3. 数据结构简介

3.1. 数据与内存

3.1.1. 基本数据类型

谈及计算机中的数据，我们会想到文本、图片、视频、语音、3D 模型等各种形式。尽管这些数据的组织形式各异，但它们都由各种基本数据类型构成。

「基本数据类型」是 CPU 可以直接进行运算的类型，在算法中直接被使用。

- 「整数」按照不同的长度分为 byte, short, int, long 。在满足取值范围的前提下，我们应该尽量选取较短的整数类型，以减小内存空间占用；
- 「浮点数」表示小数，按长度分为 float, double ，选用规则与整数相同。
- 「字符」在计算机中以字符集形式保存，char 的值实际上是数字，代表字符集中的编号，计算机通过字符集查表完成编号到字符的转换。
- 「布尔」代表逻辑中的“是”与“否”，其占用空间需根据编程语言确定。

类别	符号	占用空间	取值范围	默认值
整数	byte	1 byte	-2^7 ~ 2^7-1 (-128 ~ 127)	0
	short	2 bytes	-2^{15} ~ $2^{15}-1$	0
	int	4 bytes	-2^{31} ~ $2^{31}-1$	0
	long	8 bytes	-2^{63} ~ $2^{63}-1$	0
浮点数	**float**	4 bytes	-3.4×10^{38} ~ 3.4×10^{38}	0.0 f
	double	8 bytes	-1.7×10^{308} ~ 1.7×10^{308}	0.0
字符	**char**	2 bytes / 1 byte	0 ~ $2^{16}-1$	0
布尔	**bool**	1 byte / 1 bit	true 或 false	false

以上表格中，加粗项在算法题中最为常用。此表格无需硬背，大致理解即可，需要时可以通过查表来回忆。

整数表示方式

整数的取值范围取决于变量使用的内存长度，即字节（或比特）数。在计算机中，1 字节 (byte) = 8 比特 (bit)，1 比特即 1 个二进制位。以 int 类型为例：

1. 整数类型 int 占用 4 bytes = 32 bits ，可以表示 2^{32} 个不同的数字；
2. 将最高位视为符号位，0 代表正数，1 代表负数，一共可表示 2^{31} 个正数和 2^{31} 个负数；

3. 当所有 bits 为 0 时代表数字 0 ，从零开始增大，可得最大正数为 $2^{31} - 1$ ；
4. 剩余 2^{31} 个数字全部用来表示负数，因此最小负数为 -2^{31} ；具体细节涉及“源码、反码、补码”的相关知识，有兴趣的同学可以查阅学习；

其他整数类型 byte, short, long 的取值范围的计算方法与 int 类似，在此不再赘述。

浮点数表示方式 *

本书中，标题后的 * 符号代表选读章节。如果你觉得理解困难，建议先跳过，等学完必读章节后再单独攻克。

细心的你可能会发现：int 和 float 长度相同，都是 4 bytes，但为什么 float 的取值范围远大于 int ？按理说 float 需要表示小数，取值范围应该变小才对。

实际上，这是因为浮点数 float 采用了不同的表示方式。根据 IEEE 754 标准，32-bit 长度的 float 由以下部分构成：

- 符号位 S： 占 1 bit ；
- 指数位 E： 占 8 bits ；
- 分数位 N： 占 24 bits ，其中 23 位显式存储；

设 32-bit 二进制数的第 i 位为 b_i ，则 float 值的计算方法定义为：

$$\text{val} = (-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

转化到十进制下的计算公式为

$$\text{val} = (-1)^{\text{S}} \times 2^{\text{E}-127} \times (1 + \text{N})$$

其中各项的取值范围为

$$\begin{aligned} \text{S} \in &\{0, 1\}, \quad \text{E} \in \{1, 2, \ldots, 254\} \\ (1 + \text{N}) = &(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) \subset [1, 2 - 2^{-23}] \end{aligned}$$

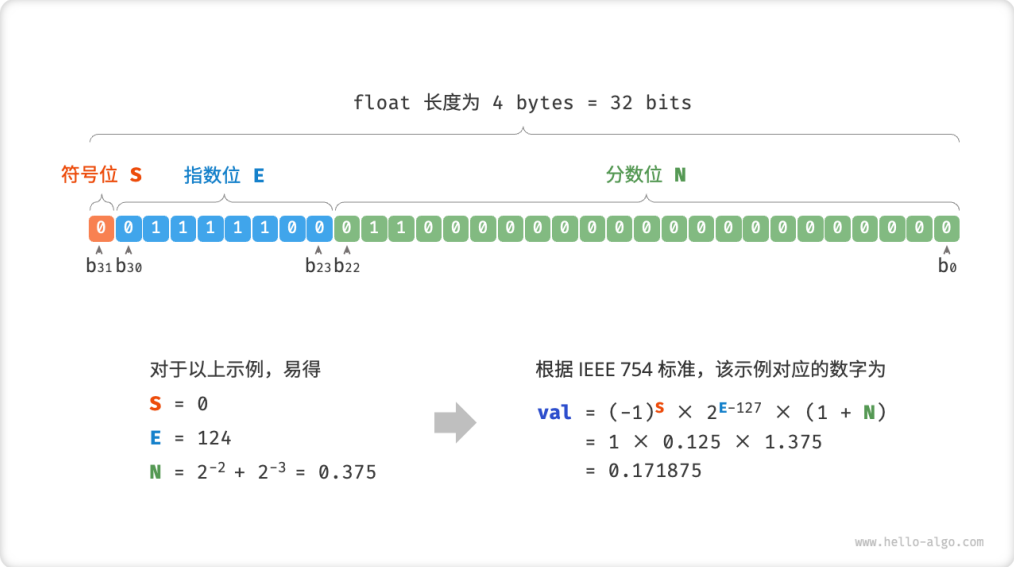

Figure 3-1. IEEE 754 标准下的 float 表示方式

以上图为例，$\mathrm{S} = 0$，$\mathrm{E} = 124$，$\mathrm{N} = 2^{-2} + 2^{-3} = 0.375$，易得

$$\text{val} = (-1)^0 \times 2^{124-127} \times (1 + 0.375) = 0.171875$$

现在我们可以回答最初的问题：**float 的表示方式包含指数位，导致其取值范围远大于 int**。根据以上计算，float 可表示的最大正数为 $2^{254-127} \times (2 - 2^{-23}) \approx 3.4 \times 10^{38}$，切换符号位便可得到最小负数。

尽管浮点数 float 扩展了取值范围，但其副作用是牺牲了精度。整数类型 int 将全部 32 位用于表示数字，数字是均匀分布的；而由于指数位的存在，浮点数 float 的数值越大，相邻两个数字之间的差值就会趋向越大。

进一步地，指数位 $E = 0$ 和 $E = 255$ 具有特殊含义，**用于表示零、无穷大、NaN 等**。

指数位 E	分数位 $\mathrm{N} = 0$	分数位 $\mathrm{N} \neq 0$	计算公式
0	± 0	次正规数	$(-1)^{\mathrm{S}} \times 2^{-126} \times (0.\mathrm{N})$
$1, 2, \dots, 254$	正规数	正规数	$(-1)^{\mathrm{S}} \times 2^{(\mathrm{E}-127)} \times (1.\mathrm{N})$
255	$\pm\infty$	NaN	

特别地，次正规数显著提升了浮点数的精度，这是因为：

- 最小正正规数为 $2^{-126} \approx 1.18 \times 10^{-38}$；
- 最小正次正规数为 $2^{-126} \times 2^{-23} \approx 1.4 \times 10^{-45}$；

双精度 double 也采用类似 float 的表示方法，此处不再详述。

基本数据类型与数据结构的关系

我们知道，**数据结构是在计算机中组织与存储数据的方式**，它的核心是“结构”，而非“数据”。如果想要表示“一排数字”，我们自然会想到使用「数组」数据结构。数组的存储方式可以表示数字的相邻关系、顺序关系，但至于具体存储的是整数 int 、小数 float 、还是字符 char ，则与“数据结构”无关。换句话说，基本数据类型提供了数据的“内容类型”，而数据结构提供了数据的“组织方式”。

```
/* 使用多种「基本数据类型」来初始化「数组」 */
int numbers[5];
float decimals[5];
char characters[5];
bool booleans[5];
```

3.1.2. 计算机内存

在计算机中，内存和硬盘是两种主要的存储硬件设备。「硬盘」主要用于长期存储数据，容量较大（通常可达到 TB 级别）、速度较慢。「内存」用于运行程序时暂存数据，速度较快，但容量较小（通常为 GB 级别）。

在算法运行过程中，相关数据都存储在内存中。下图展示了一个计算机内存条，其中每个黑色方块都包含一块内存空间。我们可以将内存想象成一个巨大的 Excel 表格，其中每个单元格都可以存储 1 byte 的数据，在算法运行时，所有数据都被存储在这些单元格中。

系统通过「内存地址 Memory Location」来访问目标内存位置的数据。计算机根据特定规则为表格中的每个单元格分配编号，确保每个内存空间都有唯一的内存地址。有了这些地址，程序便可以访问内存中的数据。

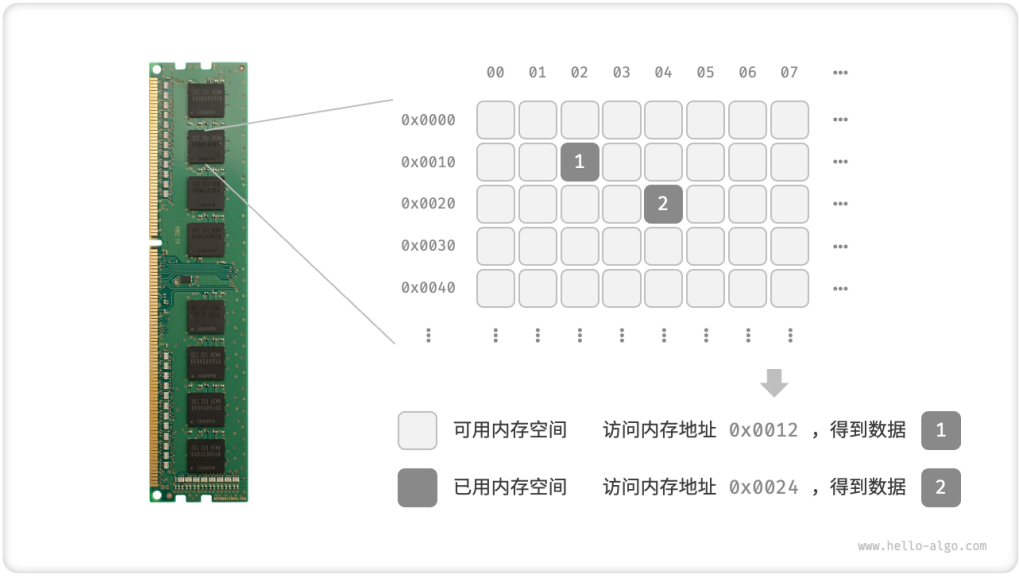

Figure 3-2. 内存条、内存空间、内存地址

在数据结构与算法的设计中，内存资源是一个重要的考虑因素。内存是所有程序的共享资源，当内存被某个程序占用时，其他程序无法同时使用。我们需要根据剩余内存资源的实际情况来设计算法。例如，算法所占用的

内存峰值不应超过系统剩余空闲内存；如果运行的程序很多并且缺少大量连续的内存空间，那么所选用的数据结构必须能够存储在离散的内存空间内。

3.2. 数据结构分类

数据结构可以从逻辑结构和物理结构两个维度进行分类。

3.2.1. 逻辑结构：线性与非线性

「**逻辑结构**」**揭示了数据元素之间的逻辑关系**。在数组和链表中，数据按照顺序依次排列，体现了数据之间的线性关系；而在树中，数据从顶部向下按层次排列，表现出祖先与后代之间的派生关系；图则由节点和边构成，反映了复杂的网络关系。

逻辑结构通常分为「线性」和「非线性」两类。线性结构比较直观，指数据在逻辑关系上呈线性排列；非线性结构则相反，呈非线性排列，例如网状或树状结构。

- **线性数据结构**：数组、链表、栈、队列、哈希表；
- **非线性数据结构**：树、图、堆、哈希表；

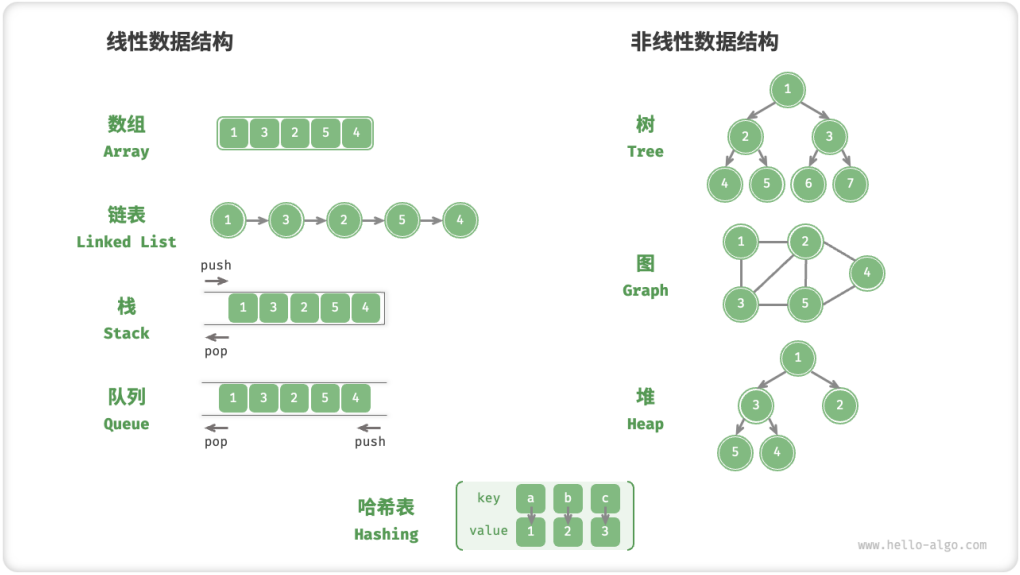

Figure 3-3. 线性与非线性数据结构

3.2.2. 物理结构：连续与离散

如若阅读起来有困难，建议先阅读下一章“数组与链表”，然后再回头理解物理结构的含义。

「**物理结构**」**体现了数据在计算机内存中的存储方式**，可以分为数组的连续空间存储和链表的离散空间存储。物理结构从底层决定了数据的访问、更新、增删等操作方法，同时在时间效率和空间效率方面呈现出互补的特点。

Figure 3-4. 连续空间存储与离散空间存储

所有数据结构都是基于数组、链表或二者的组合实现的。例如，栈和队列既可以使用数组实现，也可以使用链表实现；而哈希表的实现可能同时包含数组和链表。

- **基于数组可实现**：栈、队列、哈希表、树、堆、图、矩阵、张量（维度 ≥ 3 的数组）等；
- **基于链表可实现**：栈、队列、哈希表、树、堆、图等；

基于数组实现的数据结构也被称为「静态数据结构」，这意味着此类数据结构在初始化后长度不可变。相对应地，基于链表实现的数据结构被称为「动态数据结构」，这类数据结构在初始化后，仍可以在程序运行过程中对其长度进行调整。

数组与链表是其他所有数据结构的“底层积木”，建议读者投入更多时间深入了解这两种基本数据结构。

3.3. 小结

- 计算机中的基本数据类型包括整数 byte, short, int, long 、浮点数 float, double 、字符 char 和布尔 boolean ，它们的取值范围取决于占用空间大小和表示方式。
- 当程序运行时，数据被存储在计算机内存中。每个内存空间都拥有对应的内存地址，程序通过这些内存地址访问数据。
- 数据结构可以从逻辑结构和物理结构两个角度进行分类。逻辑结构描述了数据元素之间的逻辑关系，而物理结构描述了数据在计算机内存中的存储方式。

- 常见的逻辑结构包括线性、树状和网状等。通常我们根据逻辑结构将数据结构分为线性（数组、链表、栈、队列）和非线性（树、图、堆）两种。哈希表的实现可能同时包含线性和非线性结构。
- 物理结构主要分为连续空间存储（数组）和离散空间存储（链表）。所有数据结构都是由数组、链表或两者的组合实现的。

4. 数组与链表

4.1. 数组

「数组 Array」是一种线性数据结构，其将相同类型元素存储在连续的内存空间中。我们将元素在数组中的位置称为元素的「索引 Index」。

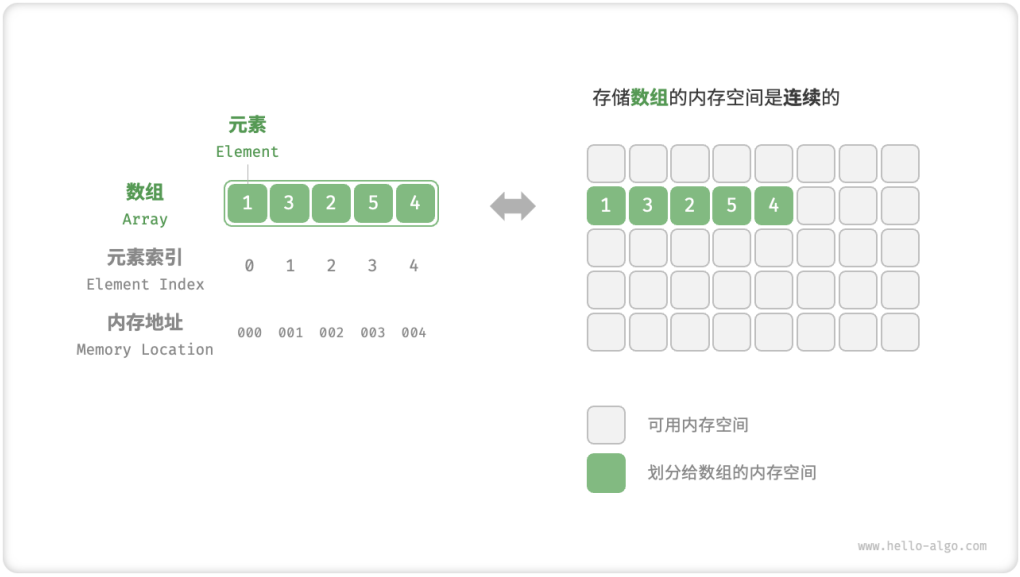

Figure 4-1. 数组定义与存储方式

数组初始化。通常有无初始值和给定初始值两种方式，我们可根据需求选择合适的方法。在未给定初始值的情况下，数组的所有元素通常会被初始化为默认值 0 。

```
// === File: array.cpp ===
/* 初始化数组 */
// 存储在栈上
int arr[5];
int nums[5] { 1, 3, 2, 5, 4 };
// 存储在堆上
int* arr1 = new int[5];
int* nums1 = new int[5] { 1, 3, 2, 5, 4 };
```

4.1.1. 数组优点

在数组中访问元素非常高效。由于数组元素被存储在连续的内存空间中，因此计算数组元素的内存地址非常容易。给定数组首个元素的地址和某个元素的索引，我们可以使用以下公式计算得到该元素的内存地址，从而直接访问此元素。

Figure 4-2. 数组元素的内存地址计算

```
# 元素内存地址 = 数组内存地址 + 元素长度 * 元素索引
elementAddr = firtstElementAddr + elementLength * elementIndex
```

为什么数组元素的索引要从 0 开始编号呢？

观察上图，我们发现数组首个元素的索引为 0 ，这似乎有些反直觉，因为从 1 开始计数会更自然。

然而，从地址计算公式的角度看，**索引本质上表示的是内存地址的偏移量**。首个元素的地址偏移量是 0 ，因此索引为 0 也是合理的。

访问元素的高效性带来了诸多便利。例如，我们可以在 $O(1)$ 时间内随机获取数组中的任意一个元素。

```
// === File: array.cpp ===
/* 随机返回一个数组元素 */
int randomAccess(int *nums, int size) {
    // 在区间 [0, size) 中随机抽取一个数字
    int randomIndex = rand() % size;
    // 获取并返回随机元素
    int randomNum = nums[randomIndex];
    return randomNum;
}
```

4.1.2. 数组缺点

数组在初始化后长度不可变。由于系统无法保证数组之后的内存空间是可用的，因此数组长度无法扩展。而若希望扩容数组，则需新建一个数组，然后把原数组元素依次拷贝到新数组，在数组很大的情况下，这是非常耗时的。

```
// === File: array.cpp ===
/* 扩展数组长度 */
int *extend(int *nums, int size, int enlarge) {
    // 初始化一个扩展长度后的数组
    int *res = new int[size + enlarge];
    // 将原数组中的所有元素复制到新数组
    for (int i = 0; i < size; i++) {
        res[i] = nums[i];
    }
    // 释放内存
    delete[] nums;
    // 返回扩展后的新数组
    return res;
}
```

数组中插入或删除元素效率低下。如果我们想要在数组中间插入一个元素，由于数组元素在内存中是“紧挨着的”，它们之间没有空间再放任何数据。因此，我们不得不将此索引之后的所有元素都向后移动一位，然后再把元素赋值给该索引。

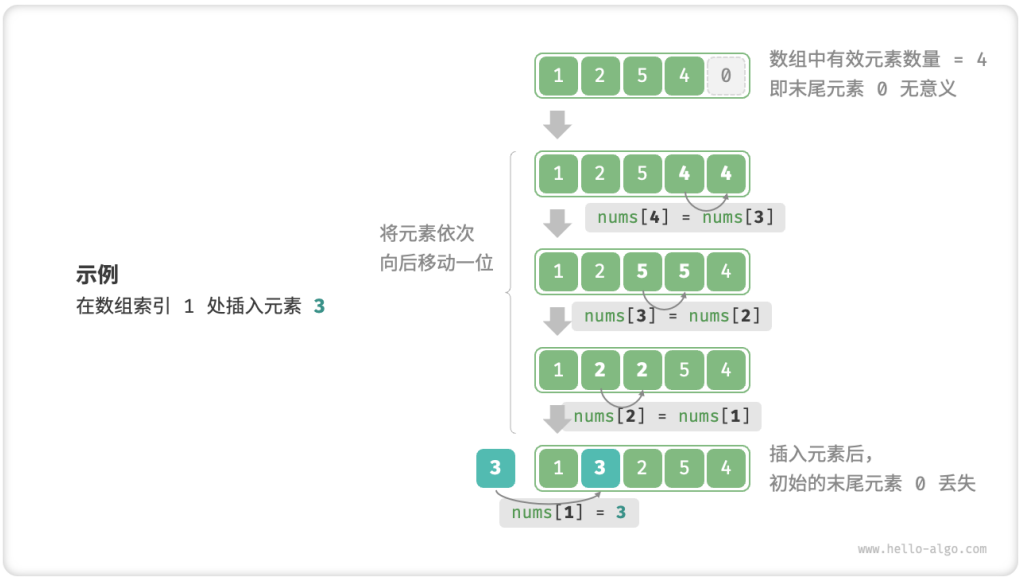

Figure 4-3. 数组插入元素

```
// === File: array.cpp ===
/* 在数组的索引 index 处插入元素 num */
void insert(int *nums, int size, int num, int index) {
    // 把索引 index 以及之后的所有元素向后移动一位
    for (int i = size - 1; i > index; i--) {
        nums[i] = nums[i - 1];
    }
    // 将 num 赋给 index 处元素
    nums[index] = num;
}
```

删除元素也类似，如果我们想要删除索引 i 处的元素，则需要把索引 i 之后的元素都向前移动一位。值得注意的是，删除元素后，原先末尾的元素变得“无意义”了，我们无需特意去修改它。

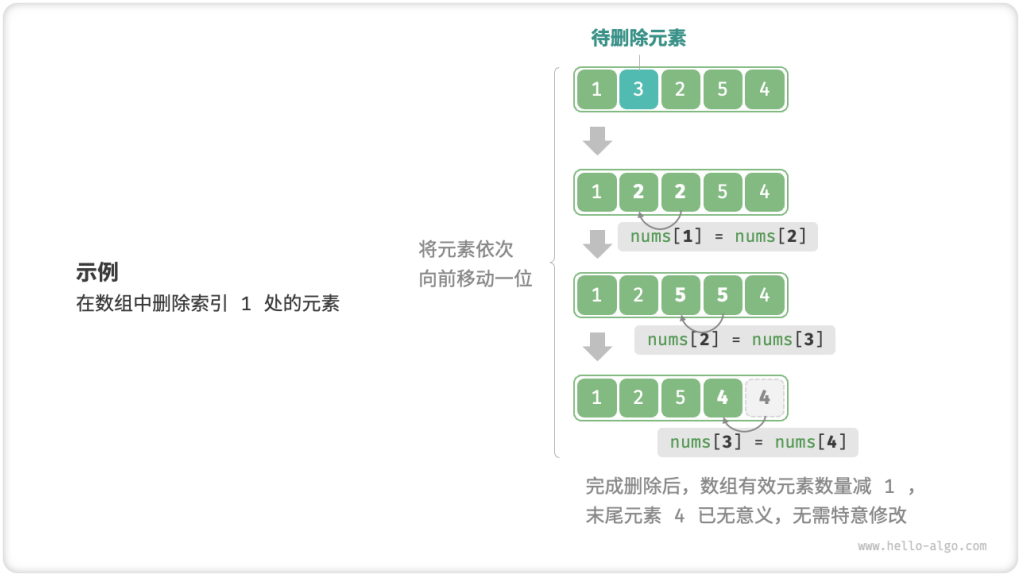

Figure 4-4. 数组删除元素

```
// === File: array.cpp ===
/* 删除索引 index 处元素 */
void remove(int *nums, int size, int index) {
    // 把索引 index 之后的所有元素向前移动一位
    for (int i = index; i < size - 1; i++) {
        nums[i] = nums[i + 1];
    }
}
```

总结来看，数组的插入与删除操作有以下缺点：

- **时间复杂度高**：数组的插入和删除的平均时间复杂度均为 $O(n)$ ，其中 n 为数组长度。
- **丢失元素**：由于数组的长度不可变，因此在插入元素后，超出数组长度范围的元素会丢失。

- **内存浪费**：我们可以初始化一个比较长的数组，只用前面一部分，这样在插入数据时，丢失的末尾元素都是我们不关心的，但这样做同时也会造成内存空间的浪费。

4.1.3. 数组常用操作

数组遍历。以下介绍两种常用的遍历方法。

```
// === File: array.cpp ===
/* 遍历数组 */
void traverse(int *nums, int size) {
    int count = 0;
    // 通过索引遍历数组
    for (int i = 0; i < size; i++) {
        count++;
    }
}
```

数组查找。通过遍历数组，查找数组内的指定元素，并输出对应索引。

```
// === File: array.cpp ===
/* 在数组中查找指定元素 */
int find(int *nums, int size, int target) {
    for (int i = 0; i < size; i++) {
        if (nums[i] == target)
            return i;
    }
    return -1;
}
```

4.1.4. 数组典型应用

随机访问。如果我们想要随机抽取一些样本，那么可以用数组存储，并生成一个随机序列，根据索引实现样本的随机抽取。

二分查找。例如前文查字典的例子，我们可以将字典中的所有字按照拼音顺序存储在数组中，然后使用与日常查纸质字典相同的“翻开中间，排除一半”的方式，来实现一个查电子字典的算法。

深度学习。神经网络中大量使用了向量、矩阵、张量之间的线性代数运算，这些数据都是以数组的形式构建的。数组是神经网络编程中最常使用的数据结构。

4.2. 链表

内存空间是所有程序的公共资源，排除已被占用的内存空间，空闲内存空间通常散落在内存各处。在上一节中，我们提到存储数组的内存空间必须是连续的，而当我们需要申请一个非常大的数组时，空闲内存中可能没

有这么大的连续空间。与数组相比，链表更具灵活性，它可以被存储在非连续的内存空间中。

「链表 Linked List」是一种线性数据结构，其每个元素都是一个节点对象，各个节点之间通过指针连接，从当前节点通过指针可以访问到下一个节点。**由于指针记录了下个节点的内存地址，因此无需保证内存地址的连续性**，从而可以将各个节点分散存储在内存各处。

链表「节点 Node」包含两项数据，一是节点「值 Value」，二是指向下一节点的「指针 Pointer」，或称「引用 Reference」。

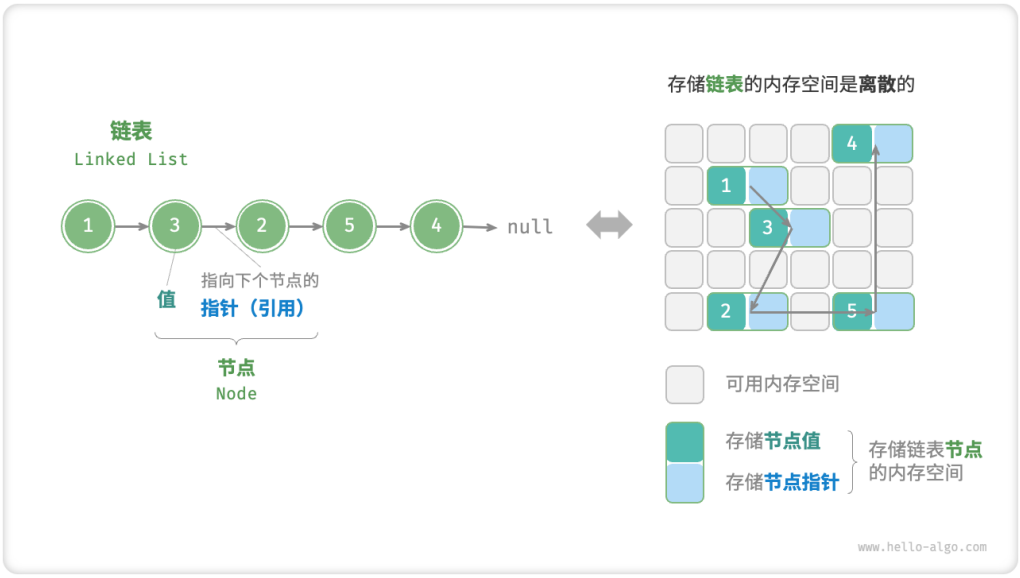

Figure 4-5. 链表定义与存储方式

```
/* 链表节点结构体 */
struct ListNode {
    int val;         // 节点值
    ListNode *next;  // 指向下一节点的指针（引用）
    ListNode(int x) : val(x), next(nullptr) {}  // 构造函数
};
```

尾节点指向什么？

我们将链表的最后一个节点称为「尾节点」，其指向的是“空”，在 Java, C++, Python 中分别记为 null , nullptr , None 。在不引起歧义的前提下，本书都使用 null 来表示空。

如何称呼链表？

在编程语言中，数组整体就是一个变量，例如数组 `nums` ，包含各个元素 `nums[0]` , `nums[1]` 等等。而链表是由多个节点对象组成，我们通常将头节点当作链表的代称，例如头节点 `head` 和链表 `head` 实际上是同义的。

链表初始化方法。建立链表分为两步，第一步是初始化各个节点对象，第二步是构建引用指向关系。完成后，即可以从链表的头节点（即首个节点）出发，通过指针 `next` 依次访问所有节点。

```
// === File: linked_list.cpp ===
/* 初始化链表 1 -> 3 -> 2 -> 5 -> 4 */
// 初始化各个节点
ListNode* n0 = new ListNode(1);
ListNode* n1 = new ListNode(3);
ListNode* n2 = new ListNode(2);
ListNode* n3 = new ListNode(5);
ListNode* n4 = new ListNode(4);
// 构建引用指向
n0->next = n1;
n1->next = n2;
n2->next = n3;
n3->next = n4;
```

4.2.1. 链表优点

链表中插入与删除节点的操作效率高。例如，如果我们想在链表中间的两个节点 `A` , `B` 之间插入一个新节点 `P` ，我们只需要改变两个节点指针即可，时间复杂度为 $O(1)$ ；相比之下，数组的插入操作效率要低得多。

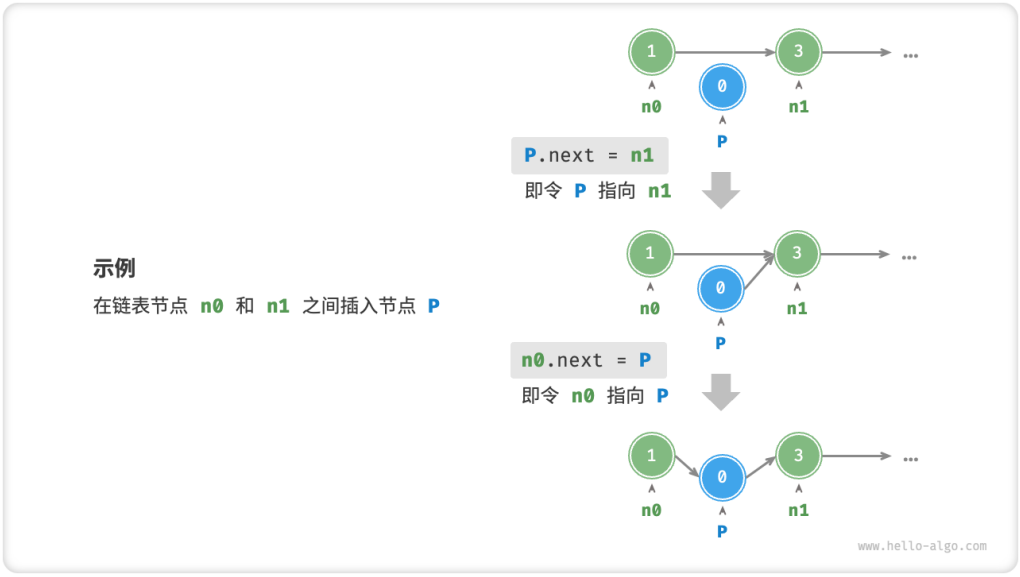

Figure 4-6. 链表插入节点

```
// === File: linked_list.cpp ===
/* 在链表的节点 n0 之后插入节点 P */
void insert(ListNode *n0, ListNode *P) {
    ListNode *n1 = n0->next;
```

```
    P->next = n1;
    n0->next = P;
}
```

在链表中删除节点也非常方便，只需改变一个节点的指针即可。如下图所示，尽管在删除操作完成后，节点 `P` 仍然指向 `n1`，但实际上 `P` 已经不再属于此链表，因为遍历此链表时无法访问到 `P`。

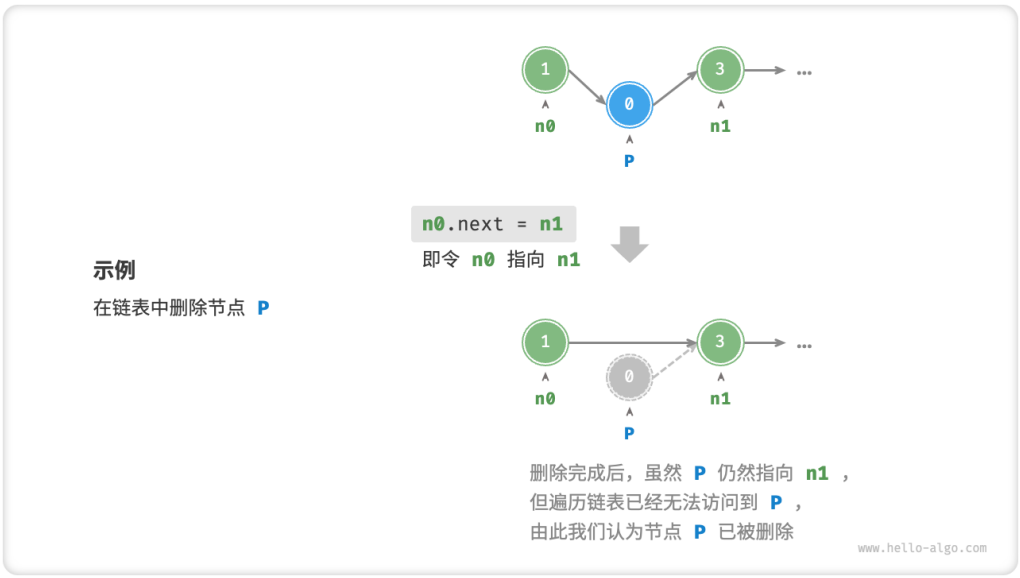

Figure 4-7. 链表删除节点

```
// === File: linked_list.cpp ===
/* 删除链表的节点 n0 之后的首个节点 */
void remove(ListNode *n0) {
    if (n0->next == nullptr)
        return;
    // n0 -> P -> n1
    ListNode *P = n0->next;
    ListNode *n1 = P->next;
    n0->next = n1;
    // 释放内存
    delete P;
}
```

4.2.2. 链表缺点

链表访问节点效率较低。如上节所述，数组可以在 $O(1)$ 时间下访问任意元素。然而，链表无法直接访问任意节点，这是因为系统需要从头节点出发，逐个向后遍历直至找到目标节点。例如，若要访问链表索引为 `index`（即第 `index + 1` 个）的节点，则需要向后遍历 `index` 轮。

```
// === File: linked_list.cpp ===
/* 访问链表中索引为 index 的节点 */
ListNode *access(ListNode *head, int index) {
    for (int i = 0; i < index; i++) {
        if (head == nullptr)
            return nullptr;
        head = head->next;
    }
    return head;
}
```

链表的内存占用较大。链表以节点为单位，每个节点除了保存值之外，还需额外保存指针（引用）。这意味着在相同数据量的情况下，链表比数组需要占用更多的内存空间。

4.2.3. 链表常用操作

遍历链表查找。遍历链表，查找链表内值为 `target` 的节点，输出节点在链表中的索引。

```
// === File: linked_list.cpp ===
/* 在链表中查找值为 target 的首个节点 */
int find(ListNode *head, int target) {
    int index = 0;
    while (head != nullptr) {
        if (head->val == target)
            return index;
        head = head->next;
        index++;
    }
    return -1;
}
```

4.2.4. 常见链表类型

单向链表。即上述介绍的普通链表。单向链表的节点包含值和指向下一节点的指针（引用）两项数据。我们将首个节点称为头节点，将最后一个节点成为尾节点，尾节点指向 null 。

环形链表。如果我们令单向链表的尾节点指向头节点（即首尾相接），则得到一个环形链表。在环形链表中，任意节点都可以视作头节点。

双向链表。与单向链表相比，双向链表记录了两个方向的指针（引用）。双向链表的节点定义同时包含指向后继节点（下一节点）和前驱节点（上一节点）的指针。相较于单向链表，双向链表更具灵活性，可以朝两个方向遍历链表，但相应地也需要占用更多的内存空间。

```
/* 双向链表节点结构体 */
struct ListNode {
    int val;         // 节点值
    ListNode *next;  // 指向后继节点的指针（引用）
    ListNode *prev;  // 指向前驱节点的指针（引用）
    ListNode(int x) : val(x), next(nullptr), prev(nullptr) {}  // 构造函数
};
```

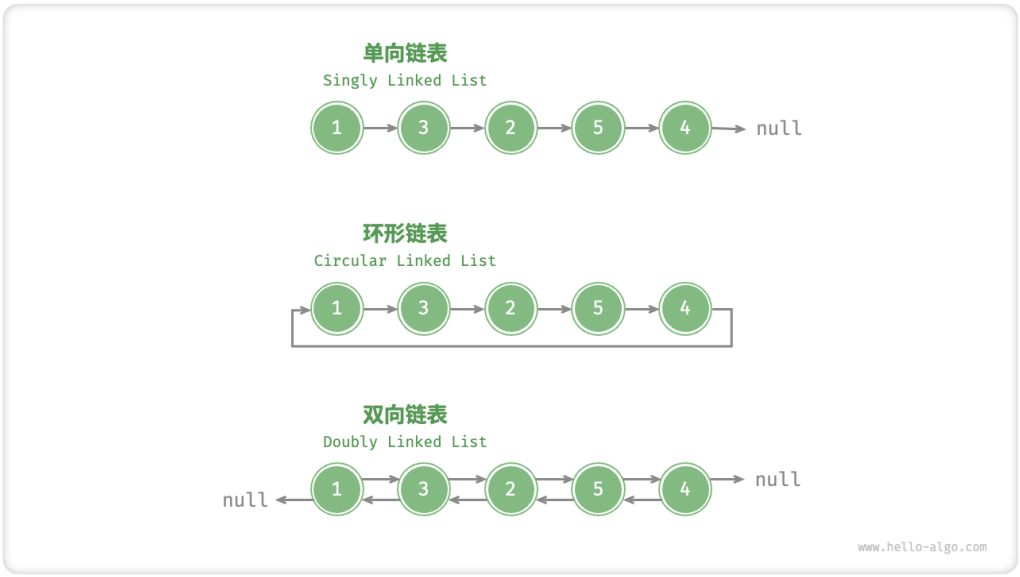

Figure 4-8. 常见链表种类

4.3. 列表

数组长度不可变导致实用性降低。在许多情况下，我们事先无法确定需要存储多少数据，这使数组长度的选择变得困难。若长度过小，需要在持续添加数据时频繁扩容数组；若长度过大，则会造成内存空间的浪费。

为解决此问题，出现了一种被称为「动态数组 Dynamic Array」的数据结构，即长度可变的数组，也常被称为「列表 List」。列表基于数组实现，继承了数组的优点，并且可以在程序运行过程中动态扩容。在列表中，我们可以自由添加元素，而无需担心超过容量限制。

4.3.1. 列表常用操作

初始化列表。通常我们会使用“无初始值”和“有初始值”的两种初始化方法。

```
// === File: list.cpp ===
/* 初始化列表 */
// 需注意，C++ 中 vector 即是本文描述的 list
// 无初始值
```

```
vector<int> list1;
// 有初始值
vector<int> list = { 1, 3, 2, 5, 4 };
```

访问与更新元素。由于列表的底层数据结构是数组，因此可以在 $O(1)$ 时间内访问和更新元素，效率很高。

```
// === File: list.cpp ===
/* 访问元素 */
int num = list[1];  // 访问索引 1 处的元素

/* 更新元素 */
list[1] = 0;  // 将索引 1 处的元素更新为 0
```

在列表中添加、插入、删除元素。相较于数组，列表可以自由地添加与删除元素。在列表尾部添加元素的时间复杂度为 $O(1)$ ，但插入和删除元素的效率仍与数组相同，时间复杂度为 $O(N)$ 。

```
// === File: list.cpp ===
/* 清空列表 */
list.clear();

/* 尾部添加元素 */
list.push_back(1);
list.push_back(3);
list.push_back(2);
list.push_back(5);
list.push_back(4);

/* 中间插入元素 */
list.insert(list.begin() + 3, 6);  // 在索引 3 处插入数字 6

/* 删除元素 */
list.erase(list.begin() + 3);      // 删除索引 3 处的元素
```

遍历列表。与数组一样，列表可以根据索引遍历，也可以直接遍历各元素。

```
// === File: list.cpp ===
/* 通过索引遍历列表 */
int count = 0;
for (int i = 0; i < list.size(); i++) {
    count++;
}

/* 直接遍历列表元素 */
count = 0;
```

```
for (int n : list) {
    count++;
}
```

拼接两个列表。给定一个新列表 `list1`，我们可以将该列表拼接到原列表的尾部。

```
// === File: list.cpp ===
/* 拼接两个列表 */
vector<int> list1 = { 6, 8, 7, 10, 9 };
// 将列表 list1 拼接到 list 之后
list.insert(list.end(), list1.begin(), list1.end());
```

排序列表。排序也是常用的方法之一。完成列表排序后，我们便可以使用在数组类算法题中经常考察的「二分查找」和「双指针」算法。

```
// === File: list.cpp ===
/* 排序列表 */
sort(list.begin(), list.end());  // 排序后，列表元素从小到大排列
```

4.3.2. 列表实现 *

为了帮助加深对列表的理解，我们在此提供一个简易版列表实现。需要关注三个核心点：

- **初始容量**：选取一个合理的数组初始容量。在本示例中，我们选择 10 作为初始容量。
- **数量记录**：声明一个变量 size，用于记录列表当前元素数量，并随着元素插入和删除实时更新。根据此变量，我们可以定位列表尾部，以及判断是否需要扩容。
- **扩容机制**：插入元素时可能超出列表容量，此时需要扩容列表。扩容方法是根据扩容倍数创建一个更大的数组，并将当前数组的所有元素依次移动至新数组。在本示例中，我们规定每次将数组扩容至之前的 2 倍。

本示例旨在帮助读者直观理解列表的工作机制。实际编程语言中，列表实现更加标准和复杂，各个参数的设定也非常有考究，例如初始容量、扩容倍数等。感兴趣的读者可以查阅源码进行学习。

```
// === File: my_list.cpp ===
/* 列表类简易实现 */
class MyList {
  private:
    int *nums;             // 数组（存储列表元素）
    int numsCapacity = 10; // 列表容量
    int numsSize = 0;      // 列表长度（即当前元素数量）
    int extendRatio = 2;   // 每次列表扩容的倍数

  public:
```

```
/* 构造方法 */
MyList() {
    nums = new int[numsCapacity];
}

/* 析构方法 */
~MyList() {
    delete[] nums;
}

/* 获取列表长度（即当前元素数量）*/
int size() {
    return numsSize;
}

/* 获取列表容量 */
int capacity() {
    return numsCapacity;
}

/* 访问元素 */
int get(int index) {
    // 索引如果越界则抛出异常，下同
    if (index < 0 || index >= size())
        throw out_of_range(" 索引越界");
    return nums[index];
}

/* 更新元素 */
void set(int index, int num) {
    if (index < 0 || index >= size())
        throw out_of_range(" 索引越界");
    nums[index] = num;
}

/* 尾部添加元素 */
void add(int num) {
    // 元素数量超出容量时，触发扩容机制
    if (size() == capacity())
        extendCapacity();
    nums[size()] = num;
    // 更新元素数量
    numsSize++;
}

/* 中间插入元素 */
```

```
void insert(int index, int num) {
    if (index < 0 || index >= size())
        throw out_of_range(" 索引越界");
    // 元素数量超出容量时，触发扩容机制
    if (size() == capacity())
        extendCapacity();
    // 索引 i 以及之后的元素都向后移动一位
    for (int j = size() - 1; j >= index; j--) {
        nums[j + 1] = nums[j];
    }
    nums[index] = num;
    // 更新元素数量
    numsSize++;
}

/* 删除元素 */
int remove(int index) {
    if (index < 0 || index >= size())
        throw out_of_range(" 索引越界");
    int num = nums[index];
    // 索引 i 之后的元素都向前移动一位
    for (int j = index; j < size() - 1; j++) {
        nums[j] = nums[j + 1];
    }
    // 更新元素数量
    numsSize--;
    // 返回被删除元素
    return num;
}

/* 列表扩容 */
void extendCapacity() {
    // 新建一个长度为 size * extendRatio 的数组，并将原数组拷贝到新数组
    int newCapacity = capacity() * extendRatio;
    int *tmp = nums;
    nums = new int[newCapacity];
    // 将原数组中的所有元素复制到新数组
    for (int i = 0; i < size(); i++) {
        nums[i] = tmp[i];
    }
    // 释放内存
    delete[] tmp;
    numsCapacity = newCapacity;
}

/* 将列表转换为 Vector 用于打印 */
```

```
    vector<int> toVector() {
        // 仅转换有效长度范围内的列表元素
        vector<int> vec(size());
        for (int i = 0; i < size(); i++) {
            vec[i] = nums[i];
        }
        return vec;
    }
};
```

4.4. 小结

- 数组和链表是两种基本数据结构，分别代表数据在计算机内存中的连续空间存储和离散空间存储方式。两者的优缺点呈现出互补的特性。
- 数组支持随机访问、占用内存较少；但插入和删除元素效率低，且初始化后长度不可变。
- 链表通过更改指针实现高效的节点插入与删除，且可以灵活调整长度；但节点访问效率低、占用内存较多。常见的链表类型包括单向链表、循环链表、双向链表。
- 动态数组，又称列表，是基于数组实现的一种数据结构。它保留了数组的优势，同时可以灵活调整长度。列表的出现极大地提高了数组的易用性，但可能导致部分内存空间浪费。
- 下表总结并对比了数组与链表的各项特性。

	数组	链表
存储方式	连续内存空间	离散内存空间
数据结构长度	长度不可变	长度可变
内存使用率	占用内存少、缓存局部性好	占用内存多
优势操作	随机访问	插入、删除

缓存局部性

在计算机中，数据读写速度排序是“硬盘 < 内存 < CPU 缓存”。当我们访问数组元素时，计算机不仅会加载它，还会缓存其周围的其他数据，从而借助高速缓存来提升后续操作的执行速度。链表则不然，计算机只能挨个地缓存各个节点，这样的多次“搬运”降低了整体效率。

- 下表对比了数组与链表在各种操作上的效率。