第 5 章　栈与队列

Abstract

栈如同叠猫猫，而队列就像猫猫排队。

两者分别代表先入后出和先入先出的逻辑关系。

5.1　栈

栈（stack）是一种遵循先入后出逻辑的线性数据结构。

我们可以将栈类比为桌面上的一摞盘子，如果想取出底部的盘子，则需要先将上面的盘子依次移走。我们将盘子替换为各种类型的元素（如整数、字符、对象等），就得到了栈这种数据结构。

如图 5-1 所示，我们把堆叠元素的顶部称为“栈顶”，底部称为“栈底”。将把元素添加到栈顶的操作叫作“入栈”，删除栈顶元素的操作叫作“出栈”。

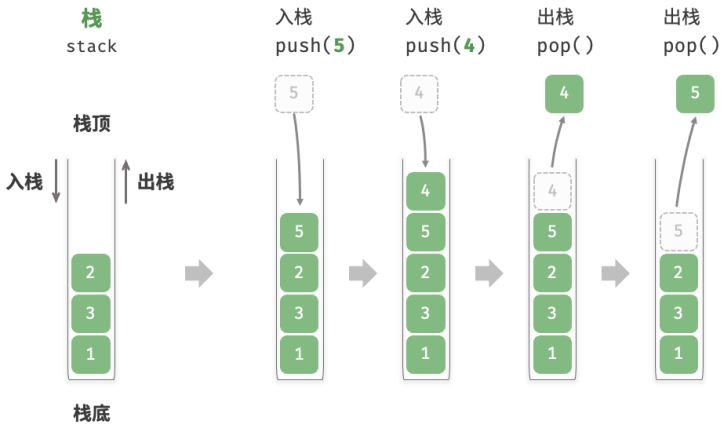


图 5-1　栈的先入后出规则

5.1.1　栈的常用操作

栈的常用操作如表 5-1 所示，具体的方法名需要根据所使用的编程语言来确定。在此，我们以常见的 `push()`、`pop()`、`peek()` 命名为例。

表 5-1　栈的操作效率

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	$O(1)$

通常情况下，我们可以直接使用编程语言内置的栈类。然而，某些语言可能没有专门提供栈类，这时我们可以将该语言的“数组”或“链表”当作栈来使用，并在程序逻辑上忽略与栈无关的操作。

```
// === File: stack.c ===

// C 未提供内置栈
```

5.1.2　栈的实现

为了深入了解栈的运行机制，我们来尝试自己实现一个栈类。

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**。换句话说，我们可以“屏蔽”数组或链表的部分无关操作，使其对外表现的逻辑符合栈的特性。

1.　基于链表的实现

使用链表实现栈时，我们可以将链表的头节点视为栈顶，尾节点视为栈底。

如图 5-2 所示，对于入栈操作，我们只需将元素插入链表头部，这种节点插入方法被称为“头插法”。而对于出栈操作，只需将头节点从链表中删除即可。

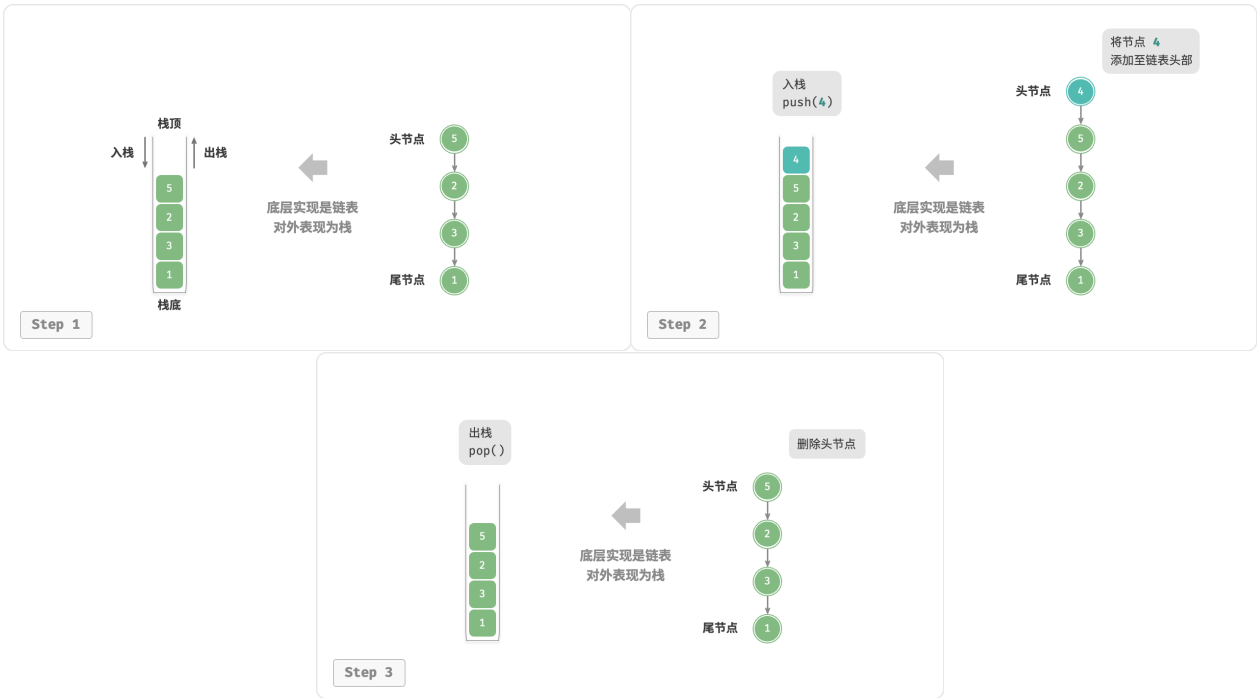


图 5-2　基于链表实现栈的入栈出栈操作

以下是基于链表实现栈的示例代码：

```c
// === File: linkedlist_stack.c ===

/* 基于链表实现的栈 */
typedef struct {
    ListNode *top; // 将头节点作为栈顶
    int size;      // 栈的长度
} LinkedListStack;

/* 构造函数 */
LinkedListStack *newLinkedListStack() {
    LinkedListStack *s = malloc(sizeof(LinkedListStack));
    s->top = NULL;
    s->size = 0;
    return s;
}

/* 析构函数 */
void delLinkedListStack(LinkedListStack *s) {
    while (s->top) {
        ListNode *n = s->top->next;
        free(s->top);
        s->top = n;
    }
    free(s);
}

/* 获取栈的长度 */
int size(LinkedListStack *s) {
    return s->size;
}

/* 判断栈是否为空 */
bool isEmpty(LinkedListStack *s) {
    return size(s) == 0;
}

/* 入栈 */
void push(LinkedListStack *s, int num) {
    ListNode *node = (ListNode *)malloc(sizeof(ListNode));
    node->next = s->top; // 更新新加节点指针域
    node->val = num;     // 更新新加节点数据域
    s->top = node;       // 更新栈顶
    s->size++;           // 更新栈大小
}

/* 访问栈顶元素 */
int peek(LinkedListStack *s) {
```

```
    if (s->size == 0) {
        printf(" 栈为空\n");
        return INT_MAX;
    }
    return s->top->val;
}

/* 出栈 */
int pop(LinkedListStack *s) {
    int val = peek(s);
    ListNode *tmp = s->top;
    s->top = s->top->next;
    // 释放内存
    free(tmp);
    s->size--;
    return val;
}
```

2. 基于数组的实现

使用数组实现栈时，我们可以将数组的尾部作为栈顶。如图 5-3 所示，入栈与出栈操作分别对应在数组尾部添加元素与删除元素，时间复杂度都为 $O(1)$ 。

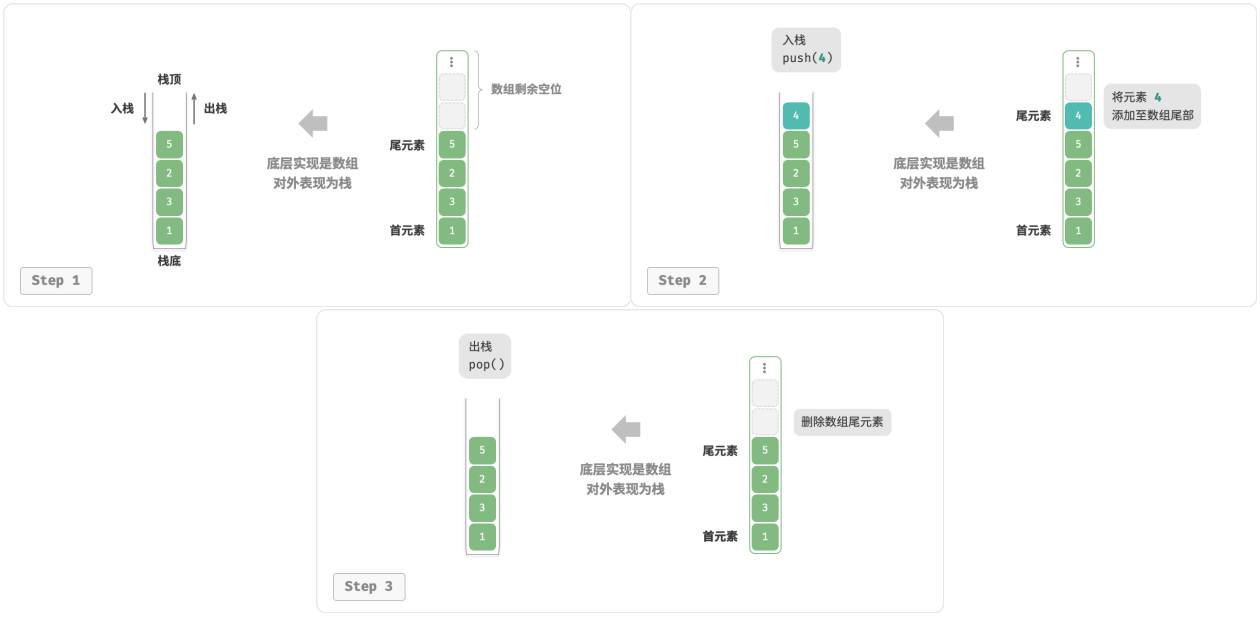


图 5-3　基于数组实现栈的入栈出栈操作

由于入栈的元素可能会源源不断地增加，因此我们可以使用动态数组，这样就无须自行处理数组扩容问题。以下为示例代码：

```c
// === File: array_stack.c ===

/* 基于数组实现的栈 */
typedef struct {
    int *data;
    int size;
} ArrayStack;

/* 构造函数 */
ArrayStack *newArrayStack() {
    ArrayStack *stack = malloc(sizeof(ArrayStack));
    // 初始化一个大容量，避免扩容
    stack->data = malloc(sizeof(int) * MAX_SIZE);
    stack->size = 0;
    return stack;
}

/* 析构函数 */
void delArrayStack(ArrayStack *stack) {
    free(stack->data);
    free(stack);
}

/* 获取栈的长度 */
int size(ArrayStack *stack) {
    return stack->size;
}

/* 判断栈是否为空 */
bool isEmpty(ArrayStack *stack) {
    return stack->size == 0;
}

/* 入栈 */
void push(ArrayStack *stack, int num) {
    if (stack->size == MAX_SIZE) {
        printf(" 栈已满\n");
        return;
    }
    stack->data[stack->size] = num;
    stack->size++;
}

/* 访问栈顶元素 */
int peek(ArrayStack *stack) {
    if (stack->size == 0) {
        printf(" 栈为空\n");
```

```
        return INT_MAX;
    }
    return stack->data[stack->size - 1];
}

/* 出栈 */
int pop(ArrayStack *stack) {
    int val = peek(stack);
    stack->size--;
    return val;
}
```

5.1.3　两种实现对比

支持操作

两种实现都支持栈定义中的各项操作。数组实现额外支持随机访问，但这已超出了栈的定义范畴，因此一般不会用到。

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。

5.1.4　栈的典型应用

- **浏览器中的后退与前进、软件中的撤销与反撤销**。每当我们打开新的网页，浏览器就会对上一个网页执行入栈，这样我们就可以通过后退操作回到上一个网页。后退操作实际上是在执行出栈。如果要同时支持后退和前进，那么需要两个栈来配合实现。

- **程序内存管理**。每次调用函数时，系统都会在栈顶添加一个栈帧，用于记录函数的上下文信息。在递归函数中，向下递推阶段会不断执行入栈操作，而向上回溯阶段则会不断执行出栈操作。

5.2　队列

队列（queue）是一种遵循先入先出规则的线性数据结构。顾名思义，队列模拟了排队现象，即新来的人不断加入队列尾部，而位于队列头部的人逐个离开。

如图 5-4 所示，我们将队列头部称为“队首”，尾部称为“队尾”，将把元素加入队尾的操作称为“入队”，删除队首元素的操作称为“出队”。

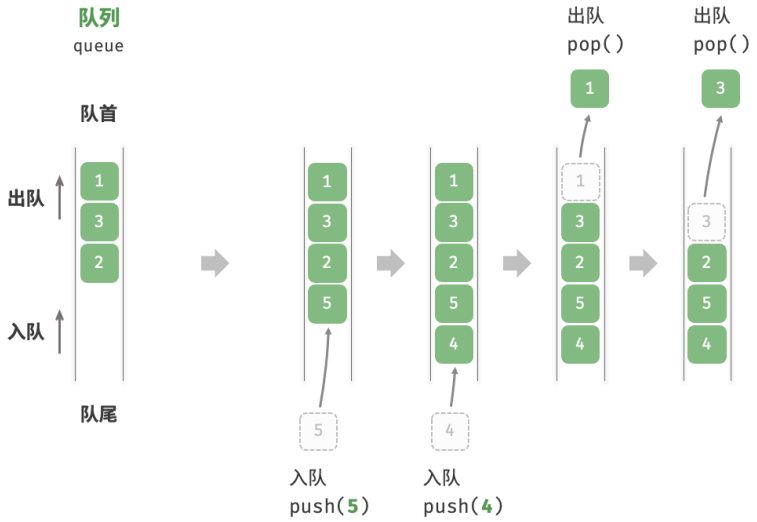


图 5-4　队列的先入先出规则

5.2.1　队列常用操作

队列的常见操作如表 5-2 所示。需要注意的是，不同编程语言的方法名称可能会有所不同。我们在此采用与栈相同的方法命名。

表 5-2　队列操作效率

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	$O(1)$

我们可以直接使用编程语言中现成的队列类：

```
// === File: queue.c ===

// C 未提供内置队列
```

5.2.2　队列实现

为了实现队列，我们需要一种数据结构，可以在一端添加元素，并在另一端删除元素，链表和数组都符合要求。

1.　基于链表的实现

如图 5-5 所示，我们可以将链表的“头节点”和“尾节点”分别视为“队首”和“队尾”，规定队尾仅可添加节点，队首仅可删除节点。

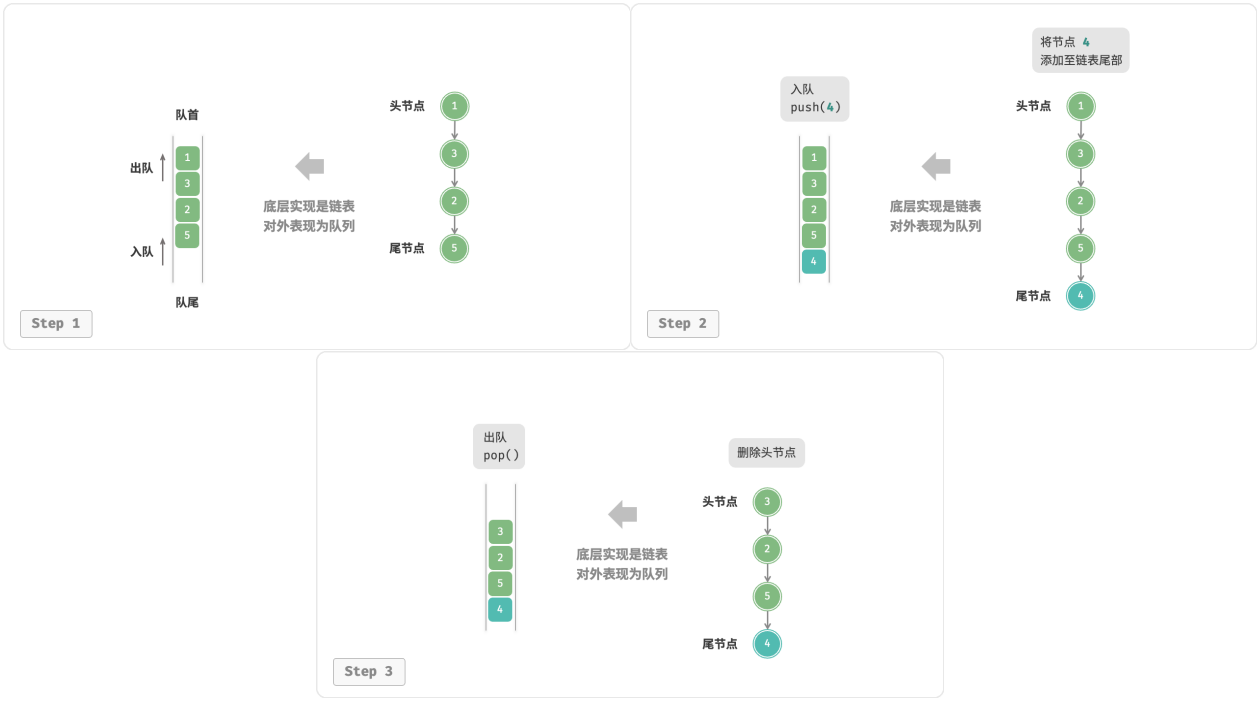


图 5-5　基于链表实现队列的入队出队操作

以下是用链表实现队列的代码：

```
// === File: linkedlist_queue.c ===

/* 基于链表实现的队列 */
typedef struct {
    ListNode *front, *rear;
    int queSize;
```

```c
} LinkedListQueue;

/* 构造函数 */
LinkedListQueue *newLinkedListQueue() {
    LinkedListQueue *queue = (LinkedListQueue *)malloc(sizeof(LinkedListQueue));
    queue->front = NULL;
    queue->rear = NULL;
    queue->queSize = 0;
    return queue;
}

/* 析构函数 */
void delLinkedListQueue(LinkedListQueue *queue) {
    // 释放所有节点
    while (queue->front != NULL) {
        ListNode *tmp = queue->front;
        queue->front = queue->front->next;
        free(tmp);
    }
    // 释放 queue 结构体
    free(queue);
}

/* 获取队列的长度 */
int size(LinkedListQueue *queue) {
    return queue->queSize;
}

/* 判断队列是否为空 */
bool empty(LinkedListQueue *queue) {
    return (size(queue) == 0);
}

/* 入队 */
void push(LinkedListQueue *queue, int num) {
    // 尾节点处添加 node
    ListNode *node = newListNode(num);
    // 如果队列为空，则令头、尾节点都指向该节点
    if (queue->front == NULL) {
        queue->front = node;
        queue->rear = node;
    }
    // 如果队列不为空，则将该节点添加到尾节点后
    else {
        queue->rear->next = node;
        queue->rear = node;
    }
```

```
    queue->queSize++;
}

/* 访问队首元素 */
int peek(LinkedListQueue *queue) {
    assert(size(queue) && queue->front);
    return queue->front->val;
}

/* 出队 */
int pop(LinkedListQueue *queue) {
    int num = peek(queue);
    ListNode *tmp = queue->front;
    queue->front = queue->front->next;
    free(tmp);
    queue->queSize--;
    return num;
}

/* 打印队列 */
void printLinkedListQueue(LinkedListQueue *queue) {
    int *arr = malloc(sizeof(int) * queue->queSize);
    // 拷贝链表中的数据到数组
    int i;
    ListNode *node;
    for (i = 0, node = queue->front; i < queue->queSize; i++) {
        arr[i] = node->val;
        node = node->next;
    }
    printArray(arr, queue->queSize);
    free(arr);
}
```

2. 基于数组的实现

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 。

可以看到，入队和出队操作都只需进行一次操作，时间复杂度均为 $O(1)$ 。

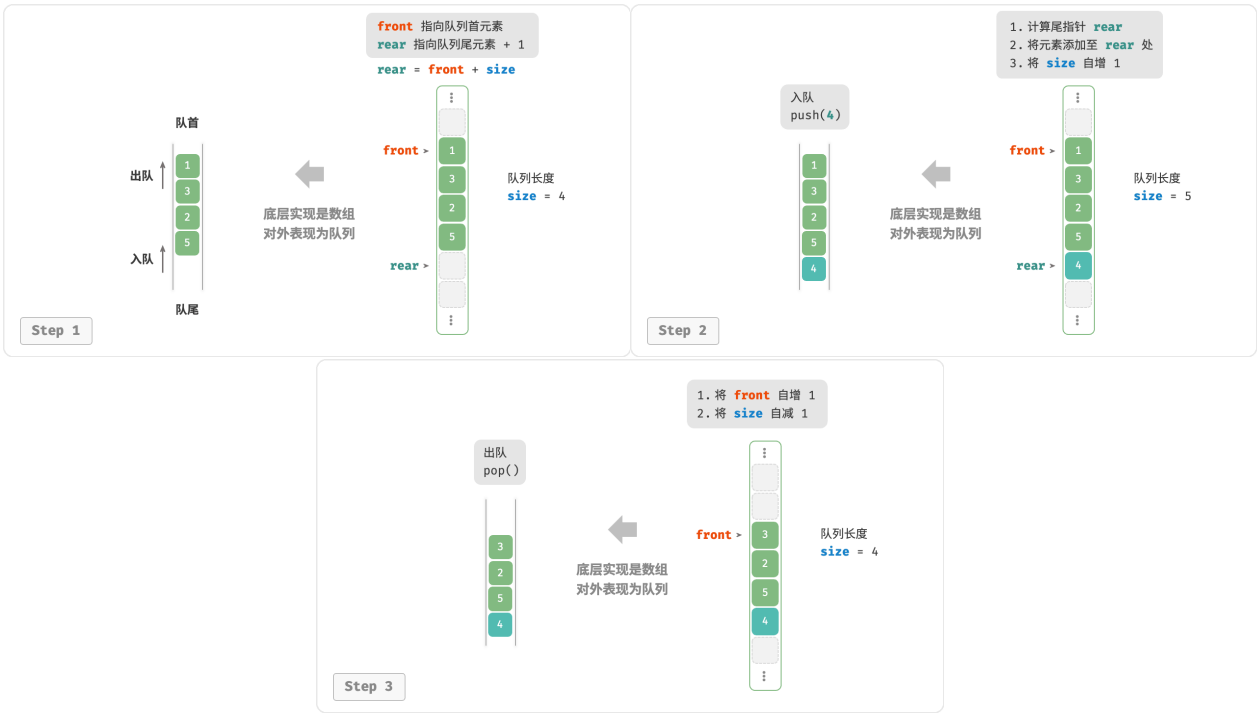


图 5-6　基于数组实现队列的入队出队操作

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：

```c
// === File: array_queue.c ===

/* 基于环形数组实现的队列 */
typedef struct {
    int *nums;       // 用于存储队列元素的数组
    int front;       // 队首指针，指向队首元素
    int queSize;     // 尾指针，指向队尾 + 1
    int queCapacity; // 队列容量
} ArrayQueue;

/* 构造函数 */
ArrayQueue *newArrayQueue(int capacity) {
    ArrayQueue *queue = (ArrayQueue *)malloc(sizeof(ArrayQueue));
    // 初始化数组
    queue->queCapacity = capacity;
    queue->nums = (int *)malloc(sizeof(int) * queue->queCapacity);
    queue->front = queue->queSize = 0;
    return queue;
}
```

```c
/* 析构函数 */
void delArrayQueue(ArrayQueue *queue) {
    free(queue->nums);
    free(queue);
}

/* 获取队列的容量 */
int capacity(ArrayQueue *queue) {
    return queue->queCapacity;
}

/* 获取队列的长度 */
int size(ArrayQueue *queue) {
    return queue->queSize;
}

/* 判断队列是否为空 */
bool empty(ArrayQueue *queue) {
    return queue->queSize == 0;
}

/* 访问队首元素 */
int peek(ArrayQueue *queue) {
    assert(size(queue) != 0);
    return queue->nums[queue->front];
}

/* 入队 */
void push(ArrayQueue *queue, int num) {
    if (size(queue) == capacity(queue)) {
        printf(" 队列已满\r\n");
        return;
    }
    // 计算队尾指针，指向队尾索引 + 1
    // 通过取余操作实现 rear 越过数组尾部后回到头部
    int rear = (queue->front + queue->queSize) % queue->queCapacity;
    // 将 num 添加至队尾
    queue->nums[rear] = num;
    queue->queSize++;
}

/* 出队 */
int pop(ArrayQueue *queue) {
    int num = peek(queue);
    // 队首指针向后移动一位，若越过尾部，则返回到数组头部
    queue->front = (queue->front + 1) % queue->queCapacity;
```

```
    queue->queSize--;
    return num;
}

/* 返回数组用于打印 */
int *toArray(ArrayQueue *queue, int *queSize) {
    *queSize = queue->queSize;
    int *res = (int *)calloc(queue->queSize, sizeof(int));
    int j = queue->front;
    for (int i = 0; i < queue->queSize; i++) {
        res[i] = queue->nums[j % queue->queCapacity];
        j++;
    }
    return res;
}
```

以上实现的队列仍然具有局限性：其长度不可变。然而，这个问题不难解决，我们可以将数组替换为动态数组，从而引入扩容机制。有兴趣的读者可以尝试自行实现。

两种实现的对比结论与栈一致，在此不再赘述。

5.2.3　队列典型应用

- **淘宝订单**。购物者下单后，订单将加入队列中，系统随后会根据顺序处理队列中的订单。在双十一期间，短时间内会产生海量订单，高并发成为工程师们需要重点攻克的问题。
- **各类待办事项**。任何需要实现“先来后到”功能的场景，例如打印机的任务队列、餐厅的出餐队列等，队列在这些场景中可以有效地维护处理顺序。

5.3　双向队列

在队列中，我们仅能删除头部元素或在尾部添加元素。如图 5-7 所示，双向队列（double-ended queue）提供了更高的灵活性，允许在头部和尾部执行元素的添加或删除操作。

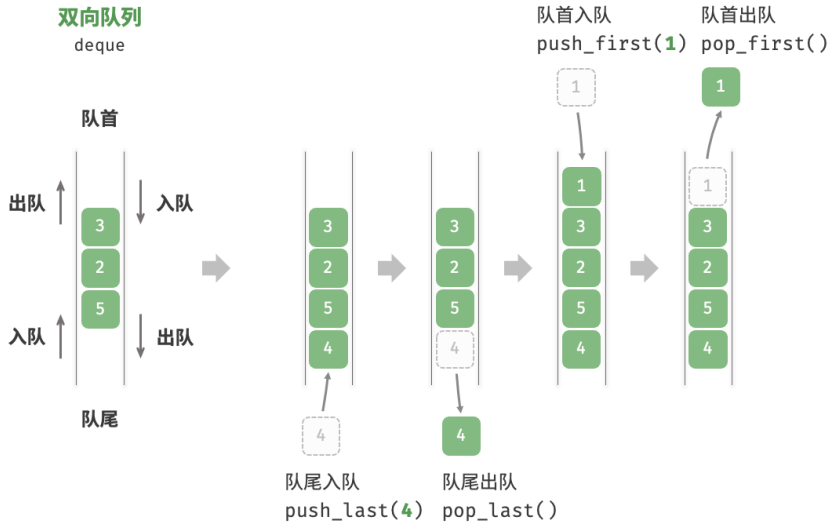


图 5-7　双向队列的操作

5.3.1　双向队列常用操作

双向队列的常用操作如表 5-3 所示，具体的方法名称需要根据所使用的编程语言来确定。

表 5-3　双向队列操作效率

方法名	描述	时间复杂度
`push_first()`	将元素添加至队首	$O(1)$
`push_last()`	将元素添加至队尾	$O(1)$
`pop_first()`	删除队首元素	$O(1)$
`pop_last()`	删除队尾元素	$O(1)$
`peek_first()`	访问队首元素	$O(1)$
`peek_last()`	访问队尾元素	$O(1)$

同样地，我们可以直接使用编程语言中已实现的双向队列类：

```c
// === File: deque.c ===

// C 未提供内置双向队列
```

5.3.2　双向队列实现 *

双向队列的实现与队列类似，可以选择链表或数组作为底层数据结构。

1. 基于双向链表的实现

回顾上一节内容，我们使用普通单向链表来实现队列，因为它可以方便地删除头节点（对应出队操作）和在尾节点后添加新节点（对应入队操作）。

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。换句话说，双向队列需要实现另一个对称方向的操作。为此，我们采用“双向链表”作为双向队列的底层数据结构。

如图 5-8 所示，我们将双向链表的头节点和尾节点视为双向队列的队首和队尾，同时实现在两端添加和删除节点的功能。

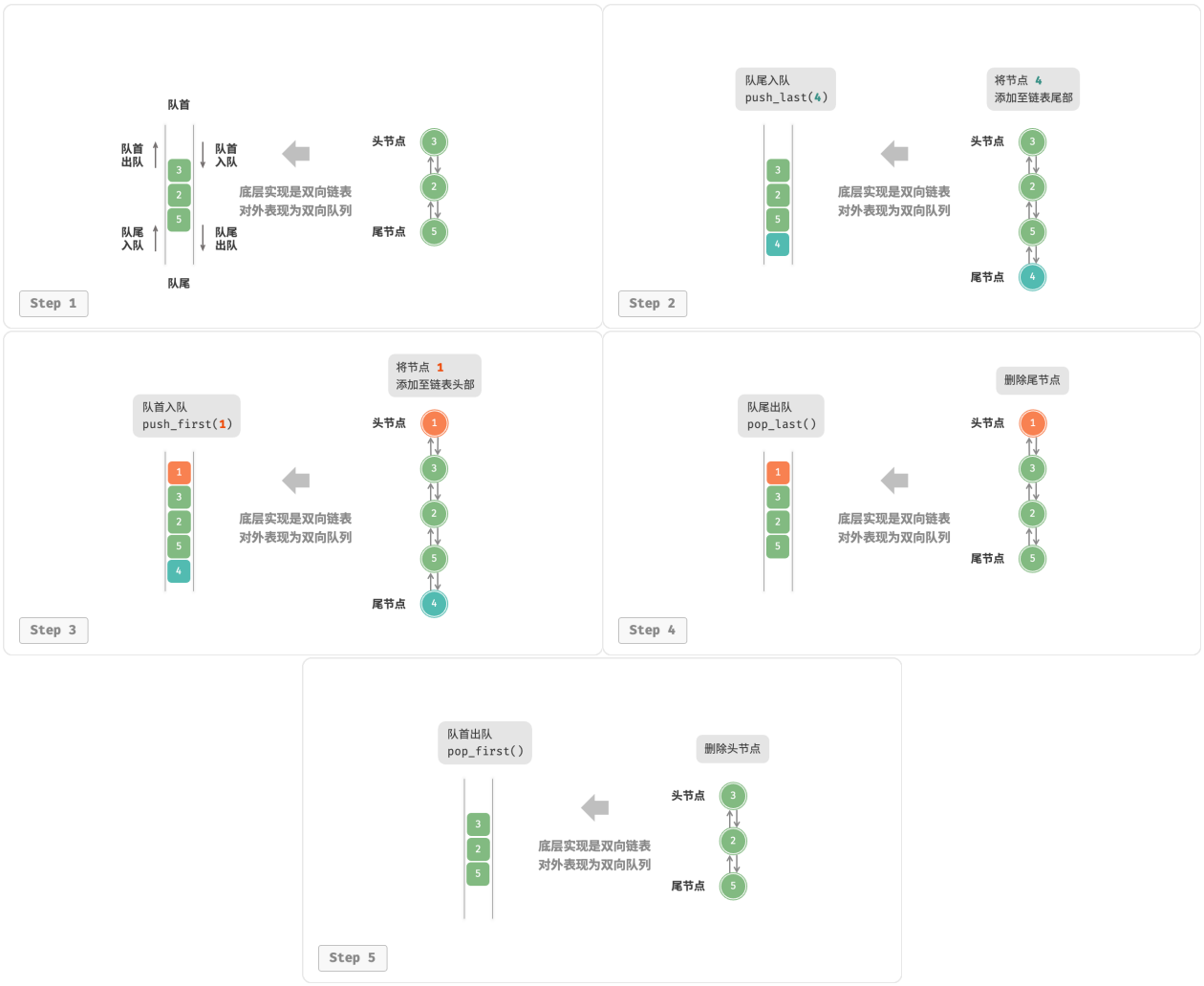


图 5-8　基于链表实现双向队列的入队出队操作

实现代码如下所示：

```
// === File: linkedlist_deque.c ===

/* 双向链表节点 */
```

```c
typedef struct DoublyListNode {
    int val;                    // 节点值
    struct DoublyListNode *next; // 后继节点
    struct DoublyListNode *prev; // 前驱节点
} DoublyListNode;

/* 构造函数 */
DoublyListNode *newDoublyListNode(int num) {
    DoublyListNode *new = (DoublyListNode *)malloc(sizeof(DoublyListNode));
    new->val = num;
    new->next = NULL;
    new->prev = NULL;
    return new;
}

/* 析构函数 */
void delDoublyListNode(DoublyListNode *node) {
    free(node);
}

/* 基于双向链表实现的双向队列 */
typedef struct {
    DoublyListNode *front, *rear; // 头节点 front ，尾节点 rear
    int queSize;                  // 双向队列的长度
} LinkedListDeque;

/* 构造函数 */
LinkedListDeque *newLinkedListDeque() {
    LinkedListDeque *deque = (LinkedListDeque *)malloc(sizeof(LinkedListDeque));
    deque->front = NULL;
    deque->rear = NULL;
    deque->queSize = 0;
    return deque;
}

/* 析构函数 */
void delLinkedListdeque(LinkedListDeque *deque) {
    // 释放所有节点
    for (int i = 0; i < deque->queSize && deque->front != NULL; i++) {
        DoublyListNode *tmp = deque->front;
        deque->front = deque->front->next;
        free(tmp);
    }
    // 释放 deque 结构体
    free(deque);
}
```

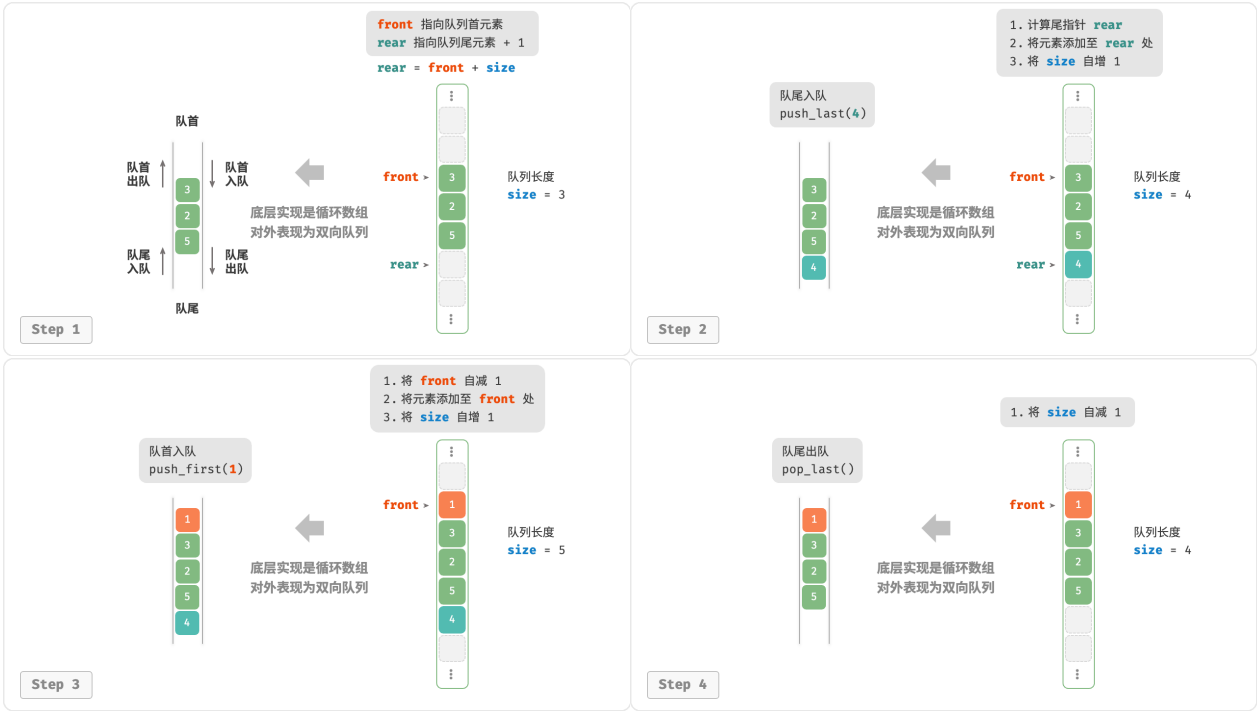
```c
/* 获取队列的长度 */
int size(LinkedListDeque *deque) {
    return deque->queSize;
}

/* 判断队列是否为空 */
bool empty(LinkedListDeque *deque) {
    return (size(deque) == 0);
}

/* 入队 */
void push(LinkedListDeque *deque, int num, bool isFront) {
    DoublyListNode *node = newDoublyListNode(num);
    // 若链表为空，则令 front 和 rear 都指向 node
    if (empty(deque)) {
        deque->front = deque->rear = node;
    }
    // 队首入队操作
    else if (isFront) {
        // 将 node 添加至链表头部
        deque->front->prev = node;
        node->next = deque->front;
        deque->front = node; // 更新头节点
    }
    // 队尾入队操作
    else {
        // 将 node 添加至链表尾部
        deque->rear->next = node;
        node->prev = deque->rear;
        deque->rear = node;
    }
    deque->queSize++; // 更新队列长度
}

/* 队首入队 */
void pushFirst(LinkedListDeque *deque, int num) {
    push(deque, num, true);
}

/* 队尾入队 */
void pushLast(LinkedListDeque *deque, int num) {
    push(deque, num, false);
}

/* 访问队首元素 */
int peekFirst(LinkedListDeque *deque) {
    assert(size(deque) && deque->front);
```

```
    return deque->front->val;
}

/* 访问队尾元素 */
int peekLast(LinkedListDeque *deque) {
    assert(size(deque) && deque->rear);
    return deque->rear->val;
}

/* 出队 */
int pop(LinkedListDeque *deque, bool isFront) {
    if (empty(deque))
        return -1;
    int val;
    // 队首出队操作
    if (isFront) {
        val = peekFirst(deque); // 暂存头节点值
        DoublyListNode *fNext = deque->front->next;
        if (fNext) {
            fNext->prev = NULL;
            deque->front->next = NULL;
        }
        delDoublyListNode(deque->front);
        deque->front = fNext; // 更新头节点
    }
    // 队尾出队操作
    else {
        val = peekLast(deque); // 暂存尾节点值
        DoublyListNode *rPrev = deque->rear->prev;
        if (rPrev) {
            rPrev->next = NULL;
            deque->rear->prev = NULL;
        }
        delDoublyListNode(deque->rear);
        deque->rear = rPrev; // 更新尾节点
    }
    deque->queSize--; // 更新队列长度
    return val;
}

/* 队首出队 */
int popFirst(LinkedListDeque *deque) {
    return pop(deque, true);
}

/* 队尾出队 */
int popLast(LinkedListDeque *deque) {
```

```
    return pop(deque, false);
}

/* 打印队列 */
void printLinkedListDeque(LinkedListDeque *deque) {
    int *arr = malloc(sizeof(int) * deque->queSize);
    // 拷贝链表中的数据到数组
    int i;
    DoublyListNode *node;
    for (i = 0, node = deque->front; i < deque->queSize; i++) {
        arr[i] = node->val;
        node = node->next;
    }
    printArray(arr, deque->queSize);
    free(arr);
}
```

2. 基于数组的实现

如图 5-9 所示，与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。



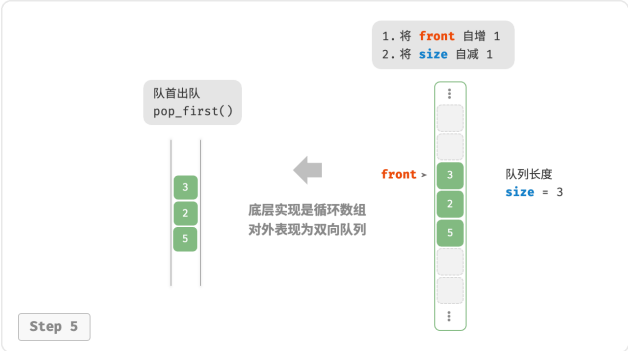


图 5-9　基于数组实现双向队列的入队出队操作

在队列的实现基础上，仅需增加“队首入队”和“队尾出队”的方法：

```
// === File: array_deque.c ===

/* 基于环形数组实现的双向队列 */
typedef struct {
    int *nums;       // 用于存储队列元素的数组
    int front;       // 队首指针，指向队首元素
    int queSize;     // 尾指针，指向队尾 + 1
    int queCapacity; // 队列容量
} ArrayDeque;

/* 构造函数 */
ArrayDeque *newArrayDeque(int capacity) {
    ArrayDeque *deque = (ArrayDeque *)malloc(sizeof(ArrayDeque));
    // 初始化数组
    deque->queCapacity = capacity;
    deque->nums = (int *)malloc(sizeof(int) * deque->queCapacity);
    deque->front = deque->queSize = 0;
    return deque;
}

/* 析构函数 */
void delArrayDeque(ArrayDeque *deque) {
    free(deque->nums);
    free(deque);
}

/* 获取双向队列的容量 */
int capacity(ArrayDeque *deque) {
    return deque->queCapacity;
}

/* 获取双向队列的长度 */
```

```c
int size(ArrayDeque *deque) {
    return deque->queSize;
}

/* 判断双向队列是否为空 */
bool empty(ArrayDeque *deque) {
    return deque->queSize == 0;
}

/* 计算环形数组索引 */
int dequeIndex(ArrayDeque *deque, int i) {
    // 通过取余操作实现数组首尾相连
    // 当 i 越过数组尾部时，回到头部
    // 当 i 越过数组头部后，回到尾部
    return ((i + capacity(deque)) % capacity(deque));
}

/* 队首入队 */
void pushFirst(ArrayDeque *deque, int num) {
    if (deque->queSize == capacity(deque)) {
        printf(" 双向队列已满\r\n");
        return;
    }
    // 队首指针向左移动一位
    // 通过取余操作实现 front 越过数组头部回到尾部
    deque->front = dequeIndex(deque, deque->front - 1);
    // 将 num 添加到队首
    deque->nums[deque->front] = num;
    deque->queSize++;
}

/* 队尾入队 */
void pushLast(ArrayDeque *deque, int num) {
    if (deque->queSize == capacity(deque)) {
        printf(" 双向队列已满\r\n");
        return;
    }
    // 计算队尾指针，指向队尾索引 + 1
    int rear = dequeIndex(deque, deque->front + deque->queSize);
    // 将 num 添加至队尾
    deque->nums[rear] = num;
    deque->queSize++;
}

/* 访问队首元素 */
int peekFirst(ArrayDeque *deque) {
    // 访问异常：双向队列为空
```

```
    assert(empty(deque) == 0);
    return deque->nums[deque->front];
}

/* 访问队尾元素 */
int peekLast(ArrayDeque *deque) {
    // 访问异常：双向队列为空
    assert(empty(deque) == 0);
    int last = dequeIndex(deque, deque->front + deque->queSize - 1);
    return deque->nums[last];
}

/* 队首出队 */
int popFirst(ArrayDeque *deque) {
    int num = peekFirst(deque);
    // 队首指针向后移动一位
    deque->front = dequeIndex(deque, deque->front + 1);
    deque->queSize--;
    return num;
}

/* 队尾出队 */
int popLast(ArrayDeque *deque) {
    int num = peekLast(deque);
    deque->queSize--;
    return num;
}

/* 返回数组用于打印 */
int *toArray(ArrayDeque *deque, int *queSize) {
    *queSize = deque->queSize;
    int *res = (int *)calloc(deque->queSize, sizeof(int));
    int j = deque->front;
    for (int i = 0; i < deque->queSize; i++) {
        res[i] = deque->nums[j % deque->queCapacity];
        j++;
    }
    return res;
}
```

5.3.3　双向队列应用

双向队列兼具栈与队列的逻辑，**因此它可以实现这两者的所有应用场景，同时提供更高的自由度**。

我们知道，软件的“撤销”功能通常使用栈来实现：系统将每次更改操作 `push` 到栈中，然后通过 `pop` 实现撤销。然而，考虑到系统资源的限制，软件通常会限制撤销的步数（例如仅允许保存 50 步）。当栈的长度超过

50 时，软件需要在栈底（队首）执行删除操作。**但栈无法实现该功能，此时就需要使用双向队列来替代栈**。请注意，“撤销”的核心逻辑仍然遵循栈的先入后出原则，只是双向队列能够更加灵活地实现一些额外逻辑。

5.4 小结

1. 重点回顾

- 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。

2. Q & A

Q：浏览器的前进后退是否是双向链表实现？

浏览器的前进后退功能本质上是“栈”的体现。当用户访问一个新页面时，该页面会被添加到栈顶；当用户点击后退按钮时，该页面会从栈顶弹出。使用双向队列可以方便地实现一些额外操作，这个在“双向队列”章节有提到。

Q：在出栈后，是否需要释放出栈节点的内存？

如果后续仍需要使用弹出节点，则不需要释放内存。若之后不需要用到，`Java` 和 `Python` 等语言拥有自动垃圾回收机制，因此不需要手动释放内存；在 `C` 和 `C++` 中需要手动释放内存。

Q：双向队列像是两个栈拼接在了一起，它的用途是什么？

双向队列就像是栈和队列的组合或两个栈拼在了一起。它表现的是栈 + 队列的逻辑，因此可以实现栈与队列的所有应用，并且更加灵活。

Q：撤销（undo）和反撤销（redo）具体是如何实现的？

使用两个栈，栈 `A` 用于撤销，栈 `B` 用于反撤销。

1. 每当用户执行一个操作，将这个操作压入栈 `A` ，并清空栈 `B` 。
2. 当用户执行“撤销”时，从栈 `A` 中弹出最近的操作，并将其压入栈 `B` 。
3. 当用户执行“反撤销”时，从栈 `B` 中弹出最近的操作，并将其压入栈 `A` 。