数据结构与算法

数据结构的基本概念

基本概念与术语

数据

这是一个最宏观的统称。所有能输入到计算机中并被程序处理的符号（数字、文字、图像、声音等）都叫数据。

数据元素

数据的基本单位。在程序中通常作为一个整体来处理。

> 在数据库里它叫“记录”，在树或图里它叫“节点” 或 “顶点”

- 类比： Excel表格中的“一行”

数据项

构成数据元素的最小单位，不可再分。

> 多个数据项组成一个数据元素。

- 类比： Excel表格中的“一个单元格”

数据对象

性质相同的数据元素的集合。

- 类比： Excel的“整张表”

> 它们之间的关系：数据项 $\rightarrow$ 数据元素 $\rightarrow$ 数据对象

数据结构

数据结构是数据元素之间的相互关系，即数据的组织方式。

包括3个方面：

1. 数据的逻辑结构
2. 数据的存储结构
3. 数据的运算

> 算法的设计取决于数据的逻辑结构；算法的实现取决于数据的存储结构

Step1 经典公式：

Pascal 之父、图灵奖得主 Niklaus Wirth 曾提出过一个著名的公式，深刻概括了编程的本质：

$$\text{程序} = \text{数据结构} + \text{算法}$$

Step2 形式化定义：

如果我们要用数学语言严谨地定义它，一个数据结构是一个二元组：

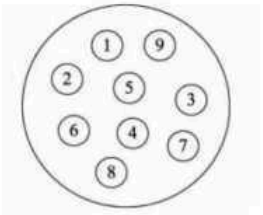
$$Data\_Structure = (D, R)$$

1. D: 是数据元素的有限集合。
2. R: 是 D 上关系的有限集合。

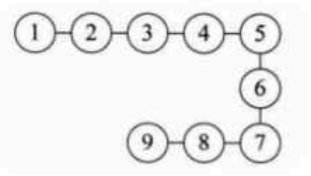
数据的逻辑结构

定义：数据的逻辑结构是从具体问题抽象出来的数学模型，是数据元素之间的逻辑关系。

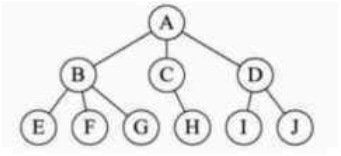
数据的逻辑结构分为线性结构和非线性结构，非线性结构又主要包括树结构和图结构。



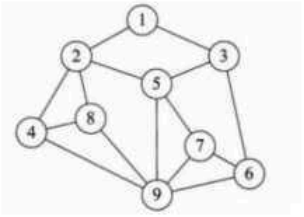
集合结构



线性结构



树形结构



图形结构

集合结构

关系映射：无特定关系

描述：

- 结构中的数据元素之间除了“同属于一个集合”的关系外，别无其他逻辑关系。
- 元素之间是离散的、无序的。

形式化表示：$R = \emptyset$

线性结构

关系映射：一对一

描述：

- 数据元素之间存在着严格的线性次序。
- 存在唯一的“第一个”元素和唯一的“最后一个”元素。
- 除头尾元素外，每个元素有且仅有一个前驱和一个后继。

> 典型代表： 线性表、栈、队列、字符串、数组。

树形结构

关系映射：一对多。

描述：

- 数据元素之间存在分层关系。
- 存在唯一的根节点。
- 除根节点外，每个节点有且仅有一个双亲，但可以有多个孩子。
- 结构内不存在环路。

> 典型代表： 二叉树、B树、堆。

图形结构

关系映射：多对多。

描述：

- 数据元素之间的关系是任意的。
- 任何两个顶点之间都可能通过边相连。
- 包含有向图和无向图。
- 可能存在环路。

> 典型代表： 社交网络图、交通路网、状态机。

数据的存储结构

数据的存储结构，也称为物理结构，是指数据的逻辑结构在计算机内存中的映像。

> 如何将逻辑上相关的数据元素，存储在物理内存单元中

在计算机科学中，主要存在四种基本的存储映射方式：

顺序存储结构

借助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。

描述：

- 将逻辑上相邻的数据元素存储在物理上连续的存储单元中。
- 随机访问： 可以通过首地址和偏移量直接计算出任一元素的内存地址。

公式如下：

$$Loc(e_i) = Loc(e_0) + i \times L（L为单个元素大小）$$

优缺点：

- 优： 存储密度高，查找速度快 $O(1)$。
- 缺： 插入和删除需要移动大量元素，要求连续的大块内存空间。

> 典型实现： 数组

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。

描述：

- 数据元素可以存储在内存中任意的存储单元。
- 可以是连续的，也可以是不连续的
- 每个数据节点包含两部分：数据域存储数据本身，指针域存储后继元素的地址。
- 顺序访问： 不支持随机访问，必须顺着指针链逐个查找。

优缺点：

- 优： 灵活利用碎片内存，插入和删除操作高效。
- 缺： 存储密度低，查找速度慢 O(n)。

> 典型实现： 链表。

索引存储结构

在存储数据元素的同时，建立附加的索引表。

描述：

- 索引表中的每一项称为索引项，通常形式为关键字或地址。
- 稠密索引：每个数据元素对应一个索引项。
- 稀疏索引：一组数据元素对应一个索引项。

优缺点：

- 优：检索速度快。
- 缺：建立索引表增加了空间开销，且增删数据时需维护索引表。

> 典型实现： 数据库的 B+ 树索引、操作系统的文件分配表。

散列存储结构

根据数据元素的关键字，通过散列函数直接计算出该元素的存储地址。

描述：

- 建立关键字与存储地址之间的一一映射关系：Address = H。
- 核心在于解决冲突，即不同关键字计算出相同地址的情况。

优缺点：

- 优：查找速度极快，理想情况下时间复杂度为 O(1)。
- 缺：仅支持精确查找，不适合范围查询或排序，可能存在哈希冲突。

> 典型实现： 哈希表

存储结构	核心特征	内存连续性	访问方式	空间效率
顺序存储	相对位置即关系	必须连续	随机访问	高
链式存储	指针链接关系	任意离散	顺序访问	较低

存储结构	核心特征	内存连续性	访问方式	空间效率
索引存储	数据 + 索引表	任意	查表访问	较低
散列存储	关键字算出地址	任意	计算访问	取决于填充因子

数据的运算

运算的定义：针对逻辑结构，指出运算的功能，即做什么。这通常在抽象数据类型中定义。

运算的实现：针对存储结构，指出运算的具体步骤，即怎么做。这需要通过具体的算法来完成。

数据的运算通常包括如下基本操作：

1. 初始化：创建一个空的数据结构，分配必要的内存空间。
2. 销毁：清楚数据结构，释放其占用的内存资源。
3. 判空：检查数据结构中是否包含数据元素。
4. 遍历：按照某种次序，访问数据结构中的每一个元素，且每个元素仅被访问一次。
5. 查找/检索：在数据结构中寻找满足特定条件的数据元素。
6. 插入：在数据结构的指定位置增加一个新的数据元素。
7. 删除：将数据结构中指定位置的元素移除。
8. 更新：修改指定位置元素的值。
9. 排序：重新排列数据元素，使其按关键字有序。
10. 合并：将两个同类的数据结构合并为一个。
11. 分裂：将一个数据结构拆分为两个。

算法

定义： 算法是解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作。

> 算法就是把“输入”转化成“输出”的一系列操作步骤。

算法的五大核心特征

并不是所有的指令序列都能叫“算法”，它必须满足以下 5 个条件：

1. 有穷性：
   - 算法必须在执行有限步骤之后结束，不能进入死循环。
   - 注意：操作系统或服务器的守护进程这种一直运行的程序，严格来说不是数学意义上的算法。
2. 确定性：
   - 每一步指令必须有明确的含义，不能有歧义。
   - 计算机不能理解“少许”、“适量”这种词。
3. 可行性：
   - 算法描述的操作必须是可以通过已经实现的基本运算执行有限次来实现的。
   - 你不能写一条指令叫“计算精确的 \pi 值”，因为 \pi 是无限不循环小数，算不完。
4. 输入：
   - 一个算法有0个或多个输入。
5. 输出：

- 一个算法必须有1个或多个输出，没有输出的算法是没有意义的。

算法与程序的关系

核心公式：

$$程序 = 数据结构 + 算法$$

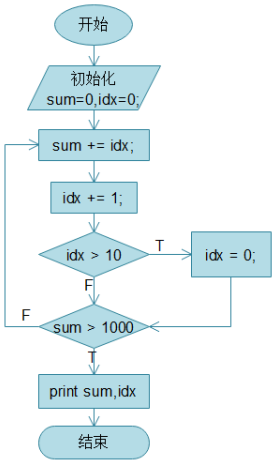
核心对比：

维度	算法	程序
本质	解决问题的逻辑思维与方法。	实现算法的计算机代码。
存在形式	可以写在纸上、脑子里、书本里。	必须存在于计算机文件（磁盘/内存）中。
语言依赖	与语言无关。同一个算法可以用 Python 写，也可以用 C++ 写。	依赖具体语言。必须符合 Python 或 C++ 的语法规则。
有穷性	必须有穷。步骤必须有限，不能死循环。	不一定有穷。例如操作系统就设计为一直在运行，直到断电。
错误处理	关注核心逻辑，通常假设输入是合法的。	必须处理所有异常，否则程序会崩。

算法的描述方法

一个算法可以用自然语言、流程图、伪代码来表示，也可以使用高级程序设计语言，如Java、C、C++来描述。

1. 自然语言：就是用我们日常说的话来描述。
2. 流程图：用标准的图形符号来表示算法的步骤。



3. 伪代码：这是介于“自然语言”和“编程语言”之间的一种描述方式。它看起来很像代码，但不需要遵守具体的语法规则

```
IF x > 0 THEN
    PRINT "是正数"
ELSE
    PRINT "是负数"
END IF
```

> 优点： 简洁清晰，很容易转化为具体的编程语言

4. 程序设计语言：直接上真代码（C, Java, Python 等）。

算法的评判标准

写出来能跑的代码不一定是好算法。我们在设计算法时，通常追求以下目标：

1. 正确性：必须能得出正确的结果。
2. 可读性：代码要容易被人读懂。
3. 健壮性：当输入了非法数据时，算法能做出适当的处理，而不是直接崩溃。
4. 高效率与低存储：
   - 时间复杂度低
   - 空间复杂度低

算法的时间复杂度

定义

算法的时间复杂度是一个函数，定性描述了该算法的运行时间如何随着问题规模 (n) 的增加而增长。它不是计算算法运行的具体时间（秒），而是计算基本操作执行次数的增长率。

假设问题规模为 n，算法中基本操作重复执行的次数是 problem size n 的某个函数，记为 $T(n)$。

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

其中 $f(n)$ 代表基本语句执行总次数。

大O表示法

大O表示法是一种用于描述函数渐进行为的数学符号。在计算机科学中，它用于表示算法复杂度的渐进上界，即在最坏情况下，算法执行时间的增长极

限。

若存在正常数 C 和 n_0，使得当 $n \geq n_0$ 时，恒有 $|T(n)| \leq C \cdot |f(n)|$，则称 $T(n)$ 是 $f(n)$ 的大O，记为：

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

在使用大O表示法时，遵循以下两条简化原则：

1. 忽略低阶项：只保留最高阶项
2. 忽略系数：去掉最高阶项前面的常数系数

示例： 若 $T(n) = 2n^2 + 3n + 1$，则时间复杂度为 $O(n^2)$。

时间复杂度排行

常见的数量级排序（效率由高到低）：

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$$

时间复杂度分析

1. 只看循环

在一个函数中，执行次数最多的通常是循环体。

```python
# 例子 1
def print_numbers(n):
    for i in range(n):   # 循环 n 次
        print(i)         # 执行 n 次
```

判定： 这是一个 $O(n)$ 算法。

2. 嵌套循环是“乘法”

如果一个循环套在另一个循环里，复杂度相乘。

```python
# 例子 2
def print_matrix(n):
    for i in range(n):       # 外层循环 n 次
        for j in range(n):   # 内层循环 n 次
            print(i, j)      # 总共执行 n * n 次
```

判定： $n \times n = n^2$，这是 $O(n^2)$ 算法。

3. 并列循环取“最大”

如果有两个分开的循环，只看量级最大的那个。

```python
# 例子 3
def complex_func(n):
    # 第一部分
    for i in range(n):
        print(i)         # 这里是 O(n)

    # 第二部分
    for i in range(n):
        for j in range(n):
            print(i, j)  # 这里是 O(n^2)
```

判定：$O(n) + O(n^2) \approx O(n^2)$。因为当 n 很大时，n 相比 n^2 微不足道。

4. 特殊情况：对数阶 $O(\log n)$

只要看到循环里，变量不是 `+1` 而是 `*2` 或 `/2`，通常就是 $\log n$。

```python
# 例子 4
i = 1
while i < n:
    i = i * 2    # 每次翻倍: 1, 2, 4, 8, 16...
```

分析： 假设 $n = 32$，只需要循环 5 次 ($2^5 = 32$)。
判定： $O(\log n)$。

5. 最好、最坏与平均情况

有些算法的快慢取决于运气。

比如：在数组里查找一个数字。

- 最好情况： 运气爆棚，第一个就是你要找的。 $\rightarrow O(1)$
- 最坏情况： 运气最差，最后一个才是，或者根本不存在。 $\rightarrow O(n)$
- 平均情况： 综合概率计算。

重要原则：在没有特殊说明时，我们通常只关注最坏情况。因为作为程序员，我们要保证系统在最倒霉的时候也不会崩溃。

算法的空间复杂度

定义：空间复杂度也是一个函数，记为 $S(n) = O(f(n))$。它衡量的是算法在运行过程中，临时占用存储空间大小与问题规模 (n) 之间的增长关系。

计算空间复杂度时，我们通常不计算“输入数据本身”占用的空间。而是关注的是额外空间

一个算法在计算机中所占用的存储空间包括：

1. 输入的数据所占用的存储空间
2. 算法本身所占用的存储空间
3. 辅助变量所占用的存储空间

> 算法的空间复杂度是指算法在运行过程中所占用的辅助存储空间大小。数据元素和算法本身所占的存储空间不属于空间复杂度的计算范畴

A. $O(1)$ - 原地工作

这是最理想的情况。无论数据规模 n 有多大，你只用了固定大小的额外空间

```python
# 例子：反转一个数组
def reverse_array(arr):
    left = 0
    right = len(arr) - 1
    while left < right:
        # 只用了 temp, left, right 这几个固定变量
        # 无论 arr 有 1万个还是 1亿个数据，额外空间都不变
        temp = arr[left]
        arr[left] = arr[right]
        arr[right] = temp
        left += 1
        right -= 1
```

B. $O(n)$ - 线性空间

额外消耗的空间与输入数据量成正比。通常发生在你需要拷贝一份数据，或者使用数据结构存所有元素时。

```python
# 例子：把数组里的偶数挑出来，存到一个新数组里
def get_even_numbers(arr):
    new_arr = []            # 创建了一个新容器
    for num in arr:
        if num % 2 == 0:
            new_arr.append(num) # 可能会存 n 个数
    return new_arr
```

判定：最坏情况下，所有数字都是偶数，`new_arr` 的大小等于 n。所以 $S(n) = O(n)$。

C. $O(n^2)$ - 二维空间

通常出现在使用了二维数组的情况。

场景： 动态规划中的状态表、图的邻接矩阵存储。

计算： $n \times n$ 的表格，空间就是 $O(n^2)$。