1. 调用栈

递归函数每次调用自身时，系统都会为新开启的函数分配内存，以存储局部变量、调用地址和其他信息等。这将导致两方面的结果。

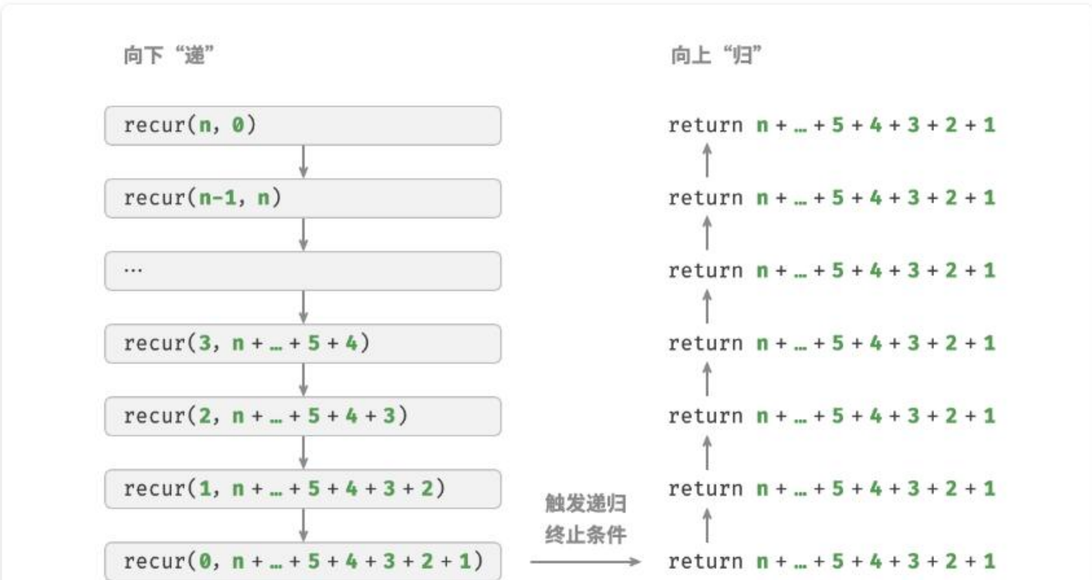
- 函数的上下文数据都存储在称为“栈帧空间”的内存区域中，直至函数返回后才会被释放。因此，**递归通常比迭代更加耗费内存空间**。
- 递归调用函数会产生额外的开销。**因此递归通常比循环的时间效率更低**。

尾递归

有趣的是，**如果函数在返回前的最后一步才进行递归调用**，则该函数可以被编译器或解释器优化，使其在空间效率上与迭代相当。这种情况被称为尾递归（tail recursion）。

- **普通递归**：当函数返回到上一层级的函数后，需要继续执行代码，因此系统需要保存上一层调用的上下文。
- **尾递归**：递归调用是函数返回前的最后一个操作，这意味着函数返回到上一层级后，无须继续执行其他操作，因此系统无须保存上一层函数的上下文。

- **普通递归**：求和操作是在“归”的过程中执行的，每层返回后都要再执行一次求和操作。
- **尾递归**：求和操作是在“递”的过程中执行的，“归”的过程只需层层返回。

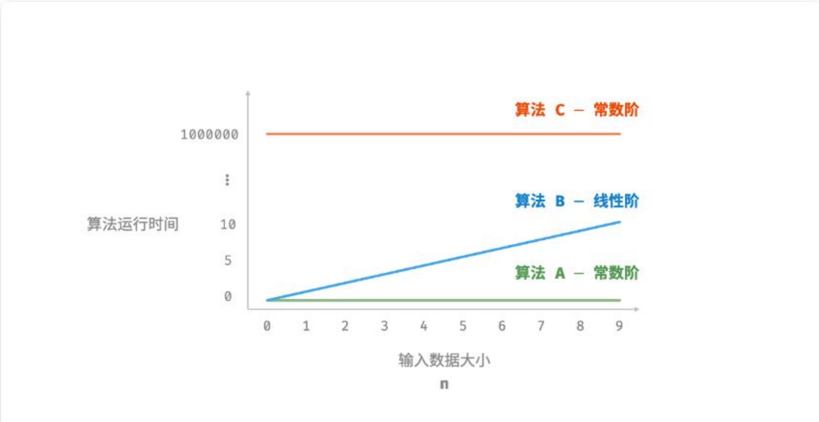


	迭代	递归
实现方式	循环结构	函数调用自身
时间效率	效率通常较高，无函数调用开销	每次函数调用都会产生开销
内存使用	通常使用固定大小的内存空间	累积函数调用可能使用大量的栈帧空间
适用问题	适用于简单循环任务，代码直观、可读性好	适用于子问题分解，如树、图、分治、回溯等，代码结构简洁、清晰

2.3.1 统计时间增长趋势

时间复杂度分析统计的不是算法运行时间，**而是算法运行时间随着数据量变大时的增长趋势**。

- 算法 A 只有 1 个打印操作，算法运行时间不随着 n 增大而增长。我们称此算法的时间复杂度为“常数阶”。
- 算法 B 中的打印操作需要循环 n 次，算法运行时间随着 n 增大呈线性增长。此算法的时间复杂度被称为“线性阶”。
- 算法 C 中的打印操作需要循环 1000000 次，虽然运行时间很长，但它与输入数据大小 n 无关。因此 C 的时间复杂度和 A 相同，仍为“常数阶”。



特点

- **时间复杂度能够有效评估算法效率。** 例如，算法 B 的运行时间呈线性增长，在 $n > 1$ 时比算法 A 更慢，在 $n > 1000000$ 时比算法 C 更慢。事实上，只要输入数据大小 n 足够大，复杂度为“常数阶”的算法一定优于“线性阶”的算法，这正是时间增长趋势的含义。
- **时间复杂度的推算方法更简便。** 显然，运行平台和计算操作类型都与算法运行时间的增长趋势无关。因此在时间复杂度分析中，我们可以简单地将所有计算操作的执行时间视为相同的“单位时间”，从而将“计算操作运行时间统计”简化为“计算操作数量统计”，这样一来估算难度就大大降低了。
- **时间复杂度也存在一定的局限性。** 例如，尽管算法 A 和 C 的时间复杂度相同，但实际运行时间差别很大。同样，尽管算法 B 的时间复杂度比 C 高，但在输入数据大小 n 较小时，算法 B 明显优于算法 C 。对于此类情况，我们时常难以仅凭时间复杂度判断算法效率的高低。当然，尽管存在上述问题，复杂度分析仍然是评判算法效率最有效且常用的方法。

时间复杂度分析本质上是计算“操作数量 $T(n)$”的渐近上界，它具有明确的数学定义。

函数渐近上界

若存在正实数 c 和实数 n_0，使得对于所有的 $n > n_0$，均有 $T(n) \leq c \cdot f(n)$，则可认为 $f(n)$ 给出了 $T(n)$ 的一个渐近上界，记为 $T(n) = O(f(n))$。

如图 2-8 所示，计算渐近上界就是寻找一个函数 $f(n)$，使得当 n 趋向于无穷大时，$T(n)$ 和 $f(n)$ 处于相同的增长级别，仅相差一个常数项 c 的倍数。

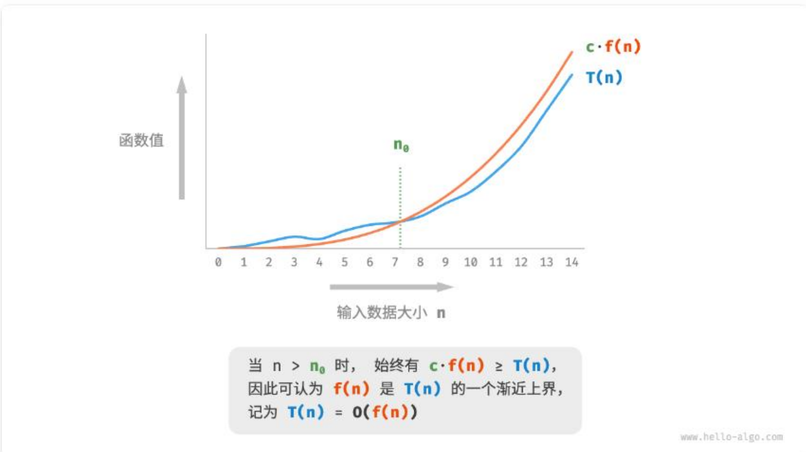


图 2-8 函数的渐近上界

根据定义，确定 $f(n)$ 之后，我们便可得到时间复杂度 $O(f(n))$ 。那么如何确定渐近上界 $f(n)$ 呢？总体分为两步：首先统计操作数量，然后判断渐近上界。

1. 第一步：统计操作数量

针对代码，逐行从上到下计算即可。然而，由于上述 $c \cdot f(n)$ 中的常数项 c 可以取任意大小，**因此操作数量 $T(n)$ 中的各种系数、常数项都可以忽略**。根据此原则，可以总结出以下计数简化技巧。

1. **忽略 $T(n)$ 中的常数项**。因为它们都与 n 无关，所以对时间复杂度不产生影响。
2. **省略所有系数**。例如，循环 $2n$ 次、$5n+1$ 次等，都可以简化记为 n 次，因为 n 前面的系数对时间复杂度没有影响。
3. **循环嵌套时使用乘法**。总操作数量等于外层循环和内层循环操作数量之积，每一层循环依然可以分别套用第 `1.` 点和第 `2.` 点的技巧。

时间复杂度由 $T(n)$ 中最高阶的项来决定。这是因为在 n 趋于无穷大时，最高阶的项将发挥主导作用，其他项的影响都可以忽略。

表 2-2 展示了一些例子，其中一些夸张的值是为了强调“系数无法撼动阶数”这一结论。当 n 趋于无穷大时，这些常数变得无足轻重。

表 2-2　不同操作数量对应的时间复杂度

操作数量 $T(n)$	**时间复杂度** $O(f(n))$
100000	$O(1)$
$3n+2$	$O(n)$
$2n^2+3n+2$	$O(n^2)$
$n^3+10000n^2$	$O(n^3)$
$2^n+10000n^{10000}$	$O(2^n)$

设输入数据大小为 n ，常见的时间复杂度类型如图 2-9 所示（按照从低到高的顺序排列）。

$$
\begin{aligned}
& O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!) \\
& \text{常数阶} < \text{对数阶} < \text{线性阶} < \text{线性对数阶} < \text{平方阶} < \text{指数阶} < \text{阶乘阶}
\end{aligned}
$$

4. 指数阶 $O(2^n)$

生物学的“细胞分裂”是指数阶增长的典型例子：初始状态为 1 个细胞，分裂一轮后变为 2 个，分裂两轮后变为 4 个，以此类推，分裂 n 轮后有 2^n 个细胞。

在实际算法中，指数阶常出现于递归函数中。例如在以下代码中，其递归地一分为二，经过 n 次分裂后停止：

5. 对数阶 $O(\log n)$ ¶

与指数阶相反，对数阶反映了“每轮缩减到一半”的情况。设输入数据大小为 n ，由于每轮缩减到一半，因此循环次数是 $\log_2 n$ ，即 2^n 的反函数。

6. 线性对数阶 $O(n \log n)$ ¶

线性对数阶常出现于嵌套循环中，两层循环的时间复杂度分别为 $O(\log n)$ 和 $O(n)$ 。相关代码如下：

算法的时间效率往往不是固定的，而是与输入数据的分布有关。假设输入一个长度为 n 的数组 `nums` ，其中 `nums` 由从 1 至 n 的数字组成，每个数字只出现一次；但元素顺序是随机打乱的，任务目标是返回元素 1 的索引。我们可以得出以下结论。

- 当 `nums = [?, ?, ..., 1]` ，即当末尾元素是 1 时，需要完整遍历数组，**达到最差时间复杂度 $O(n)$** 。
- 当 `nums = [1, ?, ?, ...]` ，即当首个元素为 1 时，无论数组多长都不需要继续遍历，**达到最佳时间复杂度 $\Omega(1)$** 。

“最差时间复杂度”对应函数渐近上界，使用大 O 记号表示。相应地，“最佳时间复杂度”对应函数渐近下界，用 Ω 记号表示：

值得说明的是，我们在实际中很少使用最佳时间复杂度，因为通常只有在很小概率下才能达到，可能会带来一定的误导性。**而最差时间复杂度更为实用，因为它给出了一个效率安全值**，让我们可以放心地使用算法。

从上述示例可以看出，最差时间复杂度和最佳时间复杂度只出现于“特殊的数据分布”，这些情况的出现概率可能很小，并不能真实地反映算法运行效率。相比之下，**平均时间复杂度可以体现算法在随机输入数据下的运行效率**，用 Θ 记号来表示。

但对于较为复杂的算法，计算平均时间复杂度往往比较困难，因为很难分析出在数据分布下的整体数学期望。在这种情况下，我们通常使用最差时间复杂度作为算法效率的评判标准。

算法在运行过程中使用的内存空间主要包括以下几种。

- **输入空间**：用于存储算法的输入数据。
- **暂存空间**：用于存储算法在运行过程中的变量、对象、函数上下文等数据。
- **输出空间**：用于存储算法的输出数据。

一般情况下，空间复杂度的统计范围是“暂存空间”加上“输出空间”。

暂存空间可以进一步划分为三个部分。

- **暂存数据**：用于保存算法运行过程中的各种常量、变量、对象等。
- **栈帧空间**：用于保存调用函数的上下文数据。系统在每次调用函数时都会在栈顶部创建一个栈帧，函数返回后，栈帧空间会被释放。
- **指令空间**：用于保存编译后的程序指令，在实际统计中通常忽略不计。

```
/* 结构体 */
struct Node {
    int val;
    Node *next;
    Node(int x) : val(x), next(nullptr) {}
};

/* 函数 */
int func() {
    // 执行某些操作...
    return 0;
}

int algorithm(int n) {          // 输入数据
    const int a = 0;            // 暂存数据（常量）
    int b = 0;                  // 暂存数据（变量）
    Node* node = new Node(0);   // 暂存数据（对象）
    int c = func();             // 栈帧空间（调用函数）
    return a + b + c;           // 输出数据
}
```

我们通常只关注最差空间复杂度。这是因为内存空间是一项硬性要求，我们必须确保在所有输入数据下都有足够的内存空间预留。

1. **以最差输入数据为准**：当 $n < 10$ 时，空间复杂度为 $O(1)$；但当 $n > 10$ 时，初始化的数组 `nums` 占用 $O(n)$ 空间，因此最差空间复杂度为 $O(n)$。
2. **以算法运行中的峰值内存为准**：例如，程序在执行最后一行之前，占用 $O(1)$ 空间；当初始化数组 `nums` 时，程序占用 $O(n)$ 空间，因此最差空间复杂度为 $O(n)$。

```
int func() {
    // 执行某些操作
    return 0;
}
/* 循环的空间复杂度为 O(1) */
void loop(int n) {
    for (int i = 0; i < n; i++) {
        func();
    }
}
/* 递归的空间复杂度为 O(n) */
void recur(int n) {
    if (n == 1) return;
    return recur(n - 1);
}
```

函数 `loop()` 和 `recur()` 的时间复杂度都为 $O(n)$，但空间复杂度不同。

- 函数 `loop()` 在循环中调用了 n 次 `function()`，每轮中的 `function()` 都返回并释放了栈帧空间，因此空间复杂度仍为 $O(1)$。
- 递归函数 `recur()` 在运行过程中会同时存在 n 个未返回的 `recur()`，从而占用 $O(n)$ 的栈帧空间。

常数阶

需要注意的是，在循环中初始化变量或调用函数而占用的内存，在进入下一循环后就会被释放，因此不会累积占用空间，空间复杂度仍为 O(1)：

2. 线性阶 $O(n)$ ¶

线性阶常见于元素数量与 n 成正比的数组、链表、栈、队列等：

3. 平方阶 $O(n^2)$

平方阶常见于矩阵和图，元素数量与 n 成平方关系：

4. 指数阶 $O(2^n)$

指数阶常见于二叉树。观察图 2-19，层数为 n 的“满二叉树”的节点数量为 $2^n - 1$，占用 $O(2^n)$ 空间：

5. 对数阶 $O(\log n)$

对数阶常见于分治算法。例如归并排序，输入长度为 n 的数组，每轮递归将数组从中点处划分为两半，形成高度为 $\log n$ 的递归树，使用 $O(\log n)$ 栈帧空间。

再例如将数字转化为字符串，输入一个正整数 n，它的位数为 $\lfloor \log_{10} n \rfloor + 1$，即对应字符串长度为 $\lfloor \log_{10} n \rfloor + 1$，因此空间复杂度为 $O(\log_{10} n + 1) = O(\log n)$。

数据结构

值得说明的是，**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。

首先需要指出，**数字是以“补码”的形式存储在计算机中的**。在分析这样做的原因之前，首先给出三者的定义。

- **原码**：我们将数字的二进制表示的最高位视为符号位，其中 0 表示正数，1 表示负数，其余位表示数字的值。
- **反码**：正数的反码与其原码相同，负数的反码是对其原码除符号位外的所有位取反。
- **补码**：正数的补码与其原码相同，负数的补码是在其反码的基础上加 1。

数组

内存上是连续的

总的来看，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“无意义”的，但这样做会造成部分内存空间浪费。

> 7. 扩容数组
>
> 在复杂的系统环境中，程序难以保证数组之后的内存空间是可用的，从而无法安全地扩展数组容量。因此在大多数编程语言中，**数组的长度是不可变的**。
>
> 如果我们希望扩容数组，则需重新建立一个更大的数组，然后把原数组元素依次复制到新数组。这是一个 $O(n)$ 的操作，在数组很大的情况下非常耗时。代码如下所示：

```
/* 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;
}
```

数组存储在连续的内存空间内，且元素类型相同。这种做法包含丰富的先验信息，系统可以利用这些信息来优化数据结构的操作效率。

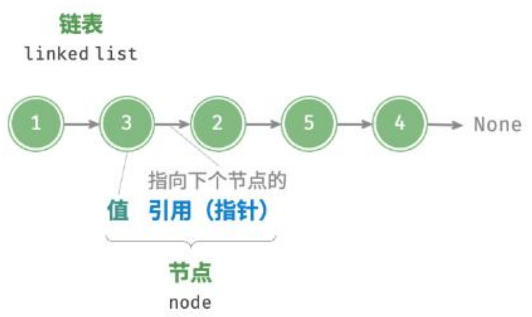
- **空间效率高**：数组为数据分配了连续的内存块，无须额外的结构开销。
- **支持随机访问**：数组允许在 $O(1)$ 时间内访问任何元素。
- **缓存局部性**：当访问数组元素时，计算机不仅会加载它，还会缓存其周围的其他数据，从而借助高速缓存来提升后续操作的执行速度。

连续空间存储是一把双刃剑，其存在以下局限性。

- **插入与删除效率低**：当数组中元素较多时，插入与删除操作需要移动大量的元素。
- **长度不可变**：数组在初始化后长度就固定了，扩容数组需要将所有数据复制到新数组，开销很大。
- **空间浪费**：如果数组分配的大小超过实际所需，那么多余的空间就被浪费了。

链表

链表（linked list）是一种线性数据结构，其中的每个元素都是一个节点对象，各个节点通过“引用”相连接。引用记录了下一个节点的内存地址，通过它可以从当前节点访问到下一个节点。
链表的设计使得各个节点可以分散存储在内存各处，它们的内存地址无须连续。



链表除了包含值以外还包含着指向下一个数据的指针，因此所占用的内存空间比

数组大。

链表初始化：

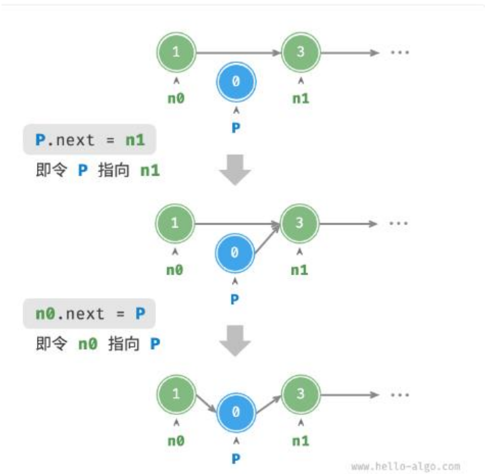
linked_list.cpp

```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);
// 构建节点之间的引用
n0->next = n1;
n1->next = n2;
n2->next = n3;
n3->next = n4;
```

链表插入

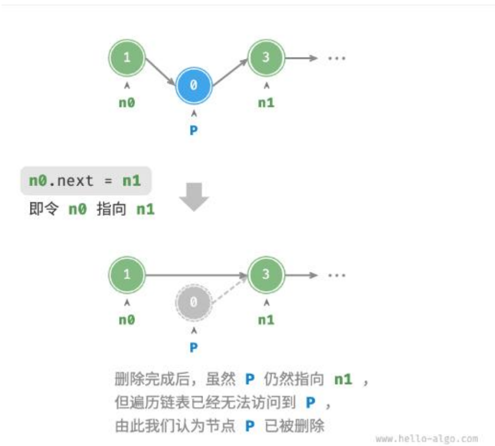
在链表中插入节点非常容易。如图 4-6 所示，假设我们想在相邻的两个节点 `n0` 和 `n1` 之间插入一个新节点 `P`，**则只需改变两个节点引用（指针）即可**，时间复杂度为 $O(1)$。

相比之下，在数组中插入元素的时间复杂度为 $O(n)$，在大数据量下的效率较低。



先取出来 N1 的地址，然后赋予 P，再把 P 的地址当作 N0 的节点。

链表删除



先取 P 的节点，赋予 N0 充当 N0 的节点

列表元素删减，其删减的元素仍然存储在内存里面。

5. 查找节点

遍历链表，查找其中值为 `target` 的节点，输出该节点在链表中的索引。此过程也属于线性查找。代码如下所示：

Python C++ Java C# Go Swift JS TS Dart Rust C Kotlin Ruby Zig

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;
}
```

二者对比

表 4-1 数组与链表的效率对比

	数组	链表
存储方式	连续内存空间	分散内存空间
容量扩展	长度不可变	可灵活扩展
内存效率	元素占用内存少、但可能浪费空间	元素占用内存多
访问元素	$O(1)$	$O(n)$
添加元素	$O(n)$	$O(1)$
删除元素	$O(n)$	$O(1)$

链表类型

- **单向链表**：即前面介绍的普通链表。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，将最后一个节点称为尾节点，尾节点指向空 `None` 。
- **环形链表**：如果我们令单向链表的尾节点指向头节点（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引用。双向链表的节点定义同时包含指向后继节点（下一个节点）和前驱节点（上一个节点）的引用（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
    ListNode *prev;  // 指向前驱节点的指针
    ListNode(int x) : val(x), next(nullptr), prev(nullptr) {}  // 构造函数
};
```

链表的应用场景

单向链表通常用于实现栈、队列、哈希表和图等数据结构。
双向链表常用于需要快速查找前一个和后一个元素的场景。
环形链表常用于需要周期性操作的场景，比如操作系统的资源调度。

栈

方法	描述	时间复杂度
`push()`	元素入栈（添加至栈顶）	$O(1)$
`pop()`	栈顶元素出栈	$O(1)$
`peek()`	访问栈顶元素	$O(1)$

```cpp
/* 初始化栈 */
stack<int> stack;

/* 元素入栈 */
stack.push(1);
stack.push(3);
stack.push(2);
stack.push(5);
stack.push(4);

/* 访问栈顶元素 */
int top = stack.top();

/* 元素出栈 */
stack.pop(); // 无返回值

/* 获取栈的长度 */
int size = stack.size();

/* 判断是否为空 */
bool empty = stack.empty();
```

栈遵循先入后出的原则，因此我们只能在栈顶添加或删除元素。然而，数组和链表都可以在任意位置添加和删除元素，**因此栈可以视为一种受限制的数组或链表**。

时间效率

在基于数组的实现中，入栈和出栈操作都在预先分配好的连续内存中进行，具有很好的缓存本地性，因此效率较高。然而，如果入栈时超出数组容量，会触发扩容机制，导致该次入栈操作的时间复杂度变为 $O(n)$ 。

在基于链表的实现中，链表的扩容非常灵活，不存在上述数组扩容时效率降低的问题。但是，入栈操作需要初始化节点对象并修改指针，因此效率相对较低。不过，如果入栈元素本身就是节点对象，那么可以省去初始化步骤，从而提高效率。

综上所述，当入栈与出栈操作的元素是基本数据类型时，例如 `int` 或 `double` ，我们可以得出以下结论。

- 基于数组实现的栈在触发扩容时效率会降低，但由于扩容是低频操作，因此平均效率更高。
- 基于链表实现的栈可以提供更加稳定的效率表现。

空间效率

在初始化列表时，系统会为列表分配"初始容量"，该容量可能超出实际需求；并且，扩容机制通常是按照特定倍率（例如 2 倍）进行扩容的，扩容后的容量也可能超出实际需求。因此，**基于数组实现的栈可能造成一定的空间浪费**。

然而，由于链表节点需要额外存储指针，**因此链表节点占用的空间相对较大**。

综上，我们不能简单地确定哪种实现更加节省内存，需要针对具体情况进行分析。

队列

队列（queue）是一种遵循先入先出规则的线性数据结构。

方法名	描述	时间复杂度
`push()`	元素入队，即将元素添加至队尾	$O(1)$
`pop()`	队首元素出队	$O(1)$
`peek()`	访问队首元素	$O(1)$

queue.cpp

```
/* 初始化队列 */
queue<int> queue;

/* 元素入队 */
queue.push(1);
queue.push(3);
queue.push(2);
queue.push(5);
queue.push(4);

/* 访问队首元素 */
int front = queue.front();

/* 元素出队 */
queue.pop();

/* 获取队列的长度 */
int size = queue.size();

/* 判断队列是否为空 */
bool empty = queue.empty();
```

队列也可以通过链表和数组表示。

在数组中删除首元素的时间复杂度为 $O(n)$ ，这会导致出队操作效率较低。然而，我们可以采用以下巧妙方法来避免这个问题。

我们可以使用一个变量 `front` 指向队首元素的索引，并维护一个变量 `size` 用于记录队列长度。定义 `rear = front + size` ，这个公式计算出的 `rear` 指向队尾元素之后的下一个位置。

基于此设计，**数组中包含元素的有效区间为 `[front, rear - 1]`**，各种操作的实现方法如图 5-6 所示。

- 入队操作：将输入元素赋值给 `rear` 索引处，并将 `size` 增加 1 。
- 出队操作：只需将 `front` 增加 1 ，并将 `size` 减少 1 。

你可能会发现一个问题：在不断进行入队和出队的过程中，`front` 和 `rear` 都在向右移动，**当它们到达数组尾部时就无法继续移动了**。为了解决此问题，我们可以将数组视为首尾相接的“环形数组”。

对于环形数组，我们需要让 `front` 或 `rear` 在越过数组尾部时，直接回到数组头部继续遍历。这种周期性规律可以通过“取余操作”来实现，代码如下所示：

双向队列

对于双向队列而言，头部和尾部都可以执行入队和出队操作。双向队列需要实现另一个对称方向的操作。为此采用“双向链表”作为双向队列的底层数据结构或者基于数组实现队列类似，我们也可以使用环形数组来实现双向队列。

- 栈是一种遵循先入后出原则的数据结构，可通过数组或链表来实现。
- 在时间效率方面，栈的数组实现具有较高的平均效率，但在扩容过程中，单次入栈操作的时间复杂度会劣化至 $O(n)$ 。相比之下，栈的链表实现具有更为稳定的效率表现。
- 在空间效率方面，栈的数组实现可能导致一定程度的空间浪费。但需要注意的是，链表节点所占用的内存空间比数组元素更大。
- 队列是一种遵循先入先出原则的数据结构，同样可以通过数组或链表来实现。在时间效率和空间效率的对比上，队列的结论与前述栈的结论相似。
- 双向队列是一种具有更高自由度的队列，它允许在两端进行元素的添加和删除操作。