数据结构
- ④ 以大 O 表示法分析算法时间复杂度

线性表 — Linear List

① 基本概念

② 顺序表

③ 链表

线性表

- 线性表举例：

- * 英文字母表 A, B, C, ⋯ , Z
- * 某单位近 5 年的计算机数量 (40, 60, 100, 150, 180)
- * 某产品淘宝的销售记录.

- 特点

- * 数据元素是多样的，但具有相同特性
- * 相邻元素之间有序偶关系 < 前驱, 后继 >

线性表

线性表是 n 个数据元素的有限序列

$$(a_1, a_2, \dots, a_i, \dots, a_n)$$

- 存在唯一的第一个数据元素，存在唯一的最后一个数据元素。
- 除第一个之外，每个数据元素只有一个前驱；除最后一个之外，每个数据元素只有一个后继。
- 线性表的长度：线性表中元素的个数，常记作 length , len , n . 当空表时， $\text{len}=0$.

线性表的两种存储方式

- ① 方案 1：顺序存储 — 称顺序表
- ② 方案 2：链式存储 — 称链表

方案 1: 顺序表 (Sequence List)

$(a_1, a_2, \dots, a_i, \dots, a_n)$

- 建一个数组，用一组地址连续的存储单元依次存储数据元素。
 - * 逻辑上相邻的数据元素，其物理存储位置也相邻

起始地址/基地址

b	a ₁	1
b + 1	a ₂	2
...	...	
b + (i - 1) · 1	a _i	i
...	...	
b + (n - 1) · 1	a _n	n
b + n · 1		空闲
...		空闲
b + (MaxLen - 1) · 1		空闲

顺序表 (Sequence List)

- 为便于维护处理，记录 3 个变量

- * 存放表元素的数组 list；

- * 表的长度 length；

- * 存储容量 maxSize；

- 常用的基本操作

- * 判断是否空: isEmpty()

- * 求长度: length()

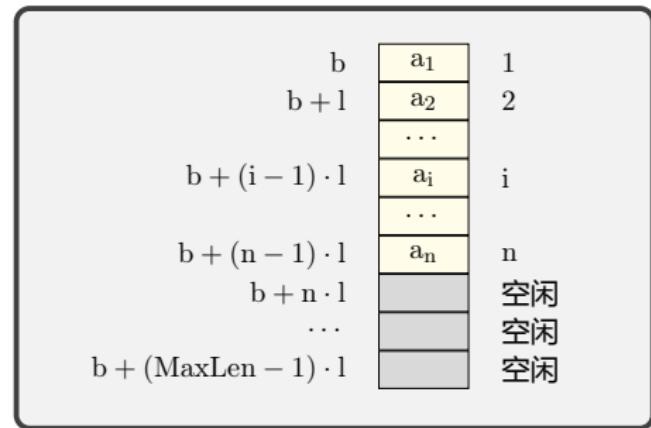
- * 取元素: get(i)

- * 插入操作: insert(i,x)

- * 删除操作: remove(i)

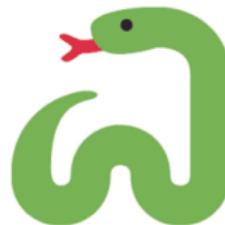
- * 查找: indexOf(x)

- * 输出: display()

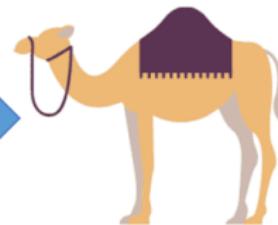


补充：常用命名规则之驼峰与蛇形命名法

snake_case



camelCase



Camel case

```
{  
    "firstName": "John",  
    "lastName": "Smith",  
    "email": "john.smith@example.com",  
    "createdAt": "2021-02-20T07:20:01",  
    "updatedAt": "2021-02-20T07:20:01",  
    "deletedAt": null  
}
```

Snake case

```
{  
    "first_name": "John",  
    "last_name": "Smith",  
    "email": "john.smith@example.com",  
    "created_at": "2021-02-20T07:20:01",  
    "updated_at": "2021-02-20T07:20:01",  
    "deleted_at": null  
}
```

```
public class SequenceList {  
    int maxSize; //最大长度  
    int length; //当前长度  
    ELEM[] list; //对象数组  
  
    public bool isEmpty()  
    public int length()  
    public ELEM get(i) throws Exception  
    public void insert(i,e)  
    public void remove(i) throws Exception  
    public int indexOf(e)  
    public void display()  
    public void clear()  
}
```

```
class SequenceList:
```

```
    def __init__(self, max_size, elements):
```

```
        self.max_size = max_size
```

```
        self.length = len(elements)
```

```
        self.elements = [0]*max_size
```

```
    for idx, value in enumerate(elements):
```

```
        self.elements[idx] = value
```

```
    def __len__(self):
```

```
        return self.length
```

```
    def __getitem__(self, i):
```

```
        return self.elements[i]
```

```
    def __setitem__(self, i, value):
```

初始化顺序表

Java/C Example

```
//初始化空表  
void initList(int size) {  
    list = new Elem[size];  
    maxSize = size; //初始存储容量  
    length = 0; //空表长度为 0  
}
```

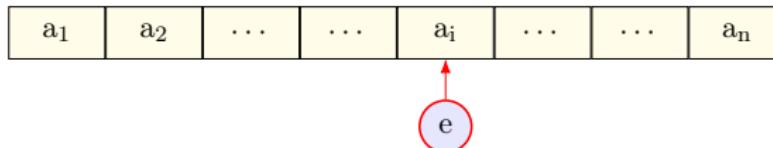
Python Example:

```
self.elements = [0]*max_size
```

在 index 位置上插入元素 e

```
boolean insert(int index, Elem e) {  
    if (length == maxSize) //当前线性表已满  
        {print("顺序表已满!"); return false;}  
    if (index < 0 || index > length) //插入位置编号不合法  
        {print("参数错误!"); return false;}  
  
    for (int j = length - 1; j >= index; j--) //向后移动元素  
        list[j + 1] = list[j];  
  
    list[index] = e; //插入元素  
    length++;  
    return true;  
}
```

插入元素的时间复杂度



- 在长为 n 的线性表中插入一个元素，所需移动元素次数的平均次数为?
 - * 有多少种可能的插入位置 ? $n + 1$
 - * 假设这些位置以同等概率出现，每个概率 $\frac{1}{n + 1}$ 。每个情形下分别移动 $n, n - 1, \dots, 0$ 次。求加权和为 $n/2$ 。
 - * 平均时间复杂度 : $T(n) = O(n)$

删除 index 位置上的元素



```
boolean remove(int index) {  
    if(index<0||index>=length)  
        return false;  
    if(getLength()==0)  
        return false;  
    for (int j=index;j<length-1;j++) //前移  
        Elem[j]=elem[j+1];  
    length--;  
}
```

删除元素的时间复杂度



- 在长度为 n 的线性表中删除一个元素：

- 共有 n 个可能的位置，每个有 $1/n$ 的概率。每个情形下分别移动 $n-1, \dots, 0$ 次。求加权和为 $(n-1)/2$ 。
- $T(n) = O(n)$

删除元素的时间复杂度



- 在长度为 n 的线性表中删除一个元素：
 - 共有 n 个可能的位置，每个有 $1/n$ 的概率。每个情形下分别移动 $n-1, \dots, 0$ 次。求加权和为 $(n-1)/2$ 。
 - $T(n) = O(n)$

结论

在顺序表中插入或删除一个元素时，平均移动一半元素，当 n 很大时，效率很低。

顺序表特点：地址连续

b	a ₁	1
b + 1	a ₂	2
	...	
b + (i - 1) · l	a _i	i
	...	
b + (n - 1) · l	a _n	n
b + n · l		空闲
...		空闲
b + (MaxLen - 1) · l		空闲

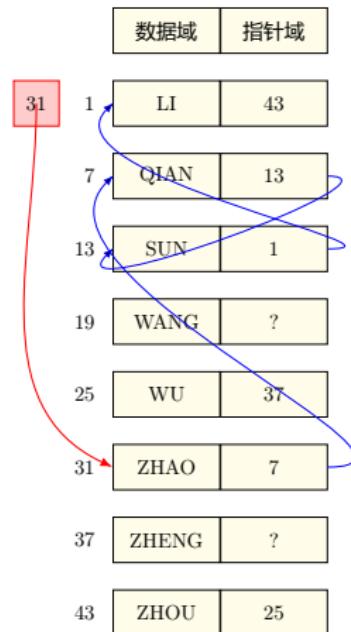
- 优点
 - * 直观
 - * 随机存储效率高

顺序表特点：地址连续

b	a ₁	1
b + 1	a ₂	2
	...	
b + (i - 1) · l	a _i	i
	...	
b + (n - 1) · l	a _n	n
b + n · l		空闲
...		空闲
b + (MaxLen - 1) · l		空闲

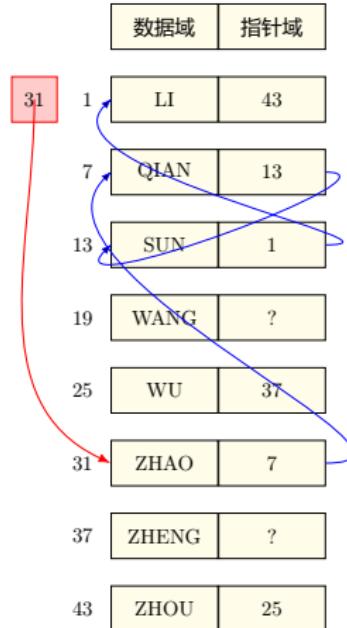
- 优点
 - * 直观
 - * 随机存储效率高
- 缺点
 - * 移动元素代价大

方案 2: 链式存储---链表



用一组任意的存储单元存储线性表的数据元素，利用指针指向直接后继的存储位置

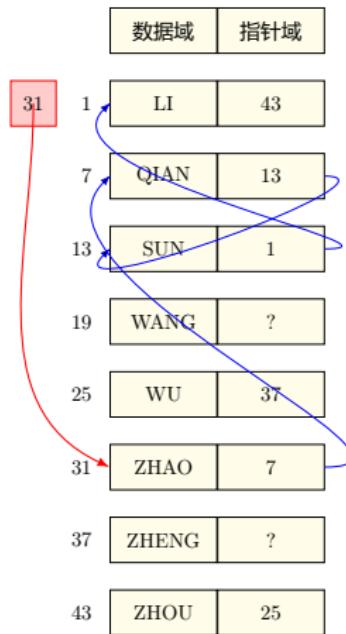
单链表的类型定义



- 单链表：每个结点有一个指针域
- 单链表可由头指针唯一确定，头指针指向第一个结点。
- Java Code:

```
class Node {  
    Object data;  
    Node next;  
}
```

单链表的类型定义



- 单链表：每个结点有一个指针域
- 单链表可由头指针唯一确定，头指针指向第一个结点。
- Python Code:

```
class Node:  
    def __init__(self, data, next=None):  
        self.data = data  
        self.next = next  
  
first = Node(1)  
second = Node(2)  
last = Node(3)  
first.next=second  
second.next = last  
print(first.next.next.data)
```

单链表上的常见操作

① 建立单链表 create()

② 求表长 length()

③ 查找 index(value)

④ 插入 insert(i,e)

⑤ 删除 remove(i)

⑥ 获取元素 get(index)

⑦ 显示 display()

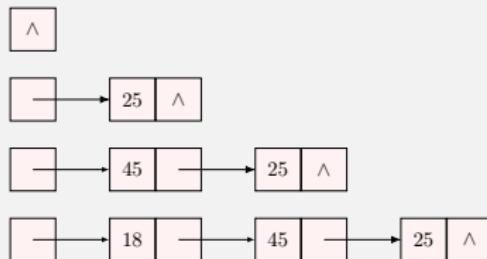
建立单链表

- 链表是**动态管理**的：链表中的每个结点占用的存储空间不是预先分配，而是运行时系统根据需求而生成的。
- 建立单链表从空表开始，每读入一个数据元素则申请一个结点，插入链表。

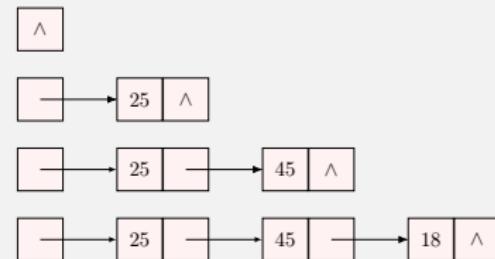
建立单链表：两种不同方式

如依次读入 25, 45, 18...

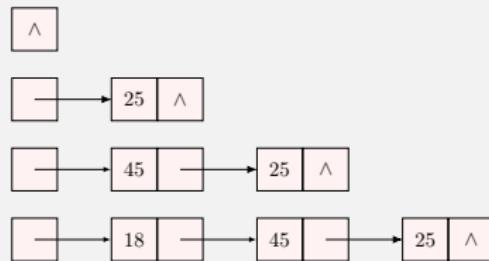
头部插入：



尾部插入：



建立单链表：头部插入

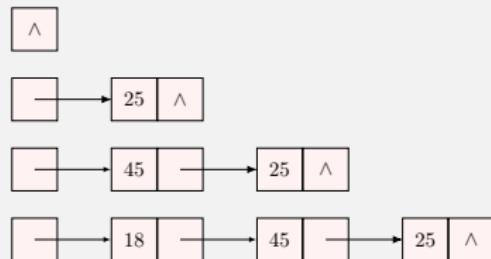


`Node s = new Node(val); // s 指向新结点`

`s.next = head; // 新结点后继为当前头结点`

`head = s; // 头指针指向新结点`

建立单链表：头部插入



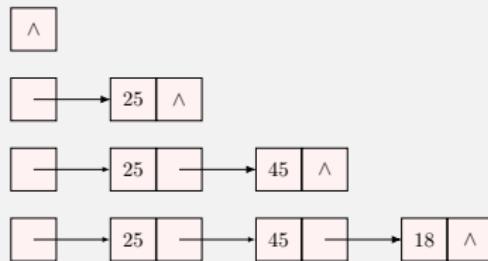
`Node s = new Node(val); //s 指向新结点`

`s.next = head; //新结点后继为当前头结点`

`head = s; //头指针指向新结点`

在空表时是否可行？

建立单链表：尾部插入



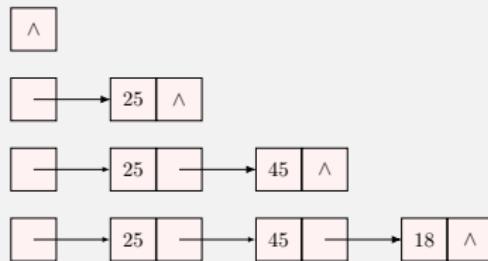
Node s=new Node(val); //s 指向新结点

Node r=Findlast(Head); //找到尾结点

r.next=s; //新结点成为尾结点的后继

r=s;

建立单链表：尾部插入



Node s=new Node(val); //s 指向新结点

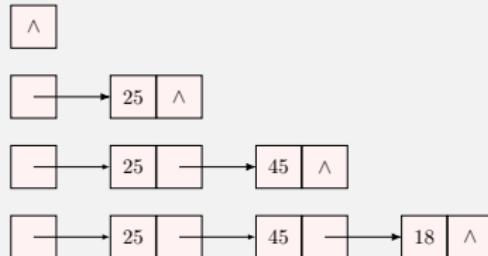
Node r=Findlast(Head); //找到尾结点

r.next=s; //新结点成为尾结点的后继

r=s;

在空表时是否可行？

建立单链表：尾部插入



```
Node s=new Node(val);
```

```
if(!head) //空表，插入第一个结点
```

```
    head=s; //新结点作为第一个结点
```

```
else //非空表
```

```
    r.next=s; //新结点作为最后一个结点的后继
```

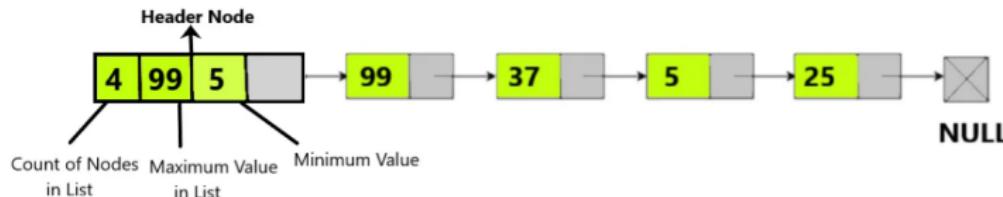
```
r=s; //尾指针 r 指向新的尾结点
```

头结点问题

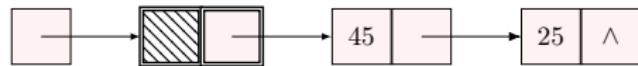
- 在上面的算法中，空表和非空表的处理是不同的
 - * 当链表为空，新结点作为第一个结点，地址放在链表头指针变量中（第一个结点没有前驱，其地址就是整个链表的地址）；
 - * 否则，新结点地址放在其前驱的指针域。
- 上述问题在很多操作中都会遇到，为方便操作，可在链表头部加一个“头结点”。

头结点问题

- 头结点的类型与数据结点一致，其数据域无定义，指针域中存放的是第一个数据结点的地址，空表时为空。头结点的数据域可以为空，也可存放线性表长度等附加信息，但此结点不能计入链表长度值。
- 加入头结点完全是为了运算的方便。有了头结点，即使为空表，头指针变量 head 也不为空，空表和“非空表”的处理成为一致。



求表长



```
int GetLength() {
    Node p = head.next;
    int len=0;
    while (p){
        p=p.next;
        len++;
    }
    return len;
}
```

求表长



```
int GetLength() {
    Node p = head.next;
    int len=0;
    while (p){
        p=p.next;
        len++;
    }
    return len;
}
```

```
class Node:
    def __init__(self, data, next=None):
        self.data = data
        self.next = next

    def get_length(self):
        p = self
        len = 0
        while(p!=None):
            p = p.next
            len = len + 1
        return len
```

```
hp = Node(None, Node(45, Node(25)))
print(get_length(hp))
```

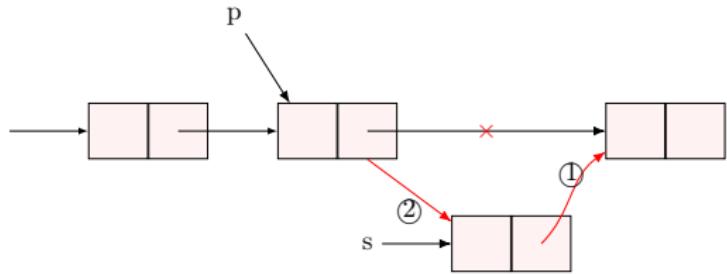
单链表的查找



- 按值查找 `index(value)`: 是否存在数据元素 X ? 序号是?
- 算法思路：“顺藤摸瓜” —— 从第一个结点开始，判断当前结点的值是否等于 x ，若是则返回该结点的指针，否则继续检查下一个，直到表尾。如果找不到则返回空。

```
public int index(value){ //在 L 中查找值为 x 的结点
    Node p=Head.next; int j=0;
    while ( p && p.data !=value){
        p=p->next; j++;
    }
    if(p) return j; else return -1;
}
```

插入新结点：在给定结点的前后

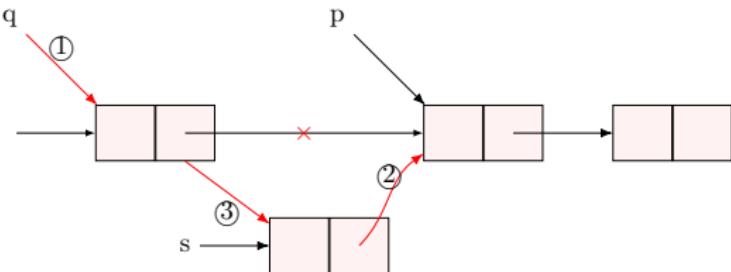


- 新结点插入到 *p* 的后面

s.next = p.next;

p.next = s;

插入新结点：在给定结点的前后



- 新结点插入到 p 的前面

找到 p 的前驱 q，在 q 之后插入 s。

`q=head;`

`while (q.next!=p) q=q.next;`

`s.next=q.next;`

`q.next=s;`

插入新结点

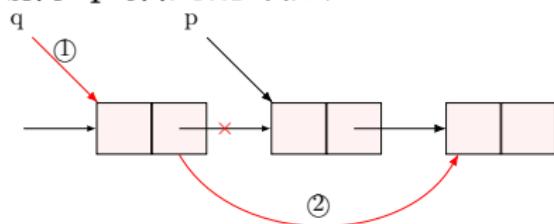
- 时间复杂度
 - * 后插操作为 $O(1)$: 不受 n 的影响
 - * 前插操作因为要先找到 p 的前驱，时间性能为 $O(n)$ 。

插入新结点

- 时间复杂度
 - * 后插操作为 $O(1)$: 不受 n 的影响
 - * 前插操作因为要先找到 p 的前驱，时间性能为 $O(n)$ 。
- 一个小技巧：
 - * 将 s 插入到 p 的后面，然后把 $p.data$ 与 $s.data$ 交换，这样能使得时间复杂性为 $O(1)$ 。

删除结点

- 删除 p 指向的结点



```
q.next=q.next.next;
```

- 首先要找到 p 的前驱结点 q , 其时间复杂性为 $O(n)$ 。
- 若要删除 p 的后继结点 (假设存在) , 则可以直接完成 :

```
p.next=p.next.next;
```
- 该操作的时间复杂性为 $O(1)$ 。

删除第 i 个结点

```
int Del(LinkList L, int i) //删除链表 L 第 i 个结点
{
    p=get(L, i-1); //查找第 i-1 个结点
    if (p==NULL) {
        printf("第 i-1 个结点不存在"); return -1;
    } else{
        if (!p->next)
            return 0; //第 i 个结点不存在
        else {
            p.next=p.next.next; //从链表中删除 i
            return 1;
        }
    }
}
```

单链表操作小结

- 在单链表上当前结点之前插入、删除一个结点，必须知道其前驱结点。
- 单链表不具有按序号随机访问的特点，只能从头指针开始一个个顺序进行。

链表特点



- 优点
 - * 插入删除方便
 - * 动态分配

链表特点



- 优点

- * 插入删除方便

- * 动态分配

- 缺点

- * 随机访问效率低

顺序表 VS 链表

顺序表优点

- 直观
- 随机访问效率高
- 没有为表达结点间的逻辑关系增加的额外开销

链表优点

- 插入删除方便
- 动态分配

链表缺点

- 为表达结点间的逻辑关系增加“指针域”
- 随机访问效率低

顺序表缺点

- 插入删除时效率低
- 需要预先分配足够大的存储空间

选取恰当的存储结构 I

① 基于存储的考虑

- ▶ 使用顺序表，在程序执行之前要明确规定它的存储规模（对 MAXSIZE 要有合适的设定），过大造成浪费，过小造成溢出。可见对线性表的长度或存储规模难以估计时，不宜采用顺序表。
- ▶ 链表不用事先估计存储规模，但链表的存储密度较低。存储密度是指一个结点中数据元素所占的存储单元和整个结点所占的存储单元之比。显然链式存储结构的存储密度是小于 1 的。

选取恰当的存储结构 II

② 基于运算的考虑

- ▶ 在顺序表中按序号访问 a_i 的时间性能时 $O(1)$ ，而链表中按序号访问的时间性能 $O(n)$ ，所以如果经常做的运算是按序号访问数据元素，显然顺序表优于链表；
- ▶ 在顺序表中做插入、删除时平均移动表中一半的元素，如果数据元素的信息量较大且表较长，不可忽视这一点；
- ▶ 在链表中作插入、删除，虽然也要找插入位置，但操作主要是比较操作，从这个角度考虑优于顺序表。

选取恰当的存储结构 III

③ 其它

- ▶ 顺序表容易实现，任何高级语言中都有数组类型，相对来讲简单直观，也是用户考虑的一个因素。

总之，两种存储结构各有长短，选择那一种由实际问题中的主要因素决定。通常“较稳定”的线性表选择顺序存储，而频繁做插入删除的话宜选择链式存储。

本堂小结

- ① 线性表的概念和两种存储方式.
- ② 顺序线性表的特征、基本操作.
- ③ 链表的特征、基本操作.
- ④ 循环链表和双向链表
- ⑤ 顺序表与链表大 PK
- ⑥ 顺序表和链表适用的场合

约瑟夫环

在罗马人占领乔塔帕特后，约瑟夫及他的 40 个战友躲到一个洞中，这些犹太人宁死也不想被敌人抓到，于是决定了一个自杀方式：

- 41 个人排成一个圆圈，由第 1 个人开始报数，每报数到第 3 人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止。
- 约瑟夫说他和另一个人逃过了这场死亡游戏: by luck or by the hand of God.
- 请问约瑟夫在这个圆圈中的位置是？

约瑟夫环

TODO