数据结构与算法B 作业6：链表、栈和排序

> **说明：**
>
> 1. **解题与记录：**
>
> 对于每一个题目，请提供其解题思路（可选），并附上使用 Python 或 C++编写的源代码（确保已在
> OpenJudge ， Codeforces，LeetCode 等平台上获得 Accepted “）。请将这些信息连同显示 Accepted”的
> 截图一起填写到下方的作业模板中。（推荐使用 Typora https://typoraio.cn 进行编辑，当然你也可以选
> 择 Word。）无论题目是否已通过，请标明每个题目大致花费的时间。
>
> 2. **提交安排：**提交时，请首先上传 PDF 格式的文件，并将.md 或.doc 格式的文件作为附件上传至右侧的
> “ ”作业评论 区。确保你的 Canvas 账户有一个清晰可见的本人头像，提交的文件为 PDF “ ”格式，并且 作业评论 区
> 包含上传的.md 或.doc 附件。
> 3. **延迟提交：**如果你预计无法在截止日期前提交作业，请提前告知具体原因。这有助于我们了解情况并可
> 能为你提供适当的延期或其他帮助。
>
> 请按照上述指导认真准备和提交作业，以保证顺利完成课程要求。

1. 题目

E24588: 后序表达式求值

Stack, http://cs101.openjudge.cn/practice/24588/
思路： 也许是这次题目个人解决起来唯一一道轻松的题目了。利用了栈进行解题，将数字添加进栈中，遇到符号就将最近的两个数字出栈，算出这次运算的答案即可。
代码：

```
def cal(m, n, sign):
    if sign == '+':
        return m + n
    elif sign == '-':
        return m - n
    elif sign == '*':
        return m * n
    elif sign == '/':
        return m / n
```

```python
def main():
    n = int(input())
    for i in range(n):
        stack = []
        string = list(input().split())
        for j in string:
            if j in ['+', '-', '*', '/']:
                q = stack.pop()
                p = stack.pop()
                stack.append(cal(p, q, j))
            else:
                stack.append(float(j))
        print(f'{stack[0]:.2f}')



if __name__ == '__main__':
    main()
```

代码运行截图：

#50361683提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```python
def cal(m, n, sign):
    if sign == '+':
        return m + n
    elif sign == '-':
        return m - n
    elif sign == '*':
        return m * n
    elif sign == '/':
        return m / n

def main():
    n = int(input())
    for i in range(n):
        stack = []
        string = list(input().split())
        for j in string:
            if j in ['+', '-', '*', '/']:
                q = stack.pop()
                p = stack.pop()
                stack.append(cal(p, q, j))
            else:
                stack.append(float(j))
        print(f'{stack[0]:.2f}')



if __name__ == '__main__':
    main()
```

基本信息

#: 50361683
题目: 24588
提交人: 22n2200011816(略约横溪)
内存: 3652kB
时间: 21ms
语言: Python3
提交时间: 2025-10-14 19:09:30

M234.回文链表

linked list, https://leetcode.cn/problems/palindrome-linked-list/

思路： 之前完成题目的时候是转化成列表存储下来，进行回文比较。现在则是遍历一遍后，将列表后半部分反转，使用两个指针对链表在两个方向上进行遍历，比较链表，这样就不用列

表进行存储了，也就顺利地把空间复杂度降下来了。
代码：

```python
class Solution:
    def isPalindrome(self, head) -> bool:
        q = head
        length = 0
        while q:
            q = q.next
            length += 1
        q = head
        temp1 = None
        for i in range(length-1):
            if i >= length // 2:
                temp = q
                if temp1:
                    q = temp1
                else:
                    q = q.next
                temp1 = q.next
                q.next = temp
            else:
                q = q.next
        first = q
        second = head
        for i in range(length//2):
            if first.val == second.val:
                first = first.next
                second = second.next
            else:
                return False
        return True
```

代码运行截图 （至少包含有"Accepted"）

M27217: 有多少种合法的出栈顺序

http://cs101.openjudge.cn/practice/27217/

思路：对这种比较考验数学思维的题目，我确实很难独立做出来。我自己是写了一个暴力解法来排列有多少种push和pop的组合，但是时间复杂度明显爆炸了。后面善用搜索才知道有卡特兰数这个东西。

对于卡特兰数，由于这是它的递推式子：

之所以卡特兰数可以用于接这道题，本质上就是对含有n个数的数组，如果其中第x个数出栈后，栈完全清空，其出栈的可能性就有 种，从而得到上文的递推式。知道这个思路后，使用回溯法就可以递推出来了。

代码：

```
def main():
    n = int(input())
    result = []
    def backtrack(n):
        temp = 0
        if n > 1:
            backtrack(n - 1)
        else:
```

```
            result.append(1)
        for i in range(n):
            temp += result[i]*result[n-i-1]
        result.append(temp)
    backtrack(n)
    print(result[-1])

if __name__ == '__main__':
    main()
```

代码运行截图 （至少包含有"Accepted"）

#50365266提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
def main():
    n = int(input())
    result = []

    def backtrack(n):
        temp = 0

        if n > 1:
            backtrack(n - 1)
        else:
            result.append(1)
        for i in range(n):
            temp += result[i]*result[n-i-1]
        result.append(temp)
    backtrack(n)
    print(result[-1])

if __name__ == '__main__':
    main()
```

基本信息

#: 50365266
题目: 27217
提交人: 22n2200011816(略约横溪)
内存: 4508kB
时间: 226ms
语言: Python3
提交时间: 2025-10-14 21:59:54

M24591:中序表达式转后序表达式

http://cs101.openjudge.cn/practice/24591/

思路： 这道题目对我而言确实难度不小。在思考时，我首先奠定的基调是：对于中序表达式中括号套括号的情况，根据之前括号匹配问题的经验，首先肯定还是需要用到栈，用于匹配括号，其次，括号中的表达式需要优先处理，整个式子就由于不同的括号变成了不同的part，因此可以使用面向对象的方式，对每个part的中序表达式调用一次函数单独处理，再塞回原来的式子中，最后输出答案。

代码

```
# 这个模块用于将符号和数字分割开
def seperate(s):
    result = []
    temp = ''
    for j in s:
        if j in ['+', '-', '*', '/', '(', ')']:
            if temp != '':
                result.append(temp)
                temp = ''
            result.append(j)
        else:
```

```
            temp += j
    if temp != '':
        result.append(temp)
    return result

# 局部的中序表达式转后续表达式
def middle_to_end(s):
    j = 0
    while j < len(s):
        if s[j] == '*' or s[j] == '/':
            s[j], s[j+1] = s[j+1], s[j]
            temp = ' '.join(s[j-1:j+2])
            del s[j-1:j+2]
            s.insert(j-1, temp)
            j -= 1
        j += 1
    j = 0
    while j < len(s):
        if s[j] == '+' or s[j] == '-':
            s[j], s[j+1] = s[j+1], s[j]
            temp = ' '.join(s[j-1:j+2])
            del s[j-1:j+2]
            s.insert(j-1, temp)
            j -= 1
        j += 1
    return s

# 使用栈的思想解决问题
def main():
    n = int(input())
    for i in range(n):
        origin = input()
        middle_num_and_sign = seperate(origin)
        stack = []
        j = 0
        while 1:
            if middle_num_and_sign[j] == '(':
                stack.append(j)
            elif middle_num_and_sign[j] == ')':
                index = stack.pop()
                temp = middle_to_end(middle_num_and_sign[index+1:j])
                del middle_num_and_sign[index:j+1]
                middle_num_and_sign.insert(index, temp[0])
                j -= (j - index + 1)
            j += 1
            if j == len(middle_num_and_sign):
                break
        print(' '.join(middle_to_end(middle_num_and_sign)))
```

```
if __name__ == '__main__':
    main()
```

（至少包含有"Accepted"）

#50384445提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
def seperate(s):
    result = []
    temp = ''
    for j in s:
        if j in ['+', '-', '*', '/', '(', ')']:
            if temp != '':
                result.append(temp)
                temp = ''
            result.append(j)
        else:
            temp += j
    if temp != '':
        result.append(temp)
    return result

def middle_to_end(s):
    j = 0
    while j < len(s):
        if s[j] == '*' or s[j] == '/':
            s[j], s[j+1] = s[j+1], s[j]
            temp = ' '.join(s[j-1:j+2])
            del s[j-1:j+2]
            s.insert(j-1, temp)
            j -= 1
        j += 1
```

基本信息

#: 50384445
题目: 24591
提交人: 22n2200011816(略约横溪)
内存: 3720kB
时间: 28ms
语言: Python3
提交时间: 2025-10-15 21:12:42

M02299:Ultra-QuickSort

merge sort, http://cs101.openjudge.cn/practice/02299/

思路：这个题目要求是相邻元素调换，多少次操作可以将列表排序，于是我去学习了一下各种排序的原理和复杂度，发现归并排序就是用相邻元素的调换完成的，于是果断copy了一份归并排序的代码，并把右边列表中的数字移动到左边的位次定义为操作的数量，最后也是AC了。

代码

```
def merge_sort(arr, os_times):
    if len(arr) <= 1:
        return arr, os_times

    # 分解：将列表分成两半
    mid = len(arr) // 2
    left_half, os_times = merge_sort(arr[:mid], os_times)  # 递归排序左半部分
    right_half, os_times = merge_sort(arr[mid:], os_times)  # 递归排序右半部分

    # 合并：将两个有序子列表合并
    return merge(left_half, right_half, os_times)

def merge(left, right, os_times):
    sorted_arr = []
```

```
    i = j = 0

    # 比较两个子列表的元素，按顺序合并
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_arr.append(left[i])
            i += 1
        else:
            sorted_arr.append(right[j])
            j += 1
            os_times += (len(left) - i)

    # 将剩余的元素添加到结果中
    sorted_arr.extend(left[i:])
    sorted_arr.extend(right[j:])
    return [sorted_arr, os_times]

# 示例
while 1:
    n = int(input())
    if n == 0:
        break
    arr = []
    for i in range(n):
        arr.append(int(input()))
    #arr = [1,2,3]
    #arr = [9, 1, 0, 5, 4]    sorted_arr = merge_sort(arr, os_times=0)
    print(sorted_arr[1])
```

（至少包含有"Accepted"）

#50420047提交状态

查看　提交　统计　提问

状态：Accepted

源代码

```
def merge_sort(arr, os_times):
    if len(arr) <= 1:
        return arr, os_times

    # 分解：将列表分成两半
    mid = len(arr) // 2
    left_half, os_times = merge_sort(arr[:mid], os_times)  # 递归排序左半
    right_half, os_times = merge_sort(arr[mid:], os_times)  # 递归排序右半

    # 合并：将两个有序子列表合并
    return merge(left_half, right_half, os_times)

def merge(left, right, os_times):
    sorted_arr = []
    i = j = 0

    # 比较两个子列表的元素，按顺序合并
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_arr.append(left[i])
            i += 1
        else:
            sorted_arr.append(right[j])
            j += 1
            os_times += (len(left) - i)
```

基本信息

#:	50420047
题目:	02299
提交人:	22n2200011816(略约横溪)
内存:	32800kB
时间:	4011ms
语言:	Python3
提交时间:	2025-10-17 20:32:20

M146.LRU 缓存

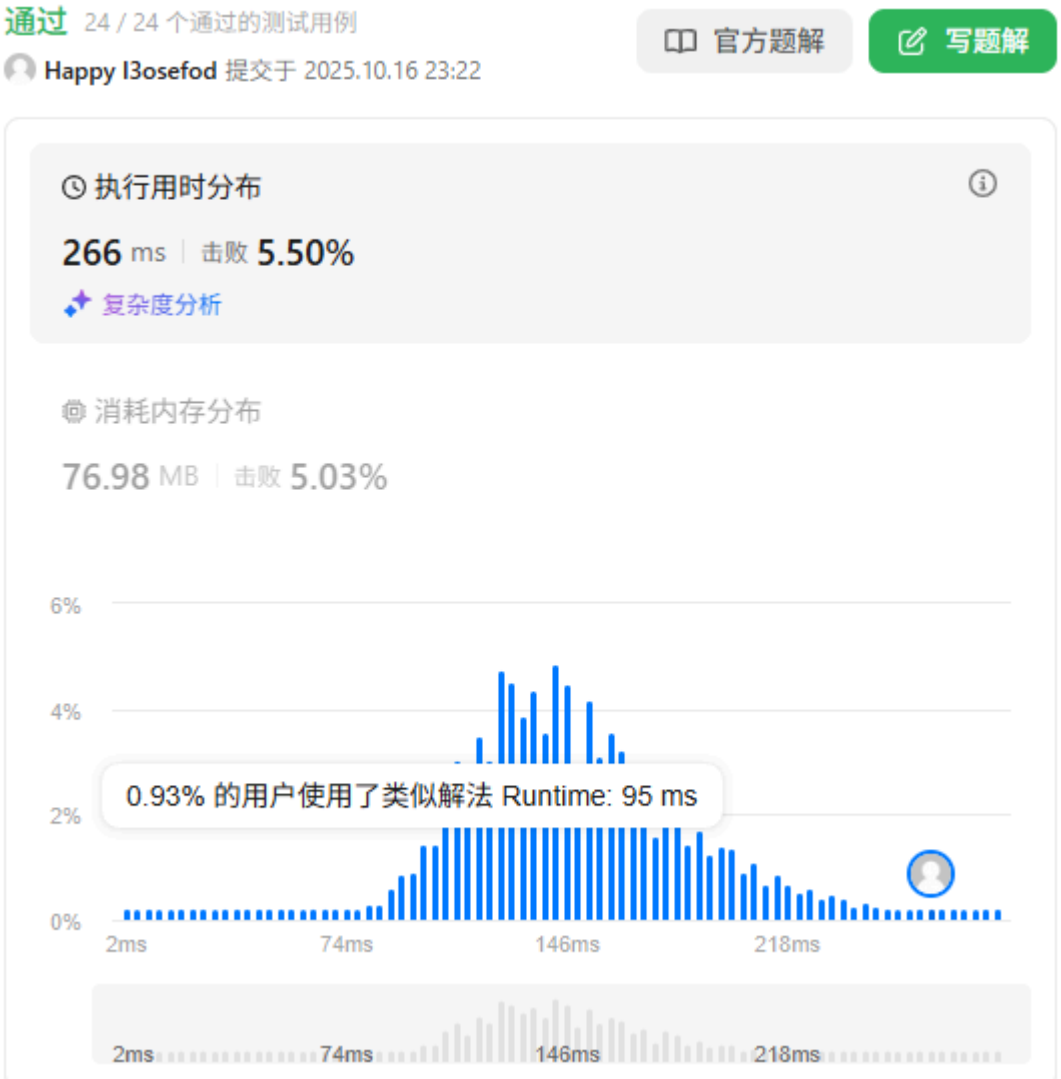
hash table, doubly-linked list, https://leetcode.cn/problems/lru-cache/
思路：这题花了很长的时间。一开始想着自己写hash表， 发现对我还是太难了。后来想到python里面字典也是用hash表实现O(1)的时间复杂度的，于是就用Dict来作为哈希表了。然后在get和put的代码上，用了双向链表来实现最近使用的键的记录，然后也是去数算的讲义里面copy了整份的双向链表的代码。尽管如此，在双向链表的append和del，并更新到字典中时也是碰了很多壁，总体有点困难。
代码：

```
# 双向链表：
class Node:
    def __init__(self, data):
        self.data = data  # 节点数据
        self.next = None  # 指向下一个节点
        self.prev = None  # 指向前一个节点


class DoublyLinkedList:
    def __init__(self):
        self.head = None  # 链表头部
        self.tail = None  # 链表尾部

    # 在链表尾部添加节点
    def append(self, data):
        new_node = Node(data)
        if not self.head:  # 如果链表为空
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
        return new_node

    # 在链表头部添加节点
    def prepend(self, data):
        new_node = Node(data)
        if not self.head:  # 如果链表为空
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
        return new_node

    # 删除链表中的指定节点
```

```
    def delete(self, node):
        temp_node_value = node.data
        if not self.head:  # 链表为空
            return
        if node == self.head:  # 删除头部节点
            self.head = node.next
            if self.head:  # 如果链表非空
                self.head.prev = None
        elif node == self.tail:  # 删除尾部节点
            self.tail = node.prev
            if self.tail:  # 如果链表非空
                self.tail.next = None
        else:  # 删除中间节点
            node.prev.next = node.next
            node.next.prev = node.prev
        node = None  # 删除节点
        return temp_node_value

    # 打印链表中的所有元素，从头到尾
    def print_list(self):
        current = self.head
        while current:
            print(current.data, end=" <-> ")
            current = current.next
        print("None")

    # 打印链表中的所有元素，从尾到头
    def print_reverse(self):
        current = self.tail
        while current:
            print(current.data, end=" <-> ")
            current = current.prev
        print("None")


class LRUCache:
    def __init__(self, capacity: int):
        self.cache = {}
        self.capacity = capacity
        self.full = 0
        self.DoubleLink = DoublyLinkedList()
    def get(self, key: int) -> int:
        if key in self.cache.keys():
            self.DoubleLink.delete(self.cache[key][1])
            temp_node = self.DoubleLink.append(key)
            self.cache[key] = [self.cache[key][0], temp_node]
            return self.cache[key][0]
            #print(self.cache, self.full)
        else:
            return -1
```

```python
    def put(self, key: int, value: int) -> None:
        if key not in self.cache.keys():
            self.full += 1
        temp_node = self.DoubleLink.append(key)
        if key in self.cache.keys():
            self.DoubleLink.delete(self.cache[key][1])
        self.cache[key] = [value, temp_node]
        if self.full > self.capacity:
            self.full -= 1
            deleted_key = self.DoubleLink.delete(self.DoubleLink.head)
            if self.DoubleLink.tail.data != deleted_key:
                del self.cache[deleted_key]
        #self.DoubleLink.print_list()
        #print(self.cache, self.full)
```

代码运行截图 （至少包含有"Accepted"）



2. 学习总结和个人收获

这一期作业的题目对我而言完成起来确实有点吃力，但是也学到了不少：

1. 对栈、链表、字典的性质和应用的认识都更加深刻。比如之前完全不知道链表的访问和删除的时间复杂度只有O(1)、字典是使用Hash表存储的等；
2. 感觉面向对象的编程更加成熟了一些。
3. 认识到数据结构与算法需要去copy一些现成的代码来用（，自己感觉好根本写不出来

3. 额外题目：

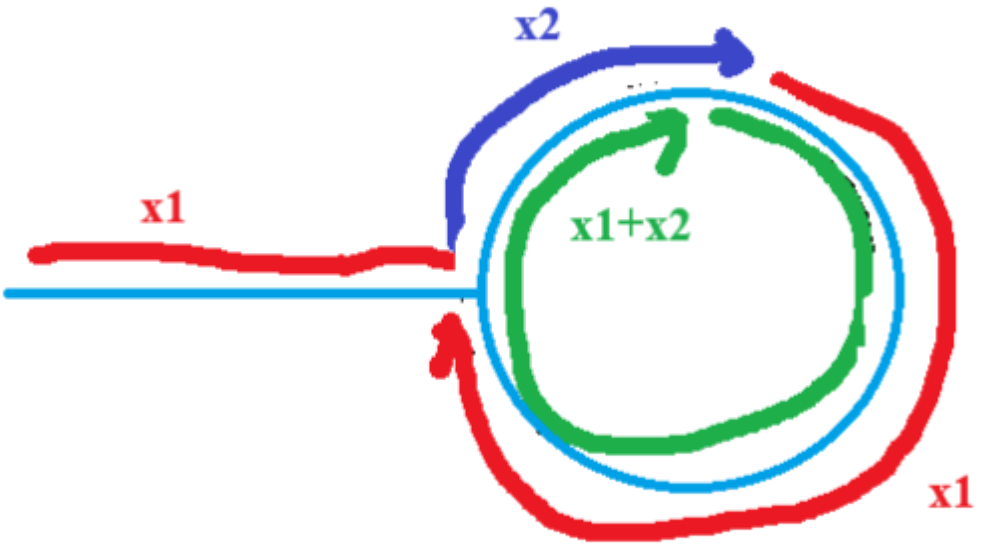
环形链表I

https://leetcode.cn/problems/linked-list-cycle/
思路：快慢指针。由于只要判断True，比较简单。
代码

```
class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        if head is None:
            return False
        fast = head
        slow = head
        while fast.next is not None and fast.next.next is not None:
            fast = fast.next
            if fast == slow:
                return True
            fast = fast.next
            slow = slow.next
            if fast == slow:
                return True
        return False
```

环形链表II

https://leetcode.cn/problems/linked-list-cycle/
思路：由于需要判断入环的index，确实想了一会儿，后来通过画图想到了思路。快指针只需要再继续绕，和新的指针从起点开始转的步数是一样的，二者相等时就是入圈的位置。当然，要注意入圈的位置就是head的情况（囧）



代码

```python
def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:
    if head is None:
        return None
    fast = head
    slow = head
    while fast.next is not None and fast.next.next is not None:
        fast = fast.next.next
        slow = slow.next
        if fast == slow:
            # 将慢指针指定为新指针
            slow = head
            while fast != slow:
                fast = fast.next
                slow = slow.next
            else:
                return fast
    return None
```