拟态语言技术手册

基础册

之恪提案

2025 年 11 月 18 日

目录

1 计算机的存储

1.1 进制

【本节目标】

- **问题驱动**：理解我们为何需要不同的计数规则。
- **掌握概念**：掌握二进制、八进制、十进制、十六进制的基本表示方法。
- **理解规律**：总结出适用于任意进制 n($n \geq 2$) 的普遍规律。
- **实践目标**：能够在小范围内进行不同进制数的识别和简单转换。

开头：我们要解决什么问题？

想象一下，你有一个巨大的仓库，里面用来堆放完全一样的箱子。为了方便管理，你会怎么记录箱子的数量呢？

- **方法 A**：每收到一个箱子，就在墙上画一道竖线 |。这样，5 个箱子就是 |||||，127 个箱子就是 127 道密密麻麻的线。这种方法非常原始，记录和清点大数字时极其困难。
- **方法 B**：你规定，每 10 个小箱子可以打包成一个“中箱”。每 10 个“中箱”又可以打包成一个“大箱”，那么，当你看到“1 个大箱、2 个中箱、7 个小箱”时，你就能立刻知道总数是 $1 \times 100 + 2 \times 10 + 7 = 127$ 个箱子。这种方法是不是清晰多了？

这个“打包规则”，就是进制。我们日常生活中习惯的“逢十进一”就是十进制。但计算机的世界和某些特定领域，使用不同的“打包规则”会更高效。本章，我们就来学习这些不同的“数字语言”。

1.1.1 十进制

十进制是我们每天都会用到的计数系统。

- **核心规则：逢十进一**。一共有 10 个基本符号：$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$。
- **比喻**：就像我们上面的仓库例子，是“按十打包”。
- **示例**：数字 305。
  - 它表示的是：3 个 $100(10^2) + 0$ 个 $10(10^1) + 5$ 个 $1(10^0)$。
  - 这里的 $100, 10, 1$ 叫做不同位数的“权重”。个位的权重是 $1(10^0)$，十位是 $10(10^1)$，百位是 $100(10^2)$，以此类推。

练习 1：数字 2024 在十进制中，代表多少？(提示：$2 \times 1000 + 0 \times 100 + 2 \times 10 + 4 \times 1$)

1.1.2 二进制

计算机的核心是电路，电路最容易表示两种状态：通电 (1) 和断电 (0)。因此，二进制是计算机世界的基石。

- **核心规则：逢二进一**。只有两个基本符号：0 和 1。
- **比喻**：不再是“按十打包”，而是“按二打包”。比如记录鸡蛋，每 2 个装一小盒，每 2 小盒装一中盒，每 2 中盒装一大盒。
- **示例**：二进制数 $(1101)_2$（我们使用 $(\text{数值})_{\text{n}}$ 表示括号内的数值由 n 进制表示，单位数有时省略括号，十进制默认不使用括号和下标）是多少?。
  - 我们同样看权重，但这里的权重是 2 的幂次方（从右向左，从 0 次方开始）。
  - 所以 $(1101)_2$ 表示：1 个 $8(2^3)$+1 个 $4(2^2)$+0 个 $2(2^1)$+1 个 $1(2^0) = 8+4+0+1 = 13$。
  - 所以，二进制里的 $(1101)_2$ 相当于我们日常说的 13。

练习 2：二进制数 $(1010)_2$ 相当于十进制的多少?

1.1.3 八进制与十六进制

直接看一长串的 0 和 1（比如 $(101010111100)_2$）非常容易眼花。于是人们发明了八进制和十六进制，作为二进制的“缩写形式”。

八进制

- **核心规则：逢八进一**。有 8 个基本符号：$0, 1, 2, 3, 4, 5, 6, 7$。
- **为何是“缩写”？**因为 3 位二进制数正好可以表示 0 到 7（$(000)_2$ 到 $(111)_2$），正好对应一位八进制数。所以，将二进制数从右向左每 3 位分一组，就能轻松转换成八进制。
- **示例：**二进制 $(101\ 110)_2$（注意分组）
  - $(101)_2 = 5$（八进制）
  - $(110)_2 = 6$（八进制）
  - 所以，$(101110)_2 = (56)_8$

十六进制

- **核心规则：逢十六进一**。有 16 个基本符号：0-9 以及 $A(10), B(11), C(12), D(13), E(14), F(15)$。

- **为何是“缩写”？**因为 4 位二进制数正好可以表示 0 到 15（$(0000)_2$ 到 $(1111)_2$），正好对应一位十六进制数。这是更常用的缩写方式。
- **示例 1：**二进制 $(1010\ 1100)_2$（注意分组）
  - $(1010)_2 = 10 = A$（十六进制）
  - $(1100)_2 = 12 = C$（十六进制）
  - 所以，$(10101100)_2 = (AC)_{16}$
- **示例 2：**我们经常在颜色代码、内存地址中看到十六进制，如 $\#FF5733$（一种橙色）。

练习 3：将二进制数 $(11100101)_2$ 转换成十六进制。（提示：先分成 $(1110)_2$ 和 $(0101)_2$ 两组）

1.1.4 普遍规律：对于任意正整数 n 的 n 进制

现在，我们来总结一下所有进制的共同规律。对于任意大于 1 的正整数 n（称为“基数”），n 进制都有以下特性：

1. **基本符号：**有 n 个基本符号，从 0 到 $(\mathrm{n}-1)$。比如二进制基数是 2，符号是 $0, 1$；十进制基数是 10，符号是 0-9。
2. **位权原理：**一个数字的大小，等于每位上的数字乘以该位的“权重”后求和。
   - **权重是：** $\mathrm{n}^{(\text{位置索引})}$。位置索引从右向左，从 0 开始。
   - **通用公式：**一个 n 进制数 $(\ldots CBA)_{\mathrm{n}}$（其中 A 是个位，B 是高位，C 是更高位...）对应的十进制值为：$\ldots + C \times \mathrm{n}^2 + B \times \mathrm{n}^1 + A \times \mathrm{n}^0$
3. **进位规则：** **逢 n 进一**。

举例验证：我们用一个不常见的**五进制**来检验这个规律。

- **五进制规则：**逢五进一，符号是 $0, 1, 2, 3, 4$。
- **问题：**五进制数 $(324)_5$ 是多少（十进制）？
- **计算：**
  - 个位（n^0 位）是 4，权重是 $5^0 = 1$，值为 $4 \times 1 = 4$
  - 高位（n^1 位）是 2，权重是 $5^1 = 5$，值为 $2 \times 5 = 10$
  - 更高位（n^2 位）是 3，权重是 $5^2 = 25$，值为 $3 \times 25 = 75$
  - 总和：$4 + 10 + 75 = 89$（十进制）

- 所以，五进制的 $(324)_5$ 等于十进制的 89。你可以理解为：3 个大包（每包 25 个）、2 个中包（每包 5 个）和 4 个散装。

终极挑战：如果有一个**七进制**数，它的基本符号是什么？数字 $(15)_7$ 是一个有效的七进制数字吗？为什么？

1.1.5 本节总结

进制	基数	基本符号	主要应用场景	特点
十进制	10	0-9	日常生活	人类最熟悉的计数法
二进制	2	0,1	计算机底层逻辑电路	便于物理实现（通电/断电），但书写冗长
八进制	8	0-7	计算机领域（历史遗留系统）	二进制的 3 位缩写（1 位八进制 =3 位二进制）
十六进制	16	0-9, A-F	计算机领域（内存地址、颜色代码）	二进制的 4 位缩写（1 位十六进制 =4 位二进制），非常常用

表 1: 常用进制对比表

理解进制的关键在于理解“位权”和“进位规则”。无论进制如何变化，其表示数值的本质都是一样的。在下一章，我们将深入学习不同进制之间如何相互转换。

练习答案与反馈

- **练习 1 答案：** $2024 = 2\times1000+0\times100+2\times10+4\times1 = 2000+0+20+4 = 2024$。你做对了吗？这看起来像是废话，但请务必理解其背后的“位权”思想。
- **练习 2 答案：** $(1010)_2 = 1\times8+0\times4+1\times2+0\times1 = 8+0+2+0 = 10$。
- **练习 3 答案：** $(11100101)_2$ 分组为 $(1110)_2$ 和 $(0101)_2$。
  - $(1110)_2 = 14 = E$
  - $(0101)_2 = 5 = 5$
  - 所以答案是 $(E5)_{16}$。

- **终极挑战答案：**七进制的基本符号是 $0, 1, 2, 3, 4, 5, 6$。数字 $(15)_7$ 是**有效**的，因为它的每一位数字 (1 和 5) 都小于基数 7。它等于 $1 \times 7^1 + 5 \times 7^0 = 7 + 5 = 12$。请注意，判断一个数字串在某进制下是否合法，要看其**每一位数字是否都小于该进制的基数**。例如，$(18)_7$ 就是非法的，因为数字 8 不小于 7。

1.2 进制之间的转换

【本节目标】

- 掌握将二进制、八进制、十六进制数转换为十进制的方法（按权展开法）。
- 掌握将十进制数转换为二进制、八进制、十六进制的方法（除基取余法）。
- 熟练运用二进制作为桥梁，进行八进制与十六进制之间的快速转换。
- 理解小数部分的进制转换原理。

现在我们已经理解了不同进制的表示方法，接下来学习如何在不同进制之间进行转换。这是编程和计算机科学中的一项基本技能。

1.2.1 其他进制转换为十进制：按权展开法

这是最简单的一种转换，我们之前已经接触过。核心思想是：将每个数位上的数字乘以该位的权重（基数的幂次），然后求和。

通用公式：对于一个 k 位的 n 进制数 $(a_{k-1}a_{k-2}\ldots a_1a_0)_n$，其十进制值为：

$$a_{k-1} \times n^{k-1} + a_{k-2} \times n^{k-2} + \ldots + a_1 \times n^1 + a_0 \times n^0$$

示例 1：将二进制数 $(10110)_2$ 转换为十进制。

解：从右向左，权重依次是 $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$。

$$\begin{aligned}(10110)_2 &= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 \\ &= 16 + 0 + 4 + 2 + 0 = 22\end{aligned}$$

所以，$(10110)_2 = (22)_{10}$。

示例 2：将十六进制数 $(2A7)_{16}$ 转换为十进制。

解：注意 $A = 10$，权重依次是 $16^0 = 1$, $16^1 = 16$, $16^2 = 256$。

$$\begin{aligned}(2A7)_{16} &= 2 \times 16^2 + A \times 16^1 + 7 \times 16^0 \\ &= 2 \times 256 + 10 \times 16 + 7 \times 1 \\ &= 512 + 160 + 7 = 679\end{aligned}$$

所以，$(2A7)_{16} = (679)_{10}$。

练习 1：将八进制数 $(47)_8$ 转换为十进制。

1.2.2 十进制转换为其他进制：除基取余法

将十进制数转换为 n 进制数，需要不断用 n 去除十进制数，并记录余数，直到商为 0 为止。最后将余数从后向前（从最后一个余数到第一个余数）排列。

示例 3：将十进制数 29 转换为二进制。

解：用 2 连续除 29，记录余数：

$$
\begin{aligned}
29 \div 2 = 14 \quad &\text{余} \quad 1 \quad \text{（最低位）} \\
14 \div 2 = 7 \quad &\text{余} \quad 0 \\
7 \div 2 = 3 \quad &\text{余} \quad 1 \\
3 \div 2 = 1 \quad &\text{余} \quad 1 \\
1 \div 2 = 0 \quad &\text{余} \quad 1 \quad \text{（最高位）}
\end{aligned}
$$

将余数从下往上排列：11101，所以 $(29)_{10} = (11101)_2$。

示例 4：将十进制数 255 转换为十六进制。

解：用 16 连续除 255，记录余数：

$$
\begin{aligned}
255 \div 16 = 15 \quad &\text{余} \quad 15 \quad (F) \\
15 \div 16 = 0 \quad &\text{余} \quad 15 \quad (F)
\end{aligned}
$$

将余数从下往上排列：FF，所以 $(255)_{10} = (FF)_{16}$。

练习 2：将十进制数 100 转换为二进制和十六进制。

1.2.3 二进制与八进制、十六进制的快捷转换

由于 $8 = 2^3$，$16 = 2^4$，二进制与八进制、十六进制之间可以通过简单的分组法快速转换。

二进制转八进制：将二进制数从右向左每 3 位一组（不足 3 位在左边补 0），然后将每组转换为对应的八进制数。

示例 5：将 $(11010111)_2$ 转换为八进制。

解：分组：011|010|111（最左边补一个 0）

$(011)_2 = 3_8$, $(010)_2 = 2_8$, $(111)_2 = 7_8$

所以，$(11010111)_2 = (327)_8$

八进制转二进制：将八进制数的每一位展开为 3 位二进制数。

示例 6：将 $(514)_8$ 转换为二进制。

解： $5_8 = (101)_2$, $1_8 = (001)_2$, $4_8 = (100)_2$

所以，$(514)_8 = (101001100)_2$

二进制转十六进制：将二进制数从右向左每 4 位一组（不足 4 位在左边补 0），然后将每组转换为对应的十六进制数。

示例 7：将 $(110110101)_2$ 转换为十六进制。
解：分组：0001|1011|0101（最左边补三个 0）
$(0001)_2 = 1_{16}$, $(1011)_2 = B_{16}$, $(0101)_2 = 5_{16}$
所以，$(110110101)_2 = (1B5)_{16}$
十六进制转二进制：将十六进制数的每一位展开为 4 位二进制数。
示例 8：将 $(E3A)_{16}$ 转换为二进制。
解： $E_{16} = (14)_{10} = (1110)_2$, $3_{16} = (0011)_2$, $A_{16} = (10)_{10} = (1010)_2$
所以，$(E3A)_{16} = (111000111010)_2$
练习 3：将二进制数 $(1011101)_2$ 转换为八进制和十六进制。

1.2.4 八进制与十六进制之间的转换

八进制与十六进制之间没有直接的快捷转换方法，但可以通过二进制作为桥梁。
转换路径：八进制 ⇄ 二进制 ⇄ 十六进制
示例 9：将 $(347)_8$ 转换为十六进制。
解：

1. 八进制转二进制：$3_8 = (011)_2$, $4_8 = (100)_2$, $7_8 = (111)_2$，得到 $(011100111)_2$
2. 二进制转十六进制：分组 0000|1110|0111，得到 $(0E7)_{16}$，即 $(E7)_{16}$

所以，$(347)_8 = (E7)_{16}$
示例 10：将 $(A9F)_{16}$ 转换为八进制。
解：

1. 十六进制转二进制：$A_{16} = (1010)_2$, $9_{16} = (1001)_2$, $F_{16} = (1111)_2$，得到 $(101010011111)_2$
2. 二进制转八进制：分组 101|010|011|111，得到 $(5237)_8$

所以，$(A9F)_{16} = (5237)_8$
练习 4：将八进制数 $(65)_8$ 转换为十六进制。

1.2.5 小数部分的进制转换

小数的进制转换稍微复杂，但原理相似。
十进制小数转 `n` 进制：用乘基取整法。不断用 `n` 乘小数部分，取整数部分，直到小数部分为 0 或达到所需精度。
示例 11：将十进制小数 0.625 转换为二进制。

解：

$$\begin{aligned} 0.625 \times 2 = 1.25 \quad &\text{取整} \quad 1 \ \text{（最高位）} \\ 0.25 \times 2 = 0.5 \quad &\text{取整} \quad 0 \\ 0.5 \times 2 = 1.0 \quad &\text{取整} \quad 1 \ \text{（最低位）} \end{aligned}$$

将整数部分从上往下排列：0.101，所以 $(0.625)_{10} = (0.101)_2$。

n **进制小数转十进制：**同样使用按权展开法，但权重是负幂次。

示例 12：将二进制小数 $(0.101)_2$ 转换为十进制。

解：

$$\begin{aligned} (0.101)_2 &= 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 1 \times 0.5 + 0 \times 0.25 + 1 \times 0.125 \\ &= 0.5 + 0 + 0.125 = 0.625 \end{aligned}$$

本节总结

- **转十进制：**按权展开，求和。
- **十进制转：**整数部分除基取余，小数部分乘基取整。
- **二、八、十六互换：**利用分组法（3 位一组或 4 位一组）。
- **八、十六互换：**以二进制为桥梁。

进制转换是计算机科学的基础，需要多加练习才能熟练掌握。

练习答案

- **练习 1（八转十）：** $(47)_8 = 4 \times 8^1 + 7 \times 8^0 = 32 + 7 = 39$
- **练习 2（十转二、十六）：**
  - 二进制：$100 \div 2 = 50$ 余 0; $50 \div 2 = 25$ 余 1; $25 \div 2 = 12$ 余 1; $12 \div 2 = 6$ 余 0; $6 \div 2 = 3$ 余 0; $3 \div 2 = 1$ 余 1; $1 \div 2 = 0$ 余 1。得 $(1100100)_2$
  - 十六进制：$100 \div 16 = 6$ 余 4; $6 \div 16 = 0$ 余 6。得 $(64)_{16}$
- **练习 3（二转八、十六）：**
  - 八进制：$(1011101)_2$ 分组为 001|011|101，得 $(135)_8$
  - 十六进制：分组为 0101|1101，得 $(5D)_{16}$
- **练习 4（八转十六）：** $(65)_8 = (110101)_2$，分组为 0011|0101，得 $(35)_{16}$

1.3 比特、字节与字长

【本节目标】

- **问题驱动**：理解计算机如何用最基本的结构单元来存储和表示庞杂的信息。
- **掌握概念**：掌握比特（Bit）、字节（Byte）、字长（Word Size）的核心定义。
- **理解差异**：理解不同进制的计算机系统中，这些基础单元的含义有何不同。
- **实践目标**：能够计算一个给定长度的字节或字所能表示的状态总数。

开头：我们要解决什么问题？

想象一下，你要设计一个巨大的自动化仓库。这个仓库的核心是什么呢?

- 首先，你需要设计最小的储物单元，比如一个可以存放一种特定零件的“格子”。这个“格子”能有几种状态（比如空、满、或存放不同零件），决定了它最基本的表达能力。
- 然后，为了方便管理，你会把一定数量的小格子固定组合成一个“标准箱”。所有库存记录都以“标准箱”为单位，而不是零散的“格子”。
- 最后，仓库的搬运机器人一次能抓取、移动多少个“标准箱”，决定了仓库的运作效率。

计算机的存储设计遵循着完全相同的逻辑。本节，我们就来认识计算机世界的“最小格子”、“标准箱”和“搬运能力”——即比特、字节和字长。

1.3.1 比特

比特（Binary Digit 的缩写）是计算机存储和处理信息的最小单位。

- **核心定义**：一个比特代表一个最小的、有两种可能状态的信息单元。
- **在二进制计算机中**：这是我们最熟悉的场景。一个比特就像一个开关，只有两种状态：**0**（断电/假/否）和 **1**（通电/真/是）。这是物理上最容易实现的状态。
- **在 n 进制计算机中（理论拓展）**：如果一个计算机系统是基于 n 进制的（$n \geq 2$），那么它的一个“比特”就不仅仅有两种状态，而是有 n 种可能的状态。例如，在三进制计算机中，一个“三态比特”（Trit）可以有 **0, 1, 2** 三种状态；在量子计算中，一个量子比特（Qubit）的状态则更为复杂。

单个比特能表达的信息非常有限。为了表示更复杂的信息（比如数字、字母、颜色），我们必须将多个比特组合起来使用。

计算机类型	比特含义	状态数
二进制计算机	常规比特（Bit）	2 种（0/1）
n 进制计算机	广义比特	n 种（0 到 n − 1）

表 2: 不同进制下的“比特”

1.3.2 字节

由于单独操作每一个比特效率极低，计算机科学家定义了一个更常用的基本单位——字节。

- **核心定义**：字节是由**一连串连续**的比特组成的一个固定长度的组合。它是计算机信息处理的基本单位。
- **历史与标准**：早期，字节的长度并不统一。但随着 IBM System/360 等主流计算机架构的推广，**1 字节 = 8 比特**成为了事实上的国际标准，并沿用至今。本书后续讨论如无特别说明，均指 8 比特的字节。
- **状态数的计算**：一个长度为 m（通常为 8）的字节，其所能表示的不同状态总数由以下公式决定：

  $$\text{状态总数} = n^m$$

  其中，n 是计算机的**进制**（即一个比特的状态数），m 是字节包含的**比特数**。
- **举例**：
  - 在二**进制**计算机中，一个 8 比特的字节能表示 $2^8 = 256$ 种不同的状态（从 $(0000\ 0000)_2$ 到 $(1111\ 1111)_2$）。
  - 如果存在一个**四进制**计算机（$n = 4$），那么它的一个 8 “比特”的字节，就能表示 $4^8 = 65536$ 种状态，其信息密度远高于二进制系统。

字节就像仓库里的“标准货箱”。无论里面的“小格子”（比特）如何，我们总是以“箱”为单位进行搬运和清点，这大大提高了管理效率。

练习 1：一个理论上的三进制计算机系统，定义了其“字节”由 6 个“三态比特”组成。请问这样的一个“字节”最多能表示多少种不同的状态?

1.3.3 字长

字长是衡量计算机性能的一个关键指标，它决定了计算机一次操作所能处理的数据量。

- **核心定义**：字长指的是计算机**一次性**能够处理、存储或传输的二进制信息的最大位数（在 n 进制计算机中，则是 n 进制信息的最大“比特”数）。它通常等于 CPU 中通用寄存器的宽度。

- **比喻**：字长就像仓库里搬运机器人的“抓取能力”。如果机器人一次能抓起 4 个“标准箱”（字节），那么我们就说这台“仓库机器人”（CPU）的字长是 4 字节。
- **与字节的关系**：字长通常是字节的整数倍。我们常用“位”或“字节”来描述它。
  - 早期的微型计算机（如 Intel 8088）字长为 16 位，即 2 字节。
  - 现代个人计算机和服务器大多是 64 位，即 8 字节。
- **重要性**：更大的字长通常意味着更强的性能，因为 CPU 一次能处理更多数据、访问更大内存（内存地址空间受字长限制，如 32 位系统最大寻址空间为 2^{32} 字节，约 4GB）。字长也决定了计算机是“32 位系统”还是“64 位系统”。

概念	比喻	核心定义	计算公式/示例
比特	最小储物格	信息的最小单元，有 n 种状态	二进制下：0 或 1
字节	标准货箱	由 m 个连续比特组成的管理单位	状态数 $= n^m$
字长	机器人的抓取能力	计算机一次操作能处理的最大数据量	64 位系统：8 字节

表 3: 比特、字节与字长对比总结

本节总结

从比特到字节再到字长，计算机的存储体系是一个自底向上、层层抽象的经典设计。

- **比特**是信息的“原子”，其状态数由计算机的进制 n 决定。
- **字节**是为了管理效率而定义的“标准箱”，由 m 个连续的比特构成，其表达能力（状态数）由幂函数 n^m 决定。
- **字长**是计算机性能的标尺，代表了 CPU 的“搬运带宽”，决定了计算机一次性能处理多少“标准箱”。

理解这三者的关系，是理解计算机如何工作的基石。

练习答案与反馈

- **练习 1 答案：**已知计算机为三进制（$n = 3$），“字节”长度为 6 个“三态比特”（$m = 6$）。根据状态数计算公式 $n^m = 3^6 = 729$。因此，这样的一个“字节”最多能表示 **729** 种不同的状态。

1.4 数胞

【本节目标】

- **引入概念**：理解“数胞”（Numerical Cell, NC）作为计算机存储空间统一抽象模型的核心思想。
- **掌握表示**：掌握数胞的二元组表示法 (n, m) 及其各种简写形式。
- **建立联系**：能够将具体的存储单元（如比特、字节）抽象为对应的数胞表示。
- **应用公式**：运用状态数公式 n^m 理解数胞的描述能力。

开头：我们要解决什么问题？

在前面的章节中，我们学习了比特、字节等存储单元。我们发现，无论是哪种进制的计算机，其存储单元都可以被概括为两个核心属性：1. 该单元所能呈现的**所有可能状态的数量**（n）。2. 该单元在某一时刻所处的**具体状态值**（m）。

为了用一种统一、简洁的数学语言来描述计算机中所有的存储空间，我们引入了“数胞”（Numerical Cell）这一抽象概念。它将帮助我们跳出二进制、十进制等具体进制的限制，从更高层面理解存储的本质。

1.4.1 数胞的定义与表示

一个数胞（NC）可以由一个二元组完整定义：

$$NC \equiv (n,\ m)$$

其中：

- n：表示该数胞的**状态数**（Number of States）。它一般是一个大于 1 的自然数，决定了该存储单元能表示多少种不同的情况。它本质上对应了前文所述的“进制”概念。
- m：表示该数胞的**值**（Value）。它是该存储单元在当前时刻的具体状态，是一个自然数，且必须满足 $0 \le m < n$。

1.4.2 数胞的简写符号

为了书写方便，我们定义了一套简写符号：

- NC：数胞（Numerical Cell）的通用缩写。
- NC_n：表示所有状态数为 n 的同一类型数胞的集合。它强调了存储单元的“类型”或“容量”。

- $NC_{\mathtt{n}}(\mathtt{m})$：表示一个状态数为 n，且当前值为 m 的特定数胞。这是最完整的表示形式，同时包含了存储单元的属性和当前状态。

示例：

- $NC_2(1)$：表示一个状态数为 2（即二进制）、当前值为 1 的数胞。这其实就是我们熟悉的一个比特（Bit），其值为 1。
- $NC_{256}(65)$：表示一个状态数为 256、当前值为 65 的数胞。这恰好可以表示一个值为 65 的字节（Byte），因为一个 8 位二进制字节有 $2^8 = 256$ 种状态。
- $NC_{10}(7)$：表示一个状态数为 10（即十进制）、当前值为 7 的数胞。这可以想象为一个十进制的基本存储单元。

1.4.3 数胞的抽象意义

“计算机中所有的存储空间都可以抽象为一个数胞。”这句话的含义是：无论存储空间的实际物理实现如何，我们都可以用数胞的二元组模型 (n, m) 来刻画它。

- **若一个存储单位可以表示 N 个状态**：这意味着它的“类型”是 $NC_{\mathtt{N}}$。
- **且其当前值为 M**：这意味着它的当前状态是 M，且 $0 \leq \mathtt{M} < \mathtt{N}$。
- **那么它可以被数胞 $NC_{\mathtt{N}}(\mathtt{M})$ 表示**：这个数胞的表示完全捕获了该存储单元的核心信息。

数胞的抽象威力在于其通用性。它不关心底层是二进制电路、三进制器件还是其他任何物理实现，它只关心逻辑上的状态数量和一个具体的状态值。这使得我们可以在统一的框架下讨论不同架构的计算机存储问题。

1.4.4 状态数公式的再现

一个由 k 个 $NC_{\mathtt{n}}$ 类型的数胞连续组成的存储空间，其总状态数正是我们熟悉的公式：

$$\text{总状态数} = \mathtt{n}^{\mathtt{k}}$$

这个公式解释了为什么一个由 8 个 NC_2（比特）组成的字节 ($\mathtt{n} = 2, \mathtt{k} = 8$) 有 256 种状态 ($2^8$)，也解释了一个由 2 个 NC_{10}（十进制单元）组成的存储空间有 100 种状态 (10^2)。
练习 1：有一个存储单元，它由 3 个 NC_4（四进制数胞）连续构成。请问这个存储单元总共可以表示多少种不同的状态？请用数胞的形式 $NC_{\mathtt{N}}$ 表示这个组合后的存储单元类型。
练习 2：某台计算机的一个基本存储单元是 NC_{100}。请问这个单元的值可能是 95 吗？可能是 100 吗？为什么？

本节总结

数胞 ($NC_{\mathtt{n}}(\mathtt{m})$) 是一个高度抽象的概念，它统一地描述了计算机中的存储：

- **本质**：一个存储单元就是一个有 n 种状态、当前处于第 m 种状态的物理实体。
- **表示**：$NC_{\mathtt{n}}$ 表示类型，$NC_{\mathtt{n}}(\mathtt{m})$ 表示一个具体的实例。
- **约束**：值 m 必须满足 $0 \leq \mathtt{m} < \mathtt{n}$。
- **扩展**：k 个 $NC_{\mathtt{n}}$ 组合可形成一个更大的 $NC_{\mathtt{n}^{\mathtt{k}}}$ 类型的数胞。

引入数胞的概念，为我们后续讨论内存地址、数据表示等话题提供了强大而简洁的理论工具。

练习答案与反馈

- **练习 1 答案：** 总状态数 $= 4^3 = 64$。这个由 3 个四进制数胞组成的存储单元可以表示 64 种状态，因此它的类型可以表示为 NC_{64}。
- **练习 2 答案：** 该存储单元是 NC_{100} 类型，意味着它的值 m 必须满足 $0 \leq \mathtt{m} < 100$。因此，**95** 是可能的值（因为 $0 \leq 95 < 100$），而 **100** 是不可能的值（因为 $100 \not< 100$，违反了数胞的定义约束）。

1.5 地址

【本节目标】

- **问题驱动**：理解计算机如何在海量的存储空间中精确地找到每一个数据。
- **掌握概念**：掌握地址（Address）的核心定义，理解其与字节的关系。
- **理解规律**：理解地址的值所代表的物理意义——即第一个字节的位置。
- **实践目标**：能够根据地址计算其指向的存储空间范围。

开头：我们要解决什么问题？

想象一下，一个巨大的自动化仓库，里面排列着成千上万个一模一样的“标准货箱”（对应计算机中的**字节**）。现在，管理员需要取出存放在“第 2056 号货箱”里的零件，他应该怎么做？

- **方法 A**：从第一个货箱开始，1, 2, 3, ... 一直数到 2056 个。这种方法在货箱数量巨大时，效率极其低下，几乎不可能实现。
- **方法 B**：为每一个货箱分配一个独一无二的**编号**。管理员只需输入“2056”，搬运机器人就能直接移动到对应编号的货箱位置。这个独一无二的编号，就是**地址**。

计算机的中央处理器（CPU）就像那位管理员，它需要处理存储在各个“货箱”（字节）里的数据。为了快速定位，计算机也必须为整个存储空间中的每一个字节分配一个唯一的“编号”，这个编号就是**地址**。

1.5.1 地址的定义与性质

地址是用于标识存储空间中每个字节唯一位置的整数值。
它具备以下几个核心性质：

- **唯一性**：存储空间中的每一个字节都有一个独一无二的地址，就像每个家庭都有一个唯一的门牌号。
- **有序性**：地址通常是从 0 开始连续递增的整数。地址 N 代表的字节，紧挨着地址 $N-1$ 和地址 $N+1$ 的字节，它们物理上在存储空间中顺序排列。
- **字节为单元**：地址的**基本单位是字节**。我们常说“某个地址”，指的就是“某个字节所在的位置”。这是地址计算和寻址的基础。

1.5.2 地址的值与指向关系

这是理解地址的关键：**地址的值，直接表示该地址所指向的存储位置的起始点，即其第一个字节的位置**。
比喻：假设整个存储空间是一条笔直的长街，街上每栋房子的大小都完全一样（1 字节）。那么，地址就像是门牌号。

- 地址 0：代表整条街的起点，即 1 号房子的位置。
- 地址 1：代表从起点开始，往后数 1 个房子长度的位置，即 2 号房子的位置。
- 地址 N：代表从起点开始，往后数 N 个房子长度的位置。这个位置，就是第 $(N+1)$ 个房子的起始点。

因此，当我们说“访问地址 2024”时，其含义是：**从存储空间的起始点开始，向后偏移 2024 个字节，找到的那个字节**。

1.5.3 扩展：地址与多字节数据

大部分数据的大小都超过 1 个字节。那么如何存储它们呢？答案仍然是依靠地址。

当一个数据需要占用多个连续的字节时，计算机会分配一段连续的存储空间给它，并**用这段空间第一个字节的地址来代表整个数据的存储位置**。
示例：假设一个数据类型需要占用 4 个字节。

- 计算机决定将它存放在从地址 3000 开始的连续 4 个字节中。

- 那么，这个数据的“存放地址”就是 3000。
- 这个数据实际占用了地址为 3000（第 1 字节）, 3001（第 2 字节）, 3002（第 3 字节）, 3003（第 4 字节）的四个存储单元。

CPU 知道这种数据的长度是 4 字节。所以当它要读取这个数据时，它会从“起始地址”（3000）开始，连续读取 4 个字节的内容，并将其组合还原成完整的数据。
练习：一个数据需要占用 8 个字节，它被存储在从地址 2048 开始的一段空间里。请问这个数据占用了哪些地址的字节？它的结束地址是多少？

本节总结

- **地址**是存储空间的“导航坐标”，是每个字节的唯一标识符。
- 地址的**基本单位是字节**。
- **地址的值** N，表示从存储空间起点向后偏移 N 个字节后的位置，即**该地址所指向字节的起始点**。
- 对于多字节数据，用其**第一个字节的地址**来代表整个数据的存储位置。

理解地址是指向“第一个字节”的这一概念，是理解后续指针、数组、内存布局等更高级主题的基石。

练习答案与反馈

- **练习答案：**
  - 起始地址是 2048，占用 8 个字节。
  - 占用的地址范围是：2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055。
  - 结束地址是 $2048 + 8 - 1 = 2055$。（因为起始地址本身也占一个位置）
- **反馈：**你算对了吗？关键在于理解“从地址 `N` 开始，连续 `K` 个字节”的含义，其覆盖的地址范围是从 `N` 到 $\texttt{N} + \texttt{K} - 1$。

1.6 存储对齐

【本节目标】

- **问题驱动**：理解计算机为何不总是“见缝插针”地存放数据，而是有时会留下“空位”。
- **掌握概念**：掌握存储对齐（Data Alignment）的定义和判断方法。

- **理解差异**：理解对齐规则是因“机”制宜、因“境”而异的，并非一成不变。
- **实践目标**：能够判断给定地址是否满足指定的对齐要求，并能计算满足对齐要求的地址。

开头：我们要解决什么问题？

让我们再次回到巨型自动化仓库的比喻。现在，仓库里来了一批特殊的“超长货物”，每个货物需要占用连续的 4 个货箱（字节）。

- **方法 A（不对齐）**：搬运机器人可以把这个货物从任何位置开始存放，例如从第 1001 号货箱开始，占用 $1001, 1002, 1003, 1004$ 号箱。
- **方法 B（对齐）**：仓库规定，这类 4 箱长的货物必须从“4 的倍数”号货箱开始存放，例如 $1000, 1004, 1008$ 号等，而不能从 $1001, 1002, 1003$ 号开始。

方法 B 就是“存储对齐”。这条规定看似浪费了空间（比如 $1001, 1002, 1003$ 号箱暂时空置），但却能极大提高机器人的存取效率。计算机世界也是如此，为了提升访问速度，计算机会要求数据必须从特定的地址开始存储。

1.6.1 存储对齐的定义

存储对齐是计算机系统为了高效访问数据而采用的一种技术。它要求某些类型的数据的起始地址必须是指定大小的整数倍。

- **对齐大小（Alignment Size）**：一个自然数，记为 `a`，表示对齐的字节数。常见的有 1-字节对齐、4-字节对齐、8-字节对齐等。
- **对齐规则**：如果一个数据被要求必须是 `a` 字节对齐的，那么该数据在存储空间中的起始地址 `A` 必须能被 `a` 整除。即必须满足以下数学条件：

$$\texttt{A} \bmod \texttt{a} = 0$$

- **举例**：
  - 要求 4 字节对齐：则地址 A 必须是 4 的倍数，如 $0, 4, 8, 12, \ldots$。地址 1004 满足要求（$1004 \div 4 = 251$，余数为 0），地址 1006 则不满足（$1006 \div 4 = 251.5$，余数为 2）。
  - 1 字节对齐：等同于不对齐，因为任何整数 `A` 除以 1 的余数都是 0。数据可以存放在任何地址。

1.6.2 对齐规则的多样性

非常重要的一点是：**不存在一套通用的、放之四海而皆准的对齐规则**。具体的对齐要求取决于：

1. **硬件平台（因“机”制宜）**：不同的 CPU 架构有其偏好的对齐方式。
   - 例如，某些早期的处理器严格要求访问 4 字节整数必须 4 字节对齐，如果地址不对齐，则会引发硬件异常，导致程序错误。
   - 而现代的 CPU 虽然处理不对齐的访问时不会报错，但性能会有显著下降。因此，编译器在为其生成代码时，仍然会默认采用优化的对齐策略以保证性能。
2. **编程语言与编译器（因“境”而异）**：即使在同一种硬件上，不同的编程语言和编译器也可能有不同的默认对齐规则。
   - 在某些编程语言中，可以使用编译器指令来修改默认的对齐规则，从而在节省存储空间和牺牲访问速度之间做出权衡。
3. **人为规定**：在设计和定义自己的文件格式或网络传输协议时，开发者可以为了解析方便或兼容性，自行规定其中各个数据段的对齐方式。例如，可以规定所有数据块必须从 16 字节的边界开始。

因此，理解存储对齐的关键不在于记住一套固定的规则，而在于理解“为何对齐”（提升性能）和“如何判断”（地址是否是对齐值的整数倍）这两个核心思想，并意识到规则本身是灵活可变的。

练习 1：一个数据被要求进行 8 字节对齐。请问地址 2024 和地址 2026，哪一个满足对齐要求？请用计算说明。

练习 2：在某一个系统环境下，规定所有数据必须 4 字节对齐。现在有一个 1 字节长的数据（例如一个字符）和一个紧接着存放的 4 字节长的数据（例如一个整数）。

(a) 如果字符从地址 1000 开始存放，那么整数可以从哪个地址开始存放？

(b) 这样做是否可能造成整数存放的地址违反 4 字节对齐的规定？如果可能，如何避免？

本节总结

- **是什么**：存储对齐要求数据的起始地址 `A` 是其对齐大小 `a` 的整数倍（$A \bmod a = 0$）。
- **为什么**：为了提升 CPU 访问数据的效率，或在某些架构上保证程序正确性。
- **灵活性**：对齐规则不是绝对的，它因硬件平台、编程语言、编译器设置乃至人为规定而异。

练习答案与反馈

- **练习 1 答案：**
  - $2024 \div 8 = 253$，余数为 0。故地址 2024 满足 8 字节对齐。
  - $2026 \div 8 = 253.25$，余数为 2（因为 $2026 - 253 \times 8 = 2026 - 2024 = 2$）。故地址 2026 不满足 8 字节对齐。
  - 答案是：地址 2024 满足要求。
- **练习 2 答案：**

(a) 字符占用了地址 1000。下一个可用的地址是 1001。因此，整数可以从地址 1001 开始存放。

(b) **可能造成违反规定**。地址 1001 ($1001 \bmod 4 = 1$) 不是 4 的倍数，不满足 4 字节对齐。为了避免这种情况，编译器或程序员通常会在字符数据之后插入 3 个字节的“填充”（Padding），使得整数可以从地址 1004 开始存放，从而满足对齐要求。

2 计算机的运算

2.1 伪代码

由于此技术手册不依赖特定的编程语言进行教学，因此使用伪代码的方式来模拟程序。且此技术手册的伪代码使用独有的规范和语法。之后每个章节，如果需要使用特定的功能，则会在章节中给出伪代码，以及伪代码的规范。

2.1.1 数胞变量

我们使用：

$$\texttt{var} := NC_{\texttt{n}}(\texttt{m})$$

表示变量 var 被定义为一个状态数为 n，值为 m 的数胞。

如果之后需要改变 var 的值，允许使用：

$$\texttt{var} := \texttt{a}$$

表示在这行伪代码执行后，var 的值变为 a $\bmod$ n。

如果你有两个数胞变量 $\texttt{x} = NC_{\texttt{n}_1}(\texttt{m}_1)$ 和 $\texttt{y} = NC_{\texttt{n}_2}(\texttt{m}_2)$，你可以使用：

$$\texttt{x} := \texttt{y}$$

使执行这行伪代码之后 $\texttt{x} = NC_{\texttt{n}_1}(\texttt{m}_2 \bmod \texttt{n}_1)$。

也可以使用：

$$\texttt{x} := \texttt{num}$$

其中，num 为一个自然数，使执行这行伪代码之后 $\texttt{x} = NC_{\texttt{n}_1}(\texttt{num} \bmod \texttt{n}_1)$。

如果我们需要表示 var 的状态数为 n，可以使用：

$$\texttt{var} \in NC_{\texttt{n}}$$

表示。

2.1.2 全局定义

同时我们定义几个全局变量，在之后的所有章节中都可能被使用：

- BASE 为计算机所使用的进制。
- BYTELENGTH 为计算机的一个字节所使用的位数。
- WORDSIZE 为与计算机字长等价的字节数。

接着我们定义一个特殊的值 WORDSIZEVALUE = $\mathrm{BASE}^{\mathrm{BYTELENGTH}\times\mathrm{WORDSIZE}}$，使用此值作为状态数的数胞使用的存储空间刚好是计算机字长所占的存储空间。

我们定义一些全局函数，在之后所有章节中可能被使用：

- 设 v 为一个变量，typeof(v) 为 v 所属的数据类型，允许使用 T := typeof(v) 以让 T 存储 v 的类型信息。
- 设 v 为一个实际占用存储空间的变量，sizeof(v) 为 v 在存储空间中所占的字节数。设 T 为一个占用存储空间大小明确的数据类型，sizeof(T) 为 T 占用的存储空间大小的字节数。

如果需要在伪代码中插入说明性质的内容，则可以使用 “■任意内容”，任意内容可以是任意的文本，不参与也不影响执行。

2.2 代数运算

【本节目标】

- **问题驱动**：理解计算机中的运算为何会产生” 溢出” 现象，以及如何处理超出存储范围的结果。
- **掌握概念**：掌握数胞运算的基本原理和模运算的核心思想。
- **理解规律**：理解加法、减法、乘法、除法在数胞约束下的特殊行为。
- **实践目标**：能够计算给定数胞上的基本运算结果，理解位数扩展现象。

2.2.1 数胞的运算与循环计数

开头：我们要解决什么问题？

想象一下，你有一个只能显示 0 到 99 的两位数字计数器（这相当于一个状态数 n = 100 的数胞）。

- **情景 1**：当前显示 97，你加上 2，计数器变成 99，这是正常结果。
- **情景 2**：当前显示 97，你加上 5，理想结果是 102，但计数器只能显示两位数字，于是它从 99 之后” 归零”，显示为 02（实际上是 102 除以 100 的余数）。

这种” 归零重来” 的现象就是计算机运算中的**溢出**。由于计算机的存储空间（数胞）是有限的，任何运算结果都必须被约束在数胞能够表示的范围内。本节我们将学习数胞上的四种基本运算如何在这种约束下工作。

2.2.2 模运算：循环计数的数学基础

在计算机中，任何存储于数胞中的计算，其结果都受到数胞状态数 $\mathtt{n}$ 的严格限制。这种限制导致了计算机的运算遵循一种”循环计数”的法则，这在数学上称为**模运算**。

设一个数胞 $\mathtt{C}$ 的状态数为 $\mathtt{n}$，其当前值为 $\mathtt{m}_1$。我们对它进行某种运算，并输入另一个操作数 $\mathtt{m}_2$。该运算在常规数学下的理想结果为 $\mathtt{m}_{\mathtt{ideal}} = \mathtt{m}_1 \star \mathtt{m}_2$（其中 $\star$ 代表常规的 $+, -, \times$ 等运算）。然而，这个理想结果必须被约束到数胞 $(\mathtt{n}, \mathtt{m})$ 所能表示的 $\mathtt{n}$ 个状态（即 0 到 $\mathtt{n}-1$）的范围内，得到最终结果 $\mathtt{m}_3$。

因此，我们在数胞上定义一个新的、受约束的运算 $\odot$，其最终结果 $\mathtt{m}_3$ 由以下公式给出：

$$\mathtt{m}_3 = \mathtt{m}_1 \odot \mathtt{m}_2 = \mathtt{m}_{\mathtt{ideal}} \bmod \mathtt{n} = (\mathtt{m}_1 \star \mathtt{m}_2) \bmod \mathtt{n}$$

这个公式的意思是：先进行常规运算，然后取除以 $\mathtt{n}$ 的余数作为最终结果。

练习 1：一个状态数 $\mathtt{n} = 8$ 的数胞，计算常规运算 $10 + 5 = 15$ 在该数胞上的结果应该是多少?

2.2.3 加法运算：正向溢出与环绕

令 $\star$ 为常规加法 $+$，则数胞上受约束的加法 $\oplus$ 定义为：

$$\mathtt{m}_1 \oplus \mathtt{m}_2 = (\mathtt{m}_1 + \mathtt{m}_2) \bmod \mathtt{n}$$

现象分析：

- 当 $\mathtt{m}_1 + \mathtt{m}_2 < \mathtt{n}$ 时，结果与常规加法无异。
- 当 $\mathtt{m}_1 + \mathtt{m}_2 \geq \mathtt{n}$ 时，结果会从 0 重新开始计数，这种现象称为**正向溢出**。

示例：对于一个状态数 $\mathtt{n} = 256$ 的数胞（如一个字节），计算 $254 \oplus 3$：

$$254 \oplus 3 = (254 + 3) \bmod 256 = 257 \bmod 256 = 1$$

这就解释了为何在计算机中，$254 + 3$ 的结果不是 257 而是 1。加法器”数过了头”，从最大值 255 之后又回到了 0，然后继续数到 1。

在伪代码中，假设我们已有两个数胞变量 $\mathtt{x} = NC_{\mathtt{n}_1}(\mathtt{m}_1)$ 和 $\mathtt{y} = NC_{\mathtt{n}_2}(\mathtt{m}_2)$，允许使用：

$$\mathtt{x} :=^{\oplus} \mathtt{y}$$

使执行这行伪代码之后 $\mathtt{x} = NC_{\mathtt{n}_1}((\mathtt{m}_1 + \mathtt{m}_2) \bmod \mathtt{n}_1)$。

或者当存在自然数 $\mathtt{num}$ 时：

$$\mathtt{x} :=^{\oplus} \mathtt{num}$$

使执行这行伪代码之后 $\mathtt{x} = NC_{\mathtt{n}_1}((\mathtt{m}_1 + \mathtt{num}) \bmod \mathtt{n}_1)$。

假设我们已有两个数胞变量 $\mathtt{a} = NC_{\mathtt{n}}(\mathtt{m}_1)$ 和 $\mathtt{b} = NC_{\mathtt{n}}(\mathtt{m}_2)$，那么定义：

$$\mathtt{a} \oplus \mathtt{b} = NC_{\mathtt{n}}((\mathtt{m}_1 + \mathtt{m}_2) \bmod \mathtt{n})$$

这个运算要求 a 和 b 的状态数相同。

或者当存在自然数 num 时：

$$\mathtt{a} \oplus \mathtt{num} = NC_{\mathtt{n}}((\mathtt{m}_1 + \mathtt{num}) \bmod \mathtt{n})$$

a 和 num 的位置可以交换，即满足交换律。

2.2.4 减法运算：负向溢出与反向环绕

令 $\star$ 为常规减法 $-$，则数胞上受约束的减法 $\ominus$ 定义为：

$$\mathtt{m}_1 \ominus \mathtt{m}_2 = (\mathtt{m}_1 - \mathtt{m}_2) \bmod \mathtt{n}$$

现象分析：

- 当 $\mathtt{m}_1 - \mathtt{m}_2 \geq 0$ 时，结果是正常差值。
- 当 $\mathtt{m}_1 - \mathtt{m}_2 < 0$ 时，模运算会使其加上 n，从而从最大值 $\mathtt{n} - 1$ 开始反向循环，这种现象称为**负向溢出**。

示例：在同一数胞中（$\mathtt{n} = 256$），计算 $2 \ominus 3$：

$$2 \ominus 3 = (2 - 3) \bmod 256 = (-1) \bmod 256 = 255$$

结果是 255，即从 0 之下”绕到了”最大值。可以理解为：从 2 往回数 3 步：$2 \to 1 \to 0 \to 255$。

在伪代码中，假设我们已有两个数胞变量 $\mathtt{x} = NC_{\mathtt{n}_1}(\mathtt{m}_1)$ 和 $\mathtt{y} = NC_{\mathtt{n}_2}(\mathtt{m}_2)$，允许使用：

$$\mathtt{x} :=^{\ominus} \mathtt{y}$$

使执行这行伪代码之后 $\mathtt{x} = NC_{\mathtt{n}_1}((\mathtt{m}_1 - \mathtt{m}_2) \bmod \mathtt{n}_1)$。

或者当存在自然数 num 时：

$$\mathtt{x} :=^{\ominus} \mathtt{num}$$

使执行这行伪代码之后 $\mathtt{x} = NC_{\mathtt{n}_1}((\mathtt{m}_1 - \mathtt{num}) \bmod \mathtt{n}_1)$。

假设我们已有两个数胞变量 $\mathtt{a} = NC_{\mathtt{n}}(\mathtt{m}_1)$ 和 $\mathtt{b} = NC_{\mathtt{n}}(\mathtt{m}_2)$，那么定义：

$$\mathtt{a} \ominus \mathtt{b} = NC_{\mathtt{n}}((\mathtt{m}_1 - \mathtt{m}_2) \bmod \mathtt{n})$$

这个运算要求 a 和 b 的状态数相同。

或者当存在自然数 num 时：

$$\mathtt{a} \ominus \mathtt{num} = NC_{\mathtt{n}}((\mathtt{m}_1 - \mathtt{num}) \bmod \mathtt{n})$$

$$\mathtt{num} \ominus \mathtt{a} = NC_{\mathtt{n}}((\mathtt{num} - \mathtt{m}_1) \bmod \mathtt{n})$$

练习 2：在 $\mathtt{n} = 16$ 的数胞上，计算 $5 \ominus 8$ 的结果是多少?

2.2.5 乘法运算：位数扩展与结果截断

令 $\star$ 为常规乘法 $\times$，则数胞上的受约束乘法 $\otimes$ 定义为：

$$\mathtt{m}_1 \otimes \mathtt{m}_2 = (\mathtt{m}_1 \times \mathtt{m}_2) \bmod \mathtt{n}$$

现象分析：乘法可以视为重复的加法，因此它也继承了溢出的特性。一个很大的乘积结果会被” 截断”，只保留其除以 n 后的余数部分。
位数特性：在讨论乘法时，一个至关重要的特性是中间结果的位数扩展：

- 在任意 b 进制下，两个位数均为 k 的数字相乘，其理想结果的最大可能位数是 2k。
- **二进制示例**：一个 8 位 ($\mathtt{k}=8$) 二进制数最大值为 $(11111111)_2 = 255$。$255 \times 255 = 65025$，其二进制表示为 $(1111111000000001)_2$，总共需要 16 位来存储，正好是原来位数的两倍。
- **十进制示例**：一个 2 位数（$\mathtt{k}=2$）的最大值是 99，$99 \times 99 = 9801$，这是一个 4 位数（$2 \times 2 = 4$）。

计算机的运算器必须设计有足够的临时空间来处理这种位宽扩展的中间结果，然后再将最终结果根据目标数胞的大小进行截断（取模）并存储。这解释了为什么编程中即使一个字节的两个数相乘，也常常需要用两个字节来临时存放结果。

在伪代码中，假设我们已有两个数胞变量 $\mathtt{x} = NC_{\mathtt{n}_1}(\mathtt{m}_1)$ 和 $\mathtt{y} = NC_{\mathtt{n}_2}(\mathtt{m}_2)$，允许使用：

$$\mathtt{x} :=^{\otimes} \mathtt{y}$$

使执行这行伪代码之后 $\mathtt{x} = NC_{\mathtt{n}_1}((\mathtt{m}_1 \times \mathtt{m}_2) \bmod \mathtt{n}_1)$。

或者当存在自然数 num 时：

$$\mathtt{x} :=^{\otimes} \mathtt{num}$$

使执行这行伪代码之后 $\mathtt{x} = NC_{\mathtt{n}_1}((\mathtt{m}_1 \times \mathtt{num}) \bmod \mathtt{n}_1)$。

假设我们已有两个数胞变量 $\mathtt{a} = NC_{\mathtt{n}}(\mathtt{m}_1)$ 和 $\mathtt{b} = NC_{\mathtt{n}}(\mathtt{m}_2)$，那么定义：

$$\mathtt{a} \otimes \mathtt{b} = NC_{\mathtt{n}}((\mathtt{m}_1 \times \mathtt{m}_2) \bmod \mathtt{n})$$

这个运算要求 a 和 b 的状态数相同。

或者当存在自然数 num 时：

$$\mathtt{a} \otimes \mathtt{num} = NC_{\mathtt{n}}((\mathtt{m}_1 \times \mathtt{num}) \bmod \mathtt{n})$$

a 和 num 的位置可以交换，即满足交换律。
练习 3：在 $\mathtt{n}=100$ 的数胞（如两个十进制位）上，计算 $85 \otimes 12$。先计算常规乘积，再取模。

2.2.6 除法运算：向下取整与截断误差

令 $\star$ 为常规除法 $\div$，则数胞上的受约束除法 $\oslash$ 定义为：

$$\mathtt{m}_1 \oslash \mathtt{m}_2 = \lfloor \frac{\mathtt{m}_1}{\mathtt{m}_2} \rfloor$$

说明：

1. 与加、减、乘不同，除法通常不定义为取模运算。在计算机中，对于整数的除法，其结果是直接对精确商进行向下取整，即舍弃小数部分，只保留整数部分。$\lfloor x \rfloor$ 表示不大于 x 的最大整数。

2. **现象**：除法是" 缩减" 操作，不会产生向上溢出，但会产生截断误差（小数部分丢失）和除零错误（当 $\mathtt{m}_2 = 0$ 时）。

3. **示例**：在任意数胞中，$7 \oslash 2 = \lfloor 3.5 \rfloor = 3$，$5 \oslash 3 = \lfloor 1.666... \rfloor = 1$。

在伪代码中，假设我们已有两个数胞变量 $\mathtt{x} = NC_{\mathtt{n}_1}(\mathtt{m}_1)$ 和 $\mathtt{y} = NC_{\mathtt{n}_2}(\mathtt{m}_2), \mathtt{m}_2 \neq 0$，允许使用：

$$\mathtt{x} :=^{\oslash} \mathtt{y}$$

使执行这行伪代码之后 $\mathtt{x} = NC_{\mathtt{n}_1}(\lfloor \mathtt{m}_1 \div \mathtt{m}_2 \rfloor)$。

或者当存在自然数 num 时：

$$\mathtt{x} :=^{\oslash} \mathtt{num}$$

使执行这行伪代码之后 $\mathtt{x} = NC_{\mathtt{n}_1}(\lfloor \mathtt{m}_1 \div \mathtt{num} \rfloor)$。

假设我们已有两个数胞变量 $\mathtt{a} = NC_{\mathtt{n}}(\mathtt{m}_1)$ 和 $\mathtt{b} = NC_{\mathtt{n}}(\mathtt{m}_2)$，那么定义：

$$\mathtt{a} \oslash \mathtt{b} = NC_{\mathtt{n}}(\lfloor \mathtt{m}_1 \div \mathtt{m}_2 \rfloor)$$

这个运算要求 a 和 b 的状态数相同。

或者当存在自然数 num 时：

$$\mathtt{a} \oslash \mathtt{num} = NC_{\mathtt{n}}(\lfloor \mathtt{m}_1 \div \mathtt{num} \rfloor), \mathtt{num} \neq 0$$

$$\mathtt{num} \oslash \mathtt{a} = NC_{\mathtt{n}}(\lfloor \mathtt{num} \div \mathtt{m}_1 \rfloor), \mathtt{m}_1 \neq 0$$

我们再额外定义一个运算：

$$\mathtt{a} \text{ rem } \mathtt{b} = NC_{\mathtt{n}}(\mathtt{a} \bmod \mathtt{b})$$

同样要求 a 和 b 的状态数相同。

或者当存在自然数 num 时：

$$\mathtt{a} \text{ rem } \mathtt{num} = NC_{\mathtt{n}}(\mathtt{m}_1 \bmod \mathtt{num}), \mathtt{num} \neq 0$$

$$\texttt{num}\ \text{rem}\ \texttt{a} = NC_{\texttt{n}}(\texttt{num} \bmod \texttt{m}_1), \texttt{m}_1 \neq 0$$

练习 4：在任意数胞上（因为除法结果不受 n 约束，只要结果小于 n 即可），计算 $17 \oslash 5$ 和 $9 \oslash 4$。

练习 5：假设我们有两个数胞变量 $\texttt{a} = NC_8(5)$ 和 $\texttt{b} = NC_8(3)$。编写伪代码序列执行以下操作：

$$\texttt{a} :=^{\oplus} \texttt{b}$$

$$\texttt{c} := \texttt{a} \ominus 1 \otimes \texttt{b}$$

要求：解释第一步操作后 a 的值和第二部操作后 c 的值（以 $NC_{\texttt{n}}(\texttt{m})$ 形式表示），并说明运算中的溢出或截断现象。

本节总结

- **核心思想**：数胞运算 = 常规运算 + 模约束（除法则为取整）。
- **加法**：可能正向溢出，结果环绕到 0 重新开始。
- **减法**：可能负向溢出，结果从最大值反向环绕。
- **乘法**：中间结果位数扩展，最终结果截断。
- **除法**：向下取整，产生截断误差。

理解数胞运算的特性对于编程中防止溢出错误、理解数据类型限制至关重要。

练习答案与反馈

- **练习 1 答案：** $(10 + 5) \bmod 8 = 15 \bmod 8 = 7$。因为 15 除以 8 商 1 余 7。
- **练习 2 答案：** $(5 - 8) \bmod 16 = (-3) \bmod 16 = 13$。因为 $-3 + 16 = 13$。
- **练习 3 答案：** 常规乘积：$85 \times 12 = 1020$。取模：$1020 \bmod 100 = 20$。所以结果是 20。
- **练习 4 答案：**
  - $17 \oslash 5 = \lfloor 3.4 \rfloor = 3$
  - $9 \oslash 4 = \lfloor 2.25 \rfloor = 2$
- **练习 5 答案：**
  - 执行 $\texttt{a} :=^{\oplus} \texttt{b}$ 之后 $\texttt{a} = NC_8((5 + 3) \bmod 8) = NC_8(0)$，产生了溢出现象。
  - 执行 $\texttt{c} := \texttt{a} \ominus 1 \otimes \texttt{b}$ 之后 $\texttt{c} = NC_8(((0 - 1) \times 3) \bmod 8) = NC_8(5)$，产生了溢出现象和截断现象。

2.3 数胞的扩展：有符号数的表示与运算

【本节目标】

- **问题驱动**：理解计算机如何用只能表示正数的存储单元来表示负数。
- **掌握概念**：掌握补码表示法的原理和优势。
- **理解规律**：理解有符号数和无符号数在硬件层面的统一运算机制。
- **实践目标**：能够在有符号和无符号两种解释下转换和理解同一个数胞的值。

开头：我们要解决什么问题？

想象一下，我们的仓库计数器只能显示 0 到 99（相当于一个状态数 $\mathtt{n}=100$ 的数胞）。现在我们需要记录温度变化，既有零上温度也有零下温度。我们该如何用这个只能显示正数的计数器来表示负数呢?

- **方法 A**：预留一些数字表示负数。如让 0-59 表示 0 到 59 度，60-99 表示 -40 到 -1 度。
- **方法 B**：更普遍的方法——补码表示法，这也是计算机实际采用的方法。

本节我们将学习计算机如何通过重新解释数胞状态的语义来表示负数，以及这种表示法带来的运算优势。

2.3.1 补码表示法

之前的章节将数胞视为无符号数，其值 $\mathtt{m}$ 的范围是 0 到 $\mathtt{n}-1$。但在实际应用中，我们需要表示负数。我们可以通过重新解释数胞状态的语义来定义有符号数。

对于一个状态数为 $\mathtt{n}$ 的数胞 $NC_{\mathtt{n}}(\mathtt{m})$，我们将其所能表示的 $\mathtt{n}$ 个状态进行划分，以此来表示正数和负数：

- 状态 0 到 $\lfloor\frac{\mathtt{n}-1}{2}\rfloor$ 解释为非负数 0 到 $\lfloor\frac{\mathtt{n}-1}{2}\rfloor$。
- 状态 $\lfloor\frac{\mathtt{n}-1}{2}\rfloor+1$ 到 $\mathtt{n}-1$ 解释为负数 $-(\mathtt{n}-1-\lfloor\frac{\mathtt{n}-1}{2}\rfloor)$ 到 -1。

为什么叫”补码”？

补码的命名来源于一个数学特性：一个负数 $-x$ 的补码表示等于 $\mathtt{n}-x$。例如，在 $\mathtt{n}=256$ 的情况下，-1 的补码是 255，-2 的补码是 254，依此类推。

二进制计算机的特殊情况

为了最大化表示范围且保证对称性，通常在二进制计算机中令 $\mathtt{n}=2^k$（即数胞刚好有 2^k 个状态）。此时，划分点位于 2^{k-1}，表示范围是对称的 $[-2^{k-1},2^{k-1}-1]$。

示例：一个状态数 $\mathtt{n}=256$ 的数胞（$256=2^8$，所以 $k=8$）。

- 状态 0 到 127 解释为 0 到 127（非负数）。
- 状态 128 到 255 解释为 -128 到 -1（负数）。
- 一个数胞的值 $\mathtt{m} = 150$，若解释为无符号数，值是 150；若解释为有符号数，其值为 $150 - 256 = -106$。

练习 1：在一个 8 位二进制数胞（$\mathtt{n} = 256$）中，数值 200 在有符号解释下表示什么数？

2.3.2 补码的优势

补码表示法最精妙的特性在于：有符号数和无符号数的加法、减法和乘法运算，在硬件层面使用完全相同的电路和操作。运算过程本身对数值的解释是”无知”的，它只是在执行模 $\mathtt{n}$ 的算术。

这意味着之前为无符号数定义的这些运算规则（$\oplus, \ominus, \otimes$）完全适用于有符号数。区别仅在于，程序员或编译器在解释运算结果时，需要选择是采用无符号的视角还是有符号的视角。

在伪代码中，我们使用 $SNC_{\mathtt{n}}(\mathtt{m})$ 表示用补码解释为有符号数的数胞。SNC 可以视为一种特殊的数胞，所有对于 NC 的伪代码语句，都可以应用于 SNC。并且在对于 SNC 的运算中，其中的操作数可以是整数。但是 NC 和 SNC 不能同时出现在一个算式中。如果有这方面的需求，假设已有 $\mathtt{a} = NC_{\mathtt{n}_1}(\mathtt{m}_1)$ 和 $\mathtt{b} = SNC_{\mathtt{n}_2}(\mathtt{m}_2)$，我们定义函数：

$$\mathrm{SNC}(\mathtt{a}) = SNC_{\mathtt{n}_1}(\mathtt{m}_1)$$

$$\mathrm{NC}(\mathtt{b}) = NC_{\mathtt{n}_2}(\mathtt{m}_2)$$

加减法运算的统一性

加法/减法示例（数胞的状态数为 256）：

- **无符号解释**：$200 \oplus 100 = (200 + 100) \bmod 256 = 300 \bmod 256 = 44$
- **有符号解释**：将 200 解释为 $200 - 256 = -56$，将 100 解释为 100。$(-56) \oplus 100 = (-56 + 100) \bmod 256 = 44 \bmod 256 = 44$

结果在两种解释下都是 44。运算的过程完全一致，硬件不需要为有符号数和无符号数设计不同的加法器。

乘法运算的统一性

乘法运算 $\otimes$ 的规则同样统一。硬件执行的是模 $\mathtt{n}$ 的乘法，无论操作数被解释为有符号数还是无符号数。

乘法示例（数胞的状态数为 256）：

- **无符号解释**：$254 \otimes 127 = (254 \times 127) \bmod 256 = 32258 \bmod 256 = 2$

- **有符号解释**：将 254 解释为 $254 - 256 = -2$，将 127 解释为 127。$(-2) \otimes 127 = (-2 \times 127) \bmod 256 = -254 \bmod 256 = 2$

结果在两种解释下都是 2，这个例子清晰地表明，硬件提供的是模运算的结果。
练习 2：在 8 位二进制数胞中，计算 $240 \oplus 20$，并分别用无符号和有符号解释验证结果的正确性。

2.3.3 除法的特殊性：向零取整

有符号数的除法 $\oslash$ 具有特殊性，通常遵循” 向零取整” 的规则，这与无符号数” 向下取整” 的规则略有不同。

向零取整 vs 向下取整

- **向下取整**（无符号数）：总是向更小的整数方向取整。$\lfloor 3.7 \rfloor = 3$，$\lfloor -3.7 \rfloor = -4$。
- **向零取整**（有符号数）：总是向零的方向取整。$3.7 \to 3$，$-3.7 \to -3$。

示例：

- $(-7) \oslash 2 = -3$（向零取整，而不是向下取整的 -4）
- $7 \oslash (-2) = -3$（同样向零取整）

取余运算的符号规则

取余运算的结果符号通常与被除数的符号相同：

- (-7) rem $2 = -1$（因为 $-7 = (-4) \times 2 + 1$，但更常见的实现是 $-7 = (-3) \times 2 + (-1)$）
- 7 rem $(-2) = 1$

在有符号数胞中，同样遵循向零取整的特殊性。包括 “$\oslash$”、“$:=^{\oslash}$” 和 “rem” 运算。
练习 3：计算 $(-9) \oslash 4$ 和 (-9) rem 4，注意有符号除法的向零取整特性。
练习 4：定义一个数胞变量 $\mathtt{x} = NC_{16}(8)$

1. 分别解释 $\mathtt{x}$ 和 $\mathrm{SNC}(\mathtt{x})$ 的值（即计算无符号值和有符号值）
2. 设已有数胞变量 $\mathtt{y} = NC_{16}(14)$ 和 $\mathtt{z} = NC_{16}(3)$，尝试编写伪代码，分别使用一次 $\mathtt{y}$ 和 $\mathtt{z}$，使执行伪代码之后 $\mathtt{x} = NC_{16}(4)$。
3. 设已有数胞变量 $\mathtt{y} = SNC_{16}(-5)$ 和 $\mathtt{z} = NC_{16}(14)$，尝试编写伪代码，分别使用一次 $\mathtt{y}$ 和 $\mathtt{z}$，使执行伪代码之后 $\mathtt{x} = NC_{16}(14)$。

本节总结

- **补码表示法**：通过重新解释数胞状态的语义来表示负数，状态数 n 时，表示范围为 $[-\lfloor \frac{n}{2} \rfloor, \lfloor \frac{n-1}{2} \rfloor]$。
- **统一运算**：有符号数和无符号数的加、减、乘法使用相同的硬件电路，只是结果解释不同。
- **除法特殊性**：有符号除法采用向零取整，而无符号除法采用向下取整。
- **硬件优势**：补码表示法极大简化了 CPU 的算术单元设计。

理解补码表示法对于理解计算机如何高效处理有符号数运算至关重要。

练习答案与反馈

- **练习 1 答案**：200 在有符号解释下表示 $200 - 256 = -56$。
- **练习 2 答案**：
  - 无符号：$240 + 20 = 260$，$260 \bmod 256 = 4$
  - 有符号：240 解释为 -16，$-16 + 20 = 4$，$4 \bmod 256 = 4$
  - 两种解释结果一致，都是 4
- **练习 3 答案**：
  - (-9) 向零取整：$-9/4 = -2.25$，向零取整为 -2
  - $(-9) \text{ rem } 4 = -9 - ((-2) \times 4) = -9 - (-8) = -1$
- **练习 4 答案**：
  - 解释结果：x 的值为 8，SNC(x) 的值为 -8。
  - 第二小题，答案不唯一！例如：

    $$\mathtt{x} :=^{\ominus} \mathtt{y} \oslash \mathtt{z}$$

    又例如：

    $$\mathtt{x} :=^{\oslash} \mathtt{y} \text{ rem } \mathtt{z}$$

  - 第三小题，例如：

    $$\mathtt{x} :=^{\ominus} \mathrm{NC}(\mathtt{y}) \otimes \mathtt{z}$$

2.4 布尔值与逻辑运算

【本节目标】

- **问题驱动**：理解计算机如何通过简单的真/假判断来进行复杂的逻辑决策。
- **掌握概念**：掌握布尔值、真值表的概念，以及四种基本逻辑运算的规则。
- **理解规律**：理解一元运算与二元运算的区别，以及不同逻辑运算的语义含义。
- **实践目标**：能够根据真值表计算逻辑运算的结果，理解不同运算的现实对应关系。

开头：我们要解决什么问题？

想象一下，一个智能门禁系统的开门条件：”只有在本小区住户 **并且**不是黑名单人员 **并且** (密码正确 **或者**刷卡成功) 的情况下，才允许开门”。

计算机如何判断这样复杂的条件呢？答案就是通过**逻辑运算**。逻辑运算处理的是最简单的真/假判断，但通过组合这些基本运算，计算机能够做出复杂的逻辑决策。本节我们将学习计算机中的布尔逻辑运算体系。

2.4.1 布尔值

在计算机逻辑中，我们使用**布尔值**来表示真伪判断。布尔值只有两种可能状态：

- **真** (**T**)：表示条件成立、是、正确等肯定意义
- **假** (**F**)：表示条件不成立、否、错误等否定意义

在计算机内部，我们通常用数字来表示这两种状态：

- 用数字 `1` 表示“真”
- 用数字 `0` 表示“假”

这种表示方法使得逻辑运算可以直接通过数字电路来实现。

在伪代码中，我们定义函数 bool，规定若 $\mathtt{a} \in NC_{\mathtt{n}}$并且$\mathtt{a} \neq NC_{\mathtt{n}}(0)$，或者 $\mathtt{a} \in SNC_{\mathtt{n}}$并且$\mathtt{a} \neq SNC_{\mathtt{n}}(0)$，那么 $\text{bool}(\mathtt{a}) = \mathbf{T}$，否则 $\text{bool}(\mathtt{a}) = \mathbf{F}$。即，数胞的值不为 0，那么 bool 的输出为 **T**，数胞的值为 0，那么输出为 **F**。

我们继续定义函数 boolvalue：

$$\text{boolvalue}(\mathbf{T}) = 1$$

$$\text{boolvalue}(\mathbf{F}) = 0$$

我们使用 NC_2 来表示布尔值类型，其中 $\mathbf{T} = NC_2(1)$，$\mathbf{F} = NC_2(0)$。

2.4.2 运算的种类：一元与二元

根据参与运算的布尔值个数，逻辑运算分为两类：

- **一元运算**：只对一个布尔值进行操作的运算。
  - 输入：1 个值（0 或 1）
  - 输出：2 种可能（0 或 1）
  - 理论上存在 $2^2 = 4$ 种不同的一元运算
  - 最常用的是**非**运算 (NOT)
- **二元运算**：对两个布尔值进行操作的运算。
  - 输入：2 个值，共有 $2^2 = 4$ 种输入组合
  - 输出：每种输入组合都有 2 种输出可能
  - 理论上存在 $2^4 = 16$ 种不同的二元运算
  - 最常用的是**与** (AND)、**或** (OR)、**异或** (XOR)

练习 1：为什么一元运算有 4 种可能，而二元运算有 16 种可能?

2.4.3 真值表

为了清晰地展示逻辑运算的规则，我们使用**真值表**这一工具。真值表列出了所有可能的输入组合，以及对应每一种组合所得到的输出结果。

真值表的结构通常为：

- 左边列：所有可能的输入组合
- 右边列：对应的输出结果
- 每一行：一种特定的输入情况及其输出

接下来，我们将通过真值表来学习四种最基本的逻辑运算。

2.4.4 非运算 (NOT) - 一元运算

非运算的作用是” 取反” 或” 否定”。如果输入是” 真”，结果就是” 假”；如果输入是” 假”，结果就是” 真”。

运算规则：输出与输入相反　　**现实类比**：就像是一个开关的” 关闭” 状态对应” 打开” 的否定

非运算的真值表：

输入 A	输出 $\overline{A}$
0	1
1	0

符号表示：在变量上加一横线，如 $\overline{A}$，或者在变量前方加上 ¬，如 ¬A，表示对 A 进行非运算。

在伪代码中，我们使用符号“¬”表示非运算，且作用对象只能是布尔值。例如，$\neg\mathbf{T} = \mathbf{F}, \neg\mathbf{F} = \mathbf{T}$。

练习 2：如果 $A = \mathbf{T}$，那么 $\overline{A}$ 的值是多少?

2.4.5 与运算 (AND) - 二元运算

与运算的规则是：**只有当所有输入都为”真”时，结果才为”真”；否则，结果为”假”**。

运算规则：全真为真，有假即假　　**现实类比**：就像串联电路 - 所有开关都闭合时灯才亮

与运算的真值表：

输入 A	输入 B	输出 $A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

符号表示：常用 $\wedge$ 或 $\cdot$，如 $A \wedge B$ 或 $A \cdot B$

在伪代码中，我们使用符号“∧”表示与运算，且作用对象只能是布尔值。例如，$\mathbf{T} \wedge \mathbf{T} = \mathbf{T}, \mathbf{T} \wedge \mathbf{F} = \mathbf{F}$。

练习 3：计算 $(\mathbf{T} \wedge \mathbf{F}) \wedge \mathbf{T}$ 的结果

2.4.6 或运算 (OR) - 二元运算

或运算的规则是：**只要有一个输入为”真”，结果就为”真”；只有当所有输入都为”假”时，结果才为”假”**。

运算规则：有真即真，全假为假　　**现实类比**：就像并联电路 - 任意一个开关闭合时灯就亮

或运算的真值表：

输入 A	输入 B	输出 $A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

符号表示：常用 $\vee$ 或 $+$，如 $A \vee B$ 或 $A + B$

在伪代码中，我们使用符号“∨”表示与运算，且作用对象只能是布尔值。例如，$\mathbf{T} \vee \mathbf{T} = \mathbf{T}, \mathbf{T} \vee \mathbf{F} = \mathbf{T}$。

练习 4：如果有三个输入 $\mathtt{A} = \mathbf{T}, \mathtt{B} = \mathbf{F}, \mathtt{C} = \mathbf{T}$，计算 $(\mathtt{A} \vee \mathtt{B}) \vee \mathtt{C}$

2.4.7 异或运算 (XOR) - 二元运算

异或运算的规则是：**如果两个输入的值不同，结果为”真”；如果两个输入的值相同，结果为”假”**。

运算规则：不同为真，相同为假　　**现实类比**：就像判断两个开关状态是否一致 - 状态一致时灯灭，状态不同时灯亮

异或运算的真值表：

输入 A	输入 B	输出 $\mathtt{A} \oplus \mathtt{B}$
0	0	0
0	1	1
1	0	1
1	1	0

符号表示：常用 $\veebar$，如 $A \veebar B$

在伪代码中，我们使用符号“$\veebar$”表示与运算，且作用对象只能是布尔值。例如，$\mathbf{T} \veebar \mathbf{T} = \mathbf{F}, \mathbf{T} \veebar \mathbf{F} = \mathbf{T}$。

练习 5：计算 $(\mathbf{T} \veebar \mathbf{F}) \veebar \mathbf{T}$ 的结果，并与 $(\mathbf{T} \veebar \mathbf{T}) \veebar \mathbf{F}$ 比较

练习 6：假设有两个布尔变量 $\mathtt{p} = NC_2(1), \mathtt{q} = NC_2(0)$，计算下列式的结果

- $\text{bool}(\mathtt{p}) \wedge \neg(\text{bool}(\mathtt{q}))$
- $(\text{bool}(\mathtt{p}) \vee \text{bool}(\mathtt{q})) \wedge \text{bool}(\mathtt{p})$
- $\neg(\text{bool}(\mathtt{p}) \veebar \text{bool}(\mathtt{q}))$

本节总结

- **布尔值**：计算机逻辑的基础，只有真 (1) 和假 (0) 两种值
- **真值表**：展示逻辑运算规则的有效工具
- **四种基本运算**：
  - 非 (NOT)：取反运算，一元操作
  - 与 (AND)：全真为真，有假即假
  - 或 (OR)：有真即真，全假为假

  - 异或 (XOR)：不同为真，相同为假

- **运算优先级**：NOT > AND > OR（可用括号改变优先级）

这些基本的逻辑运算通过组合可以构建出复杂的逻辑判断系统，是计算机能够进行智能决策的基础。

练习答案与反馈

- **练习 1 答案**：一元运算有 1 个输入（2 种可能），每个输入对应 2 种输出选择，所以有 $2^2 = 4$ 种可能的一元运算。二元运算有 2 个输入（4 种组合），每种组合对应 2 种输出选择，所以有 $2^4 = 16$ 种可能的二元运算。

- **练习 2 答案**：如果 $\mathtt{A} = \mathbf{T}$，那么 $\overline{\mathtt{A}} = \mathbf{F}$（取反操作）。

- **练习 3 答案**：先计算 $\mathbf{T} \wedge \mathbf{F} = \mathbf{F}$，然后计算 $\mathbf{F} \wedge \mathbf{T} = \mathbf{F}$，所以结果是 $\mathbf{F}$。

- **练习 4 答案**：先计算 $\mathtt{A} \vee \mathtt{B} = \mathbf{T} \vee \mathbf{F} = \mathbf{T}$，然后计算 $\mathbf{T} \vee \mathtt{C} = \mathbf{T} \vee \mathbf{T} = \mathbf{T}$，所以结果是 $\mathbf{T}$。

- **练习 5 答案**：$(\mathbf{T} \veebar \mathbf{F}) \veebar \mathbf{T} = \mathbf{T} \veebar \mathbf{T} = \mathbf{F}$; $(\mathbf{T} \veebar \mathbf{T}) \veebar \mathbf{F} = \mathbf{F} \veebar \mathbf{F} = \mathbf{F}$。两个表达式结果相同。

- **练习 6 答案**：

  - $\text{bool}(\mathtt{p}) \wedge \neg(\text{bool}(\mathtt{q})) = \mathbf{T} \wedge \neg(\mathbf{F}) = \mathbf{T} \wedge \mathbf{T} = \mathbf{T}$
  - $(\text{bool}(\mathtt{p}) \vee \text{bool}(\mathtt{q})) \wedge \text{bool}(\mathtt{p}) = (\mathbf{T} \vee \mathbf{F}) \wedge \mathbf{T} = \mathbf{T} \wedge \mathbf{T} = \mathbf{T}$
  - $\neg(\text{bool}(\mathtt{p}) \veebar \text{bool}(\mathtt{q})) = \neg(\mathbf{T} \veebar \mathbf{F}) = \neg(\mathbf{T}) = \mathbf{F}$

2.5 比较运算

【本节目标】

- **问题驱动**：理解计算机如何像人类一样进行大小、相等关系的判断。

- **掌握概念**：掌握六种基本比较运算的含义和用法。

- **理解关系**：理解比较运算与布尔逻辑运算的紧密联系。

- **实践目标**：能够编写复杂的条件判断表达式，理解复合条件的逻辑关系。

开头：我们要解决什么问题？

想象一下，你在网上购物时需要筛选商品：

- “价格在 100 元到 500 元之间”

- “评分 4.5 星以上”
- “销量大于 1000 件”

这些筛选条件都涉及到数值的比较。计算机如何理解并执行这些判断呢？这就是比较运算要解决的问题。

比较运算指的是对两个数值进行对比，判断它们之间的大小、相等关系。比较运算的结果不是具体的数字，而是一个布尔值（`True` 或 `False`），表示这个判断是否成立。

2.5.1 六种基本比较运算

计算机中常用的比较运算主要有以下六种，它们构成了所有复杂判断的基础：

- **等于** ($=$)：判断两个数是否完全相等
- **不等于** ($\neq$)：判断两个数是否不相等
- **大于** ($>$)：判断左边的数是否大于右边的数
- **小于** ($<$)：判断左边的数是否小于右边的数
- **大于等于** ($\geq$)：判断左边的数是否大于或等于右边的数
- **小于等于** ($\leq$)：判断左边的数是否小于或等于右边的数

实际应用示例

假设有两个变量：$\mathtt{A}=5, \mathtt{B}=3$。我们来看这些比较运算的具体结果：

运算	符号	表达式	结果
等于	$=$	$\mathtt{A}=\mathtt{B}$	**F** (0)
不等于	$\neq$	$\mathtt{A}\neq\mathtt{B}$	**T** (1)
大于	$>$	$\mathtt{A}>\mathtt{B}$	**T** (1)
小于	$<$	$\mathtt{A}<\mathtt{B}$	**F** (0)
大于等于	$\geq$	$\mathtt{A}\geq\mathtt{B}$	**T** (1)
小于等于	$\leq$	$\mathtt{A}\leq\mathtt{B}$	**F** (0)

在伪代码中，假设我们有两个数胞变量 $\mathtt{a}=NC_{\mathtt{n}_1}(\mathtt{m}_1), \mathtt{b}=NC_{\mathtt{n}_2}(\mathtt{m}_2)$，那么我们定义数胞之间的比较运算就是其值之间的比较运算。例如，$a=b$ 相当于判断 $\mathtt{m}_1=\mathtt{m}_2$。

同理，有符号数胞 SNC 也可以进行比较，唯一不同之处在于有符号数胞的值可能是负数，从而影响判断结果。例如，假设有两个数胞变量 $\mathtt{c}=NC_{10}(7), \mathtt{d}=NC_{10}(4)$，$\mathtt{c}<\mathtt{d}=\mathbf{F}$，而 $\mathrm{SNC}(\mathtt{c})<\mathrm{SNC}(\mathtt{d})=\mathbf{T}$，因为 $\mathrm{SNC}(\mathtt{c})$ 的值被解释为 -3，而 $\mathrm{SNC}(\mathtt{d})$ 的值依然是 4，$-3<4=\mathbf{T}$。

进而，数胞与常数之间也可以进行比较，NC 允许与自然数进行比较，SNC 允许与整数进行比较，它们的结果都是布尔值。

特别地，类型信息之间的比较也是允许的，假设有 $\mathtt{t}_1 \in NC_8$ 和 $\mathtt{t}_2 \in NC_{10}$，那么有：

$\text{typeof}(\mathtt{t}_1) = NC_8$ 的输出为 **T**；

$\text{typeof}(\mathtt{t}_1) \neq NC_8$ 的输出为 **F**；

$\text{typeof}(\mathtt{t}_1) = \text{typeof}(\mathtt{t}_2)$ 的输出为 **F**。

练习 1：如果 $\mathtt{X} = 8$，$\mathtt{Y} = 8$，那么 $\mathtt{X} \geq \mathtt{Y}$ 的结果是什么？$\mathtt{X} < \mathtt{Y}$ 呢？

2.5.2 比较运算与布尔逻辑的完美结合

比较运算和布尔逻辑运算就像一对默契的搭档：比较运算负责生成判断结果，布尔逻辑运算负责组合这些判断。

关系一：比较运算产生布尔值

每个比较运算本身就是一个布尔表达式。例如，$\mathtt{A} > \mathtt{B}$ 的结果要么是 True，要么是 False，可以直接作为逻辑运算的输入。

关系二：布尔逻辑组合多个比较条件

这才是两者结合最强大的地方。现实中的判断往往需要多个条件同时满足或至少满足一个。

示例 1：范围判断

判断一个数 $\mathtt{X}$ 是否在 10 到 20 之间（包含端点）：

$$(\mathtt{X} \geq 10) \wedge (\mathtt{X} \leq 20)$$

真值表分析：

$\mathtt{X}$	$\mathtt{X} \geq 10$	$\mathtt{X} \leq 20$	最终结果
5	**F**	**T**	**F**
15	**T**	**T**	**T**
25	**T**	**F**	**F**

示例 2：多选一条件

判断今天是否是周末（星期六或星期日）：

$$(\text{今天是星期六}) \vee (\text{今天是星期日})$$

示例 3：条件取反

判断是否” 不在家”：

$$\overline{(\text{我在家})}$$

练习 2：编写一个表达式来判断成绩 $\mathtt{score}$ 是否优秀（大于等于 90 分）并且不是满分（等于 100 分）。

2.5.3 比较运算的优先级与括号使用

当表达式变得复杂时，我们需要考虑运算的优先级：

- **比较运算**的优先级高于**布尔逻辑运算**
- 具体优先级：比较运算 $(=, \neq, >, <, \geq, \leq)$ > NOT > AND > OR

示例分析

表达式：$\texttt{X} > 5 \wedge \texttt{Y} < 10 \vee \texttt{Z} = 0$

在忽视优先级的情况下，这个表达式可能产生歧义。尽管与的优先级高于或的优先级，但是为了清晰的可读性，应该使用括号：

$$(\texttt{X} > 5 \wedge \texttt{Y} < 10) \vee \texttt{Z} = 0$$

或者

$$\texttt{X} > 5 \wedge (\texttt{Y} < 10 \vee \texttt{Z} = 0)$$

括号不仅消除了歧义，还让表达式更易读。

练习 3：为表达式"$\texttt{A} = 1 \vee \texttt{B} > 5 \wedge \texttt{C} < 3$" 添加括号，使其可读性更清晰。

本节总结

- **六种比较运算**：等于、不等于、大于、小于、大于等于、小于等于
- **结果类型**：比较运算的结果是布尔值（True/False）
- **组合使用**：通过 AND、OR、NOT 组合多个比较条件形成复杂判断
- **优先级**：比较运算 > NOT > AND > OR，建议使用括号确保清晰
- **实际应用**：条件判断、数据筛选、流程控制等都依赖比较运算

比较运算与布尔逻辑的结合，让计算机能够理解并执行复杂的条件判断，这是编程和算法设计的基础。

练习答案与反馈

- **练习 1 答案**：因为 $\texttt{X} = \texttt{Y} = 8$，所以：
  - $\texttt{X} \geq \texttt{Y}$：True（因为 $8 \geq 8$ 成立）
  - $\texttt{X} < \texttt{Y}$：False（因为 $8 < 8$ 不成立）
- **练习 2 答案**：$(\texttt{score} \geq 90) \wedge (\texttt{score} \neq 100)$
- **练习 3 答案**：有两种可能的解释：

- $(\texttt{A} = 1) \vee (\texttt{B} > 5 \wedge \texttt{C} < 3)$
- $(\texttt{A} = 1 \vee \texttt{B} > 5) \wedge \texttt{C} < 3$

具体使用哪种取决于实际需求，这就是为什么需要括号来明确意图。

3 程序的控制流

3.1 为什么需要控制流

【本节目标】

- **问题驱动**：理解为什么程序不能只是简单地从上到下执行每一条指令。
- **掌握概念**：掌握顺序、分支、循环三种基本控制结构的概念和用途。
- **理解差异**：理解不同控制结构如何影响程序的执行路径。
- **实践目标**：能够分析简单程序段中控制流的走向。

开头：我们要解决什么问题？

想象一下，你要指导一个机器人完成”冲泡咖啡”的任务。如果你只是简单地列出所有步骤：

1. 取咖啡杯
2. 加入咖啡粉
3. 加入热水
4. 加入牛奶
5. 加入糖

这样的指令存在几个问题：

- **缺乏灵活性**：如果用户不想加糖怎么办？机器人还是会机械地执行第 5 步。
- **无法处理异常**：如果咖啡粉用完了怎么办？机器人还是会继续执行后续步骤。
- **重复劳动**：如果需要冲泡 3 杯咖啡，就要把同样的指令写 3 遍。

这就是我们需要**控制流**的原因：让程序能够根据不同的情况选择不同的执行路径，或者重复执行某些指令。控制流是程序的”决策大脑”。

3.1.1 三种基本控制结构

计算机科学证明，任何复杂的程序都可以由以下三种基本控制结构组合而成：

1. 顺序结构

顺序结构是最简单的控制结构，程序按照指令的书写顺序依次执行。

步骤	指令	说明
1	取咖啡杯	第一步
2	加入咖啡粉	第二步
3	加入热水	第三步

特点：

- 每条指令都会被执行
- 执行顺序固定不变
- 就像烹饪食谱中的步骤列表

在伪代码中，默认上一行执行完后执行上一行的下一行。

2. 分支结构

分支结构让程序能够根据条件选择不同的执行路径。分支结构有多种形式：

单分支结构（如果）：

条件判断	执行内容
如果” 用户要加糖”	加入糖
	继续下一步

在伪代码中，使用以下结构表示单分支结构：

$$\begin{cases} \text{分支内容} & \text{如果}：\text{执行条件} \end{cases}$$

其中 分支内容 是任意的伪代码，执行条件 是任意的可以代表布尔值的伪代码。

双分支结构（如果-否则）：

条件判断	路径 A	路径 B
如果” 用户要加糖”	加入糖 继续下一步	不加糖 继续下一步

在伪代码中，使用以下结构表示双分支结构：

$$\begin{cases} \text{分支内容 1} & \text{如果}：\text{执行条件} \\ \text{分支内容 2} & \text{否则} \end{cases}$$

其中 分支内容 1 和 分支内容 2 是任意的伪代码，执行条件 是任意的可以代表布尔值的伪代码。

多分支结构（如果-否则如果-否则）：

条件判断	路径 A	路径 B	路径 C
根据咖啡类型选择	美式咖啡 加少量水 继续下一步	拿铁咖啡 加牛奶 继续下一步	卡布奇诺 加奶泡 继续下一步

在伪代码中，使用以下结构表示多分支结构：

$$\begin{cases} \text{分支内容 1} & \text{如果：执行条件 1} \\ \text{分支内容 2} & \text{否则如果：执行条件 2} \\ \vdots & \vdots \\ \text{分支内容 n} & \text{否则如果：执行条件 n} \\ \text{分支内容 n+1} & \text{否则} \end{cases}$$

其中 分支内容 1 到 分支内容 n+1 是任意的伪代码，执行条件 1 到 执行条件 n 是任意的可以代表布尔值的伪代码，“分支内容 n+1　否则”可选择忽略不写。

在伪代码中，若书写空间不够，可以把“如果：...”、“否则如果：...”、“否则”提到对应分支内容的第一行。

分支结构的数学表示：

$$\text{执行路径} = \begin{cases} \text{路径 A} & \text{当条件} C_1 \text{成立时} \\ \text{路径 B} & \text{当条件} C_2 \text{成立时} \\ \text{路径 C} & \text{当条件} C_3 \text{成立时} \\ \vdots & \vdots \\ \text{默认路径} & \text{当所有条件都不成立时} \end{cases}$$

特点：

- 基于条件进行选择
- 每次只执行其中一个分支
- 就像十字路口的选择：向左转、向右转或直行

3. 循环结构

循环结构让程序能够重复执行某段指令。根据条件判断的时机，循环分为两种类型：

先判断型循环（当型循环）：

条件检查	循环体	后续步骤
检查 `i` < 3	执行冲泡步骤 `i` 增加 1	循环结束 继续后续操作

在伪代码中，使用以下结构表示先判断型循环：

$$\begin{bmatrix} \text{当}：\text{循环条件} \\ \quad \text{循环内容} \end{bmatrix}$$

其中 循环内容 是任意的伪代码，循环条件 是任意的可以代表布尔值的伪代码。

后判断型循环（直到型循环）：

循环体	条件检查	后续步骤
执行冲泡步骤	检查 `i` < 3 如果成立则继续	循环结束 继续后续操作

在伪代码中，使用以下结构表示先判断型循环：

$$\begin{bmatrix} \quad \text{循环内容} \\ \text{当}：\text{循环条件} \end{bmatrix}$$

其中 循环内容 是任意的伪代码，循环条件 是任意的可以代表布尔值的伪代码。

两种循环的区别：

- **先判断型循环**：可能一次都不执行（如果初始条件就不成立）
- **后判断型循环**：至少执行一次循环体
- 两种结构可以相互转换，但语义略有不同

循环结构的数学表示：

当条件 C 成立时，重复执行：指令序列 S

执行指令序列 S，直到条件 C 不成立

特点：

- 重复执行相同的指令序列
- 必须有终止条件，避免无限循环
- 就像工厂的流水线：重复相同的操作

3.1.2 控制结构的组合使用

实际程序往往是三种结构的复杂组合。例如，一个完整的咖啡冲泡程序可能包含：

1. **顺序结构**：基本冲泡步骤（取杯 → 加粉 → 加热水）
2. **分支结构**：根据用户选择决定是否加糖、加多少糖

3. **循环结构**：重复冲泡多杯咖啡
4. **嵌套分支**：在循环内部根据咖啡类型选择不同的冲泡参数
5. **循环中的分支**：在每次冲泡时检查原料是否充足

这种组合使得程序能够处理复杂的现实问题。
练习：分析”自动售货机”的工作流程，识别其中的顺序、分支和循环结构，并说明使用了哪种类型的循环。

3.1.3 控制流的重要性

控制流是程序能够”智能”应对不同情况的关键：

- **适应性**：程序能够根据输入数据或环境状态调整行为
- **效率性**：通过循环避免代码重复，提高开发效率
- **健壮性**：能够处理异常情况和边界条件
- **可维护性**：清晰的控制结构使程序更易理解和修改

本节总结

- **顺序结构**：指令按书写顺序依次执行
- **分支结构**：包括单分支、双分支和多分支三种形式
- **循环结构**：包括先判断型和后判断型两种类型
- **组合使用**：三种结构可嵌套组合形成复杂程序

理解控制流是理解程序如何工作的基础，也是学习编程的重要起点。

练习答案与反馈

- **练习答案示例**：
  - **顺序结构**：收钱 → 出货 → 找零的基本流程
  - **分支结构**：根据用户选择的商品类型提供不同的商品
  - **先判断型循环**：等待用户投币，直到金额足够（先检查金额是否足够）
  - **后判断型循环**：至少提供一次商品选择机会（先显示商品选择界面）
  - **嵌套结构**：在出货过程中检查库存，如果缺货则提示并退款

3.2 霍尔逻辑

【本节目标】

- **问题驱动**：理解如何用数学方法严格描述程序的行为和正确性。
- **掌握概念**：掌握霍尔三元组的基本形式和语义含义。
- **理解关系**：理解前置条件、程序代码和后置条件之间的逻辑关系。
- **实践目标**：能够用霍尔三元组描述简单程序的正确性要求。

开头：我们要解决什么问题？

当我们编写程序时，如何确保程序确实完成了我们期望的任务？传统的测试方法只能验证有限的情况，而数学逻辑可以为我们提供更严格的保证。

霍尔逻辑（Hoare Logic）由英国计算机科学家东尼 · 霍尔（Tony Hoare）于 1969 年提出，是一种用于程序正确性验证的形式系统。它的核心工具是**霍尔三元组**，能够用数学语言精确描述程序的行为。

3.2.1 霍尔三元组的基本概念

霍尔三元组是霍尔逻辑的基本构件，其形式如下：

$$\{P\}\ C\ \{Q\}$$

其中：

- P：**前置条件**（Precondition），描述程序执行前必须满足的条件
- C：**程序代码**（Command），要验证的程序片段
- Q：**后置条件**（Postcondition），描述程序执行后应该满足的条件

霍尔三元组的数学含义是：**如果程序 C 在满足条件 P 的状态下开始执行，并且程序能够正常终止，那么执行结束后必然满足条件 Q。**

3.2.2 霍尔三元组的严格定义

为了精确理解霍尔三元组，我们需要形式化地定义其语义。设程序状态空间为 Σ，每个状态 $\sigma \in \Sigma$ 是程序变量的一个赋值。

定义 1（程序语义）：程序 C 的语义是一个部分函数 $[\![C]\!] : \Sigma \rightharpoonup \Sigma$，表示从初始状态到终止状态的映射。

定义 2（条件满足）：对于条件 P 和状态 σ，$P(\sigma)$ 表示在状态 σ 下条件 P 成立。

定义 3（霍尔三元组成立）：霍尔三元组 $\{P\}\ C\ \{Q\}$ 成立（记作 $\models \{P\}\ C\ \{Q\}$）当且仅当：

$$\forall \sigma \in \Sigma.\ \big(P(\sigma) \wedge [\![C]\!](\sigma) \text{ 有定义}\big) \Rightarrow Q([\![C]\!](\sigma))$$

这个定义可以理解为：对于所有满足前置条件 P 的初始状态 σ，如果程序 C 能够从 σ 正常终止于某个状态 σ'，那么 σ' 必须满足后置条件 Q。

3.2.3 霍尔三元组的实例分析

示例：赋值语句

$$\{\mathrm{x}, \mathrm{y} \in NC_{\mathrm{n}}, \mathrm{n} > 8, \mathrm{x} = 5\}\ \mathrm{y} := \mathrm{x} \oplus 3\ \{\mathrm{y} = 8\}$$

- **前置条件**：x, y 为状态数大于 8 的同类型数胞，x 的值等于 5
- **程序**：将 $\mathrm{x} \oplus 3$ 的值赋给 y
- **后置条件**：y 的值等于 8
- **验证**：如果 $\mathrm{x} = 5$，那么 $\mathrm{x} \oplus 3 = 8$，赋值后 $\mathrm{y} = 8$ 成立

3.2.4 霍尔三元组的强度关系

霍尔三元组之间存在逻辑上的强弱关系，这对于程序推理很重要：

定理 1（前置条件强化）：如果 $\models \{P\}\ C\ \{Q\}$ 且 $P' \Rightarrow P$，那么 $\models \{P'\}\ C\ \{Q\}$。

定理 2（后置条件弱化）：如果 $\models \{P\}\ C\ \{Q\}$ 且 $Q \Rightarrow Q'$，那么 $\models \{P\}\ C\ \{Q'\}$。

示例：

- 已知 $\mathrm{x}, \mathrm{y} \in NC_{\mathrm{n}}$，$\mathrm{n}$ 满足执行“$\mathrm{y} := \mathrm{x} \oplus 1$”之后不会产生溢出，$\{\mathrm{x} > 0\}\ \mathrm{y} := \mathrm{x} \oplus 1\ \{\mathrm{y} > 1\}$
- 由于 $\mathrm{x} > 5 \Rightarrow \mathrm{x} > 0$，根据定理 1 可得：$\{\mathrm{x} > 5\}\ \mathrm{y} := \mathrm{x} \oplus 1\ \{\mathrm{y} > 1\}$
- 由于 $\mathrm{y} > 1 \Rightarrow \mathrm{y} > 0$，根据定理 2 可得：$\{\mathrm{x} > 0\}\ \mathrm{y} := \mathrm{x} \oplus 1\ \{\mathrm{y} > 0\}$

练习：验证霍尔三元组 $\{\mathrm{x} \in NC_{\mathrm{n}}, \mathrm{n} > 25, \mathrm{x} = 10\}\ \mathrm{y} := \mathrm{x} \otimes 2; \mathrm{z} := \mathrm{y} \oplus 5\ \{\mathrm{z} = 25\}$ 是否成立。

3.2.5 霍尔逻辑的应用价值

霍尔逻辑为程序验证提供了坚实的数学基础：

- **形式化验证**：可以用数学方法证明程序的正确性，而不仅依赖测试
- **程序推理**：为编译器优化和程序分析提供理论依据
- **软件开发**：支持契约式设计和形式化方法的应用
- **安全关键系统**：在航空、医疗等需要高可靠性的领域有重要应用

本节总结

- **霍尔三元组**：$\{P\}\ C\ \{Q\}$ 描述程序 C 在前提 P 下执行后保证 Q
- **严格定义**：基于程序状态和语义函数的形式化描述
- **实例分析**：通过具体例子展示霍尔三元组的应用
- **强度关系**：前置条件可以强化，后置条件可以弱化
- **应用价值**：为程序正确性验证提供数学基础

霍尔逻辑将程序行为转化为数学命题，使得我们可以用逻辑推理的方法来确保程序的正确性，这是计算机科学中形式化方法的重要基石。

练习答案与反馈

- **练习答案**：
  - 前置条件：$\mathtt{x} \in NC_{\mathtt{n}}, \mathtt{n} > 25, \mathtt{x} = 10$
  - 第一步：$\mathtt{y} := \mathtt{x} \otimes 2$，得到 $\mathtt{y} = 20$，且不会产生溢出
  - 第二步：$\mathtt{z} := \mathtt{y} \oplus 5$，得到 $\mathtt{z} = 25$，且不会产生溢出
  - 后置条件：$\mathtt{z} = 25$ 成立
  - 因此该霍尔三元组成立

3.3 顺序结构的证明

【本节目标】

- **问题驱动**：理解如何用霍尔逻辑证明顺序执行的多条语句的正确性。
- **掌握概念**：掌握顺序结构的推理规则，理解霍尔三元组的衔接方式。
- **理解规律**：理解如何通过中间条件将多个三元组连接起来覆盖整个顺序结构。
- **实践目标**：能够为简单的顺序程序段构造完整的证明链。

开头：我们要解决什么问题？

程序中的顺序结构由多条语句依次执行组成。要证明整个顺序结构的正确性，我们需要确保每条语句的执行都满足其前后的条件，并且这些条件能够首尾衔接。霍尔逻辑提供了一条专门的推理规则来处理顺序结构。

3.3.1 顺序结构的推理规则

顺序结构的推理规则是霍尔逻辑中最基本的规则之一。对于两个顺序执行的语句 C_1 和 C_2，规则如下：

$$\frac{\{P\}\ C_1\ \{R\} \quad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}}$$

这个规则的含义是：如果从 P 执行 C_1 后能得到 R，并且从 R 执行 C_2 后能得到 Q，那么从 P 执行 C_1 后接着执行 C_2，就能得到 Q。

这里的 R 称为**中间条件**（Intermediate Condition），它既是 C_1 的后置条件，又是 C_2 的前置条件。

3.3.2 顺序结构的证明示例

示例：已知 $\mathtt{x}, \mathtt{y} \in NC_{\mathtt{n}}, n > 6$，证明程序段 $\mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{x} \oplus 1$ 在初始条件 $\mathtt{y} = 5$ 下执行后，满足 $\mathtt{y} = 6$。

我们需要找到合适的中间条件来连接两个赋值语句。

1. 第一个赋值语句：$\{\mathtt{y} = 5\}\ \mathtt{x} := \mathtt{y}\ \{\mathtt{x} = 5 \wedge \mathtt{y} = 5\}$
2. 第二个赋值语句：$\{\mathtt{x} = 5 \wedge \mathtt{y} = 5\}\ \mathtt{y} := \mathtt{x} \oplus 1\ \{\mathtt{y} = 6 \wedge \mathtt{x} = 5\}$

根据顺序规则，我们可以得到：

$$\{\mathtt{y} = 5\}\ \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{x} \oplus 1\ \{\mathtt{y} = 6 \wedge \mathtt{x} = 5\}$$

特别地，后置条件中的 $\mathtt{y} = 6$ 正是我们想要的结论。

3.3.3 多语句顺序的推广

对于更多语句的顺序执行，推理规则可以推广为：

$$\frac{\{P\}\ C_1\ \{R_1\} \quad \{R_1\}\ C_2\ \{R_2\} \quad \cdots \quad \{R_{n-1}\}\ C_n\ \{Q\}}{\{P\}\ C_1; C_2; \cdots; C_n\ \{Q\}}$$

这意味着我们需要找到一系列中间条件 $R_1, R_2, \ldots, R_{n-1}$，使得每个三元组都成立，从而保证整个顺序结构的正确性。

关键点：中间条件的选择是证明的关键，它需要精确捕获每个步骤执行后的程序状态。

练习：已知 $\mathtt{a}, \mathtt{b}, \mathtt{c} \in NC_{\mathtt{n}}$，证明程序段 $\mathtt{a} := \mathtt{b}; \mathtt{b} := \mathtt{c}; \mathtt{c} := \mathtt{a}$ 在初始条件 $\mathtt{a} = \mathtt{A} \wedge \mathtt{b} = \mathtt{B} \wedge \mathtt{c} = \mathtt{C}$ 下执行后，满足 $\mathtt{a} = \mathtt{B} \wedge \mathtt{b} = \mathtt{C} \wedge \mathtt{c} = \mathtt{B}$。

本节总结

- **推理规则**：顺序结构的证明通过中间条件衔接多个霍尔三元组
- **中间条件**：既是前一个语句的后置条件，又是后一个语句的前置条件
- **证明步骤**：为每个语句找到合适的三元组，确保条件链的连续性
- **推广性**：规则可推广到任意多个顺序执行的语句

顺序结构的证明是霍尔逻辑中最直接的部分，为更复杂的控制结构证明奠定了基础。

练习答案与反馈

- **练习答案**：
  - 第一步：$\{\mathtt{a} = \mathtt{A} \land \mathtt{b} = \mathtt{B} \land \mathtt{c} = \mathtt{C}\}\ \mathtt{a} := \mathtt{b}\ \{\mathtt{a} = \mathtt{B} \land \mathtt{b} = \mathtt{B} \land \mathtt{c} = \mathtt{C}\}$
  - 第二步：$\{\mathtt{a} = \mathtt{B} \land \mathtt{b} = \mathtt{B} \land \mathtt{c} = \mathtt{C}\}\ \mathtt{b} := \mathtt{c}\ \{\mathtt{a} = \mathtt{B} \land \mathtt{b} = \mathtt{C} \land \mathtt{c} = \mathtt{C}\}$
  - 第三步：$\{\mathtt{a} = \mathtt{B} \land \mathtt{b} = \mathtt{C} \land \mathtt{c} = \mathtt{C}\}\ \mathtt{c} := \mathtt{a}\ \{\mathtt{a} = \mathtt{B} \land \mathtt{b} = \mathtt{C} \land \mathtt{c} = \mathtt{B}\}$
- **反馈**：这个练习揭示了霍尔逻辑的一个重要价值——它具有强大的推理能力。

3.4 分支结构的证明

【本节目标】

- **问题驱动**：理解如何证明条件语句的正确性，无论执行哪个分支。
- **掌握概念**：掌握分支结构的推理规则，理解条件语句的证明方法。
- **理解规律**：理解如何确保每个分支都建立相同的后置条件。
- **实践目标**：能够为条件语句构造完整的霍尔三元组证明。

开头：我们要解决什么问题？

程序中的分支结构根据条件选择不同的执行路径。要证明分支结构的正确性，我们需要确保无论条件如何评估，每个分支的执行都能建立相同的后置条件。霍尔逻辑提供了专门的推理规则来处理条件语句。

3.4.1 分支结构的推理规则

对于条件语句 如果B 则C_1 否则C_2，推理规则如下：

$$\frac{\{P \land B\}\ C_1\ \{Q\} \quad \{P \land \neg B\}\ C_2\ \{Q\}}{\{P\}\ \text{如果}B\ \text{则}C_1\ \text{否则}C_2\ \{Q\}}$$

这个规则的含义是：如果当条件 B 为真时执行 C_1 能从 P 得到 Q，并且当条件 B 为假时执行 C_2 也能从 P 得到 Q，那么整个条件语句能从 P 得到 Q。

3.4.2 分支结构的证明示例

示例：已知 $\mathtt{x}, \mathtt{y} \in NC_n$，证明程序段：

$$\begin{cases} \mathtt{max} := \mathtt{x} & \text{如果}: \mathtt{x} > \mathtt{y} \\ \mathtt{max} := \mathtt{y} & \text{否则} \end{cases}$$

在初始条件 $\mathbf{T}$ 下执行后，满足 $\mathtt{max} = \max(\mathtt{x}, \mathtt{y})$。
我们需要证明两个分支都建立相同的后置条件。

1. 真分支：$\{\mathtt{x} > \mathtt{y}\}\ \mathtt{max} := \mathtt{x}\ \{\mathtt{max} = \max(\mathtt{x}, \mathtt{y})\}$
2. 假分支：$\{\neg(\mathtt{x} > \mathtt{y})\}\ \mathtt{max} := \mathtt{y}\ \{\mathtt{max} = \max(\mathtt{x}, \mathtt{y})\}$

根据分支规则，我们可以得到：

$$\{\mathbf{T}\}\ \text{如果}\mathtt{x} > \mathtt{y}\ \text{则}\mathtt{max} := \mathtt{x}\ \text{否则}\mathtt{max} := \mathtt{y}\ \{\mathtt{max} = \max(\mathtt{x}, \mathtt{y})\}$$

3.4.3 多分支情况的处理

对于多分支条件语句（如果-否则如果-否则），推理规则可以推广为：

$$\frac{\{P \wedge B_1\}\ C_1\ \{Q\} \quad \{P \wedge B_2\}\ C_2\ \{Q\} \quad \cdots \quad \{P \wedge B_n\}\ C_n\ \{Q\}}{\{P\}\ \text{如果}B_1\ \text{则}C_1\ \text{否则如果}B_2\ \text{则}C_2 \cdots\ \text{否则}C_n\ \{Q\}}$$

其中 $B_1, B_2, \ldots, B_n$ 是互斥且完备的条件覆盖。
关键点：每个分支都必须建立相同的后置条件 Q，这要求我们在设计程序时确保所有分支的一致性。
练习：已知 $\mathtt{x}, \mathtt{y} \in SNC_n$，证明以下程序段在初始条件 $\mathbf{T}$ 下执行后，满足 $\mathtt{y} = |\mathtt{x}|$。

$$\begin{cases} \mathtt{y} := -\mathtt{x} & \text{如果}: \mathtt{x} < 0 \\ \mathtt{y} := \mathtt{x} & \text{否则} \end{cases}$$

本节总结

- **推理规则**：分支结构的证明需要每个分支都建立相同的后置条件
- **条件处理**：真分支和假分支分别处理，但最终目标一致
- **多分支扩展**：规则可推广到任意多个分支的情况
- **设计启示**：要求所有分支都指向相同的后置条件，这促进了程序的一致性设计

分支结构的证明确保了程序的条件逻辑能够正确实现预期功能，无论运行时条件如何评估。

练习答案与反馈

- **练习答案**：
  - 真分支：$\{\mathtt{x} < 0\}\ \mathtt{y} := -\mathtt{x}\ \{\mathtt{y} = |\mathtt{x}|\}$
  - 假分支：$\{\neg(\mathtt{x} < 0)\}\ \mathtt{y} := \mathtt{x}\ \{\mathtt{y} = |\mathtt{x}|\}$
  - 因此：$\{\mathbf{T}\}$ 如果$\mathtt{x} < 0$ 则$\mathtt{y} := -\mathtt{x}$ 否则$\mathtt{y} := \mathtt{x}\ \{\mathtt{y} = |\mathtt{x}|\}$
- **反馈**：这个例子展示了霍尔逻辑如何用于证明基本算法的正确性，如计算绝对值。

3.5 循环结构的证明

【本节目标】

- **问题驱动**：理解如何证明循环语句的正确性，包括终止性和部分正确性。
- **掌握概念**：掌握循环不变式的概念和作用，理解循环的推理规则。
- **理解规律**：理解循环不变式如何在迭代过程中保持成立。
- **实践目标**：能够为简单循环找出合适的不变式并构造证明。

开头：我们要解决什么问题？

循环结构是程序中最复杂的控制结构之一，因为它涉及不确定次数的重复执行。要证明循环的正确性，我们需要找到一种在循环每次迭代前后都成立的条件——循环不变式（Loop Invariant）。霍尔逻辑提供了基于循环不变式的推理规则。

3.5.1 循环不变式的概念

循环不变式是一个在循环执行过程中保持”不变”的条件。具体来说：

- 在循环开始前成立
- 在循环每次迭代后仍然成立
- 当循环终止时，可以帮助我们建立最终的后置条件

循环不变式捕获了循环的抽象本质，是理解循环行为的关键。

3.5.2 循环结构的推理规则

对于循环语句 当B 执行C，推理规则如下：

$$\frac{\{I \wedge B\}\ C\ \{I\}}{\{I\}\ \text{当}B\ \text{执行}C\ \{I \wedge \neg B\}}$$

这个规则的含义是：如果循环体 C 在条件 $I \wedge B$ 下执行后能保持 I 成立，那么整个循环在初始满足 I 的情况下执行后，将满足 $I \wedge \neg B$。
其中 I 是循环不变式，B 是循环条件。

3.5.3 循环结构的证明示例

示例：已知 n 为任意自然数，证明以下计算阶乘的程序段满足后置条件 $\mathtt{f} = \mathtt{n}!$。

$$\mathtt{i} := NC_\infty(1)$$

$$\mathtt{f} := NC_\infty(1)$$

$$\left[\begin{array}{l} \text{当} : \mathtt{i} \leq \mathtt{n} \\ \quad \mathtt{f} :=^{\otimes} \mathtt{i} \\ \quad \mathtt{i} :=^{\oplus} 1 \end{array}\right]$$

在这个程序段中，我们使用了 NC_∞ 这个数胞类型，这是一种理想中的数胞，它的状态数无穷大，等价于自然数集。因此这种数胞无法代表存储空间，因为没有存储空间拥有无限的状态数量。实际上，NC_∞ 是可以用数据结构实现的，在学习数据结构之后可做解释。在这里，我们因暂时需要，假设类型为 NC_∞ 的变量就是自然数变量，且我们不关心它的实现。

我们需要找到合适的循环不变式。观察发现：在循环过程中，f 总是等于 $\mathtt{i} - 1$ 的阶乘，即 $\mathtt{f} = (\mathtt{i} - 1)!$。
因此，循环不变式 I 为：$\mathtt{f} = (\mathtt{i} - 1)! \wedge \mathtt{i} \leq \mathtt{n} + 1$
证明步骤：

1. 初始化：在循环开始前，$\mathtt{i} = 1, \mathtt{f} = 1$，有 $\mathtt{f} = (1-1)! = 0! = 1$ 成立，且 $\mathtt{i} = 1 \leq \mathtt{n}+1$ 成立

2. 保持：假设 I 成立且 $\mathtt{i} \leq \mathtt{n}$，执行循环体后：

   - $\mathtt{f} :=^{\otimes} \mathtt{i}$，此时 $\mathtt{f} = (\mathtt{i} - 1)! \times \mathtt{i} = \mathtt{i}!$
   - $\mathtt{i} :=^{\oplus} 1$，此时 i 相比于之前增加 1。
   - 因此新状态满足 $\mathtt{f} = (\mathtt{i} - 1)!$

3. 终止：当循环终止时，$\mathtt{i} > \mathtt{n}$ 且 I 成立，即 $\mathtt{f} = (\mathtt{i} - 1)! \wedge \mathtt{i} = \mathtt{n} + 1$，所以 $\mathtt{f} = \mathtt{n}!$

3.5.4 循环证明的完整性

完整的循环证明还需要考虑循环的终止性（即循环最终会结束）。霍尔逻辑的上述规则只保证了循环的**部分正确性**——如果循环终止，那么结果正确。要证明**完全正确性**，还需要证明循环一定会终止，这通常通过找到一个随循环迭代而递减的界函数来实现。

练习：已知 $\mathtt{n}$ 为任意自然数，考虑以下计算平方和的程序段。找出合适的循环不变式并证明后置条件 $\mathtt{s} = \sum_{\mathtt{k}=1}^{\mathtt{n}} \mathtt{k}^2$。

$$\mathtt{i} := NC_{\infty}(1)$$

$$\mathtt{s} := NC_{\infty}(0)$$

$$\left[\begin{array}{c} \text{当} : \mathtt{i} \leq \mathtt{n} \\ \mathtt{s} :=^{\oplus} \mathtt{i} \otimes \mathtt{i} \\ \mathtt{i} :=^{\oplus} 1 \end{array}\right]$$

本节总结

- **循环不变式**：循环证明的核心，是在循环过程中保持成立的条件
- **推理规则**：基于不变式，保证循环终止后建立期望的后置条件
- **证明步骤**：初始化、保持、终止三个步骤
- **完全正确性**：还需要证明循环终止性（通过界函数）

循环结构的证明是霍尔逻辑中最具挑战性的部分，但也是最能体现形式化方法威力的部分。

练习答案与反馈

- **练习答案**：
  - 循环不变式：$\mathtt{s} = \sum_{\mathtt{k}=1}^{\mathtt{i}-1} \mathtt{k}^2 \wedge \mathtt{i} \leq \mathtt{n} + 1$
  - 初始化：$\mathtt{i} = 1, \mathtt{s} = 0$，有 $\mathtt{s} = \sum_{\mathtt{k}=1}^{0} \mathtt{k}^2 = 0$ 成立
  - 保持：假设不变式成立且 $\mathtt{i} \leq \mathtt{n}$，执行循环体后：
    - 执行 $\mathtt{s} :=^{\oplus} \mathtt{i} \otimes \mathtt{i}$ 之后 $\mathtt{s} = \sum_{\mathtt{k}=1}^{\mathtt{i}-1} \mathtt{k}^2 + \mathtt{i}^2 = \sum_{\mathtt{k}=1}^{\mathtt{i}} \mathtt{k}^2$
    - $\mathtt{i} :=^{\oplus} 1$
    - 新状态满足 $\mathtt{s} = \sum_{\mathtt{k}=1}^{\mathtt{i}-1} \mathtt{k}^2$
  - 终止：当循环终止时，$\mathtt{i} > \mathtt{n}$ 且 $\mathtt{i} = \mathtt{n} + 1$，所以 $\mathtt{s} = \sum_{\mathtt{k}=1}^{\mathtt{n}} \mathtt{k}^2$
- **反馈**：这个练习展示了循环不变式如何捕获循环的累积效应，是理解循环行为的强大工具。

4 算法与数据结构

4.1 指针

【本节目标】

- **问题驱动**：理解如何通过间接方式访问和操作存储空间中的数据。
- **掌握概念**：掌握指针的基本定义、指针运算的含义和规则。
- **理解关系**：理解指针与存储空间地址的对应关系。
- **实践目标**：能够分析指针运算的结果，理解指针的间接访问特性。

开头：我们要解决什么问题？

想象一下，你有一个巨大的仓库，里面存放着各种物品。每个物品都有一个唯一的编号（地址）。现在，你需要告诉搬运机器人去取某个特定编号的物品。你有两种方法：

- **直接方式**：直接告诉机器人” 去取 2056 号物品”
- **间接方式**：在一张纸条上写下”2056”，然后告诉机器人” 去取纸条上写的编号对应的物品”

第二种方法就是指针的基本思想。指针就像是那张写着地址的纸条，它本身不包含实际数据，而是告诉我们到哪里可以找到数据。

4.1.1 指针的基本概念

指针是一种特殊的变量，它的值不是直接的数据内容，而是另一个数据在存储空间中的地址。

- **指针变量**：存储空间中的一个存储单元，专门用来存放地址值，本质上是一个存放地址的数胞变量，这个数胞变量的状态数通常是 $\mathtt{BASE}^{\mathtt{BYTELENGTH}\times\mathtt{WORDSIZE}}$，即计算机字长代表的存储空间的状态数。在按规范操作的情况下，这个值不会溢出。因此计算指针的地址时通常忽略状态数的限制。
- **指针值**：指针变量中存储的具体地址数值
- **指向关系**：如果指针 `P` 的值是地址 A，我们就说 `P` 指向地址 A 处的数据
- **间接访问**：通过指针访问其指向的数据称为间接访问或解引用

比喻理解：

- **地址**：就像房子的门牌号（如” 人民路 123 号”）

- **数据**：就像房子里住的人或存放的物品
- **指针**：就像一张写着地址的便签（上面写着” 人民路 123 号”）
- **间接访问**：按照便签上的地址去找对应的房子

示例：

- 假设在地址 1000 处存储了一个整数 42
- 在地址 2000 处有一个指针变量 P，其值为 1000
- 那么我们就说：指针 P（位于地址 2000）指向地址 1000 处的整数 42

4.1.2 指针的基本运算

在伪代码中，我们使用 PTR 表示指针类型：

$$PTR_{\mathtt{T}}$$

用于表示指向 T 类型数据的指针。并使用 $\odot$ 表示空指针，表示不指向任何有效地址的指针：

因而我们可以写出以下伪代码：

$$\mathtt{p} := PTR_{NC_2}(\odot)$$

表示变量 p 被定义为一个指向 NC_2 类型数据的指针，指针值为空指针的值，其中默认指针值的类型是 $NC_{\mathtt{WORDSIZEVALUE}}, \mathtt{WORDSIZEVALUE} = \mathtt{BASE}^{\mathtt{BYTELENGTH}\times\mathtt{WORDSIZE}}$。

$$\mathtt{p} \in PTR_{NC_2}$$

表示判断变量 p 是否是一个指向 NC_2 类型数据的指针。

在伪代码中，说明几种与指针相关的特殊的运算，这些运算都与地址计算相关：

1. 取地址运算

取地址运算获取某个数据的存储地址。

- **符号**：用 & 表示
- **含义**：设变量 X 的类型为 T，X 的存储地址为 a，$\&\mathtt{X} = PTR_{\mathtt{T}}(\mathtt{a})$
- **结果类型**：取地址运算的结果是一个指针

示例：如果 NC_6 数胞变量 X 存储在地址 3000 处，那么 $\&\mathtt{X} = PTR_{\mathtt{NC_6}}(3000)$。

2. 解引用运算

解引用运算通过指针访问其指向的数据。

- **符号**：用 $*$ 表示
- **含义**：$*$P 表示访问指针 P 所指向地址处的数据。
- **要求**：P 必须是一个有效的指针（即其值是一个合法的地址，其类型是一个有效的类型）

示例：如果 P $= PTR_{NC_{100}}(3000)$，且地址 3000 处存储着 42，那么 $*$P 的结果就是 42。

3. 申请空间运算

申请空间运算用于从计算机中获取一块存储空间，这块存储空间一定是之前没有被使用过的，且申请得到的地址一定是有效的。

- **符号**：用 $\uparrow$ 表示
- **含义**：$\uparrow$ T 表示申请一块大小为 sizeof(T) 个字节的存储空间，并输出存储空间的起始地址，其中 T 为数据类型。

4. 释放空间运算

释放空间运算用于将申请空间运算获取的存储空间释放回计算机，使用申请空间运算后必须使用释放空间运算把存储空间返回给计算机。如果示例程序中出现申请后未释放的情况，则说明示例程序将释放空间运算省略了。

- **符号**：用 $\downarrow$ 表示
- **含义**：$\downarrow$P 表示释放存储空间 P 所占的空间，释放的大小取决于之前使用申请空间运算获取的存储空间的字节数。
- **要求**：P 必须为申请空间运算获取的指针，且 P 没有被释放过

3. 指针算术运算

指针算术运算基于指向的数据类型大小进行地址计算。

指针加减整数：

- P$\boxplus$n：输出指针 P 的值增加 n $\times$ sizeof(数据类型) 个字节后的值
- P$\boxminus$n：输出指针 P 的值减少 n $\times$ sizeof(数据类型) 个字节后的值

- 其中 sizeof(数据类型) 由先前的申请空间运算决定，如果使用的是 ↑ T 的方式申请，那么 sizeof(数据类型) = sizeof(T)，否则默认 sizeof(数据类型) = 1。

示例：假设 P 指向一个占用 4 字节的数胞，且 P 的当前值是 1000。且地址 0 到 1015 都是有效的。

- P ⊞ 1 的结果是 1004（$1000 + 1 \times 4$）
- P ⊞ 3 的结果是 1012（$1000 + 3 \times 4$）
- P ⊟ 2 的结果是 992（$1000 - 2 \times 4$）

注意：请在确保指针加减之后的结果不会导致溢出时，再进行指针加减运算。例如，已知指针 P 指向有效的地址，且 P ⊞ 3 依然指向有效的地址，此时才允许进行指针加减运算。

指针相减：

- P ⊟ Q：计算两个指针之间相差的数据元素个数
- 结果 = (SNC(P) ⊖ SNC(Q)) ÷ sizeof(数据类型)
- 其中要求 P 和 Q 指向的数据类型是相同的。例如，若 P 指向 NC_{345} 类型的数据，则 Q 必须也指向 NC_{345} 类型的数据。

示例：如果 P 指向地址 1000，Q 指向地址 1012，且指向的数据的类型相同，数据类型大小为 4 字节。

- P ⊟ Q = $(1000 - 1012) \div 4 = (-12) \div 4 = -3$
- 这表示 P 在 Q 之前 3 个元素的位置

4. 指针比较运算

指针可以进行比较运算，判断它们之间的位置关系。

- P = Q：判断两个指针是否指向同一个地址
- P ≠ Q：判断两个指针是否指向不同的地址
- P < Q：判断 P 是否指向 Q 之前的地址
- P > Q：判断 P 是否指向 Q 之后的地址

注意：指针比较通常只在指向同一块连续存储空间时才有意义。

指针运算的意义

这些公式表明，指针运算会自动考虑数据类型的大小，这使得程序员可以以数据元素为单位进行思考，而不需要关心具体的字节偏移量。

练习 1：假设有一个指针 P 指向地址 2000，数据类型大小为 8 字节，地址 0 到 2024 都是有效的。

(a) 计算 P 田 2 的值

(b) 计算 P 曰 1 的值

(c) 如果另一个指针 Q 的值是 2024，计算 Q 曰 P 的值

练习 2：考虑以下情景：

- 变量 X 存储在地址 3000 处，值为 100
- 指针 P 存储在地址 4000 处，其值为 3000
- 指针 Q 存储在地址 5000 处，其值也是 3000

请回答：

(a) ∗P 的值是多少?

(b) P = Q 的结果是真还是假?

(c) &X 的值是多少?

重要注意事项：

- **空指针**：不指向任何有效地址的指针，通常用特殊值（如 0）表示
- **野指针**：指向无效或已释放存储空间的指针，使用野指针会导致错误
- **类型安全**：指针通常与特定数据类型关联，确保访问的正确性

本节总结

- **指针本质**：存储地址的变量，提供间接访问数据的能力
- **基本运算**：取地址 (&)、解引用 (∗)、算术运算 (田, 曰)、比较运算
- **运算规则**：基于数据类型大小进行地址计算
- **核心价值**：实现动态性、共享访问和复杂数据结构的构建
- **安全使用**：注意空指针和野指针的问题

理解指针是理解计算机如何高效管理存储空间的关键，也是学习更高级数据结构和算法的基础。

练习答案与反馈

- **练习 1 答案**:
  (a) `P` 田 $2 = 2000 + 2 \times 8 = 2000 + 16 = 2016$
  (b) `P` 曰 $1 = 2000 - 1 \times 8 = 2000 - 8 = 1992$
  (c) `Q` 曰 `P` $= (2024 - 2000) \div 8 = 24 \div 8 = 3$
- **练习 2 答案**:
  (a) `*P` 访问地址 3000 处的值，所以是 100
  (b) `P` 和 `Q` 的值都是 3000，所以 `P` = `Q` 为真
  (c) `&X` 是变量 `X` 的地址，所以是 3000
- **反馈**：指针运算的关键是理解" 地址" 和" 数据" 的区别，以及指针运算自动考虑数据类型大小的特性。

4.2 在伪代码中定义规则

到目前为止，我们已经在伪代码中定义了很多规则。但是，这些规则在面对越来越多的需求时，一定是不够方便的。因此，我们允许伪代码的编写者自行定义规则。但是要满足以下要求:

- 新的规则本身不会产生歧义或意义不明确的情况。
- 新的规则不会与已有的规则冲突，即不会产生歧义或意义不明确的情况。

我们规定新的定义满足以下格式:

定义 : 定义内容

其中 定义内容 可以是任意描述定义的自然语言或数学语言。但是需要保证满足以上两点要求。

4.3 结构体

【本节目标】

- **问题驱动**：理解如何将不同类型的数据组织成一个逻辑整体。
- **掌握概念**：掌握结构体的定义、成员访问和对齐规则。
- **理解特性**：理解结构体在存储空间中的布局方式。
- **实践目标**：能够计算结构体的大小，理解对齐规则对存储空间使用的影响。

开头：我们要解决什么问题？

在实际应用中，我们经常需要将多个相关的数据项组合在一起。例如，描述一个学生可能需要姓名、学号、年龄等多个信息。如果使用单独的变量来表示，这些数据之间缺乏逻辑联系，管理起来很不方便。结构体就是为了解决这个问题而引入的。

4.3.1 结构体的基本概念

结构体是一种用户自定义的数据类型，它允许将多个不同类型的数据项组合成一个单一的逻辑单元。

- **结构体定义**：描述结构体包含哪些数据项以及它们的类型
- **结构体实例**：根据结构体定义创建的具体数据对象
- **成员变量**：结构体中包含的各个数据项
- **成员访问**：通过结构体实例访问其成员变量的操作

比喻理解：

- **结构体定义**：就像一张表格模板，规定了需要填写哪些信息
- **结构体实例**：就像按照模板填写好的一张具体表格
- **成员变量**：就像表格中的各个填写项（如姓名、年龄等）

4.3.2 结构体的存储布局

结构体在存储空间中是连续存储的，但有一个重要的特性：**对齐规则**。

对齐规则的基本原理：

- 某些计算机体系结构要求特定类型的数据必须从特定的地址边界开始存储
- 例如，4 字节整数可能要求从 4 的倍数的地址开始存储
- 对齐可以提高数据访问的效率，但可能导致存储空间浪费

示例：考虑一个包含两个成员的结构体：

- 成员 1：1 字节的小数胞
- 成员 2：4 字节的大数胞

无对齐情况（理论上的紧凑布局）：

```
地址：1000 [小数胞]
地址：1001 [大数胞字节1]
地址：1002 [大数胞字节2]
地址：1003 [大数胞字节3]
地址：1004 [大数胞字节4]
总大小：5字节
```

有对齐情况（实际中的典型布局）：

```
地址：1000 [小数胞]
地址：1001 [填充字节] -- 对齐填充
地址：1002 [填充字节] -- 对齐填充
地址：1003 [填充字节] -- 对齐填充
地址：1004 [大数胞字节1] -- 从4的倍数地址开始
地址：1005 [大数胞字节2]
地址：1006 [大数胞字节3]
地址：1007 [大数胞字节4]
总大小：8字节
```

对齐规则不是固定不变的，它取决于具体的执行环境。

4.3.3 结构体大小的计算

计算结构体大小需要遵循以下步骤：

计算步骤：

1. 确定每个成员的类型大小和对齐要求
2. 按照声明顺序放置成员，每个成员从合适对齐的地址开始
3. 在成员之间插入必要的填充字节以满足对齐要求
4. 在结构体末尾添加填充字节，使总大小成为最大对齐要求的整数倍

示例计算：

- 对齐规则：按类型大小对齐，结构体的对齐要求等于其最大成员的对齐要求

结构体定义：结构体 Example 有三个成员，分别为 1 字节的 a，4 字节的 b 和 2 字节的 c。

大小计算过程：

1. 起始地址：0

2. 成员 a：大小 1 字节，地址 0（0 是 1 的倍数）

3. 成员 b：大小 4 字节，需要 4 的倍数地址，下一个可用地址是 4

4. 在 a 和 b 之间插入 3 字节填充（地址 1-3）

5. 成员 c：大小 2 字节，地址 8（8 是 2 的倍数）

6. 结构体总大小：目前到地址 9

7. 最大对齐要求是 4，总大小需要是 4 的倍数

8. 在末尾添加 2 字节填充（地址 10-11），总大小 =12 字节

4.3.4 结构体的操作

结构体支持以下几种基本操作：

- **成员访问**：通过结构体变量访问其成员
- **赋值操作**：整个结构体可以作为单位进行赋值

在伪代码中，我们通过如下格式定义结构体：

$$\left[\begin{array}{c} \text{结构体}:\text{结构体名称}\{\text{类型 1},\ldots,\text{类型 n}\} \\ \text{成员 1} \in \text{成员 1 的类型} \\ \cdots \\ \text{成员 n} \in \text{成员 n 的类型} \end{array}\right]$$

其中 结构体名称 可以自由命名，成员 1 到 成员 n 也可以自由命名，成员 1 的类型 到 成员 n 的类型 都是可以代表数据类型的集合，如 NC_2、SNC_8。
{类型 1,...,类型 n} 可以被省略，如果未被省略，类型 1 到 类型 n 可以自由命名，用于代表一个数据类型。结构体中 成员 1 的类型 到 成员 n 的类型 可以使用 类型 1 到 类型 n 指代具体的类型。在除了定义结构体时，{类型 1,...,类型 n} 中的类型必须明写为确定的数据类型。已定义的结构体也可以被视为一个类型，可写为：

$$\text{结构体名称}\{\text{类型 1},\ldots,\text{类型 n}\}$$

假设已定义结构体 S，我们可以使用：

$$\text{s} \in \text{S}$$

当 s 在前文中未定义时，意为声明新变量 s 是一个 S 的实例。当 s 在前文中已定义时，意为判断 s 的类型是否为 S，输出布尔值。

假设已定义结构体 S，S 中存在成员 m，对于一个 S 的实例 s，可以使用：

$$\mathrm{s} \to m$$

访问实例 s 中的成员变量 m。

假设已定义结构体 S，对于两个 S 的实例 $\mathrm{s}_1, \mathrm{s}_2$，可以使用：

$$\mathrm{s}_1 := \mathrm{s}_2$$

使实例 s_1 中的所有成员变量等于 s_2 中所有同名的成员变量。

假设已定义结构体 S，对于两个 S 的实例 $\mathrm{s}_1, \mathrm{s}_2$，可以使用：

$$\mathrm{s}_1 = \mathrm{s}_2$$

来判断 s_1 的所有成员变量是否等于 s_2 中所有同名的成员变量，输出为布尔值。

$$\mathrm{s}_1 \neq \mathrm{s}_2$$

来判断是否存在至少一个 s_1 的成员变量不等于 s_2 中同名的成员变量，输出为布尔值。

练习 1：考虑此结构体定义：结构体 S1 有三个成员，分别为 1 字节的 a，2 字节的 b 和 1 字节的 c。

对齐规则：成员按类型大小对齐，设计算机字长为 4 字节，结构体的大小是字长的整数倍。

(a) 计算这个结构体的大小

(b) 画出结构体在存储空间中的布局图

练习 2：同样的结构体定义，但在紧凑打包模式下（无对齐填充）：

(a) 计算这个结构体的大小

(b) 比较两种模式下的大小差异

4.3.5 结构体的应用场景

结构体在程序设计中有着广泛的应用：

- **数据记录**：表示数据库记录、配置文件项等
- **复杂对象**：表示图形界面中的控件、游戏中的实体等

重要注意事项：

- **内存使用**：成员顺序影响结构体大小，合理安排成员顺序可以节省内存

本节总结

- **结构体本质**：将相关数据项组织成逻辑整体的复合数据类型
- **对齐规则**：成员按特定边界对齐，提高访问效率但可能浪费空间
- **环境依赖性**：对齐规则因硬件架构、编译器设置等因素而异
- **大小计算**：需要考虑成员顺序、对齐要求和填充字节
- **应用价值**：提高代码可读性、组织复杂数据、映射硬件等

理解结构体及其对齐规则对于编写高效、可移植的代码至关重要，特别是在系统编程和性能敏感的应用中。

练习答案与反馈

- **练习 1 答案**：
  (a) 大小计算：
    1. 成员 a：地址 0，大小 1 字节
    2. 成员 b：需要 2 的倍数地址，下一个是 2，地址 1 填充 1 字节
    3. 成员 b：地址 2-3，大小 2 字节
    4. 成员 c：地址 4，大小 1 字节
    5. 总大小目前 5 字节，最大对齐要求 4 字节，需要 4 的倍数
    6. 地址 5-7 填充 3 字节，总大小 =8 字节

  (b) 布局图：

```
地址0: [a]
地址1: [填充]
地址2: [b字节1]
地址3: [b字节2]
地址4: [c]
地址5: [填充]
地址6: [填充]
地址7: [填充]
```

- **练习 2 答案**：
  (a) 紧凑模式下：$1 + 2 + 1 = 4$ 字节
  (b) 差异：紧凑模式节省 4 字节（8 vs 4），但可能降低访问效率

- **反馈**：结构体对齐的关键是理解”对齐边界”的概念，以及如何通过合理安排成员顺序来优化存储空间使用。在实际编程中，需要在空间效率和时间效率之间做出权衡。

4.4 联合体

【本节目标】

- **理解概念**：掌握联合体的基本定义和核心特性。
- **发现不同**：理解联合体与结构体的本质区别。
- **掌握计算**：能够计算联合体的大小。
- **理解原理**：理解联合体成员共享存储空间的含义。

开头：我们要解决什么问题？

想象一下，你有一个盒子，这个盒子的大小刚好能放下你最大的玩具。今天你可能想在里面放一个皮球，明天你可能想换成一堆积木。虽然每次只能放一种玩具，但盒子还是那个盒子——这就是联合体的基本思想。

联合体让不同的数据可以共用同一个”盒子”（存储空间），根据你的需要，这个”盒子”可以被解释为不同的类型。

4.4.1 什么是联合体？

联合体是一种让多个不同类型的数据共享同一块存储空间的方式。

- **共享空间**：所有成员都从同一个位置开始存储
- **单次使用**：同一时间只能存放其中一个成员的值
- **大小适配**：联合体的大小刚好能放下最大的成员

简单对比：

- **结构体**：像一个大书包，每个隔层放不同的书本，所有书本同时存在
- **联合体**：像一个小袋子，每次只能放一本书，但可以换不同的书

4.4.2 联合体的大小

联合体的大小由其最大的成员决定。如果一个联合体有 $\texttt{k}$ 个成员，它们的大小分别是 $\texttt{size}_1, \texttt{size}_2, \ldots, \texttt{size}_\texttt{k}$，那么联合体的大小是：

$$\texttt{size}_{\text{union}} = \max(\texttt{size}_1, \texttt{size}_2, \ldots, \texttt{size}_\texttt{k})$$

例子：

- 如果联合体有一个占 4 字节的成员和一个占 8 字节的成员
- 那么联合体的大小就是 8 字节（两个中较大的那个）

4.4.3 联合体如何工作？

联合体的所有成员都从内存中的同一个地址开始存储。这意味着：

- 给一个成员赋值会覆盖其他成员的值
- 读取一个成员时，解释的是同一块存储空间的数据
- 同一时间只有一个成员的值是有效的

小例子：一个联合体有两个成员，一个 4 字节的” 数胞 1”，和另一个 4 字节的” 数胞 2”. 这个联合体的两个成员共享 4 字节空间。如果你给” 数胞 1” 成员赋值，那么” 数胞 2” 的 4 个字节也会被改变，因为它们其实是同一块数据。

4.4.4 为什么需要标签？

由于联合体可以存放不同类型的值，在证明中，我们通常需要一个” 标签” 来记录当前存放的是哪种类型。

标签的作用：

- 记录当前联合体中存储的是哪个成员
- 确保我们以正确的方式使用联合体

使用规则：

1. 存入一个值后，要设置标签指明存的是哪个成员
2. 读取值时，要先检查标签确认当前存储的是哪个成员

假设我们有一个联合体，可以存放两种数据：一个数字或者一组字符。

使用方法：

- 如果要存数字：先存数字，然后把标签设为” 数字”
- 如果要读数字：先检查标签是不是” 数字”，然后再读

在伪代码中，我们通过如下格式定义联合体：

$$\left[\begin{array}{c} \text{联合体}：\text{联合体名称}\{\text{类型 1},\ldots,\text{类型 n}\} \\ \text{成员 1} \in \text{成员 1 的类型} \\ \cdots \\ \text{成员 n} \in \text{成员 n 的类型} \end{array}\right]$$

其中 联合体名称 可以自由命名，成员 1 到 成员 n 也可以自由命名，成员 1 的类型 到 成员 n 的类型 都是可以代表数据类型的集合。
{类型 1, . . . , 类型 n} 可以被省略，如果未被省略，类型 1 到 类型 n 可以自由命名，用于代表一个数据类型。联合体中 成员 1 的类型 到 成员 n 的类型 可以使用 类型 1 到 类型 n 指代具体的类型。在除了定义结构体时，{类型 1, . . . , 类型 n} 中的类型必须明写为确定的数据类型。已定义的联合体也可以被视为一个类型，可写为：

$$\text{联合体名称}\{\text{类型 1}, \ldots, \text{类型 n}\}$$

假设已定义联合体 U，我们可以使用：

$$\mathrm{u} \in \mathrm{U}$$

当 u 在前文中未定义时，意为声明新变量 u 是一个 U 的实例。当 u 在前文中已定义时，意为判断 u 的类型是否为 U，输出布尔值。

假设已定义结构体 U，U 中存在成员 m，对于一个 U 的实例 u，可以使用：

$$\mathrm{u} \to m$$

访问实例 u 中的成员变量 m。

假设已定义结构体 U，对于两个 U 的实例 $\mathrm{u}_1, \mathrm{u}_2$，可以使用：

$$\mathrm{u}_1 := \mathrm{u}_2$$

使实例 u_1 中的存储空间存储的值等于 u_2 中的存储空间存储的值。

假设已定义结构体 U，对于两个 U 的实例 $\mathrm{u}_1, \mathrm{u}_2$，可以使用：

$$\mathrm{u}_1 = \mathrm{u}_2$$

来判断 u_1 中的存储空间存储的值是否等于 u_2 中的存储空间存储的值，输出为布尔值。

$$\mathrm{s}_1 \neq \mathrm{s}_2$$

来判断 s_1 中的存储空间存储的值是否不等于 u_2 中的存储空间存储的值，输出为布尔值。

本节总结：

- **共享空间**：联合体的成员共用同一块存储空间
- **大小最大**：联合体的大小等于最大成员的大小
- **需要标签**：证明中通常需要标签来记录当前有效的成员
- **类型灵活**：同一块数据可以被解释为不同的类型

联合体提供了让数据” 一变多用” 的能力，但需要配合标签来确保正确使用。
小练习：如果一个联合体有三个成员，大小分别是 2 字节、4 字节和 6 字节，这个联合体有多大?

练习答案

- **练习答案**：联合体的大小是 6 字节，因为要能放下最大的成员。

4.5 存储空间的数学抽象

【本节目标】

- **理解概念**：掌握用数学方式描述存储空间的核心思想。
- **掌握表示**：学会使用数学符号表示存储空间的状态和状态变化。
- **发现价值**：理解数学抽象如何帮助我们精确描述和推理程序行为。
- **建立联系**：将数学抽象与之前学习的指针、结构体等概念联系起来。

开头：我们要解决什么问题？

在我们学习了指针、结构体和联合体之后，你已经知道程序是如何在存储空间这片”大画布” 上精巧地排布数据的。但有一个更深层次的问题：当我们改变一个变量的值时，到底发生了什么？我们能否用严谨的数学语言来描述这幅” 画作” 以及我们对它的每一次修改？

想象一下，你有一张可以无限复制的神奇画布。每次你想修改画上的某个部分时，你不是直接涂改原画，而是复制一份几乎完全相同的画，只在新画上做出细微的改动。这样，你就同时拥有了修改前和修改后的两个版本。这种思考方式就是数学描述存储空间变化的核心思想。

4.5.1 核心抽象：存储空间是一张” 地址-值” 对应表

数学家们最经典的模型，是将计算机的整个存储空间想象成一张巨大的、可变的对应表。

- **地址 (L)**：就像是画布上每一个格子的编号。这其实就是指针所保存的值。
- **值 (V)**：就是存放在这些格子里的数据。它可以是一个简单的数字，也可以是一个复杂的结构体或联合体。
- **状态 (σ)**：在某一时刻，整个存储空间的完整情况。数学上，它就是一个函数，将每个地址 (编号) 映射到对应的值 (格子里的内容)。我们可以把它记作 $\sigma : L \to V$。

举个例子：假设地址 2024 这个格子里存着数字 18(比如一个 `age` 变量)，地址 3000 这个格子里存着地址 2024(比如一个 $\mathrm{p_{age}}$ 指针变量)。那么此刻的存储空间状态 σ 就可以表示为：

$$\sigma = \{\ldots, 2024 \to 18, 3000 \to 2024, \ldots\}$$

4.5.2 改变存储空间的数学解释

在现实中，我们改变一个变量的值，像是把格子里的旧东西拿出来，放进新东西。但在数学的永恒世界里，”改变”这个概念被巧妙地重新解释了：它不是修改旧状态，而是基于旧状态创造一个新状态。

举个例子：

假设初始状态 σ_0 是：$\sigma_0 = \{\mathtt{l_x} \to 5, \mathtt{l_y} \to 10\}$($\mathtt{l_x}$ 和 $\mathtt{l_y}$ 是两个变量的地址)。现在执行一句代码：$\mathtt{x} := \mathtt{y} + 1$. 这个操作在数学上会产生一个全新的状态 σ_1。σ_1 在除了 $\mathtt{l_x}$ 之外的所有地址上，都和 σ_0 一模一样。唯独在 $\mathtt{l_x}$ 这个地址上，它的值被”更新”为了 $\sigma_0(\mathtt{l_y}) + 1$(也就是 $10 + 1 = 11$)。

我们用一种特殊的数学符号来表示这个”除了某一点不同，其余都相同”的新状态：

$$\sigma_1 = \sigma_0[\mathtt{l_x} \to \sigma_0(\mathtt{l_y}) + 1]$$

所以，$\sigma_1 = \{\mathtt{l_x} \to 11, \mathtt{l_y} \to 10\}$。这就好像在说：程序员每一次修改存储空间，并不是在涂改原来的画，而是重新画了一幅和之前几乎一模一样的画，只在一处做了细微的改动。数学通过这种方式，在它”静止”的世界里，精准地描述了”变化”。

4.5.3 为什么这种抽象很重要？

这种数学抽象为我们提供了强大的工具：

- **精确描述**：可以精确描述程序在任何时刻的状态
- **推理证明**：可以证明程序的正确性，确保它按预期工作
- **理解变化**：可以清晰理解每次操作如何改变存储空间

这种抽象让我们能够像数学家一样思考程序的行为，而不仅仅是像工程师一样编写代码。

练习：假设初始状态 $\sigma_0 = \{1000 \to 5, 1004 \to 8\}$。执行操作 $\mathtt{y} := \mathtt{x} + 3$ 后（假设 $\mathtt{x}$ 在地址 1000，$\mathtt{y}$ 在地址 1004），新状态 σ_1 是什么？

本节总结

- **数学视角**：存储空间可以被看作一张”地址-值”的对应表
- **状态表示**：用函数 $\sigma : L \to V$ 表示某一时刻的完整存储状态
- **变化描述**：状态变化不是修改旧状态，而是创建新状态 $\sigma_{\text{新}} = \sigma_{\text{旧}}[\text{地址} \to \text{新值}]$
- **抽象价值**：提供了精确描述和推理程序行为的数学工具
- **实际联系**：这种抽象帮助我们更好地理解指针、变量赋值等实际操作

数学抽象让我们能够用严谨的方式思考程序如何操作存储空间，这是理解计算机科学深层原理的重要一步。

练习答案

- **练习答案**：$\sigma_1 = \sigma_0[1004 \to \sigma_0(1000) + 3] = \{1000 \to 5, 1004 \to 8\}[1004 \to 5 + 3] = \{1000 \to 5, 1004 \to 8\}$

4.6 算法

【本节目标】

- **理解概念**：从数学函数出发，理解算法的基本概念和特性。
- **掌握扩展**：理解算法在计算机中的实现方式和特殊性质。
- **建立联系**：将数学中的计算过程与计算机中的算法执行联系起来。

开头：从数学到计算机

在数学中，我们学习过函数的概念。比如 $f(x) = x^2 + 1$，这个函数接受一个输入 x，经过计算后产生一个输出结果。如果 $x = 2$，那么 $f(2) = 2^2 + 1 = 5$。这个计算过程本身就是一个简单的算法。

计算机中的算法也遵循类似的思想，但实现方式更加丰富和灵活。

4.6.1 数学计算过程回顾

数学中的函数计算可以看作是一个确定的算法过程：

- 有明确的输入和输出
- 计算步骤是确定的、可重复的
- 过程是有限的，能在有限步骤内完成
- 每个步骤都是明确无歧义的

例子：

- $f(x) = 2x + 3$：输入 x，按照” 乘 2 再加 3” 的算法得到输出
- $g(x, y) = x^2 + y^2$：输入两个数，按照” 各自平方再求和” 的算法得到结果

4.6.2 计算机中的算法实现

计算机中的算法比数学计算过程更加丰富和实用：

1. 复杂流程控制

- 支持条件判断、循环等控制结构
- 能够处理分支情况和重复任务
- 算法逻辑可以更加复杂和智能

2. 状态管理

- 算法执行过程中可以修改和维护状态
- 可以通过变量存储中间结果
- 能够处理有状态的计算问题

3. 过程抽象与封装

- 复杂算法可以被封装成函数或方法
- 通过算法名称来代表这一系列操作步骤
- 使得代码模块化，提高可重用性和可维护性

4.6.3 算法的执行过程

当执行一个算法时，计算机会按照以下逻辑进行：

1. **输入处理**：接收输入数据并进行验证
2. **初始化**：设置初始状态和变量
3. **步骤执行**：按照算法定义逐步执行操作
4. **状态更新**：在执行过程中维护和更新状态
5. **结果输出**：产生最终结果并返回
6. **资源清理**：释放使用的临时资源

4.6.4 算法的特性分析

根据算法的性质，我们可以从多个维度进行分析：

确定性算法：

- 每个步骤都有明确的定义
- 相同的输入总是产生相同的输出和相同的执行路径
- 如排序算法、数学计算算法等

有状态算法：

- 算法的行为可能依赖于之前的状态
- 输出不仅取决于输入，还取决于当前状态
- 如游戏 AI、会话管理等算法

在伪代码中，我们可以使用：

$$\text{定义}：\text{模板组} = \text{类型 1}, \ldots, \text{类型 n}$$

$$\text{定义}：\text{参数组} = \text{输入 1} \in \text{输入 1 的类型}, \ldots, \text{输入 n} \in \text{输入 n 的类型}$$

$$\text{定义}：\text{算法名称}\{\text{模板组}\}(\text{参数组}) \to \text{输出的类型} = \left[\text{算法内容}\right]$$

来定义一个算法。其中，算法名称 可以自由命名，输入 1 到 输入 n 也可以自由命名，并且 输入 1 到 输入 n 作为变量只在 算法内容 有效，在算法外部是未定义的。输入 1 的类型 到 输入 n 的类型 以及 输出的类型 都是可以代表数据类型的集合，算法内容 是符合算法输入输出形式的任意伪代码。模板组 可以被省略，如果未被省略，类型 1 到 类型 n 可以自由命名，用于代表一个数据类型。算法中 输出 1 的类型 到 输出 n 的类型 可以使用 类型 1 到
类型 n 指代具体的类型，同时 算法内容 中也可以使用它们指代具体的类型。在除了定义算法时，{类型 1, . . . , 类型 n} 中的类型必须明写为确定的数据类型。在 算法内容 中，我们规定带有“输出 :”前缀的伪代码将自身的输出作为算法的输出，且执行带有这个前缀的伪代码之后算法结束。若算法中没有出现这个前缀，则默认最后被执行的一行伪代码为输出（一个算法可能有多个最后被执行的伪代码）。且算法中“→ 输出的类型”可以被省略，省略时默认算法不输出，即 输出的类型 = ∅，此时不允许在 算法内容 中使用“输出 :”前缀。算法名称{模板组}(参数组) → 输出的类型 也可以被视为一个类型，其中 算法代称 可以是任意文本，用于代表拥有相同输入输出规范的所有函数。

特别地，假设已定义算法 $f(\texttt{i} \in \texttt{T}) \to \emptyset$，且存在变量 $\texttt{var} \in \texttt{T}$，那么在 $f(\texttt{var})$ 的算法内容中，$\&\texttt{var} \neq \&\texttt{i}$，即局部变量 i 与 var 的地址不同，所以它们不是同一个变量。

我们规定，算法中的参数按值传递，即保证算法中的输入的值等于传入变量的值，但实际上输入与传入变量的地址不同，不是同一个变量。

对于两个算法 A 和 B，如果它们的算法名称相同，输入输出的类型相同，则认为它们是同名的。所以，不允许定义两个同名的算法，尽管它们的算法内容可能不同。

思考：数学中的计算过程 $f(x) = x + 1$ 对应的是确定性算法还是有状态算法？为什么？

本节总结

- **基础概念**：算法是解决问题的明确步骤序列
- **计算机实现**：支持复杂流程控制、状态管理和过程抽象
- **执行过程**：输入处理 → 初始化 → 步骤执行 → 状态更新 → 结果输出 → 资源清理
- **算法特性**：确定性算法与有状态算法的区别在于是否依赖历史状态

算法是计算机科学的核心基础，理解算法的概念和特性对于设计高效可靠的程序至关重要。

4.7 空间复杂度与时间复杂度

【本节目标】

- **理解概念**：掌握算法复杂度的基本含义。
- **掌握分析**：学会如何分析算法的资源消耗。
- **实践应用**：能够计算简单算法的时间复杂度。

开头：算法的”成本”

当我们解决一个问题时，通常有多种算法可以选择。如何判断哪个算法更好呢？除了正确性之外，我们还需要考虑算法的”成本”——主要是时间成本和空间成本。

4.7.1 什么是复杂度？

复杂度描述的是算法执行所需资源的多少，主要包括：

时间复杂度：算法执行需要的时间

空间复杂度：算法执行需要的存储空间

复杂度不是用具体的秒数或字节数来表示，而是用输入规模函数的形式来描述。

4.7.2 频度分析

分析复杂度的一种方法是统计基本操作的执行次数：
基本操作：算法中最耗时的核心操作

- 比如：比较操作、赋值操作、算术运算等
- 不同算法有不同的基本操作选择

频度：基本操作执行的次数

- 通常表示为输入规模 n 的函数 $T(n)$
- $T(n)$ 称为时间频度函数

4.7.3 大 O 表示法

我们通常关心的是当输入规模很大时，频度函数的增长趋势。大 O 表示法描述的就是这种渐进行为。
定义：若存在正常数 c 和 n_0，使得当 $n \geq n_0$ 时，$T(n) \leq c \times f(n)$，则称 $T(n) = O(f(n))$。
常见复杂度：

- $O(1)$：常数复杂度，与输入规模无关
- $O(\log n)$：对数复杂度，增长很慢
- $O(n)$：线性复杂度，与输入规模成正比
- $O(n^2)$：平方复杂度，增长较快
- $O(2^n)$：指数复杂度，增长极快

4.7.4 时间复杂度计算示例

例子 1：求数组元素和

- 算法：遍历数组，累加所有元素
- 基本操作：加法操作
- 频度：n 次加法（n 为数组长度）
- 时间复杂度：$O(n)$

例子 2：嵌套循环

- 算法：对每个元素，与所有其他元素比较

- 基本操作：比较操作
- 频度：$n \times n = n^2$ 次比较
- 时间复杂度：$O(n^2)$

4.7.5 空间复杂度分析

空间复杂度分析类似，关注算法执行过程中需要的额外存储空间：

考虑因素：

- 变量占用的空间
- 动态申请的空间
- 递归调用的栈空间

例子：递归计算斐波那契数列

- 每次递归调用都需要保存状态
- 空间复杂度为 $O(n)$

练习：分析此操作的空间复杂度和时间复杂度：计算 n 的阶乘（迭代版本）

本节总结

- **复杂度意义**：衡量算法资源消耗的指标
- **时间空间**：时间复杂度关注执行时间，空间复杂度关注存储空间
- **频度分析**：通过统计基本操作次数来分析复杂度
- **大 O 表示**：描述算法在输入规模很大时的增长趋势
- **实用价值**：帮助选择更高效的算法解决实际问题

理解复杂度分析是评估算法优劣的重要工具。

练习答案

- **练习答案**：**计算 n 的阶乘（迭代版本）**：
  - **时间复杂度**：$O(n)$
    - 需要执行 n 次乘法操作（从 1 乘到 n）
    - 操作次数与输入值 n 成正比

- **空间复杂度**：$O(1)$
  - 只需要固定数量的变量（如循环计数器、累乘结果）
  - 无论 n 多大，使用的额外空间是固定的

分析说明：

- 时间复杂度的计算基于**基本操作次数**与输入规模的关系
- 空间复杂度的计算基于**额外使用的存储空间**与输入规模的关系
- $O(1)$ 表示常数复杂度，是最理想的情况
- $O(n)$ 表示线性复杂度，随输入规模线性增长

4.8 数组

【本节目标】

- **理解概念**：掌握数组的基本定义和核心特性。
- **掌握计算**：学会计算数组中任意元素的地址。
- **发现联系**：理解数组与指针之间的紧密关系。
- **数学建模**：能够用数学方式描述数组的存储和访问。

开头：我们要解决什么问题？

在生活中，我们经常需要处理一系列相同类型的东西。比如，一个班级所有学生的成绩、一个书架上的所有图书。在编程中，为了高效地处理这种”数据的序列”，我们使用一种叫做数组的基本数据结构。

你可以把数组想象成一个有多格子的储物柜，或者一列编号的座位。

- 每个格子 (或座位) 被称为数组的一个元素。
- 所有元素都必须是同一种类型。
- 每个元素在数组中都有一个唯一的编号，叫做索引。索引通常从 0 开始计数。

4.8.1 存储空间中的数组

数组的元素一个紧挨着一个，连续地存储在计算机的存储空间中。这是数组最重要的特性。

- 假设一个数组的类型是 `T`。一个 `T` 类型的变量所占用的字节数是 $\texttt{size}_{\texttt{T}}$。
- 那么这个数组的每个元素都占用 $\texttt{size}_{\texttt{T}}$ 个字节。

- 第一个元素 (索引 0) 存放在起始地址 base。
- 第二个元素 (索引 1) 存放在地址 $\texttt{base} + 1 \times \texttt{size}_\texttt{T}$。
- 第三个元素 (索引 2) 存放在地址 $\texttt{base} + 2 \times \texttt{size}_\texttt{T}$。
- ... 以此类推。

任何一个元素 (索引为 i) 的地址 address[i] 都可以用公式 $\texttt{address}[\texttt{i}] = \texttt{base} + \texttt{i} \times \texttt{size}_\texttt{T}$ 来表示。其中 base 是整个数组的起始地址，$\texttt{size}_\texttt{T}$ 是单个元素的大小 (字节数)。

这个公式和指针的运算规则是完全一致的。数组名在很多情况下可以看作一个指向数组第一个元素的指针常量。

4.8.2 数组的使用

我们通过数组名和索引来访问特定的元素。假设数组名为 arr，且数组有 n 个元素，那么第 i 个元素 (索引为 $\texttt{i} - 1$，$\texttt{i} \leq \texttt{n}$) 的访问表示为：

$$\texttt{arr}[\texttt{i}]$$

正如前面提到的，数组名本身就是一个指向数组起始地址 (即第一个元素) 的指针。但是，数组名和普通的指针变量有一个关键区别：

- 普通指针变量的值是可以改变的 (可以指向别处)。
- 而数组名是一个常量，它的值 (数组的起始地址) 在数组的生命周期内是固定不变的。你不能让数组名指向另一个地方。

4.8.3 数组的数学模型

在我们用来描述存储空间的数学模型中，数组可以被精确定义。
设：

- base 是数组的起始地址。
- length 是数组的长度。
- $\texttt{size}_\texttt{T}$ 是每个元素的大小。
- 数组有效的地址范围 L 为：$\texttt{base} \leq \texttt{L} \leq \texttt{base} + (\texttt{length} - 1) \times \texttt{size}_\texttt{T}$。

那么，在某个状态 σ 下，读取数组索引 i 的值，可以定义为读取地址 $\texttt{base} + \texttt{i} \times \texttt{size}_\texttt{T}$ 开始的一段连续字节，并将其解释为类型 T。

$$\texttt{arr}[\texttt{i}] = \text{interpret}_\texttt{T}(\sigma(\texttt{base} + \texttt{i} \times \texttt{size}_\texttt{T}), \sigma(\texttt{base} + \texttt{i} \times \texttt{size}_\texttt{T} + 1), \dots)$$

向数组索引 i 写入一个值 v，会产生一个新的状态 σ'。这个状态 σ' 只在从地址 $\mathtt{base} + \mathtt{i} \times \mathtt{size}_{\mathtt{T}}$ 开始，长度为 $\mathtt{size}_{\mathtt{T}}$ 的这段存储空间的值发生了变化，其他所有地址的值保持不变。

$$\sigma' = \sigma[(\mathtt{base} + \mathtt{i} \times \mathtt{size}_{\mathtt{T}}) \to \mathtt{v}]$$

4.8.4 数组的常见操作

数组支持如下基本操作：

- **随机访问**：通过索引直接访问任意元素，时间复杂度为 $O(1)$

示例：假设有一个整数数组 numbers，起始地址为 1000，每个元素占 4 字节。

- $\mathtt{numbers}[0]$ 的地址：$1000 + 0 \times 4 = 1000$
- $\mathtt{numbers}[1]$ 的地址：$1000 + 1 \times 4 = 1004$
- $\mathtt{numbers}[2]$ 的地址：$1000 + 2 \times 4 = 1008$

在伪代码中，我们使用 $ARR_{\mathtt{T}}(\mathtt{n})$ 来表示一个长度为 n 的元素类型为 T 的数组。其中
n 为一个自然数，T 是可以代表数据类型的集合。$ARR_{\mathtt{T}}(\mathtt{n})$ 也可以被视为一个类型。同时，数组变量也可以被视为一个指向数组起始地址的指针。

我们可以使用：

$$\mathtt{a} \in ARR_{\mathtt{T}}(\mathtt{n})$$

当 a 在前文中未定义时，意为声明新变量 a 是一个长度为 n 的元素类型为 T 的数组。当 a 在前文中已定义时，意为判断 a 的类型是否为 $ARR_{\mathtt{T}}(\mathtt{n})$，输出布尔值。其中，n 必须是一个确定的大于 0 的自然数常数，例如 $3, 8$ 等，而不能是变量。

假设 a 在前文中未定义，我们可以使用：

$$\mathtt{a} \in\uparrow ARR_{\mathtt{T}}(\mathtt{n})$$

来表示申请一个长度为 n 的元素类型为 T 的数组，并把该数组的起始地址存储到 a 中，此时 $\mathtt{a} \in PTR_{ARR_{\mathtt{T}}(\mathtt{n})}$，或者更泛用地表示为，$\mathtt{a} \in PTR_{PTR_{\mathtt{T}}}$。与 $\mathtt{a} \in ARR_{\mathtt{T}}(\mathtt{n})$ 不同的是，此时 n 允许是大于 0 的数胞变量。此时 $*\mathtt{a} = \mathtt{a}[0]$。

假设已有数组 $\mathtt{a} \in ARR_{\mathtt{T}}(\mathtt{n})$，允许这样访问 a 的元素：

$$\mathtt{a}[\mathtt{i}]$$

其中 $\mathtt{i} < \mathtt{n}$。

练习：有一个长度为 8 以上的字符数组 chars，起始地址为 2000，每个字符占 1 字节。请计算：

1. $\mathtt{chars}[5]$ 的地址是多少？
2. 如果要访问地址 2007 处的元素，它的索引是多少？

本节总结

- **数组本质**：相同类型元素的连续存储集合
- **索引访问**：通过 `arr[i]` 访问第 `i` 个元素（从 0 开始）
- **地址计算**：元素地址 = 基地址 + 索引 × 元素大小
- **与指针关系**：数组名可视为指向首元素的常量指针
- **数学表示**：可用状态函数 σ 精确描述数组的存储和访问

数组是计算机科学中最基础也是最重要的数据结构之一，理解其存储方式和访问机制对于后续学习更复杂的数据结构和算法至关重要。

练习答案

- **练习答案**：
  1. `chars[5]` 的地址：$2000 + 5 \times 1 = 2005$
  2. 地址 2007 对应的索引：$(2007 - 2000) \div 1 = 7$

4.9 字典

【本节目标】

- **理解概念**：掌握字典作为键值对映射的基本概念。
- **掌握操作**：理解字典的核心操作及其数学含义。
- **数学建模**：能够用函数模型描述字典的行为。

开头：从现实到抽象

在日常生活中，我们经常使用字典：根据一个” 键”(如一个单词)，来查找对应的” 值”(如该单词的定义)。在计算机科学中，**字典**或称**映射**，正是对这种” 键-值” 对应关系的抽象。它是一种存储**键值对**的数据结构，并保证每个键都是唯一的。

4.9.1 字典的核心操作

字典的接口非常直观，主要包含以下操作：

- **插入/更新**：将键 `k` 和值 `v` 关联起来并存入字典。如果键 `k` 已存在，则用新值 `v` 覆盖旧值。
- **查找**：根据给定的键 `k`，查找并返回其对应的值。如果键不存在，则返回特定值表示不存在。

- **删除**：从字典中移除键 k 及其关联的值。
- **存在判断**：判断字典中是否存在键 k。

4.9.2 字典的数学模型

我们可以为字典建立一个纯粹、抽象的数学模型，这个模型只关心其逻辑行为，而不关心其实现方式。

一个字典 D 在任意时刻的逻辑状态，都可以用一个**部分函数**来精确描述：

$$D : K \to V$$

其中：

- K 是所有可能键的集合
- V 是所有可能值的集合
- 这个函数 D 的定义域 $\mathrm{dom}(D) \subseteq K$，是当前字典中所有有效键的集合

这个函数 D 清晰地刻画了键与值之间的映射关系。对于任何一个键 $\mathtt{k} \in \mathrm{dom}(D)$，$D(\mathtt{k})$ 唯一地确定了它对应的值。

4.9.3 字典操作的形式化规范

基于这个函数模型，我们可以为每个字典操作给出严格的、形式化的规范：

插入/更新操作：

$$\{true\} \quad \mathrm{insert}(\mathtt{k}, \mathtt{v}) \quad \{D_{\mathrm{new}} = D_{\mathrm{old}}[\mathtt{k} \mapsto \mathtt{v}]\}$$

后置条件表示新函数 D_{new} 是这样定义的：

$$D_{\mathrm{new}}(\mathtt{k}') = \begin{cases} \mathtt{v} & \text{如果} \mathtt{k}' = \mathtt{k} \\ D_{\mathrm{old}}(\mathtt{k}') & \text{如果} \mathtt{k}' \in \mathrm{dom}(D_{\mathrm{old}}) \wedge \mathtt{k}' \neq \mathtt{k} \end{cases}$$

查找操作：

$$\{\mathtt{k} \in \mathrm{dom}(D)\} \quad \mathrm{get}(\mathtt{k}) \quad \{result = D_{\mathrm{old}}(\mathtt{k})\}$$

前置条件：键 k 必须存在于字典中
后置条件：返回值等于操作执行前函数 D 在 k 上的值

删除操作：

$$\{\mathtt{k} \in \mathrm{dom}(D_{\mathrm{old}})\} \quad \mathrm{remove}(\mathtt{k}) \quad \{D_{\mathrm{new}} = D_{\mathrm{old}} \setminus \{(\mathtt{k}, D_{\mathrm{old}}(\mathtt{k}))\}\}$$

前置条件：键 k 必须存在于字典中
后置条件：新函数去除对键 k 的映射

存在判断：

$$\{true\} \quad \mathrm{contains}(\mathtt{k}) \quad \{result = (\mathtt{k} \in \mathrm{dom}(D))\}$$

后置条件：返回值反映键 k 是否在字典中

本节总结

- **字典本质**：键值对的映射集合，每个键唯一对应一个值
- **数学模型**：可以用部分函数 $D : K \to V$ 精确描述
- **核心操作**：插入、查找、删除、存在判断
- **行为一致性**：无论何种实现，外在行为必须符合数学模型

字典是计算机科学中最重要的数据结构之一，理解其数学模型和实现原理对于设计高效程序至关重要。

4.10 动态数组

【本节目标】

- **问题驱动**：理解我们为什么需要使用动态数组。
- **掌握概念**：理解动态数组的结构和操作。
- **理解实现**：理解动态数组是如何实现的。
- **实践目标**：能够自主使用伪代码编写动态数组的实现。

从本节开始，我们进入了**算法与数据结构**中最难的部分。从现在开始，你需要理解大量文中的伪代码，并尝试自行用伪代码实现数据结构。

开头：我们要解决什么问题？

在程序中，我们可以使用数组存放一组相同类型的数据。但是，数组的长度是有限且不变的。假设有一个存放学生成绩的数组，班里有 10 个学生，我们会在定义这个数组时确定其长度为 10，以刚好存放所有学生的成绩。但是当班里来了 10 个新学生时，原本的数组已经无法存放更多的成绩数据了。你需要一个解决办法：

- **方法 A**：再创建一个长度为 10 的数组，用于存放新学生的成绩。
- **方法 B**：创建一个长度为 20 的数组，将原来的 10 个学生的成绩抄到新数组中，然后往后继续写新学生的成绩。
- **方法 C**：使用一个结构体来存储数组的地址和长度，当长度不够时，将结构体实例中的指针指向一个更大的数组，并将原来数组的数据复制到新的数组中。

实际上，方法 C 就是动态数组的思想。这样，无论有多少个学生，学生增加了多少，减少了多少，这个结构体都能通过调整指针和长度变量来实现动态存储。

但是，这个方法还是太简陋了，有一些问题没有解决。接下来，我们将共同编写伪代码，实现一个真正的动态数组。如果你有想法，可以自行思考编写伪代码，以实现动态数组，然后再与文中的实现进行对比。

4.10.1 动态数组的结构

我们使用 DA 指代动态数组：

$$\text{定义}:\texttt{WORDSIZEVALUE}=\texttt{BASE}^{\texttt{BYTELENGTH}\times\texttt{WORDSIZE}}$$

$$\begin{bmatrix}\text{结构体}:DA\{\texttt{T}\}\\ addr\in PTR_{PTR_{\texttt{T}}}\\ l\in NC_{\texttt{WORDSIZEVALUE}}\\ c\in NC_{\texttt{WORDSIZEVALUE}}\end{bmatrix}$$

其中，$addr$ 是动态数组的指针，用于指向当前存放类型为 T 的数据的普通数组，l 是指向的普通数组的目前长度，c 是指向的普通数组的实际容量，即使用申请空间运算时指定的长度。l 和 c 之间满足 $l\le c$，且由于 $addr$、l 和 c 的类型都是以计算机字长计算得到状态数的数胞，理论上按规范操作不会产生溢出现象。

4.10.2 动态数组的初始化

动态数组的初始化，实际上就是使用申请空间运算创建一个普通数组，并返回一个指向这个普通数组的指针。

$$\text{定义}:DA_{\texttt{T}}=DA\{\texttt{T}\}$$

$$\text{定义}:\text{FillZero}\{\texttt{T}\}(\texttt{p}\in PTR_{\texttt{T}}),$$

执行此算法之后，p指向的数据所占的存储空间的每个字节都被设为 0

$$\text{定义}:\text{InitialDA}\{\texttt{T}\}(\texttt{l}\in NC_{\texttt{WORDSIZEVALUE}})\to DA_{\texttt{T}}=$$

$$\begin{bmatrix}\texttt{addr}:=\uparrow ARR_{\texttt{T}}(\texttt{l})\\ \text{FillZero}\{ARR_{\texttt{T}}(\texttt{l})\}(\texttt{addr})\\ \texttt{instance}\in DA\\ \texttt{instance}\to addr:=\texttt{addr}\\ \texttt{instance}\to l:=0\\ \texttt{instance}\to c:=\texttt{l}\\ \text{输出}:\texttt{instance}\end{bmatrix}$$

InitialDA的输入l满足$\texttt{l}>0$

$$\text{定义}:\uparrow DA_{\texttt{T}}(\texttt{n})=\text{InitialDA}\{\texttt{T}\}(\texttt{n}),\texttt{n}\in NC_{\texttt{WORDSIZEVALUE}}$$

4.10.3 动态数组的销毁

由于初始化动态数组时，我们使用了申请空间运算，因此当动态数组不再使用时，需要使用释放空间运算将其销毁。

$$\text{定义}:\text{DestroyDA}\{\texttt{T}\}(\texttt{instance}\in DA_{\texttt{T}})=$$

$$\left[\downarrow(\texttt{instance} \to addr)\right]$$

$$\text{定义}:\downarrow \texttt{instance} = \text{DestroyDA}\{\texttt{T}\}(\texttt{instance}), \texttt{instance} \in DA_{\texttt{T}}$$

我们规定，对于一个结构体或联合体的实例 instance，如果它的初始化需要使用申请空间运算，且初始化之后并未释放。且存在语法规则 ↓ instance，我们默认这个语句会完成 instance 的销毁操作。因此，当在算法中出现一个结构体或联合体的实例 instance 时，如果算法结束后，不存在任何途径能访问到 instance，那么这个实例会被自动销毁，即便没有显式地书写 “↓ instance”，我们默认此时执行了 ↓ instance。

4.10.4 访问动态数组的元素

访问动态数组的元素，本质上是通过动态数组的指针，访问当前指向的普通数组的元素。

$$\text{定义}:\texttt{instance}[\texttt{i}] = (\texttt{instance} \to addr)[\texttt{i}]$$

$$\texttt{i} \in NC_{\texttt{WORDSIZEVALUE}}, \texttt{instance} \in DA, \texttt{i} < \texttt{instance} \to l$$

4.10.5 重新设定动态数组的长度

重新设定动态数组的长度，分为三种情况。第一种，新长度小于等于原容量，且小于等于原长度，直接将动态数组的长度设置为新长度。第二种，新长度小于等于原容量，但大于原长度，则将动态数组的长度设置为新长度，并把延伸出的新的几个元素的值都设置为 0。第三种，新长度大于原容量，此时动态数组的容量不足以容纳新长度的数组，需要对动态数组进行扩容。

对于扩容，在这里使用的策略是不断将容量扩大为原来的 2 倍，直到容量足以容纳新长度的数组。

$$\text{定义}:\text{SetLengthOfDA}\{\texttt{T}\}(\texttt{instance} \in DA_{\texttt{T}}, \texttt{n} \in NC_{\texttt{WORDSIZEVALUE}}) =$$

$$\left[\texttt{instance} \to l := \texttt{n}\right]$$

$$\text{定义}:\text{SetLengthAndFillOfDA}\{\texttt{T}\}(\texttt{instance} \in DA_{\texttt{T}}, \texttt{n} \in NC_{\texttt{WORDSIZEVALUE}}) =$$

$$\left[\begin{array}{c} \texttt{i} := \texttt{instance} \to l \\ \left[\begin{array}{c} \text{当}:\texttt{i} < \texttt{n} \\ \text{FillZero}\{\texttt{T}\}((\texttt{instance} \to addr)[\texttt{i}]) \\ \texttt{i} :=^{\oplus} 1 \end{array}\right] \\ \texttt{instance} \to l := \texttt{n} \end{array}\right]$$

$$\text{定义}:\text{ExtendDAInResize}\{\texttt{T}\}(\texttt{instance} \in DA_{\texttt{T}}, \texttt{n} \in NC_{\texttt{WORDSIZEVALUE}}) =$$

$$\left[\begin{array}{c} \left[\begin{array}{c} 当:\texttt{instance} \to c < \texttt{n} \\ \texttt{instance} \to c :=^{\otimes} 2 \end{array}\right] \\ \texttt{newarr} :=\uparrow ARR_{\texttt{T}}(\texttt{instance} \to c) \\ \text{FillZero}\{ARR_{\texttt{T}}(\texttt{instance} \to c)\}(\texttt{newarr}) \\ \texttt{i} := NC_{\texttt{WORDSIZEVALUE}}(0) \\ \left[\begin{array}{c} 当:\texttt{i} < \texttt{instance} \to l \\ \texttt{newarr[i]} := (\texttt{instance} \to addr)[\texttt{i}] \\ \texttt{i} :=^{\oplus} 1 \end{array}\right] \\ \downarrow(\texttt{instance} \to addr) \\ \texttt{instance} \to l := \texttt{n} \\ \texttt{instance} \to addr := \texttt{newarr} \end{array}\right]$$

$$定义:\text{ResizeOfDA}\{\texttt{T}\}(\texttt{instance} \in DA_{\texttt{T}}, \texttt{n} \in NC_{\texttt{WORDSIZEVALUE}}) =$$

$$\left[\begin{cases} \text{SetLengthOfDA}\{\texttt{T}\}(\texttt{instance}, \texttt{n}) & 如果:\texttt{n} \le \texttt{instance} \to l \\ \text{SetLengthAndFillOfDA}\{\texttt{T}\}(\texttt{instance}, \texttt{n}) & 否则如果:\texttt{n} \le \texttt{instance} \to c \\ \text{ExtendDAInResize}\{\texttt{T}\}(\texttt{instance}, \texttt{n}) & 否则 \end{cases}\right]$$

$$定义:\texttt{instance} \to \text{resize}(\texttt{n}) = \text{ResizeOfDA}\{\texttt{T}\}(\texttt{instance}, \texttt{n}),$$

$$\texttt{instance} \in DA_{\texttt{T}}, \texttt{n} \in NC_{\texttt{WORDSIZEVALUE}}$$

4.10.6 重新设定动态数组的容量

重新设定动态数组的容量，分为两种情况。第一种，新容量小于等于原容量，什么都不做，不需要更改动态数组的容量。第二种，新容量大于原容量，则将动态数组的容量扩展为新容量，并进行扩容，但是长度不变。

$$定义:\text{ExtendDAInReserve}\{\texttt{T}\}(\texttt{instance} \in DA_{\texttt{T}}, \texttt{n} \in NC_{\texttt{WORDSIZEVALUE}}) =$$

$$\left[\begin{array}{c} \texttt{newarr} :=\uparrow ARR_{\texttt{T}}(\texttt{n}) \\ \text{FillZero}\{ARR_{\texttt{T}}(\texttt{n})\}(\texttt{newarr}) \\ \texttt{i} := NC_{\texttt{WORDSIZEVALUE}}(0) \\ \left[\begin{array}{c} 当:\texttt{i} < \texttt{instance} \to l \\ \texttt{newarr[i]} := (\texttt{instance} \to addr)[\texttt{i}] \\ \texttt{i} :=^{\oplus} 1 \end{array}\right] \\ \downarrow(\texttt{instance} \to addr) \\ \texttt{instance} \to addr := \texttt{newarr} \end{array}\right]$$

$$定义:\text{ReserveOfDA}\{\texttt{T}\}(\texttt{instance} \in DA_{\texttt{T}}, \texttt{n} \in NC_{\texttt{WORDSIZEVALUE}}) =$$

$$\left[\begin{cases} \text{ExtendDAInReserve}(\texttt{instance}, \texttt{n}) & 如果:\texttt{n} > \texttt{instance} \to c \\ \texttt{instance} \to c := \texttt{n} \end{cases}\right]$$

$$\text{定义}:\texttt{instance} \to \text{reserve}(\texttt{n}) = \text{ReserveOfDA}\{\texttt{T}\}(\texttt{instance},\texttt{n}),$$

$$\texttt{instance} \in DA_{\texttt{T}}, \texttt{n} \in NC_{\texttt{WORDSIZEVALUE}}$$

4.10.7 为动态数组添加新元素

添加新元素的本质就是，使动态数组的长度增加 1，并把新元素放在数组的末尾。

$$\text{定义}:\text{AppendOfDA}\{\texttt{T}\}(\texttt{instance} \in DA_{\texttt{T}}, \texttt{element} \in \texttt{T}) =$$

$$\left[\begin{cases}\text{如果}:\texttt{instance} \to l = \texttt{instance} \to c \\ \text{ExtendDAInResize}\{\texttt{T}\}(\texttt{instance}, (\texttt{instance} \to l) \oplus 1) \\ (\texttt{instance} \to l) :=^{\oplus} 1 \quad \text{否则}\end{cases} \right.$$
$$\left. (\texttt{instance} \to addr)[(\texttt{instance} \to l) \ominus 1] := \texttt{element} \right]$$

$$\text{定义}:\texttt{instance} \to \text{append}(\texttt{element}) = \text{AppendOfDA}\{\texttt{T}\}(\texttt{instance},\texttt{element}),$$

$$\texttt{instance} \in DA_{\texttt{T}}, \texttt{element} \in \texttt{T}$$

4.10.8 为动态数组插入元素

插入元素的本质就是，增加动态数组的长度，然后把指定位置及之后的位置的所有元素后移，最后插入元素到指定位置。

$$\text{定义}:\text{InsertOfDA}\{\texttt{T}\}(\texttt{instance} \in DA_{\texttt{T}}, \texttt{element} \in \texttt{T}, \texttt{position} \in NC_{\texttt{WORDSIZEVALUE}}) =$$

$$\left[\begin{cases}\text{如果}:\texttt{instance} \to l = \texttt{instance} \to c \\ \text{ExtendDAInResize}\{\texttt{T}\}(\texttt{instance}, (\texttt{instance} \to l) \oplus 1) \\ (\texttt{instance} \to l) :=^{\oplus} 1 \quad \text{否则}\end{cases} \right.$$
$$\left\{\begin{array}{l}\text{如果}:(\texttt{instance} \to l) > 0 \\ \texttt{i} := (\texttt{instance} \to l) \ominus 1 \\ \left[\begin{array}{c}\text{当}:\texttt{i} \geq \texttt{position} \wedge \texttt{i} < (\texttt{instance} \to l) \\ (\texttt{instance} \to addr)[\texttt{i} \oplus 1] := (\texttt{instance} \to addr)[\texttt{i}] \\ \texttt{i} :=^{\ominus} 1\end{array}\right] \end{array}\right.$$
$$\left. (\texttt{instance} \to addr)[\texttt{position}] := \texttt{element} \right]$$

$$\text{定义}:\text{InsertOfDA}\{\texttt{T}\}(\texttt{instance} \in DA_{\texttt{T}},$$

$$\texttt{elements} \in PTR_{\texttt{T}}, \texttt{length} \in NC_{\texttt{WORDSIZEVALUE}}, \texttt{position} \in NC_{\texttt{WORDSIZEVALUE}}) =$$

$$\left[\begin{array}{c}\left\{\begin{array}{l}\text{如果}:(\texttt{instance}\to l)\oplus\texttt{length}>\texttt{instance}\to c\\ \text{ExtendDAInResize}\{\texttt{T}\}(\texttt{instance},(\texttt{instance}\to l)\oplus\texttt{length})\\ (\texttt{instance}\to l):=^{\oplus}\texttt{length}\quad\text{否则}\end{array}\right.\\ \left\{\begin{array}{l}\text{如果}:(\texttt{instance}\to l)>0\\ \texttt{i}:=(\texttt{instance}\to l)\ominus 1\\ \left[\begin{array}{c}\text{当}:\texttt{i}\geq\texttt{position}\wedge\texttt{i}<(\texttt{instance}\to l)\\ (\texttt{instance}\to addr)[\texttt{i}\oplus\texttt{length}]:=(\texttt{instance}\to addr)[\texttt{i}]\\ \texttt{i}:=^{\ominus}1\end{array}\right]\end{array}\right.\\ \texttt{j}:=NC_{\texttt{WORDSIZEVALUE}}(0)\\ \left[\begin{array}{c}\text{当}:\texttt{j}<\texttt{length}\\ (\texttt{instance}\to addr)[\texttt{j}\oplus\texttt{position}]:=*(\texttt{elements}\boxplus\texttt{j})\\ \texttt{j}:=^{\oplus}1\end{array}\right]\end{array}\right]$$

InsertOfDA的输入$\texttt{position}$满足$\texttt{position}\leq\texttt{instance}\to l$

第二个InsertOfDA的输入$\texttt{elements},\texttt{length}$满足$\texttt{elements}$指向的连续$\texttt{length}$个数据有效

$$\text{定义}:\texttt{instance}\to\text{insert}(\texttt{element},\texttt{position})=$$

$$\text{InsertOfDA}\{\texttt{T}\}(\texttt{instance},\texttt{element},\texttt{position}),$$

$$\texttt{instance}\in DA_{\texttt{T}},\texttt{element}\in\texttt{T},\texttt{position}\in NC_{\texttt{WORDSIZEVALUE}}$$

$$\text{定义}:\texttt{instance}\to\text{insert}(\texttt{elements},\texttt{length},\texttt{position})=$$

$$\text{InsertOfDA}\{\texttt{T}\}(\texttt{instance},\texttt{elements},\texttt{length},\texttt{position}),\texttt{instance}\in DA_{\texttt{T}},$$

$$\texttt{elements}\in PTR_{\texttt{T}},\texttt{length}\in NC_{\texttt{WORDSIZEVALUE}},\texttt{position}\in NC_{\texttt{WORDSIZEVALUE}}$$

4.10.9 为动态数组删除元素

删除元素的本质就是，把指定位置及之后的位置的所有元素前移，然后减少数组的长度。

$$\text{定义}:\text{RemoveOfDA}\{\texttt{T}\}(\texttt{instance}\in DA_{\texttt{T}})$$

$$\left[(\texttt{instance}\to l):=^{\ominus}1\right]$$

$$\text{ResizeOfDA}\{\texttt{T}\}(\texttt{instance}\in DA_{\texttt{T}})\text{要求}\texttt{instance}\to l>0$$

$$\text{定义}:\text{RemoveOfDA}\{\texttt{T}\}(\texttt{instance}\in DA_{\texttt{T}},\texttt{index}\in NC_{\texttt{WORDSIZEVALUE}})=$$

$$\left[\begin{array}{c}\texttt{i}:=\texttt{index}\oplus 1\\ \left[\begin{array}{c}\text{当}:\texttt{i}<\texttt{instance}\to l\\ (\texttt{instance}\to addr)[\texttt{i}\ominus 1]:=(\texttt{instance}\to addr)[\texttt{i}]\\ \texttt{i}:=^{\oplus}1\end{array}\right]\\ (\texttt{instance}\to l):=^{\ominus}1\end{array}\right]$$

$$\text{RemoveOfDA}\{\texttt{T}\}(\texttt{instance} \in DA_{\texttt{T}}, \texttt{index} \in NC_{\texttt{WORDSIZEVALUE}})\text{要求}\texttt{index} < \texttt{instance} \to l$$

$$\text{定义}:\text{RemoveOfDA}\{\texttt{T}\}$$

$$(\texttt{instance} \in DA_{\texttt{T}}, \texttt{index} \in NC_{\texttt{WORDSIZEVALUE}}, \texttt{length} \in NC_{\texttt{WORDSIZEVALUE}}) =$$

$$\left[\begin{array}{c} \texttt{i} := \texttt{index} \oplus \texttt{length} \\ \left[\begin{array}{c} \text{当}:\texttt{i} < \texttt{instance} \to l \\ (\texttt{instance} \to addr)[\texttt{i} \ominus \texttt{length}] := (\texttt{instance} \to addr)[\texttt{i}] \\ \texttt{i} :=^{\oplus} 1 \end{array}\right] \\ (\texttt{instance} \to l) :=^{\ominus} \texttt{length} \end{array}\right]$$

$$\text{RemoveOfDA}\{\texttt{T}\}(\texttt{instance} \in DA_{\texttt{T}}, \texttt{index} \in NC_{\texttt{WORDSIZEVALUE}},$$

$$\texttt{length} \in NC_{\texttt{WORDSIZEVALUE}})\text{要求}\texttt{index} < \texttt{instance} \to l \wedge \texttt{index} \oplus \texttt{length} < \texttt{instance} \to l$$

$$\text{定义}:\texttt{instance} \to \text{remove}() = \text{RemoveOfDA}\{\texttt{T}\}(\texttt{instance}), \texttt{instance} \in DA_{\texttt{T}}$$

$$\text{定义}:\texttt{instance} \to \text{remove}(\texttt{index}) = \text{RemoveOfDA}\{\texttt{T}\}(\texttt{instance}, \texttt{index}),$$

$$\texttt{instance} \in DA_{\texttt{T}}, \texttt{index} \in NC_{\texttt{WORDSIZEVALUE}}$$

$$\text{定义}:\texttt{instance} \to \text{remove}(\texttt{index}, \texttt{length}) = \text{RemoveOfDA}\{\texttt{T}\}(\texttt{instance}, \texttt{index},$$

$$\texttt{length}), \texttt{instance} \in DA_{\texttt{T}}, \texttt{index} \in NC_{\texttt{WORDSIZEVALUE}}, \texttt{length} \in NC_{\texttt{WORDSIZEVALUE}}$$

4.10.10 动态数组的比较

动态数组的比较，本质上就是按一定的规则对元素进行比较，这些规则甚至可以自定义，以下是对于 DA 的标准比较。

$$\text{EqualBetweenDA}\{\texttt{T}\}(\texttt{left} \in DA_{\texttt{T}}, \texttt{right} \in DA_{\texttt{T}}) \to NC_2 =$$

$$\left[\begin{array}{c} \left\{\text{输出}:\mathbf{F} \quad \text{如果}:\texttt{left} \to l \neq \texttt{right} \to l \right. \\ \texttt{i} := NC_{\texttt{WORDSIZEVALUE}}(0) \\ \left[\begin{array}{c} \text{当}:\texttt{i} < \texttt{left} \to l \\ \left\{\text{输出}:\mathbf{F} \quad \text{如果}:(\texttt{left} \to addr)[\texttt{i}] \neq (\texttt{right} \to addr)[\texttt{i}] \right. \\ \texttt{i} :=^{\oplus} 1 \end{array}\right] \\ \text{输出}:\mathbf{T} \end{array}\right]$$

$$\text{LessBetweenDA}\{\texttt{T}\}(\texttt{left} \in DA_{\texttt{T}}, \texttt{right} \in DA_{\texttt{T}}) \to NC_2 =$$

$$\left[\begin{array}{c}\begin{cases}\text{输出}:\mathbf{T} & \text{如果}:\texttt{left}\to l<\texttt{right}\to l\\ \text{输出}:\mathbf{F} & \text{如果}:\texttt{left}\to l>\texttt{right}\to l\end{cases}\\ \texttt{i}:=NC_{\texttt{WORDSIZEVALUE}}(0)\\ \left[\begin{array}{c}\text{当}:\texttt{i}<\texttt{left}\to l\\ \begin{cases}\text{输出}:\mathbf{T} & \text{如果}:(\texttt{left}\to addr)[\texttt{i}]<(\texttt{right}\to addr)[\texttt{i}]\\ \text{输出}:\mathbf{F} & \text{如果}:(\texttt{left}\to addr)[\texttt{i}]>(\texttt{right}\to addr)[\texttt{i}]\end{cases}\\ \texttt{i}:=^{\oplus}1\end{array}\right]\\ \text{输出}:\mathbf{F}\end{array}\right]$$

$$\text{定义}:(\texttt{left}=\texttt{right})=\text{EqualBetweenDA}\{\texttt{T}\}(\texttt{left},\texttt{right}),\texttt{left}\in DA_{\texttt{T}},\texttt{right}\in DA_{\texttt{T}}$$

$$\text{定义}:(\texttt{left}<\texttt{right})=\text{LessBetweenDA}\{\texttt{T}\}(\texttt{left},\texttt{right}),\texttt{left}\in DA_{\texttt{T}},\texttt{right}\in DA_{\texttt{T}}$$

$$\text{定义}:(\texttt{left}\neq\texttt{right})=\neg(\texttt{left}=\texttt{right})$$

$$\text{定义}:(\texttt{left}\leq\texttt{right})=((\texttt{left}=\texttt{right})\vee(\texttt{left}<\texttt{right}))$$

$$\text{定义}:(\texttt{left}>\texttt{right})=\neg(\texttt{left}\leq\texttt{right})$$

$$\text{定义}:(\texttt{left}\geq\texttt{right})=\neg(\texttt{left}<\texttt{right})$$

4.11 链表

【本节目标】

- **问题驱动**：理解我们为什么需要使用链表。
- **掌握概念**：理解链表的结构和操作。
- **理解实现**：理解链表是如何实现的。
- **实践目标**：能够自主使用伪代码编写链表的实现。

开头：我们要解决什么问题？

在动态数组的章节中，我们详细介绍了动态数组的使用场景，以及其结构和操作。

但是，如果我们需要频繁在一个较长的动态数组中插入元素，那么每次插入的时间复杂度都是线性复杂度。当进行大量插入时，明显很不划算。

因此，我们需要一个在任意位置插入元素的成本都很低的数据结构，链表就是这样一种结构，只要明确了插入位置，它的插入操作的时间复杂度就是 $O(1)$。

让我们来看一下该如何实现链表。

4.11.1 链表的结构

我们使用 LL 表示链表。

链表存在很多种类，常见的有单链表、双链表、循环链表等。链表不像数组一样，可以随意访问任意位置的元素，而是只能从一个元素访问相邻的元素。对于单链表，只能从前一个元素访问到相邻的后一个元素，反之则不行，因此单链表支持顺序遍历，但不支持倒序遍历。对于双链表，由于其可以从一个元素访问到相邻的两个元素，因此同时支持顺序遍历和倒序遍历。对于循环链表，它的第一个元素和最后一个元素是相邻的，就像一个环，因此如果不停止，可以进行次数无限的循环遍历。我们通常把承载链表的元素的结构称为节点。

注意：所有链表都不支持元素的乱序访问！

在本节中，我们只进行双链表的实现，对于单链表和循环链表的实现，可以自行思考。

$$\begin{bmatrix} \text{结构体} : LLnode\{\mathtt{T}\} \\ data \in \mathtt{T} \\ prev \in PTR_{LLnode\{\mathtt{T}\}} \\ next \in PTR_{LLnode\{\mathtt{T}\}} \end{bmatrix}$$

$$\begin{bmatrix} \text{结构体} : LL\{\mathtt{T}\} \\ head \in PTR_{LLnode\{\mathtt{T}\}} \\ tail \in PTR_{LLnode\{\mathtt{T}\}} \\ l \in NC_{\mathtt{WORDSIZEVALUE}} \end{bmatrix}$$

其中，$LLnode$ 就是双链表的节点结构，拥有指向前一个节点的指针和指向后一个节点的指针。LL 为真正的链表主结构，拥有指向链表头节点的指针 $head$ 和尾节点的指针 $tail$，分别用于顺序访问和倒序访问，以及长度 l 代表链表的节点个数。

注意：在结构体和联合体的定义中，成员的类型不允许形成递归的定义，例如一个结构体 S 的成员的类型不允许是 S，也不允许两个结构互相包含对方类型的成员。但是，一个结构体 S 的成员的类型允许是 PTR_S，因为 S 的指针作为 S 成员的类型并不会导致递归定义，缘由在于指针本质上是个数字，这个规则对于任意指针都适用。

4.11.2 链表的初始化

链表的初始化，实际上就是置空头节点的指针和尾节点的指针，并令长度为 0，为之后添加元素做准备。

$$\text{定义} : LL_{\mathtt{T}} = LL\{\mathtt{T}\}$$

$$\text{定义} : \text{InitialLL}\{\mathtt{T}\}() \to LL_{\mathtt{T}} =$$

$$\left[\begin{array}{c}\texttt{instance} \in LL \\ \texttt{instance} \to head := \odot \\ \texttt{instance} \to tail := \odot \\ \texttt{instance} \to l := 0 \\ \text{输出} : \texttt{instance}\end{array}\right]$$

$$\text{定义} : \Delta LL_{\texttt{T}} = \text{InitialLL}\{\texttt{T}\}()$$

4.11.3 链表的销毁

链表的销毁，实际上就是分别使用释放空间运算释放每一个节点。

$$\text{定义} : \text{DestroyLL}\{\texttt{T}\}(\texttt{instance} \in LL_{\texttt{T}}) =$$

$$\left[\begin{array}{c}\left\{\text{输出} : \emptyset \quad \text{如果} : \texttt{instance} \to l = 0\right. \\ \texttt{node} := \texttt{instance} \to head \\ \left[\begin{array}{c}\text{当} : \texttt{node} \neq \texttt{instance} \to tail \\ \downarrow \texttt{node} \\ \texttt{node} := \texttt{node} \to next \end{array}\right] \\ \downarrow \texttt{node} \\ \texttt{instance} \to head := \odot \\ \texttt{instance} \to tail := \odot \\ \texttt{instance} \to l := 0\end{array}\right]$$

$$\text{定义} : \downarrow \texttt{instance} = \text{DestroyLL}\{\texttt{T}\}(\texttt{instance}), \texttt{instance} \in LL_{\texttt{T}}$$

4.11.4 为链表添加新元素

添加新元素，实际上就是创建一个新节点，并把这个节点插入到链表的尾部。

$$\text{定义} : \text{AppendOfLL}\{\texttt{T}\}(\texttt{instance} \in LL_{\texttt{T}}, \texttt{value} \in \texttt{T}) =$$

$$
\left[\begin{cases}
\text{如果} : \texttt{instance} \to l = 0 \\
\texttt{node} :=\uparrow LLnode_{\texttt{T}} \\
(*\texttt{node}) \to prev := \odot \\
(*\texttt{node}) \to next := \odot \\
(*\texttt{node}) \to data := \texttt{value} \\
\texttt{instance} \to head := \texttt{node} \\
\texttt{instance} \to tail := \texttt{node} \\
\texttt{instance} \to l :=^{\oplus} 1 \\
\text{输出} : \emptyset
\end{cases}
\begin{array}{c}
\texttt{node} :=\uparrow LLnode_{\texttt{T}} \\
\texttt{tail} := \texttt{instance} \to tail \\
(*\texttt{node}) \to prev := \texttt{tail} \\
(*\texttt{node}) \to next := \odot \\
(*\texttt{node}) \to data := \texttt{value} \\
(*\texttt{tail}) \to next := \texttt{node} \\
\texttt{instance} \to tail := \texttt{node} \\
\texttt{instance} \to l :=^{\oplus} 1
\end{array}\right]
$$

$$
\text{定义} : \texttt{instance} \to \text{append}(\texttt{value}) =
$$

$$
\text{AppendOfLL}\{\texttt{T}\}(\texttt{instance}, \texttt{value}), \texttt{instance} \in LL_{\texttt{T}}, \texttt{value} \in \texttt{T}
$$

4.11.5 为链表插入元素

插入元素，实际上就是创建一个新节点，并把这个节点插入到链表的指定位置，并更改节点之间的关系。

$$
\text{定义} : \text{InsertOfLL}\{\texttt{T}\}(\texttt{instance} \in LL_{\texttt{T}}, \texttt{position} \in PTR_{LLnode_{\texttt{T}}}, \texttt{value} \in \texttt{T}) =
$$

$$\left[\begin{array}{c}\left\{\begin{array}{l}\text{如果}:(*\texttt{position})\to next=\odot\\ \texttt{node}:=\uparrow LLnode_{\mathrm{T}}\\ (*\texttt{node})\to prev:=(*\texttt{position})\\ (*\texttt{node})\to next:=\odot\\ (*\texttt{node})\to data:=\texttt{value}\\ (*\texttt{position})\to next:=\texttt{node}\\ \texttt{instance}\to l:=^{\oplus}1\\ \text{输出}:\emptyset\end{array}\right.\\ \texttt{backnode}:=(*\texttt{position})\to next\\ \texttt{node}:=\uparrow LLnode_{\mathrm{T}}\\ (*\texttt{node})\to prev:=(*\texttt{position})\\ (*\texttt{node})\to next:=\texttt{backnode}\\ (*\texttt{node})\to data:=\texttt{value}\\ (*\texttt{position})\to next:=\texttt{node}\\ (*\texttt{backnode})\to prev:=\texttt{node}\\ \texttt{instance}\to l:=^{\oplus}1\end{array}\right]$$

InsertOfLL{T}要求$\texttt{instance}\to l>0$,且position指向instance的节点

定义：$\texttt{instance}\to\text{insert}(\texttt{position},\texttt{value})=\text{InsertOfLL}\{\texttt{T}\}$

$(\texttt{instance},\texttt{position},\texttt{value}),\texttt{instance}\in LL_{\mathrm{T}},\texttt{position}\in PTR_{LLnode_{\mathrm{T}}},\texttt{value}\in\texttt{T}$

4.11.6 为链表删除元素

删除元素，实际上就是删除链表中的指定节点，并更改节点之间的关系。

定义：$\text{RemoveOfLL}\{\texttt{T}\}(\texttt{instance}\in LL_{\mathrm{T}},\texttt{node}\in PTR_{LLnode_{\mathrm{T}}})=$

$$\left[\begin{array}{c}\texttt{frontnode}:=(*\texttt{node})\to prev\\ \texttt{backnode}:=(*\texttt{node})\to next\\ \left\{(*\texttt{frontnode})\to next:=\texttt{backnode}\quad\text{如果}:\texttt{frontnode}\neq\odot\right.\\ \left\{(*\texttt{backnode})\to prev:=\texttt{frontnode}\quad\text{如果}:\texttt{backnode}\neq\odot\right.\\ \downarrow\texttt{node}\\ \texttt{instance}\to l:=^{\ominus}1\end{array}\right]$$

RemoveOfLL{T}要求$\texttt{instance}\to l>0$,且node指向instance的节点

定义：$\texttt{instance}\to\text{remove}(\texttt{node})=$

$\text{RemoveOfLL}\{\texttt{T}\}(\texttt{instance},\texttt{node}),\texttt{instance}\in LL_{\mathrm{T}},\texttt{node}\in PTR_{LLnode_{\mathrm{T}}}$

4.11.7 链表的比较

链表的比较，与动态数组的比较类似，因此这里省略伪代码实现。判断两个链表是否相等，先保证长度相同，再判断每个节点的值是否相等。比较两个链表的大小，也可以使用先判断长度，再比较每个节点的值的方法。

单链表的实现

思考：我们用 SLL 指代单链表，那么 $SLL\{\mathtt{T}\}$ 的结构和操作该如何实现？

4.12 二叉树

【本节目标】

- **问题驱动**：理解我们为什么需要使用树。
- **掌握概念**：理解二叉树，乃至平衡树的结构和操作。
- **理解实现**：理解二叉树以及平衡树是如何实现的。
- **实践目标**：能够自主使用伪代码编写二叉树的实现。

开头：我们要解决什么问题？

在之前介绍动态数组和链表的章节中，我们使用了两种不同方法来存储线性结构的数据，这种线性数据结构统称为线性表。

但是，你有没有想过，如果我们要在一个存储了大量元素线性表中精准找到一个元素，它的时间复杂度也是线性的。如果需要大量地定位查找元素，这个时间成本可能会非常高。

因此，我们引出一种新的数据结构——树。

4.12.1 树的基本概念

树是由 $n(n \geq 0)$ 个节点组成的有限集合。当 $n = 0$ 时，称为空树；当 $n > 0$ 时，满足以下条件：

- 有一个特殊的节点称为**根节点**（Root）
- 其余节点可分为 $m(m \geq 0)$ 个互不相交的有限集合，每个集合本身又是一棵树，称为根的**子树**（Subtree）

树的比喻：可以把树想象成一棵倒置的树，根在上，枝叶在下。

4.12.2 树的术语

为了更好地描述树结构，我们需要定义一些基本术语：

- **节点**（Node）：树的基本单位，包含数据和指向其他节点的链接
- **边**（Edge）：连接两个节点的线
- **根节点**（Root）：没有父节点的节点，是树的起点
- **父节点**（Parent）：如果一个节点有子节点，那么它是这些子节点的父节点
- **子节点**（Child）：一个节点的直接下级节点
- **兄弟节点**（Sibling）：具有相同父节点的节点
- **叶节点**（Leaf）：没有子节点的节点（也称为终端节点）
- **内部节点**（Internal Node）：至少有一个子节点的非根节点
- **度**（Degree）：一个节点拥有的子节点数目
- **树的度**：树中所有节点的度的最大值
- **层次**（Level）：从根开始定义，根为第 1 层，根的子节点为第 2 层，以此类推
- **高度/深度**（Height/Depth）：树中节点的最大层次
- **路径**（Path）：从某个节点到另一个节点所经过的边的序列
- **祖先节点**（Ancestor）：从根到该节点路径上的所有节点
- **后代节点**（Descendant）：某个节点的子树中的所有节点

4.12.3 树的性质

树结构具有一些重要的数学性质：

- **连通性**：树中任意两个节点之间有且只有一条路径
- **无环性**：树中不存在环（从某个节点出发经过若干边不能回到该节点）
- **节点数与边数的关系**：具有 n 个节点的树有 $n-1$ 条边
- **层次性**：树具有天然的层次结构，便于表达父子关系

4.12.4 树的分类

根据节点的子节点数目和排列方式，树可以分为多种类型：

1. 二叉树（Binary Tree）

- 每个节点最多有两个子节点（左子节点和右子节点）
- 是最常用的一种树结构
- 包括：满二叉树、完全二叉树、二叉搜索树等

2. 多叉树（Multi-way Tree）

- 每个节点可以有多个子节点
- 包括：B 树、B+ 树、Trie 树等
- 常用于数据库和文件系统

3. 平衡树（Balanced Tree）

- 左右子树高度差不超过某个限定值
- 包括：AVL 树、红黑树等
- 保证搜索效率，避免退化为链表

4. 堆（Heap）

- 特殊的完全二叉树，满足堆性质
- 包括：最大堆、最小堆
- 常用于优先队列和堆排序

4.12.5 树的存储表示

在计算机中，树有多种存储方式：

1. 链式存储

- 每个节点包含数据域和指针域
- 指针指向子节点（对于二叉树，通常有左指针和右指针）
- 灵活，但指针占用额外空间

2. 顺序存储

- 使用数组存储树节点
- 通过下标计算父子关系（如完全二叉树的数组表示）
- 节省空间，但需要连续存储空间

3. 父指针表示法

- 每个节点存储指向父节点的指针
- 便于找到祖先路径

4. 孩子兄弟表示法

- 节点包含指向第一个孩子和下一个兄弟的指针
- 可以将多叉树转换为二叉树表示

二叉搜索树

在这个章节中，我们着重于二叉搜索树的实现。二叉搜索树是一种基于大小排序、每个节点有两个子节点、在最坏情况下会退化成链表的树结构。
二叉树的操作的时间复杂度：

- **插入**：最好 $O(\log \mathtt{n})$，最差 $O(\mathtt{n})$(退化成链表时)
- **删除**：最好 $O(\log \mathtt{n})$，最差 $O(\mathtt{n})$(退化成链表时)
- **查找**：最好 $O(\log \mathtt{n})$，最差 $O(\mathtt{n})$(退化成链表时)

4.12.6 二叉搜索树的结构

我们使用 BST 指代二叉搜索树。

$$\begin{bmatrix} \text{结构体} : BSTnode\{\mathtt{T}\} \\ data \in \mathtt{T} \\ left \in PTR_{BSTnode\{\mathtt{T}\}} \\ right \in PTR_{BSTnode\{\mathtt{T}\}} \end{bmatrix}$$

$$\begin{bmatrix} \text{结构体} : BST\{\mathtt{T}\} \\ root \in PTR_{BSTnode\{\mathtt{T}\}} \\ size \in NC_{\mathtt{WORDSIZEVALUE}} \end{bmatrix}$$

4.12.7 二叉搜索树的初始化

$$定义 : BST_{\mathtt{T}} = BST\{\mathtt{T}\}$$

$$定义 : \text{InitialBST}\{\mathtt{T}\}() \to BST_{\mathtt{T}} =$$

$$\begin{bmatrix} \mathtt{instance} \in BST_{\mathtt{T}} \\ \mathtt{instance} \to root := \odot \\ \mathtt{instance} \to size := 0 \\ 输出 : \mathtt{instance} \end{bmatrix}$$

$$定义 : \Delta BST_{\mathtt{T}} = \text{InitialBST}\{\mathtt{T}\}()$$

4.12.8 二叉搜索树的销毁

二叉搜索树的销毁，本质上就是递归销毁一个节点的所有子树。

$$定义 : \text{DestroyBST}\{\mathtt{T}\}(\mathtt{root} \in PTR_{BSTnode\{\mathtt{T}\}}) =$$

$$\begin{bmatrix} \begin{cases} 如果 : \mathtt{root} \neq \odot \\ \text{DestroyBST}(\mathtt{root} \to left) \\ \text{DestroyBST}(\mathtt{root} \to right) \\ \downarrow \mathtt{root} \end{cases} \\ 输出 : \emptyset \end{bmatrix}$$

$$定义 :\downarrow \mathtt{instance} = \text{DestroyBST}\{\mathtt{T}\}(\mathtt{instance} \to root), \mathtt{instance} \in BST_{\mathtt{T}}$$

4.12.9 为二叉搜索树添加节点

为二叉搜索树添加节点，本质上就是通过比较运算找到一个合适的位置，将节点插入到树中。

$$定义 : \text{CreateBSTNode}\{\mathtt{T}\}(\mathtt{value} \in \mathtt{T}) \to PTR_{BSTnode\{\mathtt{T}\}} =$$

$$\begin{bmatrix} \mathtt{node} :=\uparrow BSTnode\{\mathtt{T}\} \\ (*\mathtt{node}) \to data := \mathtt{value} \\ (*\mathtt{node}) \to left := \odot \\ (*\mathtt{node}) \to right := \odot \\ 输出 : \mathtt{node} \end{bmatrix}$$

$$定义 : \text{InsertNode}\{\mathtt{T}\}(\mathtt{root} \in PTR_{BSTnode\{\mathtt{T}\}}, \mathtt{value} \in \mathtt{T}) =$$

$$\left[\begin{array}{c} \left\{\begin{array}{ll} \text{输出}:\text{CreateBSTNode}(\texttt{value}) & \text{如果}:\texttt{root}=\odot \end{array}\right. \\ \left\{\begin{array}{l} \text{如果}:\texttt{value}<(*\texttt{root})\rightarrow data \\ \texttt{root}\rightarrow left:=\text{InsertNode}\{\texttt{T}\}(\texttt{root}\rightarrow left,\texttt{value}) \end{array}\right. \\ \left\{\begin{array}{l} \text{如果}:\texttt{value}>(*\texttt{root})\rightarrow data \\ \texttt{root}\rightarrow right:=\text{InsertNode}\{\texttt{T}\}(\texttt{root}\rightarrow right,\texttt{value}) \end{array}\right. \\ \text{输出}:\texttt{root} \end{array}\right]$$

$$\text{定义}:\text{InsertOfBST}\{\texttt{T}\}(\texttt{instance}\in BST_{\texttt{T}},\texttt{value}\in\texttt{T})=$$

$$\left[\begin{array}{c} \texttt{instance}\rightarrow root:=\text{InsertNode}\{\texttt{T}\}(\texttt{instance}\rightarrow root,\texttt{value}) \\ \texttt{instance}\rightarrow size:=^{\oplus}1 \end{array}\right]$$

$$\text{定义}:\texttt{instance}\rightarrow insert(\texttt{value})=$$

$$\text{InsertOfBST}\{\texttt{T}\}(\texttt{instance},\texttt{value}),\texttt{instance}\in BST_{\texttt{T}},\texttt{value}\in\texttt{T}$$

4.12.10 为二叉搜索树删除节点

删除节点有三种情况：

- 节点是叶子节点：直接删除。
- 节点有一个子节点：用子节点替代。
- 节点有两个子节点：用右子树最小节点（或左子树最大节点）替代，然后删除那个最小（或最大）节点。

$$\text{定义}:\text{FindMinNodeOfBST}\{\texttt{T}\}(\texttt{node}\in PTR_{BSTnode\{\texttt{T}\}})\rightarrow PTR_{BSTnode\{\texttt{T}\}}=$$

$$\left[\begin{array}{c} \texttt{current}:=\texttt{node} \\ \left[\begin{array}{c} \text{当}:(*\texttt{current})\rightarrow left\neq\odot \\ \texttt{current}:=\texttt{current}\rightarrow left \end{array}\right] \\ \text{输出}:\texttt{current} \end{array}\right]$$

$$\text{定义}:\text{RemoveBSTNode}\{\texttt{T}\}(\texttt{root}\in PTR_{BSTnode\{\texttt{T}\}},\texttt{value}\in\texttt{T})\rightarrow PTR_{BSTnode\{\texttt{T}\}}=$$

$$\left[\begin{array}{l}\texttt{right} := (*\texttt{root}) \to right \\ \left\{\begin{array}{l}\text{如果} : \texttt{value} < ((*\texttt{root}) \to data) \\ (*\texttt{root}) \to left := \text{RemoveBSTNode}\{\texttt{T}\}((*\texttt{root}) \to left, \texttt{value}) \\ \text{否则如果} : \texttt{value} > ((*\texttt{root}) \to data) \\ \texttt{right} := \text{RemoveBSTNode}\{\texttt{T}\}(\texttt{right}, \texttt{value}) \\ \text{否则} : \\ \left\{\begin{array}{l}\texttt{temp} := \texttt{right} \quad \text{如果} : (*\texttt{root}) \to left = \odot \\ \downarrow \texttt{root} \\ \text{输出} : \texttt{temp} \\ \texttt{temp} := (*\texttt{root}) \to left \quad \text{否则如果} : \texttt{right} = \odot \\ \downarrow \texttt{root} \\ \text{输出} : \texttt{temp} \\ \texttt{temp} := \text{FindMinNodeOfBST}\{\texttt{T}\}(\texttt{right}) \quad \text{否则} \\ (*\texttt{root}) \to data := (*\texttt{temp}) \to data \\ \texttt{right} := \text{RemoveBSTNode}\{\texttt{T}\}(\texttt{right}, (*\texttt{temp}) \to data)\end{array}\right. \\ \text{输出} : \texttt{root}\end{array}\right.$$

$$\text{定义} : \text{RemoveOfBST}\{\texttt{T}\}(\texttt{instance} \in BST_{\texttt{T}}, \texttt{value} \in \texttt{T}) =$$

$$\left[\begin{array}{c}\texttt{instance} \to root := \text{RemoveBSTNode}\{\texttt{T}\}(\texttt{instance} \to root, \texttt{value}) \\ \texttt{instance} \to size :=^{\ominus} 1\end{array}\right]$$

$$\text{RemoveOfBST}\{\texttt{T}\}\text{要求}\texttt{instance} \to size > 0$$

$$\text{定义} : \texttt{instance} \to remove(\texttt{value}) = \text{RemoveOfBST}\{\texttt{T}\}(\texttt{instance}, \texttt{value}),$$

$$\texttt{instance} \in BST_{\texttt{T}}, \texttt{value} \in \texttt{T}$$

4.12.11 查找二叉搜索树的节点

二叉搜索树的查找，本质上就是通过比较运算找到一个节点。

$$\text{定义} : \text{FindBSTNode}\{\texttt{T}\}(\texttt{root} \in PTR_{BSTnode\{\texttt{T}\}}, \texttt{value} \in \texttt{T}) \to PTR_{BSTnode\{\texttt{T}\}} =$$

$$\left[\begin{array}{c}\texttt{current} := \texttt{root} \\ \left[\begin{array}{c}\text{当} : \texttt{current} \neq \odot \wedge (*\texttt{current}) \to data \neq \texttt{value} \\ \left\{\begin{array}{l}\texttt{current} := \texttt{current} \to left \quad \text{如果} : \texttt{value} < (*\texttt{current}) \to data \\ \texttt{current} := \texttt{current} \to right \quad \text{否则}\end{array}\right.\end{array}\right] \\ \text{输出} : \texttt{current}\end{array}\right]$$

$$\text{定义}:\text{FindOfNode}\{\texttt{T}\}(\texttt{instance}\in BST_{\texttt{T}},\texttt{value}\in\texttt{T})=\!\!\rightarrow PTR_{BSTnode\{\texttt{T}\}}=$$
$$\Big[\text{输出}:\text{FindBSTNode}(\texttt{instance}\rightarrow root,\texttt{value})\Big]$$
$$\text{定义}:\texttt{instance}\rightarrow find(\texttt{value})=\text{FindOfNode}\{\texttt{T}\}(\texttt{instance},\texttt{value}),$$
$$\texttt{instance}\in BST_{\texttt{T}},\texttt{value}\in\texttt{T}$$

树的包装

对于一颗树，如果它的节点存储的数据是一个键值对，那么就可以通过键查找值。而如果存储的数据是一个普通的值，且值不允许重复，那么它具有集合的某些性质，例如唯一性。如果需要能重复存储相同的值，那么可以将节点存储的数据进行包装，附带一个计数器，没有值时为 0，存储一个值时为 1，每次重复存储相同的值时加 1，删除时减 1，直到删除为 0 时节点才被真正删除。

4.13 扩展：平衡树与红黑树简介

【本节目标】

- **问题驱动**：理解为什么普通的二叉搜索树需要” 平衡”。
- **掌握概念**：了解平衡树的基本概念和红黑树的核心规则。
- **理解原理**：理解红黑树如何通过简单的规则保证基本平衡。
- **建立认知**：了解红黑树在现实系统中的应用价值。

开头：二叉搜索树的” 短板”

在前面的章节中，我们学习了二叉搜索树。它有一个很明显的缺点：如果插入的数据恰好是有序的（比如依次插入 1, 2, 3, 4, 5），那么树就会退化成一条” 链”：

```
1
 \
  2
   \
    3
     \
      4
       \
        5
```

这种退化的树失去了二叉搜索树的优势，查找效率从 $O(\log n)$ 降低到了 $O(n)$，就像在链表中查找一样慢。

那么，有没有办法让树” 自动” 保持平衡，避免这种退化呢？这就是**平衡树**要解决的问题。

4.13.1 什么是平衡树？

平衡树是一种特殊的二叉搜索树，它通过一定的规则和操作，保证树的高度始终保持在 $O(\log n)$ 的水平。
核心思想：让树的左右两边”差不多高”，避免一边倒的情况。
比喻理解：想象一棵真实的树，如果它长得歪向一边，就容易倒下。园丁会修剪枝条，让树保持平衡。平衡树也是这样，在插入或删除节点时，会进行”修剪”（旋转操作），让树保持平衡。

4.13.2 红黑树：一种高效的平衡树

红黑树是平衡树中最著名、应用最广泛的一种。它通过给每个节点增加一个”颜色”标记（红色或黑色），并遵循五条简单的规则，来保证树的基本平衡。
红黑树的五条规则：

1. **颜色规则**：每个节点要么是红色，要么是黑色。
2. **根节点规则**：根节点必须是黑色。
3. **叶子节点规则**：所有叶子节点（空节点）都是黑色。
4. **红色节点规则**：红色节点的两个子节点都必须是黑色（即不能有两个连续的红色节点）。
5. **路径规则**：从任一节点到其每个叶子节点的所有路径都包含相同数目的黑色节点。

这五条规则的神奇效果：

- 规则 4 和规则 5 共同作用，保证了最长的路径（红黑交替）不会超过最短路径（全黑）的两倍。
- 这意味着树的高度始终被控制在 $2\log(n+1)$ 以内，保证了 $O(\log n)$ 的查找效率。

简单例子：假设我们有一个小型红黑树，从根节点到叶子节点有 3 个黑色节点。那么：

- 最短路径：全是黑色节点，长度为 3
- 最长路径：红黑交替，如黑-红-黑-红-黑，长度为 5
- 最长路径不超过最短路径的两倍（$5 \leq 2 \times 3$）

4.13.3 红黑树如何维持平衡？

当插入或删除节点可能破坏平衡时，红黑树通过两种操作来修复：

1. 变色：改变节点的颜色（红变黑或黑变红）

2. 旋转：调整节点的位置关系（左旋或右旋）

- **左旋**：以某个节点为支点，将其右子节点” 提上来”
- **右旋**：以某个节点为支点，将其左子节点” 提上来”

维护过程（概念性理解，非具体实现）：

1. 插入新节点时，通常先设为红色（这样不会违反规则 5）
2. 如果新节点的父节点也是红色，就违反了规则 4（不能有两个连续的红色节点）
3. 这时需要通过变色和旋转来修复：可能改变父节点、叔节点、祖父节点的颜色，或者进行旋转
4. 重复这个过程，直到满足所有规则

4.13.4 红黑树的优势

- 平衡性较好，保证 $O(\log n)$ 的操作时间
- 维护成本相对较低（比 AVL 树等完全平衡树的旋转次数少）
- 在各种操作（插入、删除、查找）之间取得了很好的平衡

本节总结

- **平衡需求**：避免二叉搜索树退化为链表，保证 $O(\log n)$ 的效率
- **红黑树原理**：通过颜色标记和五条规则保证基本平衡
- **维护方式**：通过变色和旋转操作修复不平衡
- **应用广泛**：在编程语言、操作系统、数据库等系统中大量使用

红黑树展示了计算机科学中的一个重要理念：通过简单的规则和巧妙的操作，可以解决复杂的问题。虽然具体的实现比较复杂，但理解其基本思想对学习数据结构和算法很有帮助。

在伪代码中，我们不实现红黑树，但是之后我们可能会使用红黑树来实现一些算法。因为二叉搜索树不平衡，所以不使用它，红黑树沿用二叉搜索树的操作。我们用 RBT 指代红黑树，例如 $\texttt{tree} \in RBT_{\texttt{T}}$，其中 `T` 为数据类型。允许使用：

$$\texttt{tree} \to \text{setrule}(\texttt{f} \in \text{func}\{\texttt{T}\}(\texttt{left} \in \texttt{T}, \texttt{right} \in \texttt{T}) \to NC_2) \to \emptyset$$

来自定义红黑树的排序规则，红黑树的插入和删除操作，以及查找操作都将使用这个排序规则。这个算法只允许在树中没有元素的时候使用，否则可能导致冲突问题。
思考题：如果一棵红黑树有 100 个节点，根据规则 5，从根节点到叶子节点的黑色节点数量最多可能相差多少倍？为什么？

思考题提示

根据红黑树的规则 5，所有路径的黑色节点数量相同。但规则 4 允许红色节点存在，所以路径长度可能不同。最长路径（红黑交替）不会超过最短路径（全黑）的两倍。

4.14 哈希表

【本节目标】

- **问题驱动**：理解为什么需要一种比树结构更快的查找方式。
- **掌握概念**：了解哈希表的基本原理和核心组件。
- **理解操作**：掌握哈希表的插入、查找和删除操作原理。
- **认识挑战**：了解哈希冲突及其解决方法。

开头：从直接寻址到哈希表

想象一下，如果我们有一个巨大的数组，并且知道每个数据的确切位置，那么查找数据只需要 $O(1)$ 时间——直接访问数组下标即可。但现实是，数据的关键字可能范围很大（如身份证号），我们无法为所有可能的关键字分配存储空间。

哈希表提供了一种巧妙的解决方案：通过一个”魔法函数”（哈希函数）将任意大小的关键字映射到固定大小的数组索引范围内，从而实现近似 $O(1)$ 的查找效率。

4.14.1 哈希表的核心组件

哈希表由三个核心部分组成：

1. 哈希函数（Hash Function）

- 将关键字映射到数组索引的函数：`index` = hash(`key`)
- 理想特性：计算速度快、分布均匀（减少冲突）
- 简单示例：对关键字取模 hash(`key`) = `key` mod `TableSize`

2. 哈希数组（Hash Array）

- 存储数据的固定大小数组
- 每个位置称为一个”桶”（bucket）

3. 冲突解决机制（Collision Resolution）

- 当不同关键字映射到同一索引时（哈希冲突）的处理方法
- 主要方法：链地址法、开放地址法

4.14.2 哈希表的基本操作

哈希表支持三种基本操作，理想情况下都能在 $O(1)$ 时间内完成：

1. 插入操作（Insert）

1. 计算关键字的哈希值：$\texttt{index} = \text{hash}(\texttt{key})$
2. 检查该位置是否已存在其他元素（冲突检测）
3. 根据冲突解决策略存放数据：
   - **链地址法**：将新元素添加到该位置的链表中
   - **开放地址法**：按照探测序列寻找下一个空位

2. 查找操作（Lookup/Search）

1. 计算关键字的哈希值：$\texttt{index} = \text{hash}(\texttt{key})$
2. 检查该位置的元素：
   - 如果使用链地址法，遍历该位置的链表
   - 如果使用开放地址法，按照探测序列查找
3. 找到匹配关键字则返回数据，否则返回不存在

3. 删除操作（Delete）

1. 计算关键字的哈希值：$\texttt{index} = \text{hash}(\texttt{key})$
2. 找到该关键字对应的元素（类似查找过程）
3. 根据冲突解决策略移除元素：
   - **链地址法**：从链表中移除相应节点
   - **开放地址法**：特殊标记而非真正删除（避免破坏探测序列）

4.14.3 哈希冲突的解决策略

哈希冲突是不可避免的（鸽巢原理），主要有两种解决策略：

1. 链地址法（Chaining）

- 每个桶存储一个链表（或其他容器）
- 冲突元素都添加到同一桶的链表中
- 优点：简单有效，负载因子可以大于 1
- 缺点：需要额外的指针空间，缓存不友好

2. 开放地址法（Open Addressing）

- 所有元素都存放在哈希数组本身中
- 冲突时按照探测序列寻找下一个空桶
- 常见探测方法：
  - 线性探测：$h(\mathtt{k}, \mathtt{i}) = (\text{hash}(\mathtt{k}) + \mathtt{i}) \bmod \mathtt{m}$
  - 平方探测：$h(\mathtt{k}, \mathtt{i}) = (\text{hash}(\mathtt{k}) + \mathtt{c}_1\mathtt{i} + \mathtt{c}_2\mathtt{i}^2) \bmod \mathtt{m}$
  - 双重哈希：$h(\mathtt{k}, \mathtt{i}) = (\text{hash}_1(\mathtt{k}) + \mathtt{i} \cdot \text{hash}_2(\mathtt{k})) \bmod \mathtt{m}$
- 优点：不需要额外空间，缓存友好
- 缺点：负载因子必须小于 1，删除操作复杂

4.14.4 哈希表的性能因素

哈希表的性能主要取决于以下几个因素：

1. 哈希函数质量

- 应该将关键字均匀分布到各个桶中
- 减少冲突的可能性

2. 负载因子（Load Factor）

- 定义：$\alpha = \frac{\mathtt{n}}{\mathtt{m}}$，其中 $\mathtt{n}$ 是元素数量，$\mathtt{m}$ 是桶数量
- 负载因子越高，冲突概率越大
- 通常需要扩容（rehashing）当负载因子超过阈值（如 0.7-0.8）

3. 冲突解决策略

- 不同策略在不同场景下性能特征不同

本节总结

- **核心思想**：通过哈希函数将关键字映射到数组索引，实现快速访问
- **三大操作**：插入、查找、删除都能在平均 $O(1)$ 时间内完成
- **关键挑战**：哈希冲突需要通过链地址法或开放地址法解决
- **性能因素**：哈希函数质量、负载因子和冲突解决策略影响性能

哈希表是计算机科学中最重要的数据结构之一，它展示了如何通过巧妙的设计将平均时间复杂度从 $O(\log n)$（树结构）提升到 $O(1)$，这种效率提升在实际系统中往往意味着性能的数量级差异。
思考题：如果哈希函数总是将所有关键字映射到同一个桶中，哈希表的性能会如何变化？这种情况下时间复杂度是多少？

思考题提示

当哈希函数产生严重冲突时，哈希表退化为一个链表（链地址法）或需要遍历整个数组（开放地址法），时间复杂度从 $O(1)$ 退化到 $O(n)$。这强调了哈希函数均匀分布的重要性。

5 附录：伪代码规则参考

基本语法规则

1. 变量定义

- `var` := $NC_{\mathtt{n}}(\mathtt{m})$ 定义状态数为 `n`、值为 `m` 的数胞变量
- `var` := `value` 修改变量的值
- `var` $\in NC_{\mathtt{n}}$ 声明变量类型或判断类型

2. 全局变量

- `BASE`：计算机使用的进制
- `BYTELENGTH`：一个字节的位数
- `WORDSIZE`：字长的字节数
- $\mathtt{WORDSIZEVALUE} = \mathtt{BASE}^{\mathtt{BYTELENGTH}\times\mathtt{WORDSIZE}}$

运算符

1. 算术运算

- $\oplus$：模加法 $\mathtt{a}\oplus\mathtt{b} = (\mathtt{a}+\mathtt{b}) \bmod \mathtt{n}$
- $\ominus$：模减法 $\mathtt{a}\ominus\mathtt{b} = (\mathtt{a}-\mathtt{b}) \bmod \mathtt{n}$
- $\otimes$：模乘法 $\mathtt{a}\otimes\mathtt{b} = (\mathtt{a}\times\mathtt{b}) \bmod \mathtt{n}$
- $\oslash$：整数除法 $\mathtt{a}\oslash\mathtt{b} = \lfloor \mathtt{a}\div\mathtt{b} \rfloor$
- rem：取余运算

2. 逻辑运算

- $\neg$：非运算
- $\wedge$：与运算
- $\vee$：或运算
- $\veebar$：异或运算

3. 比较运算

- $=, \neq, <, >, \leq, \geq$: 标准比较运算符
- 结果返回布尔值 **T** 或 **F**

4. 指针运算

- &: 取地址运算
- $*$: 解引用运算
- $\uparrow$: 申请空间运算
- $\downarrow$: 释放空间运算
- $\boxplus$: 指针加法
- $\boxminus$: 指针减法

控制流结构

1. 顺序结构

- 默认按书写顺序执行

2. 分支结构

$$\begin{cases} \text{语句 1} & \text{如果 : 条件 1} \\ \text{语句 2} & \text{否则如果 : 条件 2} \\ \vdots & \\ \text{语句 n} & \text{否则} \end{cases}$$

3. 循环结构

- 先判断循环: $\begin{bmatrix} \text{当 : 循环条件} \\ \text{循环体} \end{bmatrix}$
- 后判断循环: $\begin{bmatrix} \text{循环体} \\ \text{当 : 循环条件} \end{bmatrix}$

数据类型定义

1. 结构体定义

$$\left[\begin{array}{l}\text{结构体}:\text{结构体名称}\{\text{类型 1},\ldots,\text{类型 n}\}\\ \qquad\text{成员 1}\in\text{成员 1 的类型}\\ \qquad\qquad\vdots\\ \qquad\text{成员 n}\in\text{成员 n 的类型}\end{array}\right]$$

2. 联合体定义

$$\left[\begin{array}{l}\text{联合体}:\text{联合体名称}\{\text{类型 1},\ldots,\text{类型 n}\}\\ \qquad\text{成员 1}\in\text{成员 1 的类型}\\ \qquad\qquad\vdots\\ \qquad\text{成员 n}\in\text{成员 n 的类型}\end{array}\right]$$

3. 算法定义

$$\text{定义}:\text{算法名称}\{\text{模板组}\}(\text{参数组})\to\text{输出类型}=\left[\text{算法内容}\right]$$

内置函数

1. 类型相关

- typeof(v)：获取变量的类型信息
- sizeof(v) 或 sizeof(T)：获取存储空间大小
- bool(a)：将值转换为布尔值（非零为 **T**，零为 **F**）
- boolvalue(**T**) = 1, boolvalue(**F**) = 0

2. 类型转换

- SNC(a)：将数胞转换为有符号数胞
- NC(b)：将有符号数胞转换为数胞

数据结构操作

1. 数组操作

- arr $\in ARR_{\mathrm{T}}$(n)：声明长度为 n 的数组

- `arr[i]`：访问数组元素（`i < n`）

2. 动态数组操作

- $\uparrow DA_{\text{T}}$(`n`)：初始化动态数组
- `da` → resize(`n`)：调整大小
- `da` → append(`element`)：添加元素
- `da` → insert(`element`, `position`)：插入元素
- `da` → remove()：删除元素

3. 链表操作

- ΔLL_{T}：初始化链表
- `ll` → append(`value`)：添加元素
- `ll` → insert(`position`, `value`)：插入元素
- `ll` → remove(`node`)：删除元素

4. 二叉树操作

- ΔBST_{T}：初始化二叉搜索树
- `bst` → insert(`value`)：插入节点
- `bst` → remove(`value`)：删除节点
- `bst` → find(`value`)：查找节点

5. 红黑树操作

- ΔRBT_{T}：初始化红黑树
- `rbt` → insert(`value`)：插入节点
- `rbt` → remove(`value`)：删除节点
- `rbt` → find(`value`)：查找节点
- `rbt` → setrule(`func`)：设置排序规则

特殊语法

1. 输出语句

- 在算法中使用 输出 : `value` 返回结果
- 如果省略输出类型，算法不返回值

2. 自定义定义

- 定义 : 定义内容：允许用户自定义规则
- 新定义不能与现有规则冲突

3. 成员访问

- `struct` → `member`：访问结构体/联合体成员
- `object` → `method`(`args`)：调用对象方法

霍尔逻辑表示法

1. 霍尔三元组

- $\{P\}\ C\ \{Q\}$：前置条件 P，程序 C，后置条件 Q
- 含义：如果 P 成立且 C 终止，则 Q 成立

2. 推理规则

- 顺序规则：$\dfrac{\{P\}\ C_1\ \{R\} \quad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}}$
- 分支规则：$\dfrac{\{P \wedge B\}\ C_1\ \{Q\} \quad \{P \wedge \neg B\}\ C_2\ \{Q\}}{\{P\}\ \text{如果}B\ \text{则}C_1\ \text{否则}C_2\ \{Q\}}$
- 循环规则：$\dfrac{\{I \wedge B\}\ C\ \{I\}}{\{I\}\ \text{当}B\ \text{执行}C\ \{I \wedge \neg B\}}$

存储空间模型

1. 状态表示

- $\sigma : L \to V$：存储空间状态函数
- L：地址集合，V：值集合

2. 状态更新

- $\sigma_{新} = \sigma_{旧}[地址 \to 新值]$
- 表示除了指定地址外，其他地址值不变

复杂度表示法

1. 大 O 表示法

- $O(1)$：常数复杂度
- $O(\log n)$：对数复杂度
- $O(n)$：线性复杂度
- $O(n^2)$：平方复杂度
- $O(2^n)$：指数复杂度

2. 频度函数

- $T(n)$：基本操作执行次数
- 用于分析时间复杂度和空间复杂度