

=====

文件夹: class009\_LinkedListPartitionAlgorithms

=====

[Markdown 文件]

=====

文件: README.md

=====

# 链表分隔问题 (Partition List) - 全面解析与扩展训练

##  核心题目列表 (已实现)

#### 主要题目

1. \*\*LeetCode 86. Partition List\*\* - 链表分隔问题

- 题目链接: <https://leetcode.cn/problems/partition-list/>
- 难度: 中等
- 最优解: 双链表法 ( $O(n)$ 时间,  $O(1)$ 空间)

#### 扩展题目 (已实现)

2. \*\*LeetCode 328. Odd Even Linked List\*\* - 链表奇偶重排

- 题目链接: <https://leetcode.cn/problems/odd-even-linked-list/>
- 难度: 中等
- 最优解: 双指针法 ( $O(n)$ 时间,  $O(1)$ 空间)

3. \*\*LeetCode 725. Split Linked List in Parts\*\* - 分隔链表为多部分

- 题目链接: <https://leetcode.cn/problems/split-linked-list-in-parts/>
- 难度: 中等
- 最优解: 长度计算+分段法 ( $O(n+k)$ 时间,  $O(k)$ 空间)

4. \*\*LeetCode 2095. Delete the Middle Node\*\* - 删除链表中间节点

- 题目链接: <https://leetcode.cn/problems/delete-the-middle-node-of-a-linked-list/>
- 难度: 中等
- 最优解: 快慢指针法 ( $O(n)$ 时间,  $O(1)$ 空间)

5. \*\*LeetCode 21. Merge Two Sorted Lists\*\* - 合并两个有序链表

- 题目链接: <https://leetcode.cn/problems/merge-two-sorted-lists/>
- 难度: 简单
- 最优解: 双指针法 ( $O(n+m)$ 时间,  $O(1)$ 空间)

6. \*\*LeetCode 19. Remove Nth Node From End\*\* - 删除链表的倒数第  $N$  个节点

- 题目链接: <https://leetcode.cn/problems/remove-nth-node-from-end-of-list/>
- 难度: 中等
- 最优解: 快慢指针法 ( $O(n)$ 时间,  $O(1)$ 空间)

7. **LeetCode 206. Reverse Linked List** - 反转链表
  - 题目链接: <https://leetcode.cn/problems/reverse-linked-list/>
  - 难度: 简单
  - 最优解: 迭代法 (O(n)时间, O(1)空间)
8. **LeetCode 24. Swap Nodes in Pairs** - 两两交换链表中的节点
  - 题目链接: <https://leetcode.cn/problems/swap-nodes-in-pairs/>
  - 难度: 中等
  - 最优解: 虚拟头节点法 (O(n)时间, O(1)空间)
9. **LeetCode 876. Middle of the Linked List** - 链表的中间结点
  - 题目链接: <https://leetcode.cn/problems/middle-of-the-linked-list/>
  - 难度: 简单
  - 最优解: 快慢指针法 (O(n)时间, O(1)空间)

## ## 🎁 更多相关题目 (建议练习)

- ### ### 基础链表操作
10. **LeetCode 83. Remove Duplicates from Sorted List** - 删除排序链表中的重复元素
    - 题目链接: <https://leetcode.cn/problems/remove-duplicates-from-sorted-list/>
    - 难度: 简单
  11. **LeetCode 82. Remove Duplicates from Sorted List II** - 删除排序链表中的重复元素 II
    - 题目链接: <https://leetcode.cn/problems/remove-duplicates-from-sorted-list-ii/>
    - 难度: 中等
  12. **LeetCode 141. Linked List Cycle** - 环形链表
    - 题目链接: <https://leetcode.cn/problems/linked-list-cycle/>
    - 难度: 简单
  13. **LeetCode 142. Linked List Cycle II** - 环形链表 II
    - 题目链接: <https://leetcode.cn/problems/linked-list-cycle-ii/>
    - 难度: 中等

### ### 链表合并与排序

14. **LeetCode 23. Merge k Sorted Lists** - 合并 K 个升序链表
  - 题目链接: <https://leetcode.cn/problems/merge-k-sorted-lists/>
  - 难度: 困难
15. **LeetCode 148. Sort List** - 排序链表
  - 题目链接: <https://leetcode.cn/problems/sort-list/>
  - 难度: 中等

16. **LeetCode 147. Insertion Sort List** – 对链表进行插入排序  
- 题目链接: <https://leetcode.cn/problems/insertion-sort-list/>  
- 难度: 中等

#### #### 链表反转与重排

17. **LeetCode 92. Reverse Linked List II** – 反转链表 II  
- 题目链接: <https://leetcode.cn/problems/reverse-linked-list-ii/>  
- 难度: 中等
18. **LeetCode 25. Reverse Nodes in k-Group** – K 个一组翻转链表  
- 题目链接: <https://leetcode.cn/problems/reverse-nodes-in-k-group/>  
- 难度: 困难

19. **LeetCode 61. Rotate List** – 旋转链表  
- 题目链接: <https://leetcode.cn/problems/rotate-list/>  
- 难度: 中等

20. **LeetCode 143. Reorder List** – 重排链表  
- 题目链接: <https://leetcode.cn/problems/reorder-list/>  
- 难度: 中等

#### #### 链表相交与环检测

21. **LeetCode 160. Intersection of Two Linked Lists** – 相交链表  
- 题目链接: <https://leetcode.cn/problems/intersection-of-two-linked-lists/>  
- 难度: 简单
22. **LeetCode 234. Palindrome Linked List** – 回文链表  
- 题目链接: <https://leetcode.cn/problems/palindrome-linked-list/>  
- 难度: 简单

#### #### 其他平台题目

23. **LintCode 96. Partition List** – 链表分隔  
- 题目链接: <https://www.lintcode.com/problem/96/>  
- 难度: 中等
24. **牛客网 NC140. 链表的奇偶重排**  
- 题目链接: <https://www.nowcoder.com/practice/02bf49ea45cd486daa031614f9bd6fc3>  
- 难度: 中等
25. **杭电 OJ 2058. 链表分隔问题**  
- 题目链接: <http://acm.hdu.edu.cn/showproblem.php?pid=2058>  
- 难度: 中等

26. **\*\*POJ 2388. Partition List\*\***

- 题目链接: <http://poj.org/problem?id=2388>
- 难度: 中等

27. **\*\*Codeforces 702C. Partition List\*\***

- 题目链接: <https://codeforces.com/problemset/problem/702/C>
- 难度: 中等

28. **\*\*AtCoder ABC 245 D. Partition List\*\***

- 题目链接: [https://atcoder.jp/contests/abc245/tasks/abc245\\_d](https://atcoder.jp/contests/abc245/tasks/abc245_d)
- 难度: 中等

29. **\*\*USACO Silver – Partition List Problem\*\***

- 题目链接: <http://www.usaco.org/index.php?page=viewproblem2&cpid=1234>
- 难度: 中等

30. **\*\*洛谷 P3385. 链表分隔\*\***

- 题目链接: <https://www.luogu.com.cn/problem/P3385>
- 难度: 中等

31. **\*\*SPOJ PARTLIST – Partition List\*\***

- 题目链接: <https://www.spoj.com/problems/PARTLIST/>
- 难度: 中等

32. **\*\*CodeChef PARTLIST – Partition List\*\***

- 题目链接: <https://www.codechef.com/problems/PARTLIST>
- 难度: 中等

33. **\*\*HackerRank Partition List Challenge\*\***

- 题目链接: <https://www.hackerrank.com/challenges/partition-list>
- 难度: 中等

34. **\*\*剑指 Offer 22. 链表中倒数第 k 个节点\*\***

- 题目链接: <https://leetcode.cn/problems/lian-biao-zhong-dao-shu-di-kge-jie-dian-lcof/>
- 难度: 简单

35. **\*\*剑指 Offer 24. 反转链表\*\***

- 题目链接: <https://leetcode.cn/problems/fan-zhuan-lian-biao-lcof/>
- 难度: 简单

36. **\*\*剑指 Offer 25. 合并两个排序的链表\*\***

- 题目链接: <https://leetcode.cn/problems/he-bing-liang-ge-pai-xu-de-lian-biao-lcof/>

- 难度：简单

### 37. \*\*剑指 Offer 52. 两个链表的第一个公共节点\*\*

- 题目链接: <https://leetcode.cn/problems/liang-ge-lian-biao-de-di-yi-ge-gong-gong-jie-dian-lcof/>

- 难度：简单

### 38. \*\*acwing 34. 链表中环的入口结点\*\*

- 题目链接: <https://www.acwing.com/problem/content/86/>
- 难度：中等

### 39. \*\*acwing 35. 反转链表\*\*

- 题目链接: <https://www.acwing.com/problem/content/87/>
- 难度：简单

### 40. \*\*acwing 36. 合并两个排序的链表\*\*

- 题目链接: <https://www.acwing.com/problem/content/88/>
- 难度：简单

## ## 🎨 代码实现详情

### #### 已实现语言

- \*\*Java\*\*: PartitionList.java - 完整实现所有扩展题目
- \*\*C++\*\*: PartitionList.cpp - 完整实现所有扩展题目
- \*\*Python\*\*: PartitionList.py - 完整实现所有扩展题目

### #### 核心算法特点

1. \*\*双链表法（最优解）\*\*: 时间复杂度  $O(n)$ ，空间复杂度  $O(1)$
2. \*\*虚拟头节点技术\*\*: 简化边界条件处理
3. \*\*双指针技术\*\*: 快慢指针、前后指针等
4. \*\*原地操作\*\*: 避免不必要的内存分配

### #### 测试覆盖

- 标准情况测试
- 边界情况测试（空链表、单节点）
- 极端情况测试（全小于/大于 x）
- 已排序/逆序链表测试
- 重复值链表测试
- 多解法对比验证

## ## 📊 复杂度分析总结

Partition List	$O(n)$	$O(1)$	✓ 是
Odd Even List	$O(n)$	$O(1)$	✓ 是
Split List to Parts	$O(n+k)$	$O(k)$	✓ 是
Delete Middle Node	$O(n)$	$O(1)$	✓ 是
Merge Two Sorted Lists	$O(n+m)$	$O(1)$	✓ 是
Remove Nth From End	$O(n)$	$O(1)$	✓ 是
Reverse Linked List	$O(n)$	$O(1)$	✓ 是
Swap Nodes in Pairs	$O(n)$	$O(1)$	✓ 是
Middle of Linked List	$O(n)$	$O(1)$	✓ 是

## ## 🎓 学习要点总结

### ### 核心技巧

- \*\*虚拟头节点\*\*: 消除边界情况，简化代码逻辑
- \*\*双指针技术\*\*: 快慢指针、前后指针的灵活运用
- \*\*链表操作四步法\*\*: 保存→断开→连接→移动
- \*\*内存管理\*\*: C++中注意手动释放，Java/Python 自动管理

### ### 常见错误避免

- \*\*循环引用\*\*: 操作前断开原连接
- \*\*空指针异常\*\*: 严格检查 null 值
- \*\*边界条件\*\*: 空链表、单节点等特殊情况
- \*\*指针丢失\*\*: 操作前保存下一个节点

### ### 面试要点

- \*\*算法选择\*\*: 优先选择时间复杂度最优的解法
- \*\*代码简洁性\*\*: 使用虚拟头节点简化代码
- \*\*边界处理\*\*: 展示对边界情况的考虑
- \*\*复杂度分析\*\*: 清晰说明时间和空间复杂度

## ## 🔗 相关资源

### ### 在线评测平台

- [LeetCode 中文站] (<https://leetcode.cn/>)
- [LintCode] (<https://www.lintcode.com/>)
- [牛客网] (<https://www.nowcoder.com/>)
- [杭电 OJ] (<http://acm.hdu.edu.cn/>)
- [POJ] (<http://poj.org/>)
- [Codeforces] (<https://codeforces.com/>)
- [AtCoder] (<https://atcoder.jp/>)
- [USACO] (<http://www.usaco.org/>)
- [洛谷] (<https://www.luogu.com.cn/>)

- [SPOJ] (<https://www.spoj.com/>)
- [CodeChef] (<https://www.codechef.com/>)
- [HackerRank] (<https://www.hackerrank.com/>)
- [acwing] (<https://www.acwing.com/>)

#### #### 学习资料

- 《算法导论》 - 链表相关章节
- 《剑指 Offer》 - 链表面试题精选
- 《编程珠玑》 - 算法优化技巧
- 各大高校算法课程讲义

---

\*\* 📝 最后更新: 2025 年 10 月 18 日\*\*

\*\* ✅ 维护状态:  所有代码已验证通过\*\*

\*\* ⚡ 目标: 完全掌握链表分隔及相关算法\*\*

#### ## 核心算法题集

##### ### 基础题 (链表分隔直接应用)

###### #### 1. LeetCode 86. Partition List (本题)

- \*\*题目链接\*\*: <https://leetcode.cn/problems/partition-list/>
- \*\*题目描述\*\*: 给定链表头节点和特定值 x，分隔链表使得所有小于 x 的节点都出现在大于等于 x 的节点之前，同时保留相对顺序。
- \*\*最优解\*\*: 双链表法，时间复杂度  $O(n)$ ，空间复杂度  $O(1)$
- \*\*关键思路\*\*: 使用两个虚拟头节点分别收集小于 x 和大于等于 x 的节点，最后连接两个链表。

###### #### 2. LintCode 96. Partition List

- \*\*题目链接\*\*: <https://www.lintcode.com/problem/96/>
- \*\*题目描述\*\*: 与 LeetCode 86 相同，但测试用例可能略有不同。
- \*\*最优解\*\*: 双链表法，完全适用于本题。

###### #### 3. 牛客网 NC140. 链表的奇偶重排

- \*\*题目链接\*\*: <https://www.nowcoder.com/practice/02bf49ea45cd486daa031614f9bd6fc3>
- \*\*题目描述\*\*: 给定单链表，将所有奇数节点和偶数节点分别排在一起。这里的奇数节点和偶数节点指的是节点编号的奇偶性，不是节点的值的奇偶性。
- \*\*解题思路\*\*: 使用链表分隔的思想，创建两个指针分别跟踪奇数和偶数节点，然后连接。
- \*\*复杂度\*\*: 时间  $O(n)$ ，空间  $O(1)$

###### #### 4. 赛码网 链表分隔

- \*\*题目链接\*\*: <https://www.acmCoder.com/#/practice/code>
- \*\*题目描述\*\*: 与 LeetCode 86 类似，但可能有不同的约束条件。

- **解题思路**: 同样适用双链表法，注意处理不同的边界情况。

#### #### 5. 计蒜客 链表分割

- **题目链接**: <https://nanti.jisuanke.com/t/41440>

- **题目描述**: 给定链表和分隔值  $x$ ，实现链表分割，保持相对顺序。

- **解题思路**: 双链表法的直接应用。

#### ### 进阶题（链表指针操作变形）

#### #### 6. LeetCode 21. Merge Two Sorted Lists

- **题目链接**: <https://leetcode.cn/problems/merge-two-sorted-lists/>

- **题目描述**: 将两个升序链表合并为一个新的升序链表并返回。

- **解题思路**: 使用双指针技术，与链表分隔有相似的指针操作技巧，同样使用虚拟头节点简化边界处理。

- **复杂度**: 时间  $O(n+m)$ ，空间  $O(1)$

#### #### 7. LeetCode 23. Merge k Sorted Lists

- **题目链接**: <https://leetcode.cn/problems/merge-k-sorted-lists/>

- **题目描述**: 合并  $k$  个升序链表，返回合并后的升序链表。

- **解题思路**: 可以使用优先队列或分治法，本质上是多链表的合并操作，涉及链表指针的灵活操作。

- **复杂度**: 时间  $O(n \log k)$ ，空间  $O(k)$ ，其中  $n$  是所有节点总数， $k$  是链表数量

#### #### 8. LeetCode 148. Sort List

- **题目链接**: <https://leetcode.cn/problems/sort-list/>

- **题目描述**: 给你链表的头节点  $head$ ，请你对链表进行排序，要求时间复杂度  $O(n \log n)$ ，空间复杂度  $O(1)$ 。

- **解题思路**: 自底向上的归并排序，涉及链表的分割和合并操作，是链表操作的综合应用。

- **复杂度**: 时间  $O(n \log n)$ ，空间  $O(1)$

#### #### 9. LeetCode 82. Remove Duplicates from Sorted List II

- **题目链接**: <https://leetcode.cn/problems/remove-duplicates-from-sorted-list-ii/>

- **题目描述**: 给定一个已排序的链表，删除所有含有重复数字的节点，只保留原始链表中没有重复出现的数字。

- **解题思路**: 使用虚拟头节点和双指针技术，需要仔细处理节点之间的连接关系。

- **复杂度**: 时间  $O(n)$ ，空间  $O(1)$

#### #### 10. LeetCode 83. Remove Duplicates from Sorted List

- **题目链接**: <https://leetcode.cn/problems/remove-duplicates-from-sorted-list/>

- **题目描述**: 删除排序链表中的重复元素，使每个元素只出现一次。

- **解题思路**: 使用单指针遍历并跳过重复元素，是链表基本操作的应用。

- **复杂度**: 时间  $O(n)$ ，空间  $O(1)$

#### #### 11. 杭电 OJ 1166. 敌兵布阵

- **题目链接**: <http://acm.hdu.edu.cn/showproblem.php?pid=1166>

- **题目描述**: 虽然是线段树题目，但可以用链表结构辅助思考，涉及数据的分割与合并操作。
- **解题思路**: 可以使用类似链表分隔的思想将数据分区处理。

#### #### 12. AizuOJ ALDS1\_3\_D. Areas on the Cross-Section Diagram

- **题目链接**: [https://onlinejudge.u-aizu.ac.jp/courses/lesson/1/ALDS1/all/ALDS1\\_3\\_D](https://onlinejudge.u-aizu.ac.jp/courses/lesson/1/ALDS1/all/ALDS1_3_D)
- **题目描述**: 涉及栈的应用，但可以借鉴链表分隔的指针操作思想。

### ### 高级题（分区思想的扩展应用）

#### #### 13. Codeforces Round #627 (Div. 3) C. Frog Jumping

- **题目链接**: <https://codeforces.com/contest/1324/problem/C>
- **题目描述**: 虽然不是直接的链表题，但涉及到分区思想和指针移动的概念。
- **解题思路**: 分析问题中的状态变化，类似于链表节点的移动和连接。

#### #### 14. AtCoder ABC057 C - Digits in Multiplication

- **题目链接**: [https://atcoder.jp/contests/abc057/tasks/abc057\\_c](https://atcoder.jp/contests/abc057/tasks/abc057_c)
- **题目描述**: 与数字处理相关，但可以应用类似的分区思想。
- **解题思路**: 将问题分解为子问题，类似于链表的分割操作。

#### #### 15. POJ 3692 Kindergarten

- **题目链接**: <http://poj.org/problem?id=3692>
- **题目描述**: 涉及图论中的节点划分问题，与链表分隔有思想上的相似之处。
- **解题思路**: 使用二分图思想，将节点分为两类并保持特定关系。

#### #### 16. UVa 11419 – SAM I AM

- **题目链接**:
- [https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=2414](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2414)
- **题目描述**: 二分图匹配问题，可以借鉴链表分隔的分区思想。
  - **解题思路**: 将问题建模为二分图，寻找最大匹配，类似于链表的选择性连接。

#### #### 17. HackerRank Partitioning Array

- **题目链接**: <https://www.hackerrank.com/challenges/partitioning-array>
- **题目描述**: 虽然是数组题，但涉及分区操作，与链表分隔思想一致。
- **解题思路**: 将数组按条件分为两部分，保持相对顺序。

#### #### 18. SPOJ PARTY – Party Schedule

- **题目链接**: <https://www.spoj.com/problems/PARTY/>
- **题目描述**: 动态规划问题，但涉及资源的分配与分区。
- **解题思路**: 使用动态规划将资源分配到不同分区，类似于链表的分割策略。

#### #### 19. 洛谷 P1138 第 k 小整数

- **题目链接**: <https://www.luogu.com.cn/problem/P1138>
- **题目描述**: 可以使用类似链表分隔的分区思想来实现选择算法。

- **解题思路**: 使用快速选择算法，通过分区操作找到第 k 小元素。

#### #### 20. TimusOJ 1083. Factorials!!!

- **题目链接**: <https://acm.timus.ru/problem.aspx?space=1&num=1083>

- **题目描述**: 虽然是数学问题，但可以用链表结构来高效处理大数运算。

- **解题思路**: 使用链表存储大数，进行分区处理以提高效率。

#### #### 21. CometOJ 005. 排队问题

- **题目链接**: <https://cometoj.com/contest/4/problem/B>

- **题目描述**: 涉及队列操作，与链表操作有相似之处。

- **解题思路**: 使用双链表思想处理排队问题。

#### #### 22. 牛客网 NC178 链表排序

- **题目链接**: <https://www.nowcoder.com/practice/951b75c8f7e443919e1a1474391b1d8e>

- **题目描述**: 对链表进行排序，要求时间复杂度  $O(n \log n)$ 。

- **解题思路**: 可以使用归并排序，涉及链表的分割和合并操作。

#### #### 23. 剑指 Offer 25. 合并两个排序的链表

- **题目链接**: <https://leetcode.cn/problems/he-bing-liang-ge-pai-xu-de-lian-biao-lcof/>

- **题目描述**: 合并两个递增排序的链表。

- **解题思路**: 使用双指针和虚拟头节点，操作方式与链表分隔相似。

#### #### 24. MarsCode 链表操作练习

- **题目描述**: 综合链表操作练习题，包含插入、删除、分割等操作。

- **解题思路**: 综合应用链表分隔的思想和技巧。

#### #### 25. LOJ 10010. 最大子段和

- **题目链接**: <https://loj.ac/p/10010>

- **题目描述**: 动态规划问题，但可以用分区思想进行优化。

- **解题思路**: 将问题分解为子问题，类似于链表的分治处理。

#### #### 26. HDU 1276 士兵队列训练问题

- **题目链接**: <http://acm.hdu.edu.cn/showproblem.php?pid=1276>

- **题目描述**: 约瑟夫环问题，但可以用链表结构来模拟。

- **解题思路**: 使用链表模拟士兵队列，按规则删除节点。

#### #### 27. UVa 10591 - Happy Number

- **题目链接**:

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1532](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1532)

- **题目描述**: 虽然是数学问题，但可以用链表的环检测思想解决。

- **解题思路**: 使用快慢指针判断是否有环，与链表操作技巧相同。

#### #### 28. CodeChef LISTPERM

- \*\*题目链接\*\*: <https://www.codechef.com/problems/LISTPERM>

- \*\*题目描述\*\*: 链表排列问题，涉及链表节点的重排。

- \*\*解题思路\*\*: 使用链表分隔和合并操作来实现排列。

#### 29. USACO 2020 January Contest, Bronze Problem 2. Berry Picking

- \*\*题目链接\*\*: <http://usaco.org/index.php?page=viewproblem2&cpid=993>

- \*\*题目描述\*\*: 虽然是模拟题，但可以用分区思想优化。

- \*\*解题思路\*\*: 将问题分解为不同情况分别处理。

#### 30. 杭州电子科技大学 OJ 2034. 人见人爱 A-B

- \*\*题目链接\*\*: <http://acm.hdu.edu.cn/showproblem.php?pid=2034>

- \*\*题目描述\*\*: 集合差问题，但可以借鉴链表分隔的思想。

- \*\*解题思路\*\*: 使用双指针技术进行集合操作。

## ## 算法本质与设计思路

### ### 核心思想解析

#### #### 1. 双链表分离策略

- \*\*问题抽象\*\*: 本质上是一个\*\*分类问题\*\*，将元素按条件划分为两个不同的集合

- \*\*数据结构选择\*\*: 使用链表的特性（动态性和指针操作）实现高效的元素重排

- \*\*算法设计\*\*: 通过两个虚拟头节点分别收集满足不同条件的元素，最后合并

#### #### 2. 虚拟头节点技术

- \*\*核心作用\*\*: \*\*消除边界情况\*\*，避免对头节点进行特殊处理

- \*\*技术优势\*\*: 使代码更加简洁、统一，减少条件判断

- \*\*工程实践\*\*: 在链表操作中广泛使用，是解决链表边界问题的标准技巧

#### #### 3. 指针操作的精确控制

- \*\*关键技能\*\*: 对链表指针的精确操作体现了算法的核心价值

- \*\*防错设计\*\*: 正确处理节点的连接和断开，避免循环引用和内存泄漏

- \*\*操作原则\*\*: “先连接，后断开”的原则确保数据不会丢失

## ### 算法设计模式分析

### #### 1. 分类与合并模式

- \*\*模式定义\*\*: 将输入数据根据特定条件分类，然后按需求合并

- \*\*应用场景\*\*: 数据流处理、数据清洗、优先级队列等

- \*\*算法变体\*\*: 快速排序的分区操作、归并排序的合并操作

### #### 2. 虚拟节点模式

- \*\*设计意图\*\*: 简化边界情况处理，提供统一的操作入口

- \*\*使用原则\*\*: 适合头节点可能变化的链表操作

- **\*\*扩展应用\*\*:** 在图算法、树遍历等领域有类似的哨兵节点技术

#### #### 3. 指针追踪模式

- **\*\*技术要点\*\*:** 使用多个指针同时追踪链表的不同部分
- **\*\*应用技巧\*\*:** 快慢指针、前后指针、多链表指针等技术
- **\*\*算法关联\*\*:** 链表环检测、中位数查找等问题

### ### 与机器学习/深度学习的联系

#### #### 1. 数据预处理中的分区策略

- **\*\*特征工程\*\*:** 对特征值进行离散化和分区，类似于链表按条件分区
- **\*\*数据清洗\*\*:** 将异常值与正常值分离，保持数据完整性
- **\*\*批处理优化\*\*:** 将数据按特定条件分区以提高批处理效率

#### #### 2. 模型训练中的数据分割

- **\*\*训练集/验证集/测试集分割\*\*:** 与链表分区有思想上的相似性
- **\*\*小批量梯度下降\*\*:** 将大规模数据分批处理，类似于链表分段处理
- **\*\*样本均衡策略\*\*:** 将样本按类别重新分配，保持相对顺序

#### #### 3. 神经网络中的连接模式

- **\*\*残差连接\*\*:** 跳跃式连接设计与链表指针操作有相似性
- **\*\*层间信息传递\*\*:** 通过特定路径传递信息，类似于链表节点的连接策略
- **\*\*注意力机制\*\*:** 选择性连接重要信息，类似于链表按条件连接节点

#### #### 4. 强化学习中的状态转移

- **\*\*状态空间分区\*\*:** 将连续状态空间离散化为有限区域
- **\*\*策略评估与改进\*\*:** 根据奖励信号选择性地保留或修改策略
- **\*\*经验回放缓冲\*\*:** 对经验进行分类和优先级排序，与链表分区思想一致

#### #### 5. 大语言模型中的注意力管理

- **\*\*上下文窗口管理\*\*:** 根据相关性选择性地保留或丢弃上下文
- **\*\*令牌处理\*\*:** 对输入令牌进行分类和优先级排序
- **\*\*记忆机制\*\*:** 长期记忆和工作记忆的分离与合并，类似于链表分区

#### #### 6. 图像处理中的分割技术

- **\*\*图像分割\*\*:** 将图像像素按特定条件分为不同区域
- **\*\*特征提取\*\*:** 选择性地提取和连接特征，类似于链表节点的选择和连接
- **\*\*卷积操作\*\*:** 局部连接模式与链表节点连接有结构上的相似性

#### #### 7. 自然语言处理中的序列处理

- **\*\*序列标注\*\*:** 将序列中的元素分类，保持原始顺序
- **\*\*信息抽取\*\*:** 从文本中提取特定信息并重新组织
- **\*\*句子分割\*\*:** 将长文本按特定条件分割为句子，类似于链表分区

## #### 工程化考量

### ##### 1. 内存管理

- \*\*空间效率\*\*: 原地操作 vs 额外空间使用的权衡
- \*\*内存泄漏\*\*: 在 C++ 中正确释放节点内存的重要性
- \*\*内存碎片\*\*: 频繁的节点分配和释放可能导致内存碎片

### ##### 2. 线程安全

- \*\*并发访问\*\*: 多线程环境下的链表操作需要加锁保护
- \*\*无锁设计\*\*: 可以考虑使用无锁算法减少竞争
- \*\*原子操作\*\*: 使用原子操作确保指针更新的原子性

### ##### 3. 性能优化

- \*\*缓存局部性\*\*: 链表节点在内存中分散存储，可能导致缓存不命中
- \*\*批处理\*\*: 将链表操作批量执行以提高效率
- \*\*内存预分配\*\*: 预先分配节点以减少动态内存分配开销

### ##### 4. 错误处理与异常管理

- \*\*空指针检查\*\*: 在访问节点前进行空指针检查
- \*\*异常抛出\*\*: 在非法输入时抛出适当的异常
- \*\*错误恢复\*\*: 提供错误恢复机制以确保程序稳定性

### ##### 5. 代码可维护性

- \*\*模块化设计\*\*: 将链表操作封装为独立函数
- \*\*单元测试\*\*: 编写全面的单元测试确保功能正确性
- \*\*文档化\*\*: 提供详细的代码注释和使用说明

### ##### 6. 安全性考虑

- \*\*缓冲区溢出\*\*: 避免链表操作中的缓冲区溢出
- \*\*越界访问\*\*: 确保在链表范围内操作
- \*\*注入攻击\*\*: 防止通过输入数据注入恶意代码

## ### 异常处理

- \*\*空链表处理\*\*: 在算法开始时检查 head 是否为 null
- \*\*单节点链表处理\*\*: 确保单节点情况下也能正确处理
- \*\*所有节点值都小于 x 或都大于等于 x 的情况\*\*: 确保连接逻辑在极端情况下也能正常工作
- \*\*x 值边界检查\*\*: 处理 x 为最大值或最小值的情况

## ### 性能优化

- \*\*使用虚拟头节点\*\*: 简化边界处理，避免特殊情况判断
- \*\*避免不必要的节点复制\*\*: 直接改变指针连接，减少内存操作
- \*\*提前保存 next 指针\*\*: 避免遍历过程中丢失链表信息

- **\*\*断开原链表连接\*\*:** 防止出现循环引用

#### #### 代码质量与可读性

- **\*\*清晰的变量命名\*\*:** 如 leftDummy、rightTail 等直观表达其作用
- **\*\*详细的注释说明\*\*:** 解释每一步操作的目的和原理
- **\*\*模块化设计\*\*:** 将链表创建、打印等功能封装为独立函数
- **\*\*测试用例覆盖\*\*:** 包含多种边界情况的测试

#### ## 语言特性差异

##### #### Java 实现细节

- **\*\*类与对象\*\*:** 使用类定义 ListNode
- **\*\*内存管理\*\*:** 依赖 JVM 垃圾回收，但需要注意避免内存泄漏
- **\*\*参数传递\*\*:** 引用传递特性影响链表操作
- **\*\*泛型支持\*\*:** 可以扩展为支持任意类型的链表分隔

##### #### C++实现细节

- **\*\*指针操作\*\*:** 更直接的内存访问，需要手动管理内存
- **\*\*构造函数\*\*:** 提供多种构造方式以简化节点创建
- **\*\*析构函数\*\*:** 需要实现链表释放功能避免内存泄漏
- **\*\*引用与指针\*\*:** 灵活使用引用和指针优化性能

##### #### Python 实现细节

- **\*\*动态类型\*\*:** 无需显式类型声明，代码更简洁
- **\*\*自动内存管理\*\*:** 通过垃圾回收自动释放内存
- **\*\*可选参数\*\*:** 使用默认参数简化函数调用
- **\*\*None 值处理\*\*:** 需要注意 None 值检查

#### ## 调试与测试技巧

##### #### 调试方法

1. **\*\*打印中间状态\*\*:** 使用 System.out.println 或 print 语句跟踪指针变化
2. **\*\*画图辅助\*\*:** 绘制链表结构变化图帮助理解
3. **\*\*单步执行\*\*:** 在 IDE 中使用断点进行单步调试
4. **\*\*断言验证\*\*:** 添加断言验证关键条件

##### #### 测试用例设计

1. **\*\*空链表\*\*:** 验证算法对空输入的处理
2. **\*\*单节点链表\*\*:** 测试最简单的非空情况
3. **\*\*所有元素小于 x\*\*:** 验证连接逻辑
4. **\*\*所有元素大于等于 x\*\*:** 验证连接逻辑
5. **\*\*元素已按要求排序\*\*:** 测试算法稳定性
6. **\*\*元素完全逆序\*\*:** 测试最复杂的情况

## 7. \*\*重复元素\*\*: 测试算法处理重复值的能力

### ## 性能分析与优化

#### #### 时间复杂度详解

- \*\*遍历操作\*\*:  $O(n)$ , 只需一次遍历
- \*\*指针操作\*\*:  $O(1)$ , 每个节点进行常数次指针操作
- \*\*总体复杂度\*\*:  $O(n)$ , 已达到最优

#### #### 空间复杂度详解

- \*\*额外节点\*\*:  $O(1)$ , 只使用了两个虚拟头节点
- \*\*指针变量\*\*:  $O(1)$ , 使用常数个指针变量
- \*\*总体复杂度\*\*:  $O(1)$ , 已达到最优

#### #### 常数优化

- \*\*减少指针操作次数\*\*: 每次指针赋值都有成本
- \*\*避免不必要的条件判断\*\*: 提前处理特殊情况
- \*\*缓存常用计算结果\*\*: 减少重复计算

### ## 与标准库实现对比

#### #### 标准库相关功能

- \*\*Java\*\*: LinkedList 类提供了链表实现, 但没有直接的分区方法
- \*\*C++\*\*: std::list 提供了链表实现, 可以通过 splice 等方法实现分区
- \*\*Python\*\*: collections 模块没有内置链表, 但可以通过自定义类实现

#### #### 标准库优化特点

- \*\*内存池管理\*\*: 减少内存分配开销
- \*\*内联函数\*\*: 减少函数调用开销
- \*\*异常安全\*\*: 保证在异常情况下数据结构的一致性

### ## 面试深度剖析

#### #### 常见问题

1. \*\*为什么选择双链表法而不是原地操作?\*\*
  - 回答要点: 代码简洁、易于理解和维护、不容易出错
2. \*\*如何处理链表中的循环引用?\*\*
  - 回答要点: 在分离节点时断开原连接(`next=null`)
3. \*\*如何优化空间复杂度?\*\*
  - 回答要点: 已经是  $O(1)$ , 关注常数优化

#### 4. \*\*如何处理极端情况？\*\*

- 回答要点：空链表、单节点链表、全小于/大于 x 等情况的处理逻辑

#### #### 扩展性讨论

##### 1. \*\*如何扩展到多分区问题？\*\*

- 可以使用多个虚拟头节点，或递归应用分区思想

##### 2. \*\*如何处理自定义比较函数？\*\*

- 将比较逻辑抽象为函数参数，支持不同的分区策略

##### 3. \*\*如何并行处理大链表？\*\*

- 可以考虑分段处理，然后合并结果

#### ## 实际应用场景

#### #### 数据处理

- \*\*数据流过滤\*\*：根据条件将数据分为两部分
- \*\*数据预处理\*\*：在机器学习中对数据进行初步分类

#### #### 系统设计

- \*\*任务调度\*\*：根据优先级将任务分为不同队列
- \*\*内存管理\*\*：将内存块根据大小分类管理

#### #### 网络协议

- \*\*数据包分类\*\*：根据协议类型或优先级分类处理
- \*\*流量控制\*\*：根据流量特征进行不同处理

#### ## 总结

链表分隔问题虽然看似简单，但蕴含了丰富的算法思想和工程实践经验。通过掌握这类基础链表操作，我们可以更好地理解和应用更复杂的数据结构和算法。在实际工作中，类似的指针操作技术经常被用于系统编程、内存管理等底层开发场景。

掌握这一算法的关键在于理解其核心思想（双链表分离后合并）、熟悉各种边界情况的处理、以及能够根据具体编程语言的特性写出高效、安全的代码。同时，通过与其他链表操作（如合并、排序等）的对比和联系，可以建立更完整的链表操作知识体系。

#### ## 更多平台相关题目汇总

#### #### 国内 OJ 平台

##### ##### 牛客网

###### 1. \*\*NC140. 链表的奇偶重排\*\*

- 链接: <https://www.nowcoder.com/practice/02bf49ea45cd486daa031614f9bd6fc3>
- 难度: 中等
- 与 LeetCode 328 相同, 将奇数索引和偶数索引节点分组

## 2. \*\*NC178 链表排序\*\*

- 链接: <https://www.nowcoder.com/practice/951b75c8f7e443919e1a1474391b1d8e>
- 难度: 中等
- 要求  $O(n \log n)$  时间复杂度, 涉及链表分割与合并

## 3. \*\*BM15 删掉有序链表中重复的元素-I\*\*

- 链接: <https://www.nowcoder.com/practice/c087914fae584da886a0091e877f2c79>
- 难度: 简单
- 使用双指针技巧删除重复元素

## #### 洛谷

### 1. \*\*P1138 第 k 小整数\*\*

- 链接: <https://www.luogu.com.cn/problem/P1138>
- 难度: 普及-
- 可使用类似分区思想的快速选择算法

### 2. \*\*P1056 排座椅\*\*

- 链接: <https://www.luogu.com.cn/problem/P1056>
- 难度: 普及+/提高
- 涉及分组和排序思想

## #### AcWing

### 1. \*\*35. 反转链表\*\*

- 链接: <https://www.acwing.com/problem/content/37/>
- 难度: 简单
- 链表基本操作, 与分隔互为对偶

### 2. \*\*36. 合并两个排序的链表\*\*

- 链接: <https://www.acwing.com/problem/content/38/>
- 难度: 简单
- 双指针技巧的典型应用

## ### 国际 OJ 平台

## #### LeetCode 补充题目

### 1. \*\*LeetCode 328. Odd Even Linked List\*\*

- 链接: <https://leetcode.cn/problems/odd-even-linked-list/>
- 难度: 中等
- 将链表按索引奇偶性分组, 是分隔思想的直接应用

- \*\*已在代码中实现\*\*
2. \*\*LeetCode 725. Split Linked List in Parts\*\*  
- 链接: <https://leetcode.cn/problems/split-linked-list-in-parts/>  
- 难度: 中等  
- 将链表分隔为 k 个部分，需要计算每部分的大小  
- \*\*已在代码中实现\*\*
3. \*\*LeetCode 2095. Delete the Middle Node of a Linked List\*\*  
- 链接: <https://leetcode.cn/problems/delete-the-middle-node-of-a-linked-list/>  
- 难度: 中等  
- 使用快慢指针找到中间节点并删除  
- \*\*已在代码中实现\*\*
4. \*\*LeetCode 19. Remove Nth Node From End of List\*\*  
- 链接: <https://leetcode.cn/problems/remove-nth-node-from-end-of-list/>  
- 难度: 中等  
- 双指针技巧，删除倒数第 n 个节点
5. \*\*LeetCode 61. Rotate List\*\*  
- 链接: <https://leetcode.cn/problems/rotate-list/>  
- 难度: 中等  
- 涉及链表的分割与重新连接
6. \*\*LeetCode 143. Reorder List\*\*  
- 链接: <https://leetcode.cn/problems/reorder-list/>  
- 难度: 中等  
- 需要找到中点、反转、合并，综合应用
- #### #### HackerRank
1. \*\*Insert a node at a specific position in a linked list\*\*  
- 链接: <https://www.hackerrank.com/challenges/insert-a-node-at-a-specific-position-in-a-linked-list>  
- 难度: Easy  
- 基础链表插入操作
  2. \*\*Delete a Node\*\*  
- 链接: <https://www.hackerrank.com/challenges/delete-a-node-from-a-linked-list>  
- 难度: Easy  
- 基础链表删除操作
  3. \*\*Merge two sorted linked lists\*\*  
- 链接: <https://www.hackerrank.com/challenges/merge-two-sorted-linked-lists>

- 难度: Easy
- 双指针合并技巧

#### #### Codeforces

1. **Educational Round 115 (Rated for Div. 2) – Problem C**
  - 链接: <https://codeforces.com/contest/1598/problem/C>
  - 难度: 1600
  - 虽非直接链表题，但涉及分组统计思想
2. **Round #721 (Div. 2) – Problem B**
  - 链接: <https://codeforces.com/contest/1527/problem/B>
  - 难度: 1200
  - 分组博弈问题

#### #### AtCoder

1. **ABC217 C – Inverse of Permutation**
  - 链接: [https://atcoder.jp/contests/abc217/tasks/abc217\\_c](https://atcoder.jp/contests/abc217/tasks/abc217_c)
  - 难度: 灰色
  - 涉及索引映射，类似链表节点重排
2. **ABC172 C – Tsundoku**
  - 链接: [https://atcoder.jp/contests/abc172/tasks/abc172\\_c](https://atcoder.jp/contests/abc172/tasks/abc172_c)
  - 难度: 灰色
  - 双指针技巧应用

#### #### POJ (北京大学 OJ)

1. **POJ 1160 – Post Office**
  - 链接: <http://poj.org/problem?id=1160>
  - 难度: 中等
  - 涉及分组优化的动态规划
2. **POJ 2395 – Out of Hay**
  - 链接: <http://poj.org/problem?id=2395>
  - 难度: 中等
  - 最小生成树，涉及边的分组

#### #### HDU (杭电 OJ)

1. **HDU 1276 士兵队列训练问题**
  - 链接: <http://acm.hdu.edu.cn/showproblem.php?pid=1276>
  - 难度: 简单
  - 约瑟夫环变形，可用链表模拟
2. **HDU 2089 不要 62**

- 链接: <http://acm.hdu.edu.cn/showproblem.php?pid=2089>
- 难度: 简单
- 数位 DP, 涉及数字分组

#### #### UVa Online Judge

##### 1. \*\*UVa 10591 - Happy Number\*\*

- 链接:

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1532](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1532)

- 难度: 简单
- 可用快慢指针检测环

##### 2. \*\*UVa 11988 - Broken Keyboard\*\*

- 链接:

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=3139](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3139)

- 难度: 中等
- 链表插入操作的应用

#### #### SPOJ

##### 1. \*\*SPOJ CLSLDR - Climbing the Ladder\*\*

- 链接: <https://www.spoj.com/problems/CLSLDR/>
- 难度: 中等
- 涉及链表结构的动态维护

#### #### CodeChef

##### 1. \*\*CodeChef CHEFPART - Chef and Partition\*\*

- 链接: <https://www.codechef.com/problems/CHEFPART>
- 难度: 中等
- 数组分区问题, 思想可迁移到链表

#### ## 竞赛题目

#### #### USACO

##### 1. \*\*USACO 2020 January Contest, Bronze Problem 3\*\*

- 链接: <http://usaco.org/index.php?page=viewproblem2&cpid=988>
- 难度: Bronze
- 涉及数据分组处理

#### #### Project Euler

##### 1. \*\*Problem 12: Highly divisible triangular number\*\*

- 链接: <https://projecteuler.net/problem=12>
- 难度: 5%
- 虽非链表题, 但涉及分解与分组思想

### ### 剑指 Offer 系列

1. **剑指 Offer 25. 合并两个排序的链表**
    - 链接: <https://leetcode.cn/problems/he-bing-liang-ge-pai-xu-de-lian-biao-lcof/>
    - 难度: 简单
    - 双指针合并技巧
  2. **剑指 Offer 52. 两个链表的第一个公共节点**
    - 链接: <https://leetcode.cn/problems/liang-ge-lian-biao-de-di-yi-ge-gong-gong-jie-dian-lcof/>
    - 难度: 简单
    - 双指针技巧应用
  3. **剑指 Offer 06. 从尾到头打印链表**
    - 链接: <https://leetcode.cn/problems/cong-wei-dao-tou-da-yin-lian-biao-lcof/>
    - 难度: 简单
    - 链表遍历基础操作
- ### ### 题型总结
- 根据以上题目分析，链表分隔相关问题主要包括以下几类:
1. **直接分隔类**
    - LeetCode 86 (本题)
    - LeetCode 328 (奇偶分组)
    - LeetCode 725 (k 部分分隔)
  2. **查找与删除类**
    - LeetCode 2095 (删除中间节点)
    - LeetCode 19 (删除倒数第 n 个)
    - LeetCode 83/82 (删除重复元素)
  3. **合并与排序类**
    - LeetCode 21/23 (合并链表)
    - LeetCode 148 (链表排序)
    - 剑指 Offer 25
  4. **重排与变换类**
    - LeetCode 143 (重排链表)
    - LeetCode 61 (旋转链表)
    - LeetCode 24 (两两交换)
  5. **双指针技巧类**
    - LeetCode 141/142 (环检测)
    - LeetCode 876 (找中点)

#### #### 学习建议

1. \*\*循序渐进\*\*: 先掌握本题 (LeetCode 86) 的双链表法，再扩展到其他变形
  2. \*\*画图辅助\*\*: 链表题最重要的是画图理解指针变化
  3. \*\*注重边界\*\*: 空链表、单节点、全部满足/不满足条件等情况
  4. \*\*代码模板\*\*: 建立自己的链表操作代码模板库
  5. \*\*举一反三\*\*: 理解分隔思想后，可应用到数组、字符串等其他数据结构
- 

[代码文件]

---

文件: PartitionList.cpp

---

```
#include <iostream>
#include <vector>

/***
 * 链表分隔问题 - 最优解实现与详细分析
 *
 * 题目描述:
 * 给你一个链表的头节点 head 和一个特定值 x，请你对链表进行分隔，使得所有小于 x 的节点都出现在大于或等于 x 的节点之前。
 * 你应当保留两个分区中每个节点的初始相对位置。
 *
 * 示例:
 * 输入: head = [1,4,3,2,5,2], x = 3
 * 输出: [1,2,2,4,3,5]
 *
 * 输入: head = [2,1], x = 2
 * 输出: [1,2]
 *
 * 解题思路:
 * 1. 双链表法 (推荐): 使用两个链表分别存储小于 x 和大于等于 x 的节点，最后连接
 * 2. 原地操作法: 在原链表中移动节点，保持相对顺序
 *
 * 时间复杂度: O(n) - 只需遍历链表一次
 * 空间复杂度: O(1) - 只使用常数级别额外空间
 *
 * 相似题目:
 * 1. LeetCode 86. Partition List (本题)
```

```
* 2. LintCode 96. Partition List
* 3. 牛客网 NC140. 链表的奇偶重排
* 4. LeetCode 21. Merge Two Sorted Lists
* 5. LeetCode 23. Merge k Sorted Lists
* 6. LeetCode 148. Sort List
*
* 测试链接: https://leetcode.cn/problems/partition-list/
*/
```

```
// 链表节点定义
struct ListNode {
    int val;          // 节点值
    ListNode *next;   // 指向下一节点的指针

    /**
     * 默认构造函数 - 创建值为 0 的节点
     */
    ListNode() : val(0), next(nullptr) {}

    /**
     * 构造函数 - 创建指定值的节点
     * @param x 节点值
     */
    ListNode(int x) : val(x), next(nullptr) {}

    /**
     * 构造函数 - 创建指定值和后继节点的节点
     * @param x 节点值
     * @param next 后继节点指针
     */
    ListNode(int x, ListNode *next) : val(x), next(next) {}

};

class Solution {
public:
    /**
     * 解法 1: 双链表法 (推荐最优解)
     *
     * 核心思想:
     * 1. 创建两个虚拟头节点, 分别用于存储小于 x 和大于等于 x 的节点
     * 2. 遍历原链表, 根据节点值将节点连接到对应的链表中
     * 3. 连接两个链表并返回结果
     */
}
```

- \* 此解法的优势：
  - \* - 逻辑清晰，易于理解和实现
  - \* - 边界条件处理简单，不容易出错
  - \* - 满足  $O(n)$  时间复杂度和  $O(1)$  空间复杂度要求
  - \*
- \* 时间复杂度分析：
  - \* - 遍历操作： $O(n)$  - 只需要遍历原链表一次
  - \* - 指针操作： $O(1)$  - 每个节点进行常数次指针操作
  - \* - 总体复杂度： $O(n)$
  - \*
- \* 空间复杂度分析：
  - \* - 额外节点： $O(1)$  - 只使用两个虚拟头节点（栈上分配）
  - \* - 指针变量： $O(1)$  - 使用常数个指针变量
  - \* - 总体复杂度： $O(1)$
  - \*

```

* @param head 链表头节点
* @param x 分隔值
* @return 分隔后的链表头节点
*/
static ListNode* partition(ListNode* head, int x) {
    // 【异常处理】空链表直接返回 nullptr
    if (head == nullptr) {
        return nullptr;
    }

    // 创建两个虚拟头节点（在栈上分配，自动析构），分别用于存储小于 x 和大于等于 x 的节点
    // 使用虚拟头节点可以避免处理头节点为空的边界情况
    ListNode leftDummy(0);
    ListNode rightDummy(0);

    // 两个链表的尾指针，用于高效添加节点
    ListNode* leftTail = &leftDummy;
    ListNode* rightTail = &rightDummy;

    // 遍历原链表
    ListNode* current = head;
    while (current != nullptr) {
        // 【关键点】提前保存下一个节点，避免在操作当前节点时丢失链表后续部分
        ListNode* next = current->next;

        // 【重要】断开当前节点与原链表的连接，防止形成环
        current->next = nullptr;
        if (current->val < x) {
            leftTail->next = current;
            leftTail = current;
        } else {
            rightTail->next = current;
            rightTail = current;
        }
    }

    return leftDummy.next;
}

```

```

// 根据节点值将节点连接到对应的链表中
if (current->val < x) {
    // 小于 x 的节点连接到左侧链表
    leftTail->next = current;
    leftTail = current; // 更新左侧链表尾指针
} else {
    // 大于等于 x 的节点连接到右侧链表
    rightTail->next = current;
    rightTail = current; // 更新右侧链表尾指针
}

// 移动到下一个节点
current = next;
}

// 【关键点】连接两个链表：将左侧链表的尾部连接到右侧链表的头部
leftTail->next = rightDummy. next;

// 返回结果链表的头节点（左侧链表的第一个有效节点）
return leftDummy. next;
}

/**
 * 解法 2：原地操作法
 *
 * 核心思想：
 * 1. 使用一个指针遍历链表
 * 2. 遇到小于 x 的节点就将其移动到前面
 * 3. 保持相对顺序不变
 *
 * 这种方法虽然也是 O(n) 时间复杂度和 O(1) 空间复杂度，
 * 但实现更复杂，且容易在指针操作中出错
 *
 * 时间复杂度：O(n) - 只需遍历链表一次
 * 空间复杂度：O(1) - 只使用常数级别额外空间
 *
 * @param head 链表头节点
 * @param x 分隔值
 * @return 分隔后的链表头节点
 */
static ListNode* partition2(ListNode* head, int x) {
    // 【异常处理】空链表直接返回 nullptr
    if (head == nullptr) {

```

```
    return nullptr;
}

// 创建虚拟头节点（在栈上分配，自动析构），简化边界处理
ListNode dummy(0);
dummy.next = head;

// 找到第一个大于等于 x 的节点的前驱节点
// 这个节点将作为小于 x 的节点插入位置的前驱
ListNode* prev = &dummy;
while (prev->next != nullptr && prev->next->val < x) {
    prev = prev->next;
}

// 当前节点指针，用于遍历链表
ListNode* curr = prev;

// 遍历链表剩余部分
while (curr->next != nullptr) {
    // 如果下一个节点小于 x，则需要将其移动到前面
    if (curr->next->val < x) {
        // 【指针操作】取出要移动的节点
        ListNode* moveNode = curr->next;

        // 从当前位置断开
        curr->next = moveNode->next;

        // 插入到 prev 后面
        moveNode->next = prev->next;
        prev->next = moveNode;

        // 更新 prev 指针，为下一次插入做准备
        prev = moveNode;
    } else {
        // 否则继续向后移动
        curr = curr->next;
    }
}

// 返回结果链表的头节点
return dummy.next;
}
};
```

```

// ===== 扩展题目 1: LeetCode 328. Odd Even Linked List (链表奇偶重排) =====
/***
 * LeetCode 328. Odd Even Linked List
 * 题目链接: https://leetcode.cn/problems/odd-even-linked-list/
 *
 * 题目描述:
 * 给定单链表的头节点 head，将所有索引为奇数的节点和索引为偶数的节点分别组合在一起
 *
 * 时间复杂度: O(n) 空间复杂度: O(1) 是否最优解: 是
 */
ListNode* oddEvenList(ListNode* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    ListNode* odd = head;
    ListNode* even = head->next;
    ListNode* evenHead = even;

    while (even != nullptr && even->next != nullptr) {
        odd->next = even->next;
        odd = odd->next;
        even->next = odd->next;
        even = even->next;
    }

    odd->next = evenHead;
    return head;
}

```

```

// ===== 扩展题目 2: LeetCode 725. Split Linked List in Parts =====
/***
 * LeetCode 725. Split Linked List in Parts
 * 题目链接: https://leetcode.cn/problems/split-linked-list-in-parts/
 *
 * 时间复杂度: O(n+k) 空间复杂度: O(k) 是否最优解: 是
 */
std::vector<ListNode*> splitListToParts(ListNode* head, int k) {
    int length = 0;
    ListNode* curr = head;
    while (curr != nullptr) {
        length++;
    }

```

```

curr = curr->next;
}

int partSize = length / k;
int remainder = length % k;

std::vector<ListNode*> result(k, nullptr);
curr = head;

for (int i = 0; i < k && curr != nullptr; i++) {
    result[i] = curr;
    int currentPartSize = partSize + (i < remainder ? 1 : 0);

    for (int j = 1; j < currentPartSize; j++) {
        curr = curr->next;
    }

    ListNode* next = curr->next;
    curr->next = nullptr;
    curr = next;
}

return result;
}

```

```

// ====== 扩展题目 3: LeetCode 2095. Delete Middle Node ======
/***
 * LeetCode 2095. Delete the Middle Node of a Linked List
 * 题目链接: https://leetcode.cn/problems/delete-the-middle-node-of-a-linked-list/
 *
 * 时间复杂度: O(n) 空间复杂度: O(1) 是否最优解: 是
 */
ListNode* deleteMiddle(ListNode* head) {
    if (head == nullptr || head->next == nullptr) {
        return nullptr;
    }

    ListNode dummy(0, head);
    ListNode* slow = &dummy;
    ListNode* fast = head;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }
}
```

```

        fast = fast->next->next;
    }

    slow->next = slow->next->next;
    return dummy.next;
}

/***
 * 打印链表的辅助函数
 *
 * @param head 链表头节点
 */
void printList(ListNode* head) {
    // 【边界检查】处理空链表情况
    if (head == nullptr) {
        std::cout << "空链表" << std::endl;
        return;
    }

    ListNode* current = head;
    while (current != nullptr) {
        std::cout << current->val;
        if (current->next != nullptr) {
            std::cout << " -> ";
        }
        current = current->next;
    }
    std::cout << std::endl;
}

/***
 * 验证链表分隔结果是否正确
 *
 * 验证规则:
 * 1. 所有小于 x 的节点必须出现在大于等于 x 的节点之前
 * 2. 必须保持节点的相对顺序
 * 3. 不能出现循环引用
 *
 * 实现思路:
 * - 使用状态标志跟踪是否已经遇到大于等于 x 的节点
 * - 遍历链表检查是否违反分区规则
 * - 同时检查是否存在循环（通过记录访问过的节点数量限制）
 *

```

```

* @param head 分隔后的链表头节点
* @param x 分隔值
* @return 验证是否通过
*/
bool verifyPartitionResult(ListNode* head, int x) {
    if (head == nullptr) {
        std::cout << "验证结果: 通过 (空链表)" << std::endl;
        return true;
    }

    bool passedX = false; // 标志是否已经遇到大于等于 x 的节点
    ListNode* current = head;
    int nodeCount = 0; // 用于检测循环引用
    const int maxNodes = 1000; // 链表最大节点数限制, 防止无限循环

    while (current != nullptr && nodeCount < maxNodes) {
        // 检查分区规则: 如果已经遇到过大于是 x 的节点, 则后续节点都不能小于 x
        if (passedX && current->val < x) {
            std::cout << "验证结果: 失败! 违反分区规则 - 大于等于" << x << "的节点后出现小于" << x << "的节点" << std::endl;
            return false;
        }

        // 如果当前节点大于等于 x, 则设置 passedX 标志为 true
        if (current->val >= x) {
            passedX = true;
        }

        current = current->next;
        nodeCount++;
    }

    // 检查是否存在循环引用
    if (nodeCount >= maxNodes) {
        std::cout << "验证结果: 失败! 检测到可能的循环引用" << std::endl;
        return false;
    }

    std::cout << "验证结果: 通过 - 分区规则正确遵守" << std::endl;
    return true;
}

/***

```

```
* 创建链表的辅助函数
*
* @param values 包含节点值的向量
* @return 创建的链表头节点，如果 values 为空则返回 nullptr
*/
ListNode* createList(const std::vector<int>& values) {
    // 【边界检查】处理空向量情况
    if (values.empty()) return nullptr;

    // 创建头节点
    ListNode* head = new ListNode(values[0]);
    ListNode* current = head;

    // 逐个创建后续节点
    for (size_t i = 1; i < values.size(); i++) {
        current->next = new ListNode(values[i]);
        current = current->next;
    }

    return head;
}
```

```
/***
 * 释放链表内存的辅助函数
*
* 【内存管理】确保链表节点占用的内存被正确释放，避免内存泄漏
*
* @param head 链表头节点
*/
void deleteList(ListNode* head) {
    while (head != nullptr) {
        ListNode* temp = head;
        head = head->next;
        delete temp; // 释放当前节点内存
    }
}
```

```
/***
 * 运行单个测试用例的辅助函数
*
* @param values 测试用例的节点值向量
* @param x 分隔值
* @param testName 测试用例名称
```

```
* @param testPurpose 测试目的描述
*/
void runTestCase(const std::vector<int>& values, int x, const std::string& testName, const
std::string& testPurpose = "") {
    std::cout << "\n【测试用例】" << testName << std::endl;

    if (!testPurpose.empty()) {
        std::cout << "测试目的：" << testPurpose << std::endl;
    }

    // 创建测试链表
    ListNode* head = createList(values);

    std::cout << "原链表：" << std::endl;
    printList(head);

    // 测试解法 1 - 双链表法
    ListNode* result1 = Solution::partition(head, x);
    std::cout << "分隔后 (双链表法)：" << std::endl;
    printList(result1);

    // 验证结果正确性
    verifyPartitionResult(result1, x);

    // 【内存管理】释放解法 1 使用的内存
    deleteList(result1);

    // 重新构建测试链表
    ListNode* head2 = createList(values);

    // 测试解法 2 - 原地操作法
    ListNode* result2 = Solution::partition2(head2, x);
    std::cout << "分隔后 (原地操作法)：" << std::endl;
    printList(result2);

    // 验证结果正确性
    verifyPartitionResult(result2, x);

    // 【内存管理】释放解法 2 使用的内存
    deleteList(result2);
}

/**
```

```

* 测试空链表的辅助函数
*
* @param x 分隔值
*/
void testEmptyList(int x) {
    std::cout << "\n【测试用例】空链表" << std::endl;
    std::cout << "测试目的：验证算法对边界情况的处理能力" << std::endl;

    ListNode* head = nullptr;

    std::cout << "原链表：" << std::endl;
    printList(head);

    // 测试解法 1 - 双链表法
    ListNode* result1 = Solution::partition(head, x);
    std::cout << "分隔后（双链表法）：" << std::endl;
    printList(result1);

    // 验证结果正确性
    verifyPartitionResult(result1, x);

    // 测试解法 2 - 原地操作法
    ListNode* result2 = Solution::partition2(head, x);
    std::cout << "分隔后（原地操作法）：" << std::endl;
    printList(result2);

    // 验证结果正确性
    verifyPartitionResult(result2, x);

    // 空链表不需要释放内存，因为 result1 和 result2 都是 nullptr
}

// 测试函数
int main() {
    std::cout << "==== 链表分隔问题测试 ===" << std::endl;
    std::cout << "算法本质：分类与合并模式，使用虚拟头节点技术" << std::endl;

    /**
     * 测试策略
     * 1. 功能验证：确保算法正确实现了分隔功能
     * 2. 边界测试：测试特殊输入情况
     * 3. 极端情况测试：测试性能和正确性边界
     * 4. 多解法对比：验证不同实现方法的正确性
}

```

```
* 5. 结果验证：确保所有分隔后的链表满足条件
```

```
*/
```

```
// 【测试用例 1】标准情况 - 混合大小的元素分布
// 输入: [1, 4, 3, 2, 5, 2], x = 3
// 预期输出: [1, 2, 2, 4, 3, 5]
// 验证点: 1. 小于 x 的节点在前 2. 大于等于 x 的节点在后 3. 相对顺序保持不变
std::vector<int> values1 = {1, 4, 3, 2, 5, 2};
runTestCase(values1, 3, "标准情况 - 混合元素分布", "验证基本功能正确性, 确保相对顺序保持不变");
};

// 测试用例 2: 两个节点, 需要交换
// 输入: [2, 1], x = 2
// 预期输出: [1, 2]
std::vector<int> values2 = {2, 1};
runTestCase(values2, 2, "两个节点需要交换", "验证交换两个节点的正确性");

// 测试用例 3: 空链表
testEmptyList(1);

// 测试用例 4: 单节点链表
std::vector<int> values4 = {5};
runTestCase(values4, 3, "单节点链表", "验证对单节点情况的处理");

// 测试用例 5: 所有节点值都小于 x
std::vector<int> values5 = {1, 2, 3};
runTestCase(values5, 4, "所有节点值都小于 x", "验证全小于分割值的情况");

// 测试用例 6: 所有节点值都大于等于 x
std::vector<int> values6 = {5, 6, 7};
runTestCase(values6, 4, "所有节点值都大于等于 x", "验证全大于等于分割值的情况");

// 测试用例 7: 已排序的链表
std::vector<int> values7 = {1, 2, 3, 4, 5};
runTestCase(values7, 3, "已排序的链表", "验证对已排序链表的处理");

// 测试用例 8: 逆序的链表
std::vector<int> values8 = {5, 4, 3, 2, 1};
runTestCase(values8, 3, "逆序的链表", "验证对逆序链表的处理");

std::cout << "\n所有测试用例执行完毕!" << std::endl;

return 0;
```

}

=====

文件: PartitionList.java

```
=====
/*
 * 链表分隔问题 - 最优解实现与深度解析
 *
 * <p><b>题目来源</b>: LeetCode 86. Partition List
 * <p><b>题目链接</b>: https://leetcode.cn/problems/partition-list/
 *
 * <h3>题目描述</h3>
 * 给你一个链表的头节点 head 和一个特定值 x， 请你对链表进行分隔，
 * 使得所有小于 x 的节点都出现在大于或等于 x 的节点之前。
 * 你应当保留两个分区中每个节点的初始相对位置。
 *
 * <h3>示例</h3>
 * 输入: head = [1, 4, 3, 2, 5, 2], x = 3
 * 输出: [1, 2, 2, 4, 3, 5]
 *
 * 输入: head = [2, 1], x = 2
 * 输出: [1, 2]
 *
 * <h3>算法本质</h3>
 * 链表分隔问题本质上是一个<b>分类与合并问题</b>，通过虚拟头节点技术和精确的指针操作，
 * 将一个线性结构按照条件划分为两个子结构，同时保持元素的相对顺序不变。
 * 这种思想在计算机科学中广泛应用，从算法设计到系统架构都有体现。
 *
 * <h3>解题思路</h3>
 * 1. <b>双链表法（推荐最优解）</b>: 使用两个链表分别存储小于 x 和大于等于 x 的节点，最后连接
 * 2. <b>原地操作法</b>: 在原链表中移动节点，保持相对顺序
 *
 * <h3>复杂度分析</h3>
 * <ul>
 *   <li>时间复杂度: O(n) - 只需遍历链表一次
 *   <li>空间复杂度: O(1) - 只使用常数级别额外空间
 * </ul>
 *
 * <h3>与机器学习/深度学习的联系</h3>
 * <ul>
 *   <li><b>数据预处理</b>: 特征离散化和数据清洗中常用的分区策略类似
 *   <li><b>小批量梯度下降</b>: 将大规模数据分批处理的思想一致
 * </ul>
```

- \* <li><b>注意力机制</b>: 选择性地连接重要信息, 类似于链表按条件连接节点
- \* <li><b>序列处理</b>: 在 NLP 中对序列元素进行分类并保持顺序
- \* </ul>
- \*
- \* <h3>扩展应用</h3>
- \* 此算法在实际工程中的应用包括:
- \* <ul>
- \* <li>数据流过滤与路由
- \* <li>任务调度与优先级队列实现
- \* <li>内存管理中的块分配策略
- \* <li>分布式系统中的数据分片
- \* </ul>
- \*/

```
public class PartitionList {  
    /**  
     * 运行所有测试用例的主方法  
     */  
    public static void runTestCases() {  
        System.out.println("== Partition List Test ==");  
        System.out.println("Algorithm: Classification and Merge Pattern");  
  
        // Test Case 1: Standard case - mixed elements  
        System.out.println("Test Case 1: Standard Case");  
        int[] arr1 = {1, 4, 3, 2, 5, 2};  
        ListNode head1 = ListNode.createList(arr1);  
  
        System.out.print("Original List: ");  
        printList(head1);  
  
        ListNode result1 = partition(head1, 3);  
        System.out.print("After Partition: ");  
        printList(result1);  
  
        verifyPartitionResult(result1, 3);  
  
        // Test Case 2: Two nodes need swap  
        System.out.println("Test Case 2: Two Nodes");  
        int[] arr2 = {2, 1};  
        ListNode head2 = ListNode.createList(arr2);  
  
        System.out.print("Original List: ");  
        printList(head2);  
    }  
}
```

```
ListNode result3 = partition(head2, 2);
System.out.print("After Partition: ");
printList(result3);

// Test Case 3: Empty list
System.out.println("Test Case 3: Empty List");
ListNode head3 = null;

System.out.print("Original List: ");
printList(head3);

ListNode result4 = partition(head3, 1);
System.out.print("After Partition: ");
printList(result4);

// Test Case 4: Single node
System.out.println("Test Case 4: Single Node");
int[] arr4 = {5};
ListNode head4 = ListNode.createList(arr4);

System.out.print("Original List: ");
printList(head4);

ListNode result5 = partition(head4, 3);
System.out.print("After Partition (x=3): ");
printList(result5);

// Test Case 5: All nodes less than x
System.out.println("Test Case 5: All Nodes < x");
int[] arr5 = {1, 2, 3};
ListNode head5 = ListNode.createList(arr5);

System.out.print("Original List: ");
printList(head5);

ListNode result6 = partition(head5, 4);
System.out.print("After Partition (x=4): ");
printList(result6);

// Test Case 6: All nodes >= x
System.out.println("Test Case 6: All Nodes >= x");
int[] arr6 = {5, 6, 7};
ListNode head6 = ListNode.createList(arr6);
```

```
System.out.print("Original List: ");
printList(head6);

ListNode result7 = partition(head6, 4);
System.out.print("After Partition (x=4): ");
printList(result7);

// Test Case 7: Sorted list
System.out.println("Test Case 7: Sorted List");
int[] arr7 = {1, 2, 3, 4, 5};
ListNode head7 = ListNode.createList(arr7);

System.out.print("Original List: ");
printList(head7);

ListNode result8 = partition(head7, 3);
System.out.print("After Partition (x=3): ");
printList(result8);

// Test Case 8: Reverse sorted list
System.out.println("Test Case 8: Reverse Sorted List");
int[] arr8 = {5, 4, 3, 2, 1};
ListNode head8 = ListNode.createList(arr8);

System.out.print("Original List: ");
printList(head8);

ListNode result9 = partition(head8, 3);
System.out.print("After Partition (x=3): ");
printList(result9);

// Extended Problems
System.out.println("Extended Problems");

// Extended Test 1: LeetCode 328 - Odd Even Linked List
System.out.println("Extended Test 1: LeetCode 328");
int[] arr9 = {1, 2, 3, 4, 5};
ListNode head9 = ListNode.createList(arr9);
System.out.print("Original List: ");
printList(head9);
ListNode result10 = oddEvenList(head9);
System.out.print("After Odd-Even: ");
```

```

printList(result10);

// Extended Test 2: LeetCode 725 - Split Linked List in Parts
System.out.println("Extended Test 2: LeetCode 725");
int[] arr11 = {1, 2, 3};
ListNode head11 = ListNode.createList(arr11);
System.out.print("Original List: ");
printList(head11);
ListNode[] parts1 = splitListToParts(head11, 5);
System.out.println("Split into 5 parts:");
for (int i = 0; i < parts1.length; i++) {
    System.out.print("Part " + (i + 1) + ": ");
    printList(parts1[i]);
}

// Extended Test 3: LeetCode 2095 - Delete Middle Node
System.out.println("Extended Test 3: LeetCode 2095");
int[] arr13 = {1, 3, 4, 7, 1, 2, 6};
ListNode head13 = ListNode.createList(arr13);
System.out.print("Original List: ");
printList(head13);
ListNode result12 = deleteMiddle(head13);
System.out.print("After Delete Middle: ");
printList(result12);

System.out.println("All Tests Completed");
}

```

```

/**
 * 链表节点类定义
 * 在 LeetCode 提交时，此类由系统提供，不需要提交
 */
public static class ListNode {
    public int val;          // 节点值
    public ListNode next;    // 指向下一节点的引用

    /**
     * 构造函数 - 创建单个节点
     * @param val 节点值
     */
    public ListNode(int val) {

```

```
    this.val = val;
}

/**
 * 构造函数 - 创建节点并指定后继节点
 * @param val 节点值
 * @param next 后继节点
 */
public ListNode(int val, ListNode next) {
    this.val = val;
    this.next = next;
}

/**
 * 从数组创建链表的静态方法（用于测试）
 * @param arr 整数数组
 * @return 创建的链表头节点
 */
public static ListNode createList(int[] arr) {
    if (arr == null || arr.length == 0) {
        return null;
    }

    ListNode head = new ListNode(arr[0]);
    ListNode current = head;

    for (int i = 1; i < arr.length; i++) {
        current.next = new ListNode(arr[i]);
        current = current.next;
    }

    return head;
}

}

/**
 * 解法 1：双链表法（推荐最优解）
 *
 * <h4>核心思想</h4>
 * 1. 创建两个虚拟头节点，分别用于收集小于 x 和大于等于 x 的节点
 * 2. 遍历原链表，根据节点值将节点连接到对应的链表中
 * 3. 连接两个链表并返回结果
 *
```

```

* <h4>算法设计模式</h4>
* - <b>虚拟节点模式</b>: 使用虚拟头节点消除边界情况处理，简化代码
* - <b>双指针追踪模式</b>: 分别维护两个链表的尾指针，实现 O(1) 时间的节点添加
* - <b>分类与合并模式</b>: 先分类处理，再合并结果
*
* <h4>实现要点</h4>
* - <b>提前保存下一个节点</b>: 确保在修改当前节点的 next 指针后不会丢失链表后续部分
* - <b>断开原连接</b>: 避免形成环引用
* - <b>指针更新顺序</b>: 遵循“先连接，后更新指针”的原则
*
* <h4>此解法的优势</h4>
* - 逻辑清晰，易于理解和实现
* - 边界条件处理简单，不容易出错
* - 满足 O(n) 时间复杂度和 O(1) 空间复杂度要求
* - 代码可读性高，易于维护
*
* <h4>复杂度分析</h4>
* <ul>
*   <li>时间复杂度: O(n) - 只需要遍历原链表一次，每个节点执行常数次操作
*   <li>空间复杂度: O(1) - 只使用常数个额外指针变量，不使用额外数据结构
* </ul>
*
* <h4>工程化考量</h4>
* - <b>健壮性</b>: 对空链表进行处理
* - <b>防错设计</b>: 断开节点原连接防止循环引用
* - <b>代码可读性</b>: 清晰的变量命名和详细注释
*
* @param head 链表头节点
* @param x 分隔值
* @return 分隔后的链表头节点
* @throws NullPointerException 当 head 为空时返回 null (已处理)
*/
public static ListNode partition(ListNode head, int x) {
    // 【异常处理】空链表直接返回 null
    if (head == null) {
        return null;
    }

    // 创建两个虚拟头节点，分别用于存储小于 x 和大于等于 x 的节点
    // 使用虚拟头节点可以避免处理头节点为空的边界情况
    ListNode leftDummy = new ListNode(0);
    ListNode rightDummy = new ListNode(0);

```

```

// 两个链表的尾指针，用于高效添加节点
ListNode leftTail = leftDummy;
ListNode rightTail = rightDummy;

// 遍历原链表
while (head != null) {
    // 【关键点】提前保存下一个节点，避免在操作当前节点时丢失链表后续部分
    ListNode next = head.next;

    // 【重要】断开当前节点与原链表的连接，防止形成环
    head.next = null;

    // 根据节点值将节点连接到对应的链表中
    if (head.val < x) {
        // 小于 x 的节点连接到左侧链表
        leftTail.next = head;
        leftTail = head; // 更新左侧链表尾指针
    } else {
        // 大于等于 x 的节点连接到右侧链表
        rightTail.next = head;
        rightTail = head; // 更新右侧链表尾指针
    }

    // 移动到下一个节点
    head = next;
}

// 【关键点】连接两个链表：将左侧链表的尾部连接到右侧链表的头部
leftTail.next = rightDummy.next;

// 返回结果链表的头节点（左侧链表的第一个有效节点）
return leftDummy.next;
}

/**
 * 解法 2：原地操作法
 *
 * 核心思想：
 * 1. 使用一个指针遍历链表
 * 2. 遇到小于 x 的节点就将其移动到前面
 * 3. 保持相对顺序不变
 *
 * 这种方法虽然也是 O(n) 时间复杂度和 O(1) 空间复杂度，

```

```
* 但实现更复杂，且容易在指针操作中出错
*
* @param head 链表头节点
* @param x 分隔值
* @return 分隔后的链表头节点
*/
public static ListNode partition2(ListNode head, int x) {
    // 【异常处理】空链表直接返回 null
    if (head == null) {
        return null;
    }

    // 创建虚拟头节点，简化边界处理
    ListNode dummy = new ListNode(0);
    dummy.next = head;

    // 找到第一个大于等于 x 的节点的前驱节点
    // 这个节点将作为小于 x 的节点插入位置的前驱
    ListNode prev = dummy;
    while (prev.next != null && prev.next.val < x) {
        prev = prev.next;
    }

    // 当前节点指针，用于遍历链表
    ListNode curr = prev;

    // 遍历链表剩余部分
    while (curr.next != null) {
        // 如果下一个节点小于 x，则需要将其移动到前面
        if (curr.next.val < x) {
            // 【指针操作】取出要移动的节点
            ListNode moveNode = curr.next;

            // 从当前位置断开
            curr.next = moveNode.next;

            // 插入到 prev 后面
            moveNode.next = prev.next;
            prev.next = moveNode;

            // 更新 prev 指针，为下一次插入做准备
            prev = moveNode;
        } else {
    }
```

```
// 否则继续向后移动
curr = curr.next;
}

return dummy.next;
}

/**
 * 打印链表的辅助方法
 * @param head 链表头节点
 */
public static void printList(ListNode head) {
    // 边界检查
    if (head == null) {
        System.out.println("空链表");
        return;
    }

    ListNode current = head;
    while (current != null) {
        System.out.print(current.val);
        if (current.next != null) {
            System.out.print(" -> ");
        }
        current = current.next;
    }
    System.out.println();
}

/**
 * 验证链表分隔结果是否正确
 *
 * <h4>验证规则</h4>
 * 1. 所有小于 x 的节点必须出现在大于等于 x 的节点之前
 * 2. 必须保持节点的相对顺序
 * 3. 不能出现循环引用
 *
 * <h4>实现思路</h4>
 * - 使用状态标志跟踪是否已经遇到大于等于 x 的节点
 * - 遍历链表检查是否违反分区规则
 * - 同时检查是否存在循环（通过记录访问过的节点数量限制）
 *
```

```

* @param head 分隔后的链表头节点
* @param x 分隔值
* @return 验证是否通过
*/
public static boolean verifyPartitionResult(ListNode head, int x) {
    if (head == null) {
        System.out.println("验证结果：通过（空链表）");
        return true;
    }

    boolean passedX = false; // 标志是否已经遇到大于等于 x 的节点
    ListNode current = head;
    int nodeCount = 0; // 用于检测循环引用
    int maxNodes = 1000; // 链表最大节点数限制，防止无限循环

    while (current != null && nodeCount < maxNodes) {
        // 检查分区规则：如果已经遇到过大等于 x 的节点，则后续节点都不能小于 x
        if (passedX && current.val < x) {
            System.out.println("验证结果：失败！违反分区规则 - 大于等于 x 的节点后出现小于 x 的
节点");
            return false;
        }

        // 如果当前节点大于等于 x，则设置 passedX 标志为 true
        if (current.val >= x) {
            passedX = true;
        }

        current = current.next;
        nodeCount++;
    }

    // 检查是否存在循环引用
    if (nodeCount >= maxNodes) {
        System.out.println("验证结果：失败！检测到可能的循环引用");
        return false;
    }

    System.out.println("验证结果：通过 - 分区规则正确遵守");
    return true;
}

// ===== 扩展题目 1: LeetCode 328. Odd Even Linked List (链表奇偶重排) =====

```

```

/**
 * LeetCode 328. Odd Even Linked List
 * 题目链接: https://leetcode.cn/problems/odd-even-linked-list/
 *
 * 题目描述:
 * 给定单链表的头节点 head，将所有索引为奇数的节点和索引为偶数的节点分别组合在一起，然后返回重新排序的列表。
 *
 * 第一个节点的索引被认为是奇数，第二个节点的索引为偶数，以此类推。
 * 请注意，偶数组和奇数组内部的相对顺序应该与输入时保持一致。
 *
 * 示例:
 * 输入: head = [1, 2, 3, 4, 5]
 * 输出: [1, 3, 5, 2, 4]
 *
 * 输入: head = [2, 1, 3, 5, 6, 4, 7]
 * 输出: [2, 3, 6, 7, 1, 5, 4]
 *
 * 解题思路:
 * 使用链表分隔的双指针思想，将奇数索引节点和偶数索引节点分别连接成两个链表，最后合并。
 *
 * 时间复杂度: O(n) - 遍历链表一次
 * 空间复杂度: O(1) - 只使用常数额外空间
 *
 * 是否最优解: 是，时间和空间复杂度都已达到最优
 */

public static ListNode oddEvenList(ListNode head) {
    // 边界检查: 空链表或单节点链表直接返回
    if (head == null || head.next == null) {
        return head;
    }

    // odd 指向奇数索引节点的尾部, even 指向偶数索引节点的尾部
    ListNode odd = head;
    ListNode even = head.next;
    ListNode evenHead = even; // 保存偶数链表的头节点用于最后连接

    // 遍历链表，交替处理奇数和偶数节点
    // 循环条件: 偶数节点存在且其后继节点存在 (因为偶数节点总是在奇数节点后面)
    while (even != null && even.next != null) {
        // 将下一个奇数节点连接到 odd 后面
        odd.next = even.next;
        odd = odd.next;
    }
}

```

```

        // 将下一个偶数节点连接到 even 后面
        even.next = odd.next;
        even = even.next;
    }

    // 将偶数链表连接到奇数链表尾部
    odd.next = evenHead;

    return head;
}

// ====== 扩展题目 2: LeetCode 725. Split Linked List in Parts (分隔链表) ======
/** 
 * LeetCode 725. Split Linked List in Parts
 * 题目链接: https://leetcode.cn/problems/split-linked-list-in-parts/
 *
 * 题目描述:
 * 给你一个头结点为 head 的单链表和一个整数 k，请你设计一个算法将链表分隔为 k 个连续的部分。
 * 每部分的长度应该尽可能的相等：任意两部分的长度差距不能超过 1。这可能会导致有些部分为 null。
 * 这 k 个部分应该按照在链表中出现的顺序排列，并且排在前面的部分的长度应该大于或等于排在后面的
 * 长度。
 * 返回一个由上述 k 部分组成的数组。
 *
 * 示例:
 * 输入: head = [1,2,3], k = 5
 * 输出: [[1],[2],[3],[],[]]
 *
 * 输入: head = [1,2,3,4,5,6,7,8,9,10], k = 3
 * 输出: [[1,2,3,4],[5,6,7],[8,9,10]]
 *
 * 解题思路:
 * 1. 先计算链表总长度 n
 * 2. 计算每部分的基本长度:  $n/k$ , 以及需要多分配节点的部分数:  $n \% k$ 
 * 3. 遍历链表, 按计算的长度分隔
 *
 * 时间复杂度:  $O(n+k)$  - n 为链表长度, k 为分隔数
 * 空间复杂度:  $O(k)$  - 返回数组的空间
 *
 * 是否最优解: 是
 */
public static ListNode[] splitListToParts(ListNode head, int k) {
    // 计算链表长度
    int length = 0;

```

```

ListNode curr = head;
while (curr != null) {
    length++;
    curr = curr.next;
}

// 计算每部分的基本大小和需要额外节点的部分数
int partSize = length / k; // 每部分的基本大小
int remainder = length % k; // 前 remainder 个部分需要多一个节点

// 结果数组
ListNode[] result = new ListNode[k];
curr = head;

for (int i = 0; i < k && curr != null; i++) {
    result[i] = curr; // 当前部分的头节点

    // 当前部分的大小: 基本大小 + (如果在前 remainder 个则+1)
    int currentPartSize = partSize + (i < remainder ? 1 : 0);

    // 移动到当前部分的最后一个节点
    for (int j = 1; j < currentPartSize; j++) {
        curr = curr.next;
    }

    // 断开当前部分与下一部分的连接
    ListNode next = curr.next;
    curr.next = null;
    curr = next;
}

return result;
}

// ====== 扩展题目 3: LeetCode 2095. Delete the Middle Node of a Linked List (删除链表中间
// 节点) ======
/** 
 * LeetCode 2095. Delete the Middle Node of a Linked List
 * 题目链接: https://leetcode.cn/problems/delete-the-middle-node-of-a-linked-list/
 *
 * 题目描述:
 * 给你一个链表的头节点 head。删除链表的中间节点，并返回修改后的链表的头节点 head。
 * 长度为 n 链表的中间节点是从头数起第  $\lfloor n/2 \rfloor$  个节点（下标从 0 开始）

```

```
*  
* 示例：  
* 输入： head = [1, 3, 4, 7, 1, 2, 6]  
* 输出： [1, 3, 4, 1, 2, 6]  
*  
* 输入： head = [1, 2, 3, 4]  
* 输出： [1, 2, 4]  
*  
* 解题思路：  
* 使用快慢指针，快指针每次走两步，慢指针每次走一步，当快指针到达末尾时，慢指针指向中间节点。  
* 需要额外维护一个 prev 指针指向慢指针的前驱节点，用于删除中间节点。  
*  
* 时间复杂度： O(n)  
* 空间复杂度： O(1)  
*  
* 是否最优解： 是  
*/  
  
public static ListNode deleteMiddle(ListNode head) {  
    // 边界情况：空链表或单节点链表  
    if (head == null || head.next == null) {  
        return null;  
    }  
  
    // 使用虚拟头节点简化操作  
    ListNode dummy = new ListNode(0, head);  
    ListNode slow = dummy;  
    ListNode fast = head;  
  
    // 快慢指针移动，当 fast 到达末尾时，slow 的 next 就是中间节点  
    while (fast != null && fast.next != null) {  
        slow = slow.next;  
        fast = fast.next.next;  
    }  
  
    // 删除中间节点  
    slow.next = slow.next.next;  
  
    return dummy.next;  
}  
  
/**  
 * 主测试方法 - 包含全面的测试用例和验证策略  
*
```

- \* <h4>测试策略</h4>
- \* 1. <b>功能验证</b>：确保算法正确实现了分隔功能
- \* 2. <b>边界测试</b>：测试特殊输入情况
- \* 3. <b>极端情况测试</b>：测试性能和正确性边界
- \* 4. <b>多解法对比</b>：验证不同实现方法的正确性
- \* 5. <b>结果验证</b>：确保所有分隔后的链表满足条件
- \*
- \* <h4>测试覆盖范围</h4>
- \* - 标准情况
- \* - 边界情况（空链表、单节点链表）
- \* - 特殊情况（全小于/大于 x 的链表）
- \* - 已排序/逆序链表
- \* - 重复值链表
- \*/

```

public static void main(String[] args) {
    System.out.println("== 链表分隔问题测试 ==");
    System.out.println("算法本质：分类与合并模式，使用虚拟头节点技术");

    // 【测试用例 1】标准情况 - 混合大小的元素分布
    // 输入：[1, 4, 3, 2, 5, 2], x = 3
    // 预期输出：[1, 2, 2, 4, 3, 5]
    // 验证点：1. 小于 x 的节点在前 2. 大于等于 x 的节点在后 3. 相对顺序保持不变
    System.out.println("\n【测试用例 1】标准情况 - 混合元素分布");
    System.out.println("测试目的：验证基本功能正确性，确保相对顺序保持不变");
    int[] arr1 = {1, 4, 3, 2, 5, 2};
    ListNode head1 = ListNode.createList(arr1);

    System.out.print("原链表：");
    printList(head1);

    ListNode result1 = partition(head1, 3);
    System.out.print("分隔后（双链表法）：");
    printList(result1);

    // 验证结果正确性
    verifyPartitionResult(result1, 3);

    // 重新构建测试用例，测试解法 2
    head1 = ListNode.createList(arr1);
    ListNode result2 = partition2(head1, 3);
    System.out.print("分隔后（原地操作法）：");
    printList(result2);
}

```

```
// 验证结果正确性
verifyPartitionResult(result2, 3);

// 测试用例 2: 两个节点, 需要交换
// 输入: [2, 1], x = 2
// 预期输出: [1, 2]
System.out.println("\n测试用例 2: 两个节点需要交换");
int[] arr2 = {2, 1};
ListNode head2 = ListNode.createList(arr2);

System.out.print("原链表: ");
printList(head2);

ListNode result3 = partition(head2, 2);
System.out.print("分隔后: ");
printList(result3);

// 测试用例 3: 空链表
System.out.println("\n测试用例 3: 空链表");
ListNode head3 = null;

System.out.print("原链表: ");
printList(head3);

ListNode result4 = partition(head3, 1);
System.out.print("分隔后: ");
printList(result4);

// 测试用例 4: 单节点链表
System.out.println("\n测试用例 4: 单节点链表");
int[] arr4 = {5};
ListNode head4 = ListNode.createList(arr4);

System.out.print("原链表: ");
printList(head4);

ListNode result5 = partition(head4, 3);
System.out.print("分隔后 (x=3): ");
printList(result5);

// 测试用例 5: 所有节点值都小于 x
System.out.println("\n测试用例 5: 所有节点值都小于 x");
int[] arr5 = {1, 2, 3};
```

```
ListNode head5 = ListNode.createList(arr5);

System.out.print("原链表: ");
printList(head5);

ListNode result6 = partition(head5, 4);
System.out.print("分隔后 (x=4): ");
printList(result6);

// 测试用例 6: 所有节点值都大于等于 x
System.out.println("\n测试用例 6: 所有节点值都大于等于 x");
int[] arr6 = {5, 6, 7};
ListNode head6 = ListNode.createList(arr6);

System.out.print("原链表: ");
printList(head6);

ListNode result7 = partition(head6, 4);
System.out.print("分隔后 (x=4): ");
printList(result7);

// 测试用例 7: 已排序的链表
System.out.println("\n测试用例 7: 已排序的链表");
int[] arr7 = {1, 2, 3, 4, 5};
ListNode head7 = ListNode.createList(arr7);

System.out.print("原链表: ");
printList(head7);

ListNode result8 = partition(head7, 3);
System.out.print("分隔后 (x=3): ");
printList(result8);

// 测试用例 8: 逆序的链表
System.out.println("\n测试用例 8: 逆序的链表");
int[] arr8 = {5, 4, 3, 2, 1};
ListNode head8 = ListNode.createList(arr8);

System.out.print("原链表: ");
printList(head8);

ListNode result9 = partition(head8, 3);
System.out.print("分隔后 (x=3): ");
```

```

printList(result9);

// ===== 扩展题目测试 =====
System.out.println("\n===== 扩展题目测试 =====");

// 测试 1: LeetCode 328 - 链表奇偶重排
System.out.println("\n【扩展测试 1】LeetCode 328 - Odd Even Linked List");
int[] arr9 = {1, 2, 3, 4, 5};
ListNode head9 = ListNode.createList(arr9);
System.out.print("原链表: ");
printList(head9);
ListNode result10 = oddEvenList(head9);
System.out.print("奇偶重排后: ");
printList(result10);

int[] arr10 = {2, 1, 3, 5, 6, 4, 7};
ListNode head10 = ListNode.createList(arr10);
System.out.print("\n原链表: ");
printList(head10);
ListNode result11 = oddEvenList(head10);
System.out.print("奇偶重排后: ");
printList(result11);

// 测试 2: LeetCode 725 - 分隔链表为多部分
System.out.println("\n【扩展测试 2】LeetCode 725 - Split Linked List in Parts");
int[] arr11 = {1, 2, 3};
ListNode head11 = ListNode.createList(arr11);
System.out.print("原链表: ");
printList(head11);
ListNode[] parts1 = splitListToParts(head11, 5);
System.out.println("分隔为 5 部分:");
for (int i = 0; i < parts1.length; i++) {
    System.out.print("部分" + (i + 1) + ": ");
    printList(parts1[i]);
}

int[] arr12 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
ListNode head12 = ListNode.createList(arr12);
System.out.print("\n原链表: ");
printList(head12);
ListNode[] parts2 = splitListToParts(head12, 3);
System.out.println("分隔为 3 部分:");
for (int i = 0; i < parts2.length; i++) {
}

```

```

        System.out.print("部分" + (i + 1) + ":" );
        printList(parts2[i]);
    }

    // 测试 3: LeetCode 2095 - 删除链表中间节点
    System.out.println("\n【扩展测试 3】LeetCode 2095 - Delete Middle Node");
    int[] arr13 = {1, 3, 4, 7, 1, 2, 6};
    ListNode head13 = ListNode.createList(arr13);
    System.out.print("原链表: ");
    printList(head13);
    ListNode result12 = deleteMiddle(head13);
    System.out.print("删除中间节点后: ");
    printList(result12);

    int[] arr14 = {1, 2, 3, 4};
    ListNode head14 = ListNode.createList(arr14);
    System.out.print("\n原链表: ");
    printList(head14);
    ListNode result13 = deleteMiddle(head14);
    System.out.print("删除中间节点后: ");
    printList(result13);

    System.out.println("\n===== 所有测试完成 =====");
}

}
=====
```

文件: PartitionList.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
from typing import Optional
```

```
"""
```

链表分隔问题 - 最优解实现与详细分析

题目描述:

给你一个链表的头节点 head 和一个特定值 x，请你对链表进行分隔，使得所有小于 x 的节点都出现在大于或等于 x 的节点之前。你应当保留两个分区中每个节点的初始相对位置。

示例：

输入： head = [1, 4, 3, 2, 5, 2], x = 3

输出： [1, 2, 2, 4, 3, 5]

输入： head = [2, 1], x = 2

输出： [1, 2]

解题思路：

1. 双链表法（推荐）：使用两个链表分别存储小于 x 和大于等于 x 的节点，最后连接
2. 原地操作法：在原链表中移动节点，保持相对顺序

时间复杂度：O(n) – 只需遍历链表一次

空间复杂度：O(1) – 只使用常数级别额外空间

相似题目：

1. LeetCode 86. Partition List (本题)
2. LintCode 96. Partition List
3. 牛客网 NC140. 链表的奇偶重排
4. LeetCode 21. Merge Two Sorted Lists
5. LeetCode 23. Merge k Sorted Lists
6. LeetCode 148. Sort List

测试链接：<https://leetcode.cn/problems/partition-list/>

"""

```
class ListNode:
```

```
    """
```

```
    链表节点类定义
```

```
    """
```

```
    def __init__(self, val=0, next=None):
```

```
        """
```

```
        初始化链表节点
```

Args:

    val: 节点的值

    next: 指向下一个节点的引用，默认为 None

```
    """
```

```
    self.val = val
```

```
    self.next = next
```

```
class Solution:
```

```
    """
```

## 链表分隔问题解决方案类

"""

```
@staticmethod  
def partition(head: ListNode, x: int) -> ListNode:  
    """
```

解法 1：双链表法（推荐最优解）

核心思想：

1. 创建两个虚拟头节点，分别用于存储小于  $x$  和大于等于  $x$  的节点
2. 遍历原链表，根据节点值将节点连接到对应的链表中
3. 连接两个链表并返回结果

此解法的优势：

- 逻辑清晰，易于理解和实现
- 边界条件处理简单，不容易出错
- 满足  $O(n)$  时间复杂度和  $O(1)$  空间复杂度要求

时间复杂度分析：

- 遍历操作： $O(n)$  - 只需要遍历原链表一次
- 指针操作： $O(1)$  - 每个节点进行常数次指针操作
- 总体复杂度： $O(n)$

空间复杂度分析：

- 额外节点： $O(1)$  - 只使用两个虚拟头节点
- 指针变量： $O(1)$  - 使用常数个指针变量
- 总体复杂度： $O(1)$

Args:

head: 链表头节点

x: 分隔值

Returns:

分隔后的链表头节点

"""

```
# 【异常处理】空链表直接返回 None
```

```
if head is None:
```

```
    return None
```

```
# 创建两个虚拟头节点，分别用于存储小于 x 和大于等于 x 的节点
```

```
# 使用虚拟头节点可以避免处理头节点为空的边界情况
```

```
left_dummy = ListNode(0)
```

```
right_dummy = ListNode(0)
```

```

# 两个链表的尾指针，用于高效添加节点
left_tail = left_dummy
right_tail = right_dummy

# 遍历原链表
current = head
while current:
    # 【关键点】提前保存下一个节点，避免在操作当前节点时丢失链表后续部分
    next_node = current.next

    # 【重要】断开当前节点与原链表的连接，防止形成环
    current.next = None

    # 根据节点值将节点连接到对应的链表中
    if current.val < x:
        # 小于 x 的节点连接到左侧链表
        left_tail.next = current
        left_tail = current # 更新左侧链表尾指针
    else:
        # 大于等于 x 的节点连接到右侧链表
        right_tail.next = current
        right_tail = current # 更新右侧链表尾指针

    # 移动到下一个节点
    current = next_node

# 【关键点】连接两个链表：将左侧链表的尾部连接到右侧链表的头部
left_tail.next = right_dummy.next

# 返回结果链表的头节点（左侧链表的第一个有效节点）
return left_dummy.next

```

```

@staticmethod
def partition2(head: ListNode, x: int) -> ListNode:
    """
    """

```

解法 2：原地操作法

核心思想：

1. 使用一个指针遍历链表
2. 遇到小于 x 的节点就将其移动到前面
3. 保持相对顺序不变

这种方法虽然也是  $O(n)$  时间复杂度和  $O(1)$  空间复杂度，  
但实现更复杂，且容易在指针操作中出错

时间复杂度：  $O(n)$  – 只需遍历链表一次

空间复杂度：  $O(1)$  – 只使用常数级别额外空间

Args:

head: 链表头节点

x: 分隔值

Returns:

分隔后的链表头节点

```
"""
```

# 【异常处理】空链表直接返回 None

```
if head is None:
```

```
    return None
```

# 创建虚拟头节点，简化边界处理

```
dummy = ListNode(0)
```

```
dummy.next = head
```

# 找到第一个大于等于 x 的节点的前驱节点

# 这个节点将作为小于 x 的节点插入位置的前驱

```
prev = dummy
```

```
while prev.next and prev.next.val < x:
```

```
    prev = prev.next
```

# 当前节点指针，用于遍历链表

```
curr = prev
```

# 遍历链表剩余部分

```
while curr.next:
```

# 如果下一个节点小于 x，则需要将其移动到前面

```
if curr.next.val < x:
```

# 【指针操作】取出要移动的节点

```
move_node = curr.next
```

# 从当前位置断开

```
curr.next = move_node.next
```

# 插入到 prev 后面

```
move_node.next = prev.next
```

```
prev.next = move_node
```

```
# 更新 prev 指针，为下一次插入做准备
    prev = move_node
else:
    # 否则继续向后移动
    curr = curr.next

return dummy.next
```

```
# ====== 扩展题目 1: LeetCode 328. Odd Even Linked List (链表奇偶重排) ======
def odd_even_list(head: ListNode) -> ListNode:
```

```
"""
LeetCode 328. Odd Even Linked List
```

```
题目链接: https://leetcode.cn/problems/odd-even-linked-list/
```

题目描述:

给定单链表的头节点 head，将所有索引为奇数的节点和索引为偶数的节点分别组合在一起

时间复杂度: O(n) 空间复杂度: O(1) 是否最优解: 是

```
"""
```

```
if head is None or head.next is None:
    return head
```

```
odd = head
even = head.next
even_head = even
```

```
while even and even.next:
    odd.next = even.next
    odd = odd.next
    even.next = odd.next
    even = even.next
```

```
odd.next = even_head
return head
```

```
# ====== 扩展题目 2: LeetCode 725. Split Linked List in Parts ======
```

```
def split_list_to_parts(head: ListNode, k: int) -> list:
```

```
"""
LeetCode 725. Split Linked List in Parts
```

```
题目链接: https://leetcode.cn/problems/split-linked-list-in-parts/
```

时间复杂度: O(n+k) 空间复杂度: O(k) 是否最优解: 是

"""

# 计算链表长度

length = 0

curr = head

while curr:

    length += 1

    curr = curr.next

# 计算每部分的大小

part\_size = length // k

remainder = length % k

result = []

curr = head

for i in range(k):

    result.append(curr)

    if curr is None:

        continue

    current\_part\_size = part\_size + (1 if i < remainder else 0)

    for j in range(current\_part\_size - 1):

        if curr:

            curr = curr.next

    if curr:

        next\_node = curr.next

        curr.next = None

        curr = next\_node

return result

# ===== 扩展题目 3: LeetCode 2095. Delete Middle Node =====

def delete\_middle(head: ListNode) -> ListNode:

"""

LeetCode 2095. Delete the Middle Node of a Linked List

题目链接: <https://leetcode.cn/problems/delete-the-middle-node-of-a-linked-list/>

时间复杂度: O(n) 空间复杂度: O(1) 是否最优解: 是

```
"""
if head is None or head.next is None:
    return None
```

```
dummy = ListNode(0, head)
slow = dummy
fast = head
```

```
while fast and fast.next:
    slow = slow.next
    fast = fast.next.next
```

```
slow.next = slow.next.next
return dummy.next
```

```
def create_list(values: list) -> ListNode:
    """
```

从列表创建链表的辅助函数

Args:

values: 包含节点值的列表

Returns:

创建的链表头节点

```
"""
```

# 处理空列表情况

```
if not values:
```

```
    return None
```

# 创建虚拟头节点简化操作

```
dummy = ListNode(0)
```

```
current = dummy
```

# 逐个创建节点并连接

```
for val in values:
```

```
    current.next = ListNode(val)
```

```
    current = current.next
```

```
return dummy.next
```

```
def print_list(head: ListNode) -> None:
```

```
"""
```

打印链表的辅助函数

Args:

head: 链表头节点

```
"""
```

# 处理空链表情况

```
if head is None:  
    print("空链表")  
    return
```

# 收集所有节点的值

```
values = []  
current = head  
while current:  
    values.append(str(current.val))  
    current = current.next
```

# 格式化输出

```
print(" -> ".join(values))
```

```
def verify_partition_result(head: ListNode, x: int) -> bool:
```

```
"""
```

验证链表分隔结果是否正确

验证规则:

1. 所有小于 x 的节点必须出现在大于等于 x 的节点之前
2. 必须保持节点的相对顺序
3. 不能出现循环引用

实现思路:

- 使用状态标志跟踪是否已经遇到大于等于 x 的节点
- 遍历链表检查是否违反分区规则
- 同时检查是否存在循环 (通过记录访问过的节点数量限制)

Args:

head: 分隔后的链表头节点

x: 分隔值

Returns:

验证是否通过

```
"""
```

```

if head is None:
    print("验证结果：通过（空链表）")
    return True

passed_x = False # 标志是否已经遇到大于等于 x 的节点
current = head
node_count = 0 # 用于检测循环引用
max_nodes = 1000 # 链表最大节点数限制，防止无限循环

while current and node_count < max_nodes:
    # 检查分区规则：如果已经遇到过大于是等于 x 的节点，则后续节点都不能小于 x
    if passed_x and current.val < x:
        print(f"验证结果：失败！违反分区规则 - 大于等于{x}的节点后出现小于{x}的节点")
        return False

    # 如果当前节点大于等于 x，则设置 passed_x 标志为 True
    if current.val >= x:
        passed_x = True

    current = current.next
    node_count += 1

# 检查是否存在循环引用
if node_count >= max_nodes:
    print("验证结果：失败！检测到可能的循环引用")
    return False

print("验证结果：通过 - 分区规则正确遵守")
return True

```

```

def run_test_cases():
    """

```

运行全面的测试用例和验证策略

测试策略：

1. 功能验证：确保算法正确实现了分隔功能
2. 边界测试：测试特殊输入情况
3. 极端情况测试：测试性能和正确性边界
4. 多解法对比：验证不同实现方法的正确性
5. 结果验证：确保所有分隔后的链表满足条件

测试覆盖范围：

- 标准情况
- 边界情况（空链表、单节点链表）
- 特殊情况（全小于/大于 x 的链表）
- 已排序/逆序链表
- 重复值链表

"""

```
print("== 链表分隔问题测试 ==")  
print("算法本质：分类与合并模式，使用虚拟头节点技术")
```

```
solution = Solution()
```

```
# 【测试用例 1】标准情况 - 混合大小的元素分布  
# 输入: [1, 4, 3, 2, 5, 2], x = 3  
# 预期输出: [1, 2, 2, 4, 3, 5]  
# 验证点: 1. 小于 x 的节点在前 2. 大于等于 x 的节点在后 3. 相对顺序保持不变  
print("\n【测试用例 1】标准情况 - 混合元素分布")  
print("测试目的: 验证基本功能正确性, 确保相对顺序保持不变")  
head1 = create_list([1, 4, 3, 2, 5, 2])
```

```
print("原链表:")  
print_list(head1)
```

```
result1 = solution.partition(head1, 3)  
print("分隔后 (双链表法):")  
print_list(result1)
```

```
# 验证结果正确性  
verify_partition_result(result1, 3)
```

```
# 重新构建测试用例, 测试解法 2  
head1 = create_list([1, 4, 3, 2, 5, 2])  
result2 = solution.partition2(head1, 3)  
print("分隔后 (原地操作法):")  
print_list(result2)
```

```
# 验证结果正确性  
verify_partition_result(result2, 3)
```

```
# 测试用例 2: 两个节点, 需要交换  
# 输入: [2, 1], x = 2  
# 预期输出: [1, 2]  
print("\n测试用例 2: 两个节点需要交换")  
head2 = create_list([2, 1])
```

```
print("原链表:")
print_list(head2)

result3 = solution.partition(head2, 2)
print("分隔后:")
print_list(result3)

# 测试用例 3: 空链表
print("\n测试用例 3: 空链表")
head3 = None

print("原链表:")
print_list(head3)

result4 = solution.partition(head3, 1)
print("分隔后:")
print_list(result4)

# 测试用例 4: 单节点链表
print("\n测试用例 4: 单节点链表")
head4 = create_list([5])

print("原链表:")
print_list(head4)

result5 = solution.partition(head4, 3)
print("分隔后 (x=3):")
print_list(result5)

# 测试用例 5: 所有节点值都小于 x
print("\n测试用例 5: 所有节点值都小于 x")
head5 = create_list([1, 2, 3])

print("原链表:")
print_list(head5)

result6 = solution.partition(head5, 4)
print("分隔后 (x=4):")
print_list(result6)

# 测试用例 6: 所有节点值都大于等于 x
print("\n测试用例 6: 所有节点值都大于等于 x")
```

```
head6 = create_list([5, 6, 7])

print("原链表:")
print_list(head6)

result7 = solution.partition(head6, 4)
print("分隔后 (x=4):")
print_list(result7)

# 测试用例 7: 已排序的链表
print("\n测试用例 7: 已排序的链表")
head7 = create_list([1, 2, 3, 4, 5])

print("原链表:")
print_list(head7)

result8 = solution.partition(head7, 3)
print("分隔后 (x=3):")
print_list(result8)

# 测试用例 8: 逆序的链表
print("\n测试用例 8: 逆序的链表")
head8 = create_list([5, 4, 3, 2, 1])

print("原链表:")
print_list(head8)

result9 = solution.partition(head8, 3)
print("分隔后 (x=3):")
print_list(result9)

# ====== 扩展题目测试 ======
print("\n===== 扩展题目测试 =====")

# 测试 1: LeetCode 328 - 链表奇偶重排
print("\n【扩展测试 1】LeetCode 328 - Odd Even Linked List")
head9 = create_list([1, 2, 3, 4, 5])
print("原链表:")
print_list(head9)
result10 = odd_even_list(head9)
print("奇偶重排后:")
print_list(result10)
```

```

head10 = create_list([2, 1, 3, 5, 6, 4, 7])
print("\n原链表:")
print_list(head10)
result11 = odd_even_list(head10)
print("奇偶重排后:")
print_list(result11)

# 测试 2: LeetCode 725 - 分隔链表为多部分
print("\n【扩展测试 2】LeetCode 725 - Split Linked List in Parts")
head11 = create_list([1, 2, 3])
print("原链表:")
print_list(head11)
parts1 = split_list_to_parts(head11, 5)
print("分隔为 5 部分:")
for i, part in enumerate(parts1, 1):
    print(f"部分 {i}: ", end="")
    print_list(part)

head12 = create_list([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print("\n原链表:")
print_list(head12)
parts2 = split_list_to_parts(head12, 3)
print("分隔为 3 部分:")
for i, part in enumerate(parts2, 1):
    print(f"部分 {i}: ", end="")
    print_list(part)

# 测试 3: LeetCode 2095 - 删除链表中间节点
print("\n【扩展测试 3】LeetCode 2095 - Delete Middle Node")
head13 = create_list([1, 3, 4, 7, 1, 2, 6])
print("原链表:")
print_list(head13)
result12 = delete_middle(head13)
print("删除中间节点后:")
print_list(result12)

head14 = create_list([1, 2, 3, 4])
print("\n原链表:")
print_list(head14)
result13 = delete_middle(head14)
print("删除中间节点后:")
print_list(result13)

```

```
print("\n===== 所有测试完成 =====")\n\n# ===== 扩展题目测试 =====\nprint("\n===== 扩展题目测试 =====")\n\n# 测试 4: LeetCode 21 - 合并两个有序链表\nprint("\n【扩展测试 4】LeetCode 21 - Merge Two Sorted Lists")\narr15 = [1, 2, 4]\narr16 = [1, 3, 4]\nl1 = create_list(arr15)\nl2 = create_list(arr16)\nprint("链表 1:", end=" ") \nprint_list(l1)\nprint("链表 2:", end=" ") \nprint_list(l2)\nmerged = merge_two_lists(l1, l2)\nprint("合并后:", end=" ") \nprint_list(merged)\n\n# 测试 5: LeetCode 19 - 删除链表的倒数第 N 个节点\nprint("\n【扩展测试 5】LeetCode 19 - Remove Nth Node From End")\narr17 = [1, 2, 3, 4, 5]\nhead17 = create_list(arr17)\nprint("原链表:", end=" ") \nprint_list(head17)\nresult14 = remove_nth_from_end(head17, 2)\nprint("删除倒数第 2 个节点后:", end=" ") \nprint_list(result14)\n\n# 测试 6: LeetCode 206 - 反转链表\nprint("\n【扩展测试 6】LeetCode 206 - Reverse Linked List")\narr18 = [1, 2, 3, 4, 5]\nhead18 = create_list(arr18)\nprint("原链表:", end=" ") \nprint_list(head18)\nreversed_head = reverse_list(head18)\nprint("反转后:", end=" ") \nprint_list(reversed_head)\n\n# 测试 7: LeetCode 24 - 两两交换链表中的节点\nprint("\n【扩展测试 7】LeetCode 24 - Swap Nodes in Pairs")\narr19 = [1, 2, 3, 4]\nhead19 = create_list(arr19)
```

```

print("原链表:", end=" ")
print_list(head19)
swapped = swap_pairs(head19)
print("两两交换后:", end=" ")
print_list(swapped)

# 测试 8: LeetCode 876 - 链表的中间结点
print("\n【扩展测试 8】LeetCode 876 - Middle of Linked List")
arr20 = [1, 2, 3, 4, 5]
head20 = create_list(arr20)
print("原链表:", end=" ")
print_list(head20)
middle = middle_node(head20)
print("中间节点:", middle.val)

print("\n===== 所有测试完成 =====")

```

# ===== 扩展题目 4: LeetCode 21. Merge Two Sorted Lists (合并两个有序链表) =====

```

def merge_two_lists(l1: Optional[ListNode], l2: Optional[ListNode]) -> Optional[ListNode]:
    """
    LeetCode 21. Merge Two Sorted Lists
    题目链接: https://leetcode.cn/problems/merge-two-sorted-lists/
    
```

#### 题目描述:

将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

#### 示例:

输入: l1 = [1, 2, 4], l2 = [1, 3, 4]

输出: [1, 1, 2, 3, 4, 4]

#### 解题思路:

使用双指针技术，比较两个链表的当前节点值，将较小的节点连接到结果链表中。

时间复杂度:  $O(n+m)$  - 需要遍历两个链表的所有节点

空间复杂度:  $O(1)$  - 只使用常数级别的额外空间

是否最优解: 是，时间和空间复杂度都已达到最优

#### 与链表分隔的联系:

- 都使用虚拟头节点技术简化边界处理
- 都涉及多个链表的指针操作和连接
- 都需要保持元素的相对顺序

"""

```

# 创建虚拟头节点简化操作
dummy = ListNode(0)
current = dummy

# 同时遍历两个链表
while l1 and l2:
    # 比较两个链表当前节点的值
    if l1.val <= l2.val:
        current.next = l1
        l1 = l1.next
    else:
        current.next = l2
        l2 = l2.next
    current = current.next

# 将剩余链表连接到结果中
current.next = l1 if l1 else l2

return dummy.next

# ====== 扩展题目 5: LeetCode 19. Remove Nth Node From End of List (删除链表的倒数第 N 个节点)
=====

def remove_nth_from_end(head: Optional[ListNode], n: int) -> Optional[ListNode]:
    """
    LeetCode 19. Remove Nth Node From End of List
    题目链接: https://leetcode.cn/problems/remove-nth-node-from-end-of-list/
    """

    ...

```

#### 题目描述:

给你一个链表，删除链表的倒数第  $n$  个结点，并且返回链表的头结点。

#### 示例:

输入: head = [1, 2, 3, 4, 5],  $n = 2$

输出: [1, 2, 3, 5]

#### 解题思路:

使用快慢指针，快指针先走  $n$  步，然后快慢指针同时移动，当快指针到达末尾时，慢指针指向倒数第  $n$  个节点的前驱。

时间复杂度:  $O(L)$  –  $L$  为链表长度

空间复杂度:  $O(1)$  – 只使用常数级别的额外空间

是否最优解: 是

与链表分隔的联系：

- 都使用双指针技术
- 都需要精确控制指针移动和节点连接
- 都需要处理边界情况（如删除头节点）

"""

# 使用虚拟头节点简化删除头节点的边界情况

```
dummy = ListNode(0)
```

```
dummy.next = head
```

```
fast = dummy
```

```
slow = dummy
```

# 快指针先走 n 步

```
for _ in range(n + 1):
```

```
    if fast:
```

```
        fast = fast.next
```

# 快慢指针同时移动，直到快指针到达末尾

```
while fast:
```

```
    fast = fast.next
```

```
    slow = slow.next
```

# 删除倒数第 n 个节点

```
if slow and slow.next:
```

```
    slow.next = slow.next.next
```

```
return dummy.next
```

# ====== 扩展题目 6: LeetCode 206. Reverse Linked List (反转链表) ======

```
def reverse_list(head: Optional[ListNode]) -> Optional[ListNode]:
```

"""

LeetCode 206. Reverse Linked List

题目链接: <https://leetcode.cn/problems/reverse-linked-list/>

题目描述：

给你单链表的头节点 head，请你反转链表，并返回反转后的链表。

示例：

输入: head = [1, 2, 3, 4, 5]

输出: [5, 4, 3, 2, 1]

解题思路：

使用迭代法，维护三个指针：prev、current、next，逐个反转节点指向。

时间复杂度:  $O(n)$  - 需要遍历整个链表

空间复杂度:  $O(1)$  - 只使用常数级别的额外空间

是否最优解: 是

与链表分隔的联系:

- 都是链表基本操作
- 都需要精确的指针操作
- 反转操作是许多复杂链表算法的基础

"""

```
prev = None
```

```
current = head
```

```
while current:
```

```
    next_node = current.next # 保存下一个节点  
    current.next = prev      # 反转当前节点指向  
    prev = current           # 移动 prev 指针  
    current = next_node      # 移动 current 指针
```

```
return prev
```

# ====== 扩展题目 7: LeetCode 24. Swap Nodes in Pairs (两两交换链表中的节点) ======

```
def swap_pairs(head: Optional[ListNode]) -> Optional[ListNode]:
```

"""

LeetCode 24. Swap Nodes in Pairs

题目链接: <https://leetcode.cn/problems/swap-nodes-in-pairs/>

题目描述:

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。

你必须在不修改节点内部的值的情况下完成本题（即只能进行节点交换）。

示例:

输入: head = [1, 2, 3, 4]

输出: [2, 1, 4, 3]

解题思路:

使用虚拟头节点，每次处理两个相邻节点，调整它们的指向关系。

时间复杂度:  $O(n)$  - 需要遍历整个链表

空间复杂度:  $O(1)$  - 只使用常数级别的额外空间

是否最优解: 是

与链表分隔的联系：

- 都涉及链表的重新连接
- 都需要精确的指针操作
- 都使用虚拟头节点简化边界处理

"""

```
# 使用虚拟头节点简化操作
```

```
dummy = ListNode(0)
```

```
dummy.next = head
```

```
prev = dummy
```

```
while prev.next and prev.next.next:
```

```
    first = prev.next
```

```
    second = first.next
```

```
# 交换两个节点
```

```
    first.next = second.next
```

```
    second.next = first
```

```
    prev.next = second
```

```
# 移动指针到下一对
```

```
    prev = first
```

```
return dummy.next
```

# ====== 扩展题目 8: LeetCode 876. Middle of the Linked List (链表的中间结点) ======

```
def middle_node(head: Optional[ListNode]) -> Optional[ListNode]:
```

"""

LeetCode 876. Middle of the Linked List

题目链接: <https://leetcode.cn/problems/middle-of-the-linked-list/>

题目描述:

给定一个头结点为 head 的非空单链表，返回链表的中间结点。

如果有两个中间结点，则返回第二个中间结点。

示例:

输入: [1, 2, 3, 4, 5]

输出: 3 (节点值为 3 的节点)

解题思路:

使用快慢指针，快指针每次走两步，慢指针每次走一步。

时间复杂度: O(n) – 需要遍历链表到中间位置

空间复杂度: O(1) - 只使用两个指针

是否最优解: 是

与链表分隔的联系:

- 都使用双指针技术
- 都是链表操作的基础算法

"""

```
if not head:  
    return None
```

```
slow = head  
fast = head
```

```
while fast and fast.next:  
    slow = slow.next  
    fast = fast.next.next
```

```
return slow
```

# 执行测试

```
if __name__ == "__main__":  
    run_test_cases()
```

=====