《数据结构》课程实践报告

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 大二 软工 | 姓名 | 丁彦君 | 学号 | 2329403126 |
| 实验布置日期 | | 2024/09/23 | | 提交  日期 | 2024/10/08 | | 成绩 |  |

课程实践实验2：线性表的实现和应用

## 一、问题描述及要求

**1．实现无序表和顺序表及相关功能。**

**2．实现单链表，单向循环链表和双向链表（非循环）及相关功能。**

**3．将原有N副扑克牌整合在一起看是否能凑成M副扑克牌(M<N)。**

## 二、概要设计

**（1）对实验内容的理解**

**1.无序表就是创建一个动态的数组存放数据，而顺序表就是在无序表的基础上添加一些限制条件。**

**2.主要是掌握各种链表的创建，销毁，添加，删减，查找等功能和方法。**

**3.实验旨在模拟扑克牌的分配和处理过程。具体任务包括初始化若干副扑克牌，统计每副牌中的牌数，移除指定数量的牌，并最终输出剩余的牌。我们需要实现单链表的数据结构来存储牌，并保证链表中的牌按升序排列，同时允许重复牌的存在。程序中还涉及如何高效地统计、删除和拼凑扑克牌套数等操作。**

**（2）功能列表**

1. **功能描述：**

**题目1.**

**顺序表选择功能：程序首先让用户选择操作对象是无序表还是有序表。**

* **无序表功能：**
  1. **直接插入：在表的末尾插入元素。**
  2. **按位置插入：在指定位置插入元素。**
  3. **替换：根据用户提供的位置，替换元素的值。**
  4. **按值删除：删除表中指定值的元素。**
  5. **按位置删除：删除指定位置的元素。**
* **有序表功能：**
  1. **顺序插入：按照有序性插入元素，保持递增顺序。**
  2. **按位置插入：直接按位置插入元素（不保证有序性）。**
  3. **替换：替换指定位置的元素，之后检查有序性。**
  4. **按值删除：删除表中某个值。**
  5. **按位置删除：删除指定位置的元素。**
  6. **恢复数据：恢复插入或删除前的数据状态。**

**题目2.**

**该程序实现了三种不同类型的链表操作：**

**1.单链表（SingleLink）：提供基本的插入、删除、查找等操作。**

**2.单向循环链表（IterSingleLink）：环形结构，尾部指针指向头节点，同样支持插入、删除、查找等操作。**

**3.双向链表（DoubleLink）：每个节点有前后指针，支持双向遍历和操作。**

**用户通过输入可以选择创建不同类型的链表，并输入数据进行操作。根据不同链表结构，用户可以插入、删除节点，查找节点的值或位置，打印链表内容。**

**题目3.**

**系统的主要功能：**

**初始化扑克牌： 读取文件或手动输入，构建扑克牌链表。**

**牌的插入： 根据牌的大小按顺序插入到链表中。**

**牌的删除： 删除链表中指定牌的数量。**

**统计牌数： 统计当前链表中各花色的牌数。**

**拼凑完整扑克牌套数： 从多副牌中移除能够组成完整扑克牌的套数。**

**输出剩余牌： 打印链表中剩余的牌。**

**释放内存： 清空链表，释放已分配的内存。**

**（3）程序运行的界面设计**

**题目1.**

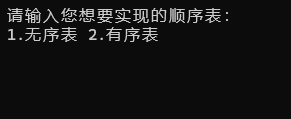
****

图 1首先选择想实现的顺序表

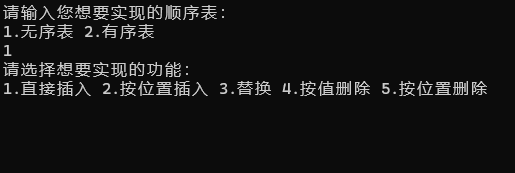
****

图 2这里先选择无序表

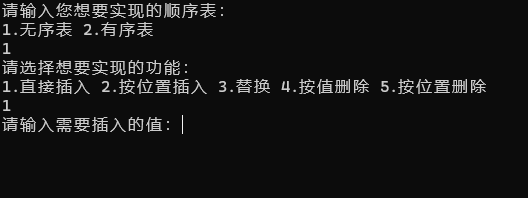
****

图 3接着会出现想要实现的功能

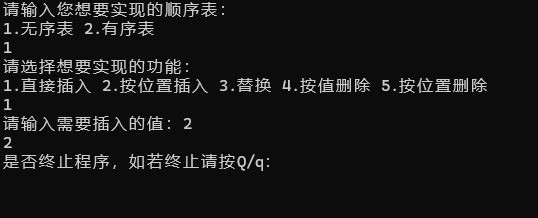
****

图 4首先运行功能1

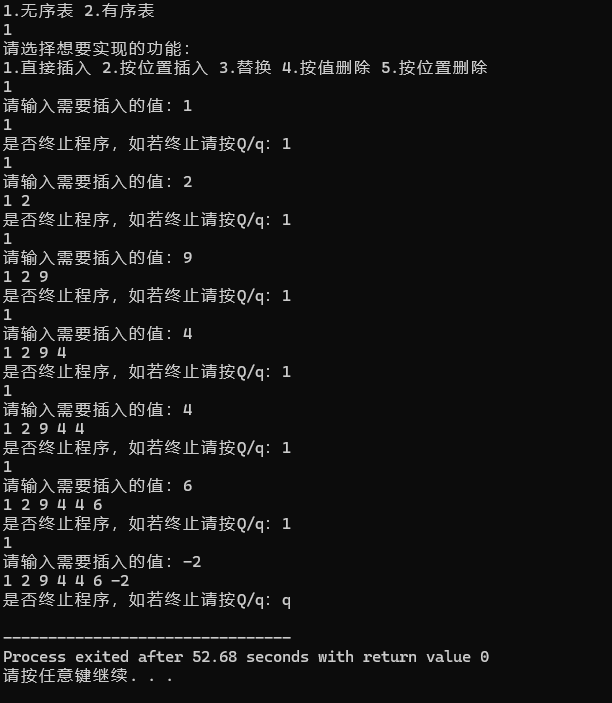
****

图5下面是功能1的演示，最后按q结束运行

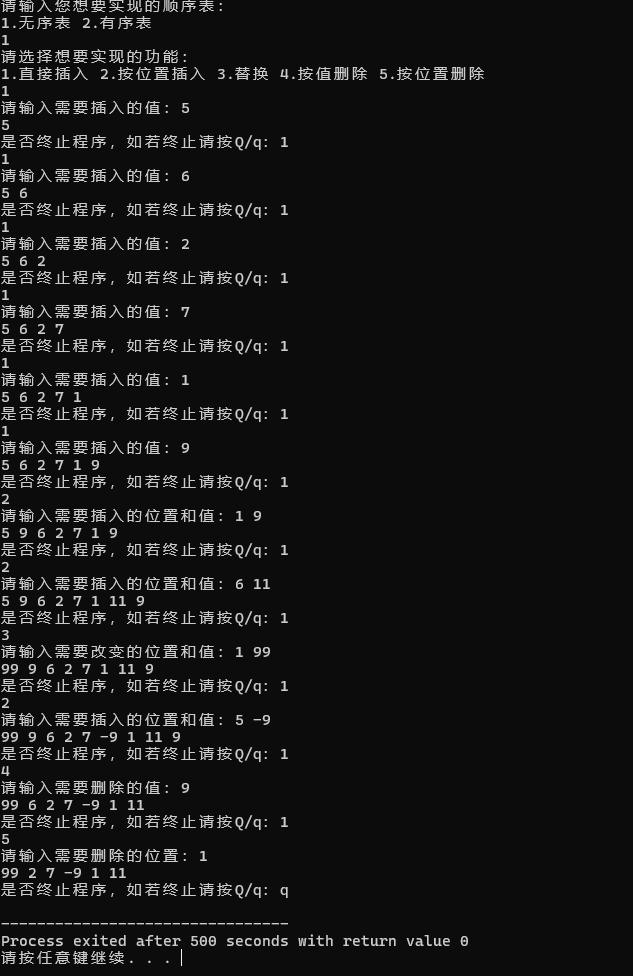
****

图6 这里是所有功能的演示

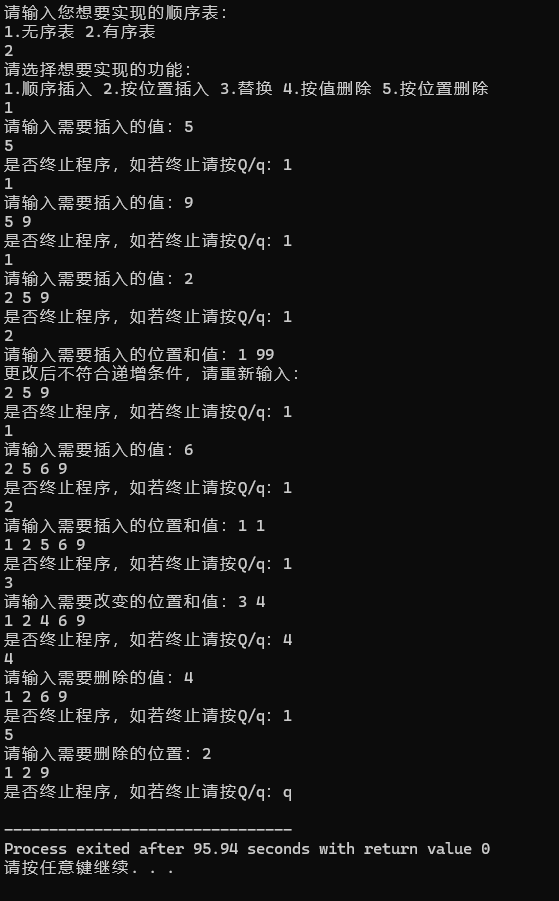


图7 这是有序表的所有功能演示

**题目2.**

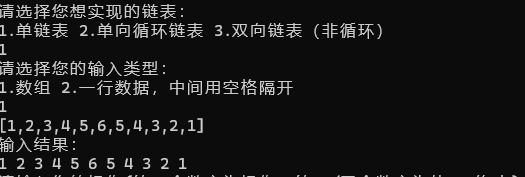
****

图1 首先选择链表种类，接着选择输入方式

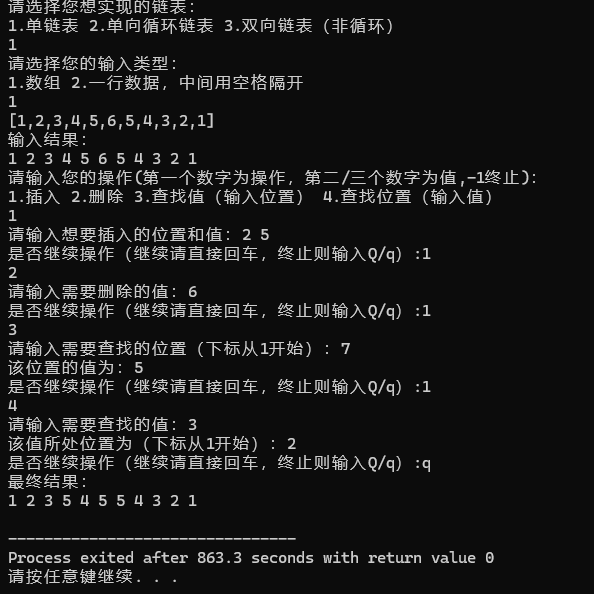
****

图2 这是数组输入的单链表功能展示

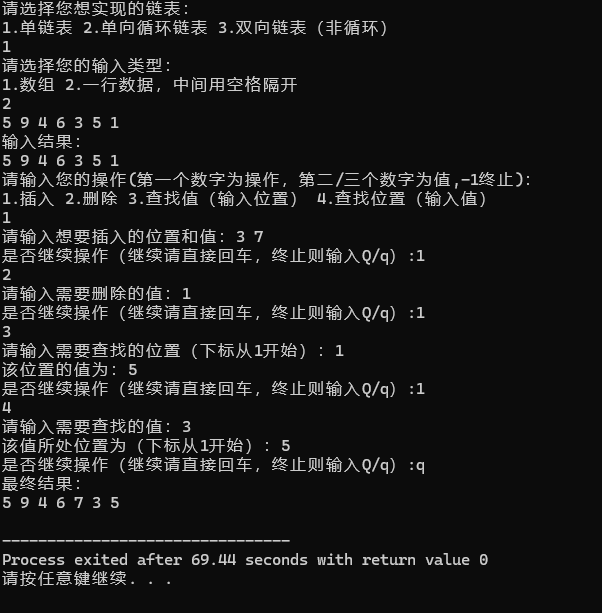
****

图3 这是一行数据输入的单链表功能展示

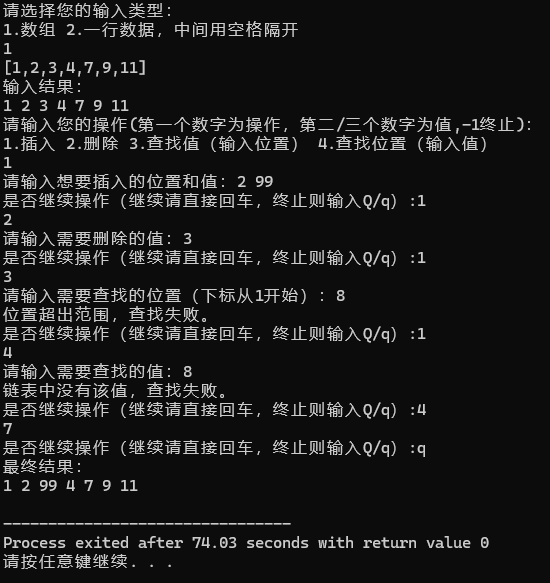
****

图1 这是单向循环链表的功能演示

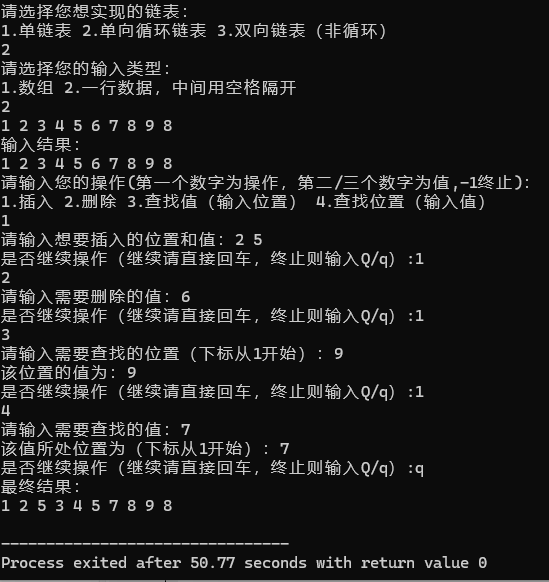


图2 这也是单向循环链表的部分

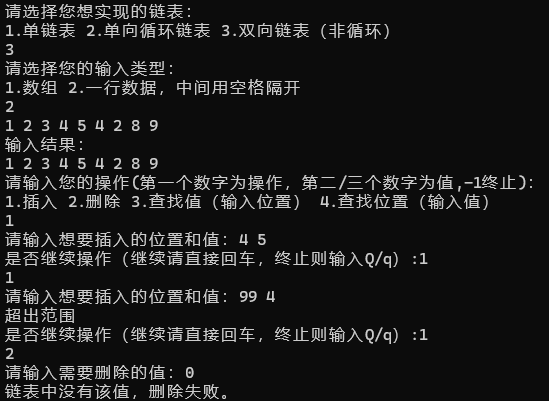


图1 双向链表的功能和前面差不多，这里就列举了一些输入超出边界的判定样例

**题目3.**

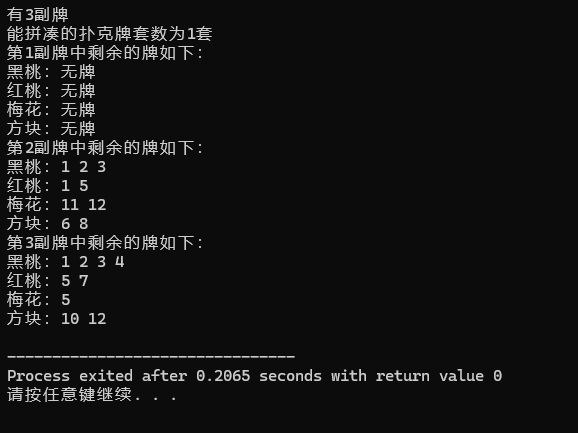
****

图1 无初始化界面设计，运行直接用了一个凑好的文件数据，正好符合条件，这里是结果

**（4）总体设计思路**

**题目1.**

**(1)数据结构：**

**1.采用顺序表，具体使用 vector<int> 作为底层容器，用于动态存储表中的元素。**

**2.无序表和有序表都使用vector，但对于有序表的插入和替换操作有严格的递增性要求。**

**(2)类设计：**

**list1：代表无序表，包含各种插入、删除、替换功能。**

**主要方法：**

**add1()：在表末尾插入元素。**

**add2()：按位置插入元素。**

**change()：替换某个位置的元素。**

**del1()：按值删除。**

**del2()：按位置删除。**

**print()：输出当前表内容。**

**list2：代表有序表，确保表中元素保持递增顺序。**

**主要方法：**

**OrderedAdd()：按递增顺序插入元素。**

**LocatedAdd()：按位置插入元素（可能破坏顺序）。**

**Change()：替换某个位置的元素。**

**del1()：按值删除。**

**del2()：按位置删除。**

**print()：输出当前表内容。**

**题目2.**

**(1)数据结构：**

**单链表：基于单向指针，每个节点包含一个值和一个指向下一个节点的指针。**

**单向循环链表：在单链表基础上，末尾节点指向头节点形成闭环。**

**双向链表：每个节点包含前后指针，使得能够在链表中向前或向后遍历。**

**(2)类的设计：**

**SingleLink类：**

**作用：实现单链表结构，提供链表的插入、删除、查找和打印功能。**

**方法介绍：**

**insert(int val)：插入值到链表中。**

**remove()：从链表中删除指定值。**

**get()：通过位置查找链表中的值。**

**search()：查找某值所在位置。**

**print()：打印当前链表内容。**

**destroy()：销毁链表，释放内存。**

**IterSingleLink类：**

**作用：实现单向循环链表，除了基本的链表操作，还处理循环结构。**

**方法介绍：与SingleLink类似，增加了循环链表的特殊处理逻辑。**

**DoubleLink类：**

**作用：实现双向链表，能够在链表中双向遍历。**

**方法介绍：**

**create(int val)：插入元素。**

**remove()：删除元素。**

**get()：通过位置查找链表中的值。**

**search()：根据值查找位置。**

**print()：打印链表。**

**destroy()：销毁链表。**

**(3)类之间的关系：**

**SingleLink、IterSingleLink和DoubleLink都是链表的不同实现。它们都实现了链表的基础功能，但数据结构和处理方式不同。**

**题目3.**

**（1）数据结构：**

**使用单链表 SingleLink 作为扑克牌的存储结构。每张牌用链表节点 CardNode 表示，按升序排列，链表允许重复牌值。**

**扑克牌的每种花色（黑桃、红桃、梅花、方块）分别存储在四个独立的链表中。**

**（2）主要类设计：**

**CardNode 类：**

**属性： 牌值 (value)、下一个节点的指针 (next)。**

**作用： 表示扑克牌链表中的每一张牌。**

**SingleLink 类：**

**方法：**

**Init()：初始化链表为空。**

**AddCard()：按升序插入一张牌到链表中。**

**PrintRemainingCards()：打印链表中的剩余牌。**

**RemoveCard()：删除链表中指定牌值的牌。**

**CountCards()：统计链表中的牌数。**

**FreeList()：释放链表的内存。**

**（3）主函数设计：**

**初始化扑克牌集合并按花色存储在不同的链表中。**

**根据用户输入或配置文件读取牌数据，并插入到对应的链表中。**

**提供操作菜单供用户选择功能，包括删除牌、打印剩余牌等。**

**（5）程序结构设计**

**题目1.**

**(1)文件划分：**

**main.cpp：程序的入口文件，负责用户输入、功能调用等。**

**list1.h / list1.cpp：定义无序表类及其方法。**

**list2.h / list2.cpp：定义有序表类及其方法，扩展有序操作。**

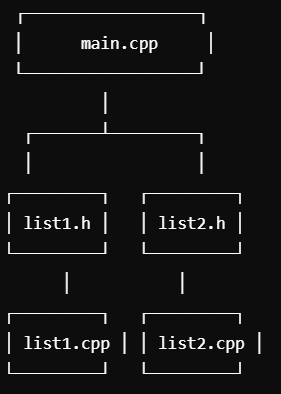
**(2)结构描述：**

**main.cpp 中引入了两个类 list1 和 list2，并通过用户选择实现对应的功能。**

**list1 类实现了基本的顺序表操作，无需维护元素的顺序。**

**list2 类在 list1 的基础上增加了递增顺序检查和顺序插入的功能，确保数据始终是有序的。**

**(3)类和模块关系图**

****

关系图

**题目2.**

**(1) 文件划分：**

**main.cpp:程序的入口文件，负责接收用户输入，调用不同类型链表的功能。根据用户选择调用对应的链表类，执行链表的插入、删除、查找等操作。**

**SingleLink.h / SingleLink.cpp:定义并实现单链表类 SingleLink，包括节点的插入、删除、查找及打印操作。该链表是单向非循环链表，适用于简单的数据结构。**

**IterSingleLink.h / IterSingleLink.cpp:定义并实现单向循环链表类 IterSingleLink，继承单链表类，并实现尾节点指向头节点的闭环结构，支持循环链表的操作。**

**DoubleLink.h / DoubleLink.cpp:定义并实现双向链表类 DoubleLink，在每个节点中维护前后指针，使得能够双向遍历链表。支持双向链表的插入、删除、查找及遍历操作。**

**(2) 结构描述：**

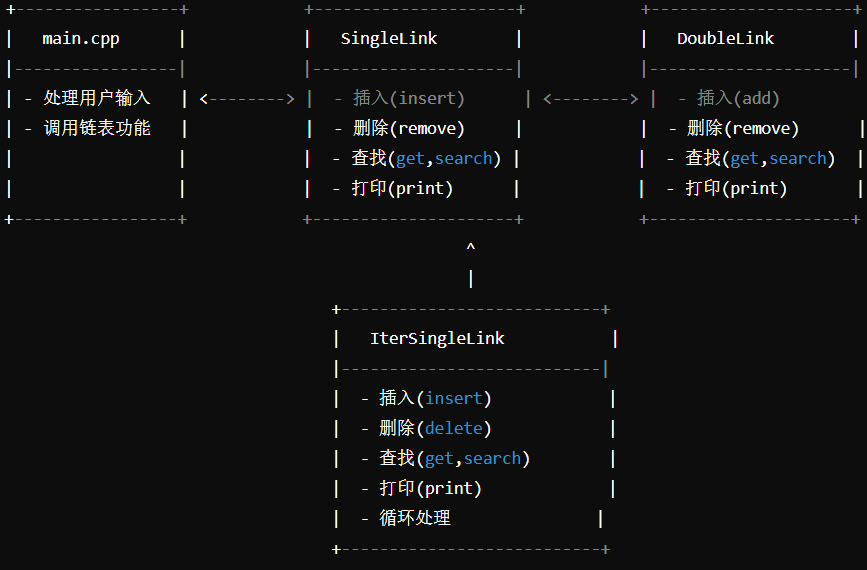
**main.cpp:main.cpp 文件作为程序的入口，包含用户输入的处理逻辑。用户可以选择链表类型（单链表、单向循环链表、双向链表），并通过不同的方式输入数据。程序基于用户选择的链表类型，调用相应的链表类进行操作。**

**SingleLink:实现了单链表的基本操作，如插入、删除、查找和打印。该链表使用单向指针，适合于较简单的数据操作。**

**IterSingleLink:在单链表的基础上增加了循环链表的特性。链表尾部指针指向头部，形成闭环，适用于需要循环访问的场景。**

**DoubleLink:实现了双向链表，每个节点包含前后指针，支持双向遍历。该链表允许用户从任意位置向前或向后操作节点，适用于复杂数据结构的场景。**

**(3) 类和模块关系图：**

****

**题目3.**

**程序文件划分如下：**

**main.cpp：主程序入口，负责用户输入、扑克牌数据的读取和操作调用。**

**SingleLink.h / SingleLink.cpp：定义扑克牌链表数据结构和操作方法。**

**CardNode.h：定义扑克牌链表节点的数据结构。**

**各部分关系：**

**主程序调用 SingleLink 类中的方法，完成扑克牌的插入、删除、打印等操作。**

**CardNode 类作为链表节点，链接形成单链表。**

## 三、详细设计

**题目1**.

**1. 主函数设计**

**主函数的作用是通过用户输入操作指令，选择对无序顺序表 (list1) 或有序顺序表 (list2) 进行操作。这部分的核心是循环提示用户进行插入、删除、替换等操作，同时在操作有序顺序表时，还要对数据的递增性进行检查。**

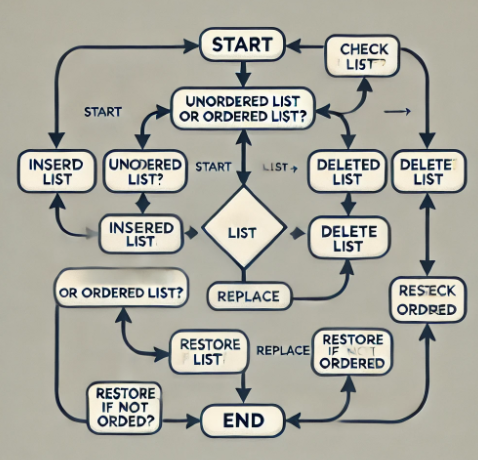
**主函数的主要流程如下：**

**步骤 1: 用户选择要操作的顺序表类型：无序表（list1）或有序表（list2）。**

**步骤 2: 进入功能选择菜单，用户可以选择插入、删除、替换等操作。**

**步骤 3: 针对有序表，在每次插入或替换后，检查是否依然符合递增要求。如果不符合，恢复之前的状态。**

**主函数流程图**

****

**2. 关键算法设计**

**（1）有序顺序表的插入算法 OrderedAdd()**

**这是一个关键算法，用于保证新插入的元素依然符合递增顺序。算法需要遍历当前的数组，找到合适的位置插入新的元素，以保持整个数组的有序性。**

**算法步骤:**

**1.输入要插入的值。**

**2.遍历当前数组，找到第一个比新值大的位置，在此位置前插入新值。**

**3.若未找到比新值大的元素，则将新值插入数组尾部。**

**流程：**

**A[开始] --> B[输入待插入值n];**

**B --> C{data是否为空？};**

**C -- 是 --> D[插入n到数组];**

**C -- 否 --> E[遍历数组，找到比n大的元素];**

**E --> F{是否找到比n大的元素？};**

**F -- 是 --> G[在该位置插入n];**

**F -- 否 --> H[将n插入数组尾部];**

**H --> I[结束];**

**G --> I;**

**D --> I;**

**（2）恢复原始状态 reuse2()**

**在list2类中，有一个设计难点，即如何在操作不符合递增顺序时恢复数组的原始状态。为此，设计了两个函数 reuse1() 和 reuse2()，分别保存和恢复原始状态。关键点在于保证每次操作前将数据备份，并在不符合递增顺序时恢复。**

**流程：**

**A[开始] --> B[将当前data备份到ans];**

**B --> C[用户操作：插入或修改];**

**C --> D{操作后是否符合递增？};**

**D -- 是 --> E[操作成功];**

**D -- 否 --> F[恢复data为ans];**

**E --> G[结束];**

**F --> G[提示用户重新输入];**

**3. 难点分析**

**（1）顺序插入的实现难点**

**对于有序顺序表，顺序插入的最大难点是如何在 O(n) 时间内找到插入位置并保证有序性。若数组较大，则插入操作的效率直接受限于数组的大小。可以考虑使用二分查找优化插入位置的查找，但这需要增加算法复杂度。**

**（2）数据恢复的逻辑**

**另一个难点在于如何在用户操作不符合要求时恢复操作前的状态。为此需要设计数据备份与恢复机制。刚开始想要用额外的一个数组来保存更改前的状态，但是出了些问题，所以后来改成在更改前判断是否合适，不合适则取消改动，这样大大减轻代码量和复杂程度。**

**题目2.**

**1. 主函数设计**

**主函数的主要流程如下：**

**1.用户选择链表类型：程序开始时，用户会根据提示选择链表类型（单链表、单向循环链表、双向链表）。**

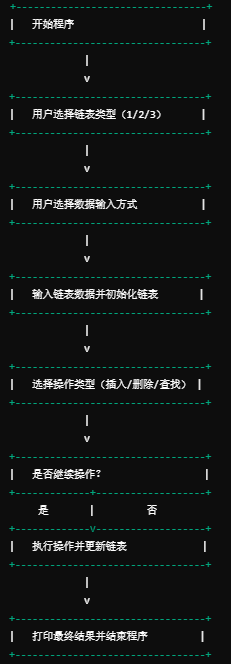
**2.用户输入数据：接着用户选择数据输入的方式，输入数组或一行数据作为链表的初始值。**

**3.执行链表操作：用户可以选择对链表进行插入、删除、查找值等操作。程序根据输入的操作选择相应的方法调用。**

**4.循环执行操作：在每次操作后，用户可以选择是否继续进行操作或结束。**

**5.打印最终结果：如果用户选择结束操作，则打印链表的最终状态，并释放内存资源。**

**主函数流程图**

****

**2. 关键算法设计**

**(1)链表插入操作:**

**单链表的插入需要找到插入点后，将新节点的指针指向下一个节点，并将前一节点的指针指向新节点。**

**单向循环链表在插入时，需要考虑尾部节点的环形结构，确保链表末尾的节点始终指向头部。**

**双向链表的插入需处理前后指针的更新，使新节点能被正确遍历。**

**(2)链表删除操作:**

**单链表删除时，需要定位到待删除节点的前一个节点，并将其指针跳过待删除节点，指向下一个节点。**

**单向循环链表需要特别处理尾节点的删除情况，确保尾节点始终指向头节点。**

**双向链表删除时，需要维护前后节点的指针关系，确保删除节点后链表仍然可双向遍历。**

**(3)链表查找操作:**

**查找操作根据索引或值遍历链表，单链表和单向循环链表均从头开始遍历。双向链表可以从任意节点进行双向查找，提高效率。**

**3. 难点分析**

**(1)循环链表的处理:**

**在单向循环链表中，确保尾部节点始终指向头部是一大难点。无论是插入还是删除操作，都需要特别处理尾部节点的指针更新，避免链表断裂。**

**(2)双向链表的双指针维护:**

**双向链表的插入和删除涉及双向指针的维护，这需要确保每个节点的前后指针在任意操作后都能正确指向邻近的节点。尤其是在链表头部和尾部的插入、删除操作中，需要特别注意。**

**(3)数据输入的处理:**

**程序允许用户通过数组或一行数据输入链表元素，尤其在输入数组时，需要解析用户输入的字符串，确保正确地分割元素并存入链表。**

**(4)边界问题的处理：**

**在遇到实现功能时插入的位置或者查找的值越界或不存在时，往往需要设立判断条件以完善功能，同时在运行中发现收尾值的调用往往被忽视，由于判断条件没有考虑到头结点和尾节点，需要细致查找并设立完善的边界条件和多元的处理方式。**

**题目3.**

**（1）主函数流程**

**主函数的大致流程如下：**

**1.初始化扑克牌链表，读取扑克牌数据文件。**

**2.按花色将扑克牌插入相应链表。**

**3.通过用户菜单选择功能进行操作。**

**4.最终输出剩余牌并释放链表内存。**

**（2）关键算法设计**

**1. 扑克牌的插入：使用单链表按升序插入牌，保持链表的有序性。**

**2. 牌的删除：遍历链表，删除指定牌值，若找到多张相同牌，根据用户输入的数量逐一删除。**

**3. 统计牌数：遍历链表，统计每个花色的牌数，用于拼凑完整扑克牌套数。**

**（3）难点分析**

**1.如何高效删除牌： 遍历链表时需考虑链表结构的变化，确保删除节点时不会导致链表断裂。**

**2.内存管理： 由于动态分配链表节点，删除牌后必须及时释放内存，避免内存泄漏。**

**3.如何正确删除牌：编写算法时最大的难点是如何做到删除牌不会删除多，经常编译运行发现把所有牌都删除了，所以花了大量精力编写函数框定范围终于解决。**

## 四、实验结果

**题目1.**

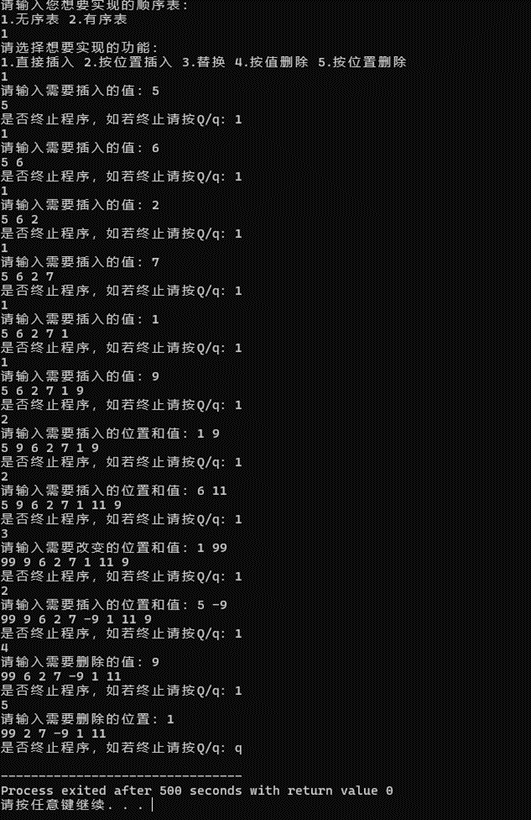
****

图 1无序表功能演示

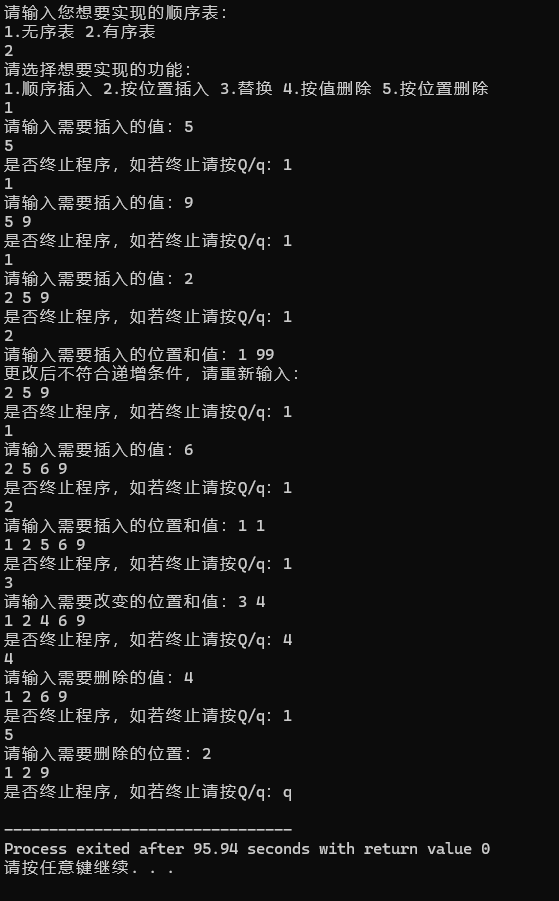


图 2有序表功能演示

**题目2.**

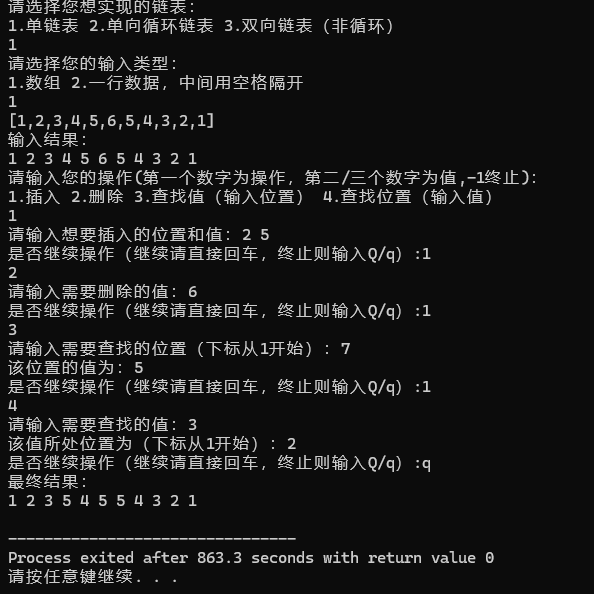
****

图1 这是数组输入的单链表功能展示

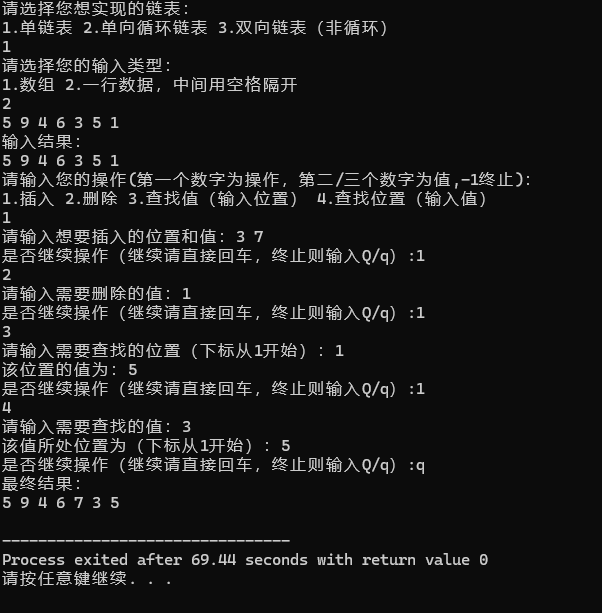
****

图2 这是一行数据输入的单链表功能展示

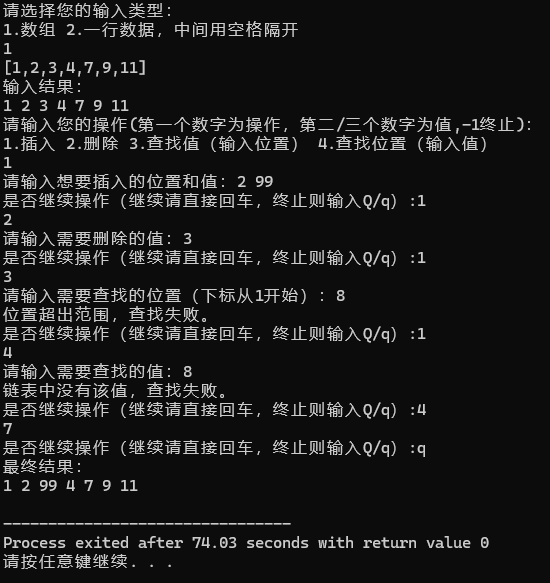
****

图1 这是单向循环链表的功能演示

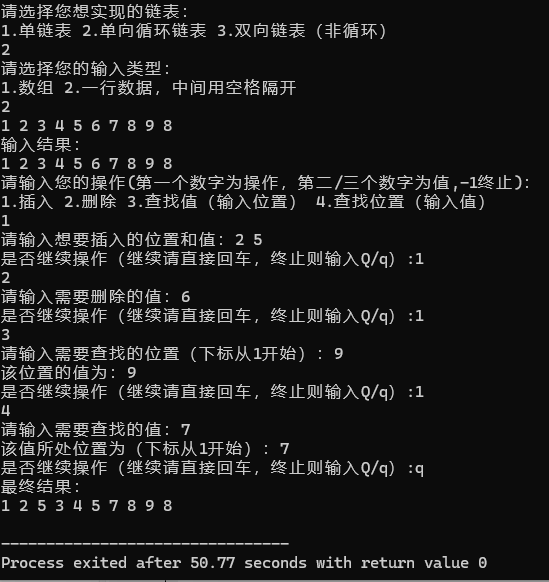


图2 这也是单向循环链表的部分

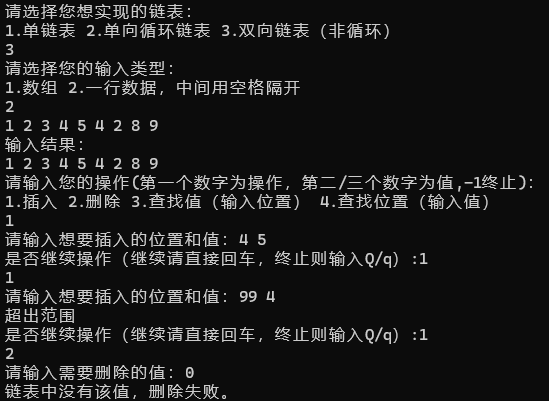


图1 双向链表的功能和前面差不多，这里就列举了一些输入超出边界的判定样例

**题目3.**

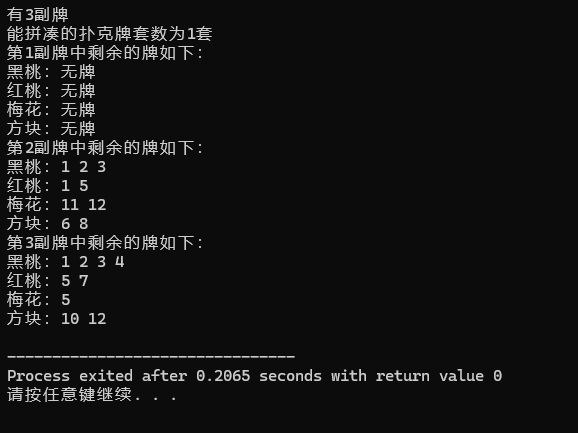
****

图1 运行直接用了一个凑好的文件数据，正好符合条件，这里是结果

## 五、实验分析与探讨

**题目1.**

**1.测试结果的分析**

**功能测试：通过手动输入数据，验证每个函数的正确性和功能。无论是顺序表list1的基本操作，还是有序表list2的插入和删除操作，都逐步进行测试。**

**边界条件测试：考虑到边界输入的特殊情况，例如空表、表头插入、表尾插入等。在list2中，还重点测试了插入元素后是否保持递增顺序的逻辑。**

**错误处理测试：模拟用户输入错误，检查程序是否能正确处理异常情况（如删除不存在的值，插入超出范围的位置等）。**

**2.结果分析**

**功能正常：在各类输入下，程序都能正常完成功能，包括插入、删除、替换和打印。尤其是在list2中，用户修改了表数据后，如果数据不再递增，能有效恢复之前的状态。**

**3.时空复杂度分析：**

**顺序表（无序）list1：**

**插入：时间复杂度为O(n)，需要在某个位置插入时需要移动后续元素。**

**删除：按值删除，最坏情况是遍历整个表，复杂度为O(n)。**

**空间复杂度：使用vector，空间动态扩展，最坏情况为O(n)。**

**有序表list2：**

**顺序插入：由于要保持递增，插入时需要遍历找到合适位置，时间复杂度为O(n)。**

**判定是否有序：遍历整个表，时间复杂度为O(n)。**

**空间复杂度：同样使用vector，复杂度为O(n)。**

**4.性能分析结论**

**对于小规模数据，该程序性能表现良好，操作较为简单高效。然而，随着数据量增大，插入、删除操作的时间消耗会增加。特别是在list2中，若频繁插入和判断有序性，性能损耗较大。**

**5.优化空间复杂度**

**可以考虑将vector换成链表结构，避免频繁的元素移动。在无序表中，这将大幅提升插入和删除的效率。**

**对于有序表，除了链表之外，平衡二叉树（如AVL树）也可以保持有序性并提高效率。**

**6.遇到的问题及解决方案**

**（1）恢复状态的问题**

**问题1：reuse1和reuse2函数在按值恢复时，无法恢复正确的data状态，导致数据被清空。**

**解决1：通过传递引用而不是值来保存状态，保证数据的实际修改，而不是在函数局部创建一个临时拷贝。使用vector<int>&解决这个问题。然而，同时我发现可以在修改数据之前对这个进行判断，如果不递增，则直接忽略，进行下一次输入，一方面大大减少了代码量，另一方面优化了代码的可读性。**

**（2）输入范围的限制**

**问题2：用户可能输入超出范围的插入位置或删除位置。**

**解决3：增加更多的输入边界检测与提示机制。例如，在插入和删除函数中，检测位置是否在允许范围内，防止异常操作。**

**（3）重复元素的处理**

**问题3：在list2中，有可能插入相同的元素，导致判断递增有误。**

**解决3：明确策略，可以规定是否允许重复元素，如果允许，则对相同元素的处理逻辑需要设计得更加严格。**

**题目2.**

**1. 测试结果分析**

**本实验实现了单链表、单向循环链表和双向链表的基本操作，包括插入、删除、查找等功能。通过用户交互选择链表类型和输入方式，程序实现了不同数据结构的创建与操作，验证了不同链表在操作效率和适用场景中的特性。**

**主要功能测试结果：**

**单链表：通过数组或空格分隔的数据输入，链表能够顺利创建并进行操作。插入和删除功能在多次测试中表现稳定，能正确处理各种边界条件。**

**单向循环链表：实现了对链表的循环遍历，确保插入和删除操作不会导致链表的断裂，能有效处理循环特性。**

**双向链表：通过双向指针的使用，删除操作能更灵活地处理节点，尤其是对尾部节点的操作，减少了复杂度。**

**2. 算法时空复杂度分析**

**时间复杂度：**

**插入操作：对于所有类型的链表，插入操作的复杂度均为 O(n)，因为在最坏情况下需要遍历整个链表找到插入位置。**

**删除操作：与插入相似，删除操作的复杂度也是 O(n)，尤其是在删除特定节点时，需要找到该节点。**

**查找操作：查找值或位置的复杂度同样为 O(n)，需要遍历链表直到找到所需的值或位置。**

**空间复杂度：**

**所有链表均为 O(n)，其中 n 为链表中节点的数量。每个节点需要存储数值和指向下一个（或前一个）节点的指针，空间占用随着节点数量的增加而线性增长。**

**3. 解决问题的其他途径**

**在实现链表操作时，有多种方式可以优化程序或简化代码结构：**

**1.使用 STL 容器：C++ 的标准模板库（STL）提供了如 std::list 和 std::vector 等容器，能够简化链表的实现与操作。STL 容器提供了更高效的内存管理与更丰富的操作接口。**

**2.结合智能指针：在链表节点的动态内存分配中，使用智能指针（如 std::shared\_ptr 和 std::unique\_ptr）可以有效管理内存，减少内存泄漏的风险。**

**4. 实验过程中遇到的问题**

**1. 输入处理问题： 在处理不同输入格式时，用户输入的不一致性可能导致程序崩溃或异常。为此，增加了输入验证机制，确保在插入数据前对输入格式进行严格检查。**

**2. 循环链表的特殊性： 在实现单向循环链表时，初期的实现未能正确处理链表的结束条件，导致出现无限循环。通过调试程序，对循环链表的操作进行了详细测试，确保指针操作的正确性。**

**3. 边界条件的处理： 链表操作中，特别是在插入和删除操作时，边界条件（如空链表或只有一个节点的链表）需要特别关注。通过增设判断逻辑，解决了在边界条件下可能出现的崩溃问题。**

**4.对知识的理解匮乏：对于动态数组的引用方法不够成熟，往往直接更改大的方面，这样往往会报错。**

**5. 进一步探讨**

**本实验通过实现不同类型的链表，帮助理解链表的基本原理和操作。然而，链表的实现依然存在改进空间。在未来的研究中，可以考虑以下几个方面：**

**1.多种数据结构的比较：深入分析链表与其他数据结构（如数组、哈希表、树等）的优劣，特别是在特定应用场景下的性能表现。**

**2.算法优化：探讨改进链表操作性能的算法，例如平衡树或跳表，提升在大数据量情况下的操作效率。**

**题目3.**

**1. 测试结果分析**

**在本实验中，程序使用单链表对扑克牌的插入、删除、统计、拼凑等操作进行了模拟与实现。通过多次测试，不同副数、不同花色的扑克牌均能正确插入链表并按升序排列，表明链表操作正确且稳定。**

**程序能够准确统计各副牌中每种花色的牌数，并正确地从多副牌中移除能够拼凑出的完整扑克牌套数，测试中得出的结果与预期相符。**

**2.主要性能分析：**

**插入操作：由于链表中的牌值始终保持升序排列，插入操作需要遍历链表直到找到合适的位置，复杂度为 O(n)，其中 n 为链表长度。在实际测试中，即使是多副牌的数据插入，性能表现良好，处理效率较高。**

**删除操作：删除操作同样需要遍历链表，找到指定牌值并删除相应数量的牌。删除过程的复杂度与插入类似，为 O(n)，但由于每次删除一张牌后需重新连接链表，时间复杂度较插入稍高。在多副牌的情况下，删除操作仍然稳定执行，没有明显的性能瓶颈。**

**3. 算法时空复杂度分析**

**时间复杂度：**

**初始化链表：对每副扑克牌初始化链表的时间复杂度为 O(1)，因为仅仅是创建空链表。**

**插入牌值：每张牌的插入操作需要遍历链表，时间复杂度为 O(n)，其中 n 为链表长度。假设总共有 m 副牌，每副牌有 k 张，那么插入所有牌的复杂度约为 O(m \* k)。**

**删除牌值：删除操作与插入类似，遍历链表查找并删除指定数量的牌，时间复杂度同样为 O(n)。**

**拼凑完整牌套：拼凑完整扑克牌套数的复杂度主要取决于统计牌数的过程，复杂度为 O(n)，之后按花色逐一删除每张牌，整体复杂度与插入和删除操作相近。**

**空间复杂度：**

**链表结构：每副牌使用单链表存储，其中每张牌的节点需要存储牌值和指针，总体空间复杂度为 O(m \* k)，即与扑克牌数量成线性关系。**

**4. 解决问题的其他途径**

**除了单链表的实现外，本实验中有其他潜在的优化方案：**

**1.使用平衡二叉搜索树：相比于链表，平衡二叉树可以在 O(log n) 时间内完成插入、删除和查找操作。如果扑克牌副数较多且链表较长，使用树结构可以提高效率，特别是在频繁插入和删除牌的场景中。**

**2.使用哈希表：哈希表可以实现 O(1) 的查找和删除操作。可以为每种牌值设置一个哈希表，将扑克牌分类存储为哈希表的键值对（牌值为键，出现次数为值）。这样可以大幅提升删除和查找的效率。**

**3.使用数组或向量：若牌的副数固定，且需要频繁查找和删除某个具体的牌值，数组或向量的随机访问速度快，可以作为链表的替代方案。特别是在需要按索引高效删除和插入数据时，向量的表现比链表好。**

**5. 实验过程中遇到的问题**

**1. 文件读取问题： 实验初期，由于扑克牌数据从文件读取，有时文件格式不一致导致读取出错。为此，采用了更加健壮的文件读取机制，确保文件的每一行数据都能正确解析。**

**2. 链表的边界问题： 链表在处理插入和删除操作时，若处理不当可能导致链表断裂或指针错误。在程序调试过程中，通过对链表的边界情况进行详细检查（如空链表、首尾节点的插入和删除），最终解决了此问题。**

**3. 编译器局限性问题：由于用的是dev-c++，所以一开始用的项目做的分.h和.cpp文件后对于链表节点的处理包括引用之类的都常常会报错，所以最后都放到一个文件里写。**

**4.删除多余牌的问题：开始时函数设计不恰当，使得每次都将所有牌要么都保留要么都删除，后来设了好多边界条件，多使用了两个函数才成功。**

**5. 进一步探讨**

**实验过程中，通过单链表的实现，我们了解了如何利用链表结构来存储、处理数据，并且通过实验验证了链表在扑克牌操作中的有效性。但链表在处理大量数据时，插入和删除操作的效率并不高。因此，未来可以尝试其他数据结构如哈希表、二叉搜索树等，以提高程序的整体性能。**

## 六、小结

**本次实验涵盖了顺序表、链表、以及扑克牌模拟这三类数据结构的设计与实现。通过功能测试、边界条件处理、以及性能分析，项目实现了各类基础操作如插入、删除、查找和排序等，帮助理解了顺序表与链表在不同场景中的适用性和效率差异。**

**完成情况：所有必做部分均已实现，顺序表与链表的基本操作测试通过，扑克牌程序准确完成牌组管理和统计功能。代码对边界条件进行了完善处理，确保功能的正确性和健壮性。**

**局限性：在链表操作的时间复杂度和扑克牌模拟的处理效率上仍有改进空间，尤其是对大数据量的处理效率较低。**

**感想与收获：通过这次实验，深刻理解了不同数据结构的应用场景和优缺点，特别是链表与顺序表在插入和删除上的效率差异。同时，在程序设计中遇到了不少调试问题，尤其是输入处理和边界条件，进一步锻炼了问题解决能力。**

**未来方向：可以通过引入更高级的数据结构（如平衡树、跳表等）进一步优化程序性能，并尝试多种编程语言及库（如STL）。同时，还可以优化输入处理机制，使程序更加鲁棒。**

## 附录：源代码

**代码见附件**