西北工業大學

算法与数据结构

Algorithms and Data Structures

软件学院 **钱锋** 编

2024 年 4 月 14 日

算法与数据结构

钱锋[1,2]

2024 年 4 月 14 日

[1]Email: strik0r.qf@gmail.com
[2]西北工业大学软件学院, School of Software, Northwestern Polytechnical University, 西安 710072

目录

第 1 章　绪论

1.1　算法与数据结构

1.1.1　数据结构的基本概念

- **数据** (data) 是描述客观事物的数值、字符以及能输入机器且能被处理的各种符号的集合.
- **数据元素** (data element) 是组成数据的基本单位, 是数据集合的个体.
- **数据对象** (data object) 是性质相同的数据元素的集合, 是数据集合的一个子集.
- **数据结构** (data structure) 是指相互之间存在一种或多种特定关系的数据元素的集合, 是带有结构的数据元素的集合, 它指的是数据元素之间的相互关系, 即数据的组织形式.
- **数据类型** (data type) 是一组性质相同的值的集合以及定义在这个集合上的一组操作的总称. 数据类型按照其"值"的特性可以分为原子类型和结构类型, 顾名思义, 所谓原子类型就是可以不可再分的数据类型.
- **抽象数据类型** (abstract data type, ADT) 定义了一个数据对象, 数据对象中各元素间的结构关系以及一组处理数据的操作. 抽象数据类型最重要的特点是数据抽象与信息隐蔽.

例 1.1.1. 整数数据的对象是集合 $\mathbb{Z} = \{0, \pm 1, \pm 2, \cdots\}$.

例 1.1.2. 在高级语言中, 整型数 `int` 的取值范围为 $[-2147483648, +2147483648) \cap \mathbb{Z}$, 即从 -2147483648 到 $+2147483647$, 在整型数中定义了加、减、乘、除和取模五种二元代数运算.

1.1.2　数据结构的内容

数据的逻辑结构是指数据元素之间的逻辑关系的描述. 数据结构的形式定义是一个序偶 (D, R), 其中, $D = \{x_1, x_2, \cdots, x_n\}$ 为数据元素的有限集合, 这里的字母 D 表示"数据"(data). R 为 D 上具有 m 元关系 $\mathcal{R}$ 的序偶的有限集合, 即 D^m 的一个有限子集, 也就是 D 中 m 元序偶 $(x_1, x_2, \cdots, x_m)$ 的有限集合. 这里的字母 R 表示"关系"(relationship).

我们假设

第 2 章　算法分析

2.1　数学基础

算法所需要的执行时间, 我们将其记作 $T(n)$, 往往是与问题规模 n 有关的函数, 在这里的问题规模 n 可以简单地理解为我们输入数据的大小或者数量级. 接下来我们引入的概念在函数之间建立了一种比较的方式, 这其实就是我们在微积分 (calculus) 中学习过的无穷大量的比较. 在接下来的讨论中, 设 $T(n), f(n), g(n)$ 是在非负整数集上有定义的函数.

定义 2.1.1. 如果 $\exists c > 0 \exists n_0 \in \mathbb{N}^+ \forall n > n_0 \, (T(n) \leqslant cf(n))$, 那么称 $T(n) = O(f(n))$, $f(n)$ 此时称为 $T(n)$ 的**上界** (upper bound).

定义 2.1.2. 如果 $\exists c > 0 \exists n_0 \in \mathbb{N}^+ \forall n > n_0 \, (T(n) \geqslant cg(n))$, 则称 $T(n) = \Omega(g(n))$, $g(n)$ 此时称为 $T(n)$ 的**下界** (lower bound).

定义 2.1.3. $T(n) = \Theta(h(n)) : \Longleftrightarrow \ ((T(n) = O(h(n))) \wedge (T(n) = \Omega(h(n))))$.

定义 2.1.4. $((T(n) = O(p(n))) \wedge (T(n) \neq \Theta(p(n)))) \Longrightarrow T(N) = o(p(n))$.

我们需要掌握以下三个结论:

定理 2.1.1. *如果* $T_1(n) = O(g(n))$, $T_2(n) = O(g(n))$, *那么*:

1° *加法规则*: $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$.

2° *乘法规则*: $T_1(n)T_2(n) = O(f(n)g(n))$.

定理 2.1.2. (多项式规则). $\forall p(n) \in \mathbb{F}[n] \left(p(n) = \Theta\left(n^{\deg p(n)}\right)\right)$.

定理 2.1.3. (对数规则). $\forall k \in \mathbb{R} \left(\log^k n = O(n)\right)$.

定理 2.1.1 告诉我们, 在大 O 表示法中, 低阶项一般可以忽略 (我们一般亲切地称其为“抓大头”), 此外, 常数也可以弃掉, 我们在这里不考虑常数的影响, 也就是说, 当两个函数的增长速率只相差常数倍时, 我们称它们具有相等的增长速率. 定理 2.1.2 告诉我们多项式的增长速度由其首项决定, 定理 2.1.3 告诉我们对数是增长地非常缓慢的.

定义 2.1.5. (无穷大量的比较). 设 $f(n), g(n)$ 是两个无穷大量, 如果:

- $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$, 则称 $f(n)$ 为 $g(n)$ 的低阶无穷大量, 即 $f(n)$ 的增长比 $g(n)$ 慢;
- $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = c \neq 0$, 则称 $f(n)$ 为 $g(n)$ 的同阶无穷大量, 即 $f(n)$ 与 $g(n)$ 的增长差不多;
- $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty$, 则称 $f(n)$ 为 $g(n)$ 的高阶无穷大量;

- $\lim\limits_{n\to\infty}$ 不存在, 也非无穷, 那么 $f(n)$ 与 $g(n)$ 之间并无无穷大量间的关系.

典型的增长率有

$$O(1) < O(\log n) < O(n) < O(n \log n) < O\left(n^2\right) < O\left(n^3\right) < O\left(a^n\right).$$

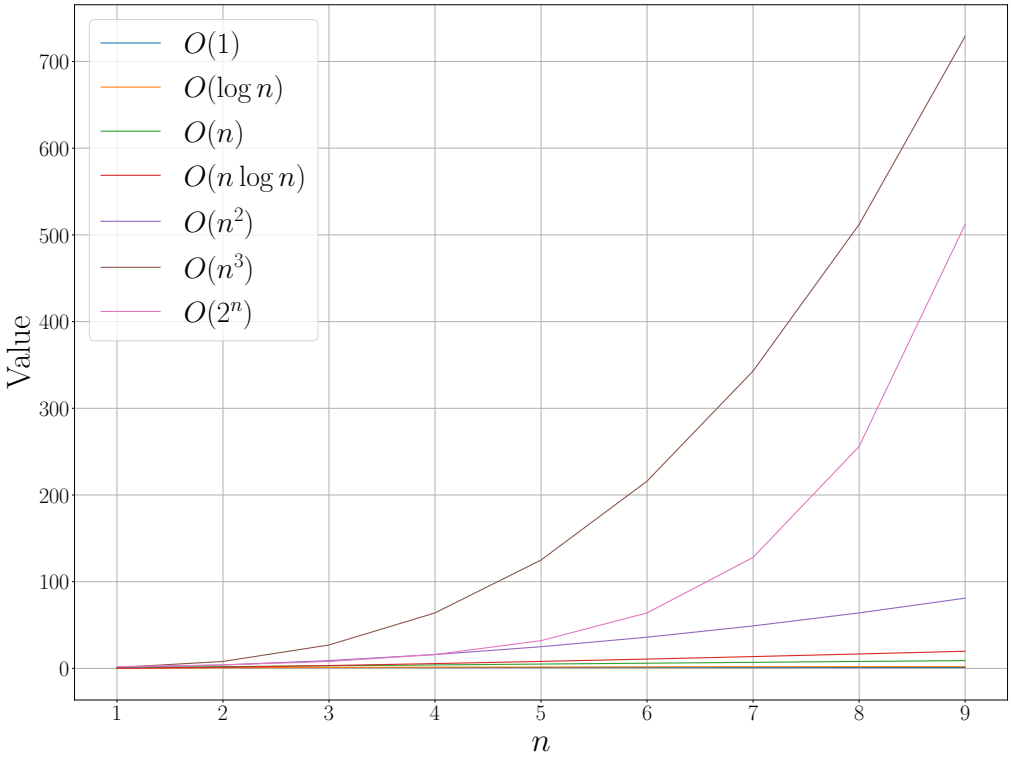

图 2.1: 常见的增长率的比较

2.2 运行时间的计算

由于大 O 分析法忽略常数倍数的影响和低阶项的影响, 因此我们在分析算法执行时间的时候实际上只需要抓出占用该算法执行时间的“大头”就可以了. 本节我们介绍一些评价算法执行时间的一般法则, 这些法则可以减轻我们分析算法执行时间时的工作量.

定理 2.2.1. *单条语句的执行时间总是 $O(1)$, 这是因为我们认为一条语句总是能在有限步骤内执行完的.*

定理 2.2.2. *顺序语句的执行时间等于中执行时间最大的那条语句 (或者语句块) 的执行时间.*

证明. 这是加法法则所决定的. □

定理 2.2.3. *一次 `for` 循环的运行时间至多是该 `for` 循环内语句 (包括测试) 的执行时间乘以迭代次数.*

定理 2.2.4. *在一组嵌套的 `for` 循环内部, 一条语句的运行时间为该语句的运行时间乘以所有这些 `for` 循环的大小的乘积.*

定理 2.2.5. 一个 `if-else` 语句块的执行时间不超过条件测试的时间加上两个可能执行的分支的执行时间的最大值. 这就是说, 程序段 `if (cond){S1} else {S2}` 的执行时间不超过 $O_{\text{cond}}(n)+\max(O_{\text{S1}}(n)+O_{\text{S2}}(n))$.

例 2.2.1. 考虑计算 $\sum_{i=1}^{N} i^3$ 的一个简单的程序片段:

```
int Sum (int N) {
    int i, partialSum;

    partialSum = 0;
    for (i=1; i<=N; i++) {
        partialSum += i*i*i;
    }
    return partialSum;
}
```

例 2.2.2. 考虑 n 阶行列式的组合定义

$$\det \boldsymbol{A} = \sum_{j_1 j_2 \cdots j_n \in S(n)} a_{1j_1} a_{2j_2} \cdots a_{nj_n}.$$

在用组合定义求解行列式的过程中, 设问题规模为 n, 那么我们首先需要进行 n 次乘法来获得其中一个求和项, 这是因为行列式是一系列乘积的和, 其中每一个乘积是由来自行列式中不同行、不同列的 n 个元素. 同理, 另外的 $(n-1)$ 个求和项也要用类似的方法取得, 这就意味着我们需要进行 n^2 次乘法. 在求和的时候, 我们还要进行 n 次加法, 所以实际的执行时间

$$T(\det \boldsymbol{A}) = O(n^2 + n) = O(n^2).$$

例 2.2.3. 求解整数 $n(n \geqslant 0)$ 的阶乘的算法如下:

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

其时间复杂度为 _____.

A. $O(\log_2 n)$ B. $O(n)$ C. $O(n \log_2 n)$ D. $O(n^2)$

例 2.2.4. 两个长度分别为 m 和 n 的升序链表, 若将它们合并为长度为 $m+n$ 的一个降序链表, 则最坏情况下的时间复杂度为 _____.

A. $O(n)$ B. $O(mn)$ C. $O(\min(m,n))$ D. $O(\max(m,n))$

本书的目的是研究如何设计出“优秀”的数据结构和算法, 什么是数据结构? 什么是算法? 简单地说, 数据结构是组织和访问数据的一种系统化方式, 算法是在有限的时间里一步步执行某些任务的过程.

2.3 实验研究

如果算法已经实现了, 可以利用 Python 中 `time` 模块的 `time()` 函数来计算算法的运行时间:

```
from time import time
start_time = time()
# 在这里执行你的算法
end_time = time()
elapsed = end_time - start_time
```

其中, `time()` 函数返回自 1970 年 1 月 1 日 00:00:00 后已经过去的秒数或分钟数, 这个时间点也被称为新纪元基准时间.

对于这一种研究算法运行效率的方法来说, 它确实能在一定程度上反应程序执行起来是快还是慢, 但是对于计算机来说, 在 `start_time` 和 `end_time` 之间的这段时间里其实有很多的进程在共用 CPU, 它们有的来自于其他应用程序, 有的则是一些维持操作系统工作所必需的进程. 我们测算得到的 `elapsed` 其实是这两部分时间的总和, 并不是算法真正占用的 CPU 时间.

所以, 尽管用执行时间来衡量算法的效率这种方法是有用的, 但是它具有三个主要的局限性:

- 由于执行时间与具体的硬件和软件环境高度相关, 所以这种方法难以直接比较两个算法的运行时间;
- 这种实验方法需要给定一组确定的测试输入, 也就是说这种方法忽略了给定输入所需要的运行时间;
- 为了计算执行时间, 我们必须把算法完全实现.

第三个缺点非常致命, 它说明了计算执行时间来衡量算法效率的本质是执行这个算法. 我们研究算法执行效率的目的是在编写算法或者实现算法以前, 选择一种效率最高的方案, 而当你已经实现了算法, 再来研究它的效率, 很明显这是本末倒置的, 谁也不想花费大量的时间, 实现了一堆低劣的算法才从中找到合适的方案. 但不可否认的是, 计算算法的运行时间仍然是一种有效的衡量程序运行速度的方式和指标, 它也是为数不多的我们能够感知到的指标.

所以, 我们的目标是寻找一种更好的能够分析算法效率的方法, 它应该满足以下的条件和要求:

- 与具体的软硬件环境无关, 它允许我们在不考虑具体的运行环境的前提下, 任意的评价两个算法的相对效率, 这有利于我们寻找出最优的方案;
- 研究不需要实现的高层次算法, 它允许我们在研究算法的效率时只需要研究一些高层次的抽象的算法, 而不需要把底层的细节也实现出来;
- 考虑所有可能的输入.

2.3.1 原子操作: 算法在执行过程中的基本操作

我们的目的是通过研究高层次的抽象算法来分析算法的执行效率, 而不是具体的实现一堆低劣的算法. 所以我们可以定义一系列的原子操作来代表算法中最基本的操作单元. 通常的讲, **原子操作**是不可再分的基本计算步骤, 比如说算术运算、比较、赋值等等.

2.3.2 获取算法运行时间的增长情况

我们把每一个算法和一个函数 $f(n)$ 联系起来, 其中, n 为给定的输入大小, 而 $f(n)$ 表示在给定的大小规模的输入下, 算法所执行的原子操作的数量. 这是算法效率的一个关键度量, 通常用 O 表示, 它表示算法的渐进复杂度.

本章后面的内容, 就将会具体介绍我们分析算法效率的方法. 我们首先要介绍在算法分析中最常用的最重要的 7 种函数, 然后介绍把算法运行时间的增长率与函数 $f(n)$ 联系起来的方法——渐进分析法. 然后介绍一些简单的数学证明的方法, 例如反证法和归纳法. 本章对读者的高等数学知识储备有一定要求.

2.4 常用的 7 种函数

2.4.1 常数函数

常数函数指的是把任何正整数 n 映射为同一个常数 $c \in \mathbb{R}$ 的函数, 即

$$f(n) = c.$$

这就是说, n 的值并不重要, $f(n)$ 总是一个定值, 即

$$\forall n \in \mathbb{N}^+ \left(f(n) = c \right).$$

在算法分析中, 常数函数描述了在计算机上需要做的基本操作的步数, 例如两个数相加、给一些变量赋值或者比较两个数的大小.

2.4.2 对数函数

对数是什么呢? 你可以这么理解: 对数之于指数就相当于除法之于乘法, 如果 $b > 1$, 并且 $b^x = n$, 那么我们定义

$$x \xlongequal{\text{def}} \log_b n,$$

现在, 请你把这句话默念三遍: 对数 $\log_b n$ 是我们为了得到 n 必须把 b 提升的幂次的数量, 也就是说, b 自乘 $\log_b n$ 次后得到了 n.

在计算机科学中, 对数函数最常见的底数是 2, 这是因为计算机里所有的数据都是用二进制序列来表示的, 并且许多算法都会出现把一个输入分成两半的操作. 所以如果我们省略底数的符号, 它一般指代底数为 2 的对数函数, 即

$$\log n \xlongequal{\text{def}} \log_2 n.$$

接下来我们给出对数的一些运算法则, 与其说是给出, 不如说是复习 (除非你是一个正在上初中却在看我的书的天才):

定理 2.4.1. *对于实数 $a, c > 0$, $b, d > 1$, 我们有:*

1° $\log_b(ac) = \log_b a + \log_b c$.

2° $\log_b \left(\dfrac{a}{c} \right) = \log_b a - \log_b c$.

3° $\log_b(a^c) = c \log_b a$.

4° $\log_b a = \dfrac{\log_d a}{\log_d b}$.

5° $b^{\log_d a} = a^{\log_d b}$, *这样写可能会让公式显得很局促, 所以我们不妨写作* $\exp_b (\log_d a) = \exp_a (\log_d b)$.

证明. 我们只证明第五条. 第五条性质的证明需要用到指数换底公式 (别跟我扯什么在这本书上是先有对数才有指数的, 你学高数的时候已经学过这个内容了, 不然的话你学高数是在浪费时间吗), 即

$$u^v = \exp\left(v \ln u \right).$$

两边都换底, 那么

$$\text{左边} = \exp\left(\log_d a \cdot \ln b \right) = \exp\left(\frac{\ln a}{\ln d} \cdot \ln b \right),$$

其中最后一个等号我们使用了性质 4, 即对数的换底公式, 同理可得

$$\text{右边} = \exp\left(\log_d b \cdot \ln a \right) = \exp\left(\frac{\ln b}{\ln d} \cdot \ln a \right),$$

显然 左边 = 右边. □

要计算一个数的对数往往是很困难的, 准确计算对数涉及微积分的应用, 最常见的方法是利用 Taylor 公式来逼近. 不过我们可以计算出大于等于 $\log_b n$ 的最小整数, 这就是我们常说的“天花板函数”或者向上取整函数 $\lceil \log_b n \rceil$. 计算它的 Python 算法很简单, 你可以自己尝试一下:

```python
def ceiling_log(b,n):
    dividend = n # 真数
    divisor = b  # 底数
    count = 1

    # 用真数反复除以底数直到商小于1
    while (dividend/divisor) >= 1:
        dividend = dividend/divisor
        count += 1

    # 能除的次数就是对数的向上取整
    return count
```

除此之外, 对数的换底公式给出了用计算器或者计算机计算任意底数的指数的方法. 比如说, 你的计算器上的 LOG 按钮计算的是 $\ln x$, 那么对于任意合法的底数 a (合法指的是它应该满足 $0 < a < 1$ 或者 $a > 1$), 我们都有

$$\log_a x = \frac{\ln x}{\ln a},$$

在 MATLAB 上, 我们就是这样计算任意底数的对数的.

第 3 章　线性表、栈和队列

3.1　线性表 ADT

3.1.1　线性表的定义

线性表 (linear list) 是具有相同数据类型的 $n(n \geqslant 0)$ 个数据元素的有限序列, 其中 n 称为表长, 当 $n = 0$ 时线性表是一个空表. 若用 L 命名线性表, 则其一般表示为

$$L = (a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n) .$$

其中下表 i 表示数据元素 a_i 的索引 (index), 它指示了元素 a_i 在表中的位置[1]. 关于线性表, 有如下概念需要大家注意:

- 不含有任何元素的线性表称为**空表** (empty list).
- a_1 是唯一的"第一个"数据元素, 称为表头, a_n 是唯一的"最后一个"数据元素, 称为表尾.
- 元素 a_{i-1} 称为元素 a_i 的前驱, a_{i+1} 称为 a_i 的后继, 显然, 除了 a_1 外的任意元素都有前驱元, 除了 a_n 以外的任何元素都有后继元.

3.1.2　线性表的基本操作

我们需要定义的线性表 ADT 上进行的基本操作有:

- `printList`: 打印线性表中的所有元素.
- `makeEmpty`: 创建一个空的线性表.
- `find`: 返回关键字 `key` 首次出现的位置, 即存储 `key` 的首个线性表元素的索引.
- `insert`: 在线性表的某个位置上插入一个元素.
- `delete`: 在线性表的某个位置上删除一个元素.
- `findKth`: 返回某个位置处的元素.

3.2　线性表的简单数组实现

我们可以用一个数组来实现一个简单的线性表. 但这并不是一个很好的方案, 这是因为我们在设计程序的时候并不知道我们将要在线性表中存储多少的元素, 因此我们不得不在程序设计阶段估计线性表的

[1] 要注意线性表的索引从 1 开始, 而数组元素的索引从 0 开始.

大小, 一般来说, 我们需要估计的大一点, 而这很有可能会造成大量的空间浪费, 或者我们的估计还不够大, 存不下用户想要存储的数据.

除此之外, 数组实现的线性表在增添和删除元素是的开销是巨大的. 具体的原因不再赘述, 给各位读者留作练习. (提示: 你在数组中间插入一个元素的时候, 是不是需要把后面的每一个元素都往后挪一个位置? 如果数组不够大了装不下了, 你是不是需要重新申请一个更大的数组, 然后把现有的数据和新的数据一个一个复制进去?)

综上所述, 我们一般不使用数组来实现线性表 ADT.

3.3 线性表的链式存储

线性表的链式存储称为**链表** (linked list), 它的基本思想是: 把表中的数据分散地存储在内存当中, 每存储一个元素, 就利用一个指针指向下一个元素. 这些指针就像一个链条一样把所有的元素"串"了起来 (这么一说, 链表这个名字还挺形象, 想象你在超市经常买的一种香肠, 它也是一个串一个最后全部连在一起的).

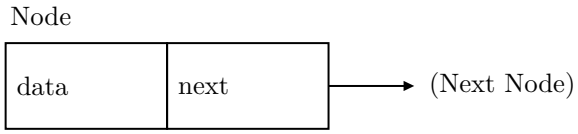

图 3.1: 链表的一个节点

如图 3.1 所示, 链表中的一个元素成为一个**节点** (node), 一个节点内包含了该元素存储的数据, 和指向该元素的后继元的一个指针, 我们称之为 `next` 指针.

3.3.1 双向链表

双向链表 (doubly linked list) 的基本思想是在每一个节点都增加一个指向其前一个节点的指针. 它增加了空间的需求, 但是使得链表的删除操作得到简化.

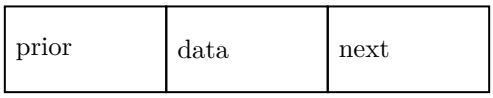

图 3.2: 双向链表的节点

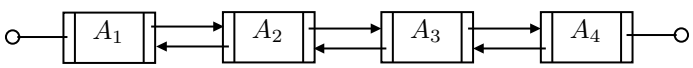

图 3.3: 一个双向链表

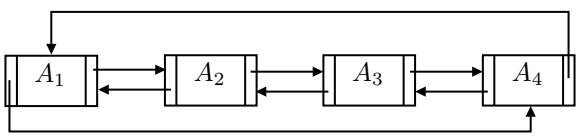

图 3.4: 一个没有表头的双向循环链表

程序清单 3.3.1.

```c
#include <stdio.h>
#include <stdlib.h>

typedef int Element;
typedef struct DNode {
    Element data;
    struct DNode *prior, *next;
};
typedef struct DNode *DNode;
// DNode 是一个指向 DNode 结构的指针, 在后续代码中一般用于指向节点

typedef struct DNode *DoubleList;
// DoubleList 是一个指向 DNode 结构的指针, 在后续程序中一般用于指定链表

// 初始化链表与销毁链表的有关函数
_Bool initList(DoubleList list);

// 插入节点与删除节点的相关函数
_Bool insert(DoubleList list, int index, Element data);
_Bool delete(DoubleList list, int index);

// 打印链表的有关函数
void printList(DoubleList list);
void printListInverse(DoubleList list);

int main() {
    // 准备工作
    struct DNode head; // 声明头节点
    DoubleList list = &head; // 将头节点绑定到链表

    // 测试初始化与插入算法 --> 测试通过
    initList(list);
    for (int i=1; i<5; i++) {
        insert(list, i,i);
        printList(list);
        printListInverse(list);
    }

    // 测试删除
    delete(list, 4);
    printList(list);
    // 应该从 1 -> 2 -> 3 -> 4 变成 1 -> 2 -> 3
    delete(list, 3);
    printList(list);
```

```
45      // 应该从 1 -> 2 -> 3 变成 1 -> 2
46      return 0;
47  }
48
49  /**
50   * 初始化双向链表的操作
51   * @param list 将要被初始化的链表的指针（实际上也就是节点的指针）
52   * @return 初始化成功则返回 1，否则返回 0.
53   * */
54  _Bool initList(DoubleList list) {
55      list->next = list->prior = NULL;
56  }
57
58  /**
59   * 双向链表的插入操作
60   * @param index 在第 Index 个节点之前插入一个新的节点
61   * @return 如果申请空间成功且成功插入节点，则返回 1.
62   */
63  _Bool insert(DoubleList list, int index, Element data) {
64      // 首先判断插入索引是否合法，如果不合法则返回 0.
65      if (index <= 0) return 0;
66
67      // 查找到第 index-1 个节点.
68      struct DNode *current = list;
69      while (--index) {
70          current = current->next;
71          if (current == NULL) return 0; // 如果当前位置 current 为空则表示已经来
                到末尾，但并未找到第 i 个节点，说明插入位置不合理.
72      }
73
74      // 申请新的节点空间并插入新节点
75      DNode newNode = malloc(sizeof(struct DNode));
76      if (newNode) {
77          newNode->data = data;
78          if (current->next) {
79              newNode->next = current->next;
80              current->next->prior = newNode;
81          } else {
82              newNode->next = NULL;
83          }
84          current->next = newNode;
85          newNode->prior = current;
86          return 1; // 如果申请空间成功且成功插入节点，则返回 1.
87      } else return 0; // 如果申请空间不成功则返回 0.
88  }
```

```
89
90  /**
91   * 删除链表中元素的函数
92   * @param list 要删除元素的链表
93   * @param index 要删除元素的索引
94   * @return 删除成功则返回 1，如果删除元素的索引不合法或者删除失败则返回 1.
95   * */
96  _Bool delete(DoubleList list, int index) {
97      // 判断待删除的元素是否合法，并将链表遍历到将要删除的节点的前一个节点
98      if (index <= 0) return 0;
99      DNode current = list;
100     while (--index) {
101         current = current->next;
102         if (current == NULL) return 0;
103     }
104
105     DNode tmp = current->next;
106     // 如果要删除的实际上是尾节点
107     if (current->next->next == NULL) {
108         current->next = NULL;
109     } else { // 要删除的不是尾节点
110         current->next = tmp->next;
111         tmp->next->prior = current;
112     }
113     free(tmp);
114 }
115
116 void printList(DoubleList list) {
117     DNode current = list->next;
118     while (current) {
119         printf("%d", current->data);
120         if (current->next) printf(" -> ");
121         current = current->next;
122     }
123     printf("\n");
124 }
125
126 // 倒序打印链表
127 void printListInverse(DoubleList list) {
128     // 遍历到链表末尾
129     DNode current = list->next;
130     DNode head = list;
131     while (current->next) current = current->next;
132     while (current->prior) {
133         printf("%d", current->data);
```

```
134         if (current->prior != head) printf(" -> ");
135         current = current->prior;
136     }
137     printf("\n");
138 }
```

表 3.1: 线性表链式存储方式的比较

找首元素节点	找表尾节点	找 p 节点的前一个节点

3.4　线性表的应用

3.4.1　多项式 ADT

1. 一元多项式的表示

次数非负的多项式 $f(x)=\sum_{i=0}^{n}a_ix^i$, 其中 $a_n\neq 0$, 则可以用一个线性表 Q 来表示, 其中

$$Q=(a_0,a_1,a_2,\cdots,a_n)\,,$$

假设 $h(x)=\sum_{i=0}^{m}b_ix^i$, 其中 $b_m\neq 0$ 且 $m<n$, 那么 $f(x)+h(x)$ 可以用线性表表示为

$$(a_0+b_0,a_1+b_1,\cdots,a_m+b_m,a_{m+1},a_{m+2},\cdots,a_n)\,.$$

2. 一元多项式的存储

如先前所述, 线性表的存储方式有顺序表和链表两种.

我们把零系数较多的多项式称为稀疏多项式. 显然, 如果对稀疏多项式采用顺序表存储, 那无疑会造成存储空间的极大浪费, 例如, 多项式 $f(x)=1+5x^{10000}+7x^{20000}$ 的顺序表存储需要 200001 个空间, 但实际有效的存储数据只有 3 个, 因此, 对于稀疏多项式来说采用链式存储更能节省储存空间.

由于非零系数较多的非稀疏多项式可以用简单的数组实现, 讨论的价值不高, 本节主要讨论稀疏多项式的链式存储. 我们采用双向链表作为存储结构来实现多项式 ADT.

3. 多项式的节点结构

在双向链表中, 节点 `Node` 含有 `prior`、`data` 和 `next` 三个数据域, 为了用一个节点存储多项式中的一项, 我们需要把 `data` 数据域明确为多项式中某项的系数 `coef` 和某项的次数 `exp`. 多项式的节点结构如图 3.5 所示.

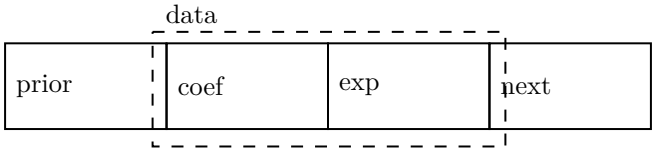

图 3.5: 一元多项式的节点结构

节点结构的声明如下:

```
struct DNode {
    double coef;
    int exp;
    struct DNode *prior, *next;
};
typedef struct DNode *DNode;
// DNode 是一个指向 DNode 结构的指针, 在后续代码中一般用于指向节点
```

4. 应用: 高精度计算圆周率

$$\frac{\pi}{2}=\sum_{k=0}^{\infty}\frac{(2k)!!}{(2k+1)!!}\left(\frac{1}{2}\right)^{k}.$$

3.5 栈 ADT

3.5.1 栈的定义

栈 (stack) 是一种限制插入和删除只能在一个位置上进行的表, 该位置是表的末端, 称为**栈顶** (stack top), 相对的, 表的第一个元素称为**栈底** (stack bottom). 对栈的基本操作主要有两种:

- `push()`: 进栈操作, 相当于插入元素.
- `pop()`: 出栈操作, 相当于取出元素, 它返回位于栈顶的元素.

由于在栈中只有位于栈顶的元素才是可访问的, 因此后进入的元素一定先被取出, 于是又称为 **后进先出表** (Last In, First Out, LIST).

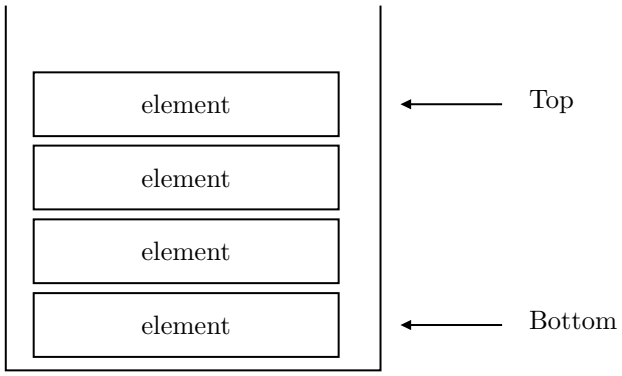

图 3.6: 栈

由于栈的限制存取的特性, 入栈操作、出栈操作的时间和空间复杂度都是 $O(1)$. 当某个数据集合只涉及在一端插入和删除数据, 并且满足后进先出、先进后出的特性的时候, 我们就应该首选栈作为数据结构.

例 3.5.1. 若进栈序列为 $1, 2, 3, 4$, 允许边进边出, 但每次出栈都要记录出栈的元素, 则不可能得到的出栈序列是?

A. $3, 2, 1, 4$ B. $3, 2, 4, 1$

C. $4, 2, 3, 1$ D. $2, 3, 4, 1$

评注. 要注意栈的后进先出原则, 如果第一个出栈元素是 4, 那么说明在此之前四个数已经全部进栈了, 4 出栈后的栈顶应该是 3. 于是本题选择 C 选项. 关于这一类题型有一个技巧, 那就是优先分析那些*靠后入栈的元素最先出栈*的序列——越往后入栈的元素越提前出来, 只能说明这前面的元素都已经入栈了, 并且此前并没有发生过出栈操作.

例 3.5.2. 假设有五个整数以 $1, 2, 3, 4, 5$ 的顺序入栈, 允许边进边出, 每次出栈记录出栈元素, 出栈序列为 $3, 5, 4, 2, 1$, 那么栈大小至少为?

A. 2 B. 3 C. 4 D. 5

评注. 能够满足题目要求的最小栈大小等于该过程中栈中元素的最大个数. 我们不妨分析一下, 首先入栈:

$$(1, 2, 3),$$

观察到出栈序列的第一个数是 3, 于是此时发生出栈操作,

$$(1, 2),$$

入栈,

$$(1, 2, 4, 5),$$

依次出栈直到栈空, 出栈序列恰为 $3, 5, 4, 2, 1$, 因此该过程是符合题目要求的操作过程. 考虑到出栈不可能增加栈中元素的个数, 因此该过程中栈内最多只有 4 个元素, 因此本题选 C.

3.5.2 栈的实现方式

数组和链表是基础的数据结构, 许多数据结构的实现都依赖于数组和链表. 栈 ADT 可以使用数组和链表来实现, 使用数组实现的栈称为顺序栈, 使用链表实现的称为链式栈.

用数组实现的顺序栈见程序清单 3.5.1. 我们首先定义了用于实现顺序栈的 `ArrayStack` 类, 然后声明了若干标识栈的状态的变量: 首先是存放栈中元素的数组 `items`, 其次是栈顶的下标 `top`, 以及栈的大小 `capacity`.

由于这段 Java 代码并不设计任何高级语法, 并且注释相对详细, 因此我们就不在正文中专门解释了.

程序清单 3.5.1. `ArrayStack.java`

```java
package cn.edu.nwpu.software.strik0r.data_structures.linear_structure.stack;
/**
 * {@code ArrayStack} 类是使用数组实现的栈 ADT.
 * @author Strik0r */
public class ArrayStack {
    private String[] items; // 存放栈中元素的数组
    private int top;        // 栈顶的下标
```

```
8       private int capacity;   // 栈的大小
9       /**
10       * 构造函数, 申请一个大小为 {@code size} 的数组空间.
11       * @param size 将要申请的数组空间的大小.*/
12      public ArrayStack(int size) {
13          this.items = new String[size]; // 申请一个大小为 capacity 的数组空间
14          this.top = -1;
15          this.capacity = size;
16      }
17      /**
18       * 入栈操作, 将元素 {@code item} 压入栈中.
19       * @param item 将要入栈的元素.
20       * @return 如果入栈成功则返回 {@code true}, 否则 (栈已满时) 返回 {@code
             false}.*/
21      public boolean push(String item) {
22          if (top+1 == capacity) return false; // 栈已满, 入栈失败
23          items[++top] = item; // 将新元素存储到栈中, 并令栈中元素的计数自增
24          return true;
25      }
26      /**
27       * 出栈操作, 返回栈顶元素.
28       * @return 如果为空栈则返回 {@code null}, 否则返回栈顶元素.*/
29      public String pop() {
30          if (top == -1) return null; // 如果是空栈则返回 null
31          String tmp = items[top];    // 将要返回的元素
32          top--;                      // 栈中元素的个数 -1.
33          return tmp;
34      }
35      /**
36       * 在不移除栈顶元素的前提下返回栈顶元素.
37       * @return 栈顶元素, 如果为空栈则返回 {@code null}.*/
38      public String getTop() {
39          if (top == -1) return null; // 如果是空栈则返回 null
40          return items[top];
41      }
42  }
```

参考文献

[1] [美] Mark Allen Weiss. 数据结构与算法分析——C 语言描述 (原书第 2 版) (典藏版) / Data Structures and Algorithm Analysis in C, Second Edition [M]. 冯舜玺译. 北京: 机械工业出版社, 2019.

[2] [美] Michael Goodrich, [美] Roberto Tamassia, [美] Michael H. Goldwasser. 数据结构与算法: Python 语言实现 / Data Structures and Algorithms in Python [M]. 张晓等译. 北京: 机械工业出版社, 2018.

[3] [美] Thomas H. Cormen, [美] Charles E. Leiserson, [美] Ronald L. Rivest, [美] Clifford Stein. 算法导论 (原书第 3 版) / Introduction to Algorithms, Third Edition [M]. 殷建平等译. 北京: 机械工业出版社, 2013.

[4] 耿国华等. 数据结构: 用 C 语言描述 (第 2 版) [M]. 北京: 高等教育出版社, 2015.

[5] 王道论坛组. 2024 年数据结构考研复习指导 [M]. 北京: 电子工业出版社, 2022.

[6] 严蔚敏, 李冬梅, 吴伟民. 数据结构: C 语言版: 双色版: 第 2 版 [M]. 北京: 人民邮电出版社, 2022.

[7] 李冬梅, 田紫微. 数据结构习题解析与实验指导: 第 2 版 [M]. 北京: 人民邮电出版社, 2022.

公诚勇毅　永矢毋忘
中华灿烂　工大无疆

2023 年　西北工业大学　中国西安