数据结构与算法

作业 3：数组与链表

您的姓名　您的学号

November 26, 2025

问题一

请在SimplifiedList 基础上，尝试实现 SimplifiedList 类的末尾删除操作 pop() 方法。这个方法应该从列表末尾删除一个元素，并返回被删除的元素，同时更新内部状态。你也应该仿照 __getitem__() 方法，添加边界检查，确保在删除时不会访问到不存在的索引（如索引小于 0 或大于等于当前长度）。如果删除成功，返回被删除的元素，否则抛出 **IndexError** 异常。

解：

```python
import ctypes
from collections import namedtuple

# 定义“管理区”
# 使用 namedtuple 来表示这个管理结构，它包含三个核心元数据：
# n: 当前存储的元素数量 (size)
# capacity: 底层数组的总容量
# data: 指向底层数组的引用 (pointer)
ListInternal = namedtuple("ListInternal", ["n", "capacity", "data"])


class SimplifiedList:
    """ 一个简化版的动态数组。"""

    def __init__(self):
        """ 初始化一个空的列表。"""
        # 初始时，列表为空，大小和容量都为 0。
```

```
        # self._internal 完美地模拟了那个“列表对象头”。
        self._internal = ListInternal(0, 0, self._make_array(0))

    def __len__(self):
        """
        获取列表长度，复杂度 O(1)。
        这完美地解释了为什么 len() 如此之快。
        """
        return self._internal.n

    def __getitem__(self, k):
        """
        通过索引获取元素，复杂度 O(1)。
        这是数组的随机访问特性。
        """
        # 边界检查
        if not 0 <= k < self._internal.n:
            raise IndexError("list index out of range")
        return self._internal.data[k]

    def append(self, obj):
        """
        在列表末尾添加元素，摊还复杂度 O(1)。
        """
        # 核心逻辑：如果空间不足，就进行扩容
        if self._internal.n == self._internal.capacity:
            self._resize(
                max(1, 2 * self._internal.capacity)
            )  # 至少扩容到 1，或当前容量的两倍

        # 将新元素放入第一个可用的空位
        self._internal.data[self._internal.n] = obj

        # 更新 size 计数器（因为 namedtuple 不可变，我们创建一个新实例）
        self._internal = self._internal._replace(n=self._internal.n + 1)


    def pop(self):
```

```
        """
        从列表末尾删除一个元素并返回它，复杂度 O(1)。
        """
        # 请在此添加您的代码

        # 边界检查


    def _resize(self, new_capacity):
        """
        私有方法，用于扩容底层数组（这是一个 O(n) 操作）。
        """
        # 1. 创建一个新的、更大的底层数组
        new_array = self._make_array(new_capacity)

        # 2. 将旧数组中的所有元素复制到新数组
        for i in range(self._internal.n):
            new_array[i] = self._internal.data[i]

        # 3. 更新内部状态，指向新的数组和容量
        self._internal = ListInternal(self._internal.n,
        ↪  new_capacity, new_array)

    def _make_array(self, capacity):
        """
        创建一个类似 C 语言的、固定大小的底层数组。
        """
        # (ctypes.py_object * capacity)()
        ↪  创建一个能存放'capacity'个Python对象的数组
        return (ctypes.py_object * capacity)()

    def __repr__(self):
        """
        提供一个美观的打印输出，类似于真实的 list。
        """
        elements = [repr(self[i]) for i in
        ↪  range(len(self))]
        return f"[{', '.join(elements)}]"
```

问题二

为什么我们说数组支持“随机访问”，而链表只支持“顺序访问”？请从它们在内存中的存储方式来解释其根本原因。

解：

问题三

简述单向链表、双向链表和循环链表各自的优缺点，并为每种结构举出一个典型的应用场景。

解：

问题四

场景分析：你正在为一个音乐 App 设计“播放队列”功能。用户有以下几种核心需求：

1. 随时查看当前播放歌曲在队列中的位置（即获取其索引）。
2. 支持“下一首”播放（即移动到下一个元素）。
3. 允许用户将任意一首歌“置顶”（即将任意一个元素移动到队列头部）。
4. 允许用户删除队列中的任意一首歌。

请分别使用数组和双向链表来分析实现这四个功能的复杂度，并最终给出你的技术选型和理由。建议使用一个表格来清晰地进行对比。

解：

问题五

给定一个单向链表的头节点 head，请设计一个算法，要求只遍历一次链表，就能找到链表的倒数第 k 个节点。请描述你的算法思路（提示：可以使用双指针法），并分析其时间复杂度和空间复杂度。

解：

问题六

请实现一个双向链表的 `insert_before` 方法，用于在指定节点前插入一个新节点。要求时间复杂度为 $O(1)$。

解：

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert_before(self, node, new_node):
        # 请在此添加您的代码
```

问题七

请实现一个双向循环链表（circular doubly linked list），其中每个节点既有 `next` 指针指向后继节点，也有 `prev` 指针指向前驱节点。要求实现基本的插入、删除和遍历操作。

解：

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class CircularDoublyLinkedList:
```

```
    # 请在此添加您的代码
```

问题八

请实现一个双向链表的 reverse 方法，用于反转链表。要求时间复杂度为 $O(n)$。例如，一个链表 A -> B -> C -> None，反转后应为 C -> B -> A -> None。

解:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def reverse(self):
        prev = None
        current = self.head
        while current:
            # 在这里补充你的代码
            ...
        self.head = prev
```

问题九

给定两个已排序的单向链表 list1 和 list2，请编写一个函数 merge_sorted_lists(list1, list2)，将它们合并成一个新的、仍然有序的链表，并返回新链表的头节点。不允许创建新的节点，只能通过修改原有两个链表的节点指针来完成。

解:

```
def merge_sorted_lists(list1, list2):
```

```
    # 在这里补充你的代码
```