北京郵電大學

实验报告

题目：实验一：链表-栈-队列综合应用实验

班　　级：2023211XXX

学　　号：2023XXXXXXX

姓　　名：Yokumi

学　　院：计算机学院

2024 年 10 月 20 日

一、实验目的

1、 练习链表的实现及操作，学习用链表解决实际问题；

2、 练习栈与队列的实现及操作，学习用栈与队列解决实际问题；

3、 锻炼程序编写及调试的能力；

4、 学习自己查找相关资料以解决实际问题的能力。

二、实验环境

操作系统为 MacOS 14.5 (23F79)，处理器架构为 arm64。C 标准采用 c17，编译器为 clang-1500.3.9.4。

```
(base)  yokumi@YokumideMacBook-Air  ~  sw_vers
ProductName:            macOS
ProductVersion:         14.5
BuildVersion:           23F79
```

图 1 操作系统

```
(base)  yokumi@YokumideMacBook-Air  ~  clang --version
Apple clang version 15.0.0 (clang-1500.3.9.4)
Target: arm64-apple-darwin23.5.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

图 2 编译器环境

三、实验内容

场景设定：

假设有一副扑克牌，用两位字符来表示扑克牌的牌面，第一位字符用 A/B/C/D 代表四种花色，第二位字符用 1……9、0(代表 10)、J、Q、K 表示数值，用 XX 和 YY 代表大小王。初始扑克牌的序列为：A1、A2、……、AK、B1、B2……、DK、XX、YY。

要求：

- 将代表 54 张扑克牌的初始序列放入到顺序表中。
- 洗牌：将 54 张扑克牌打乱顺序，可以调研学习一下洗牌算法（Fisher-Yates Shuffle 算法或 Knuth-Durstenfeld Shuffle 算法）
- 插入循环链表：依次将洗牌后的元素从顺序表中取出，插入到一个循环链表中。
- 定位：在循环链表中定位到第一个出现的大王或小王牌。

- 报数+出牌：从上一步定位到牌开始从 1 计数，当计数到 M（预先设定的一个数字）时，将牌打出。然后从下一张牌重新开始从 1 计数，直至剩余一张牌时游戏结束，请输出最后剩余的一张牌。（约瑟夫环问题）
- 到游戏结束时，最后被打出的五张 A 开头的牌是什么？（可使用队列记录最多五张打出的 A 开头的牌）
- **附加要求**：假如当打出的牌是大小王时，可以取回最多五张（不足五张时全部取走，超过五张时取五张）最近打出的牌并插入到循环链表中，那么最后剩余的一张牌以及最后被打出的五张 A 开头的牌又是什么。（可使用栈记录打出的牌）

四、实验步骤（基础 80 分+附加 10 分）

4.1 高层数据结构设计

4.1.1 顺序表

顺序表，是在计算机内存中以数组的形式保存的线性表，是指用一组地址连续的存储单元依次存储数据元素的线性结构。

由于自定义二维数组储存牌堆信息较为烦琐，效率较低，本实验直接采用 C 语言中的二维数组实现顺序表结构如下：

```
// 定义顺序表（二维数组）储存牌堆并进行初始化
char UnshuffledCards[54][3];
Generate_UnshuffledCards(UnshuffledCards);
```

初始化顺序表的伪代码如下：

```
// 初始化未打乱的牌堆
函数 Generate_UnshuffledCards(二维字符数组 UnshuffledCards)
    定义整型变量 k = 0
    定义字符数组 values = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '0', 'J', 'Q', 'K']

    // 遍历花色 (4 种花色)
    对于 i 从 0 到 3:
        // 遍历数值 (13 张牌)
        对于 j 从 0 到 12:
            // 设置牌的花色
            UnshuffledCards[k][0] = 'A' + i （转换为字符）
            // 设置牌的数值
            UnshuffledCards[k][1] = values[j]
            // 添加字符串结束符
```

```
            UnshuffledCards[k][2] = '\0'
            // 递增 k
            k++

    // 单独处理大王和小王
    复制字符串 "XX" 到 UnshuffledCards[52]
    复制字符串 "YY" 到 UnshuffledCards[53]

    返回
结束函数
```

打印顺序表的伪代码如下：

```
// 打印顺序表储存的牌堆
函数 Print_Cards(二维字符数组 UnshuffledCards)
    // 遍历所有 54 张牌
    对于 i 从 0 到 53:
        打印 第 i 张牌

    打印换行符
结束函数
```

顺序表还涉及查找特定元素等操作，由于较为简单且本次实验中无需使用，故函数定义略。

4.1.2 循环链表

循环链表是一种链式存储结构，它的最后一个结点指向头结点，形成一个环。因此，从循环链表中的任何一个结点出发都能找到任何其他结点。

本实验中数据结构的元素节点定义如下：

```
// 记录牌堆信息的结构体节点
typedef struct PokerCard
{
    /*
    * char Data[3]，Data[0]表示花色，Data[1]表示数值，Data[2]表示结束符'\0'，若两位前两位都为X(Y)则表示大王(小王)
    * PtrToCard Next，指向下一张牌的指针
    */
    char Data[3];
    PtrToCard Next;
}Card;
```

链表的结构体定义如下：

```
// 以链表结构储存的牌堆
```

```
2. typedef struct
3. {
4.     PtrToCard Head;
5.     PtrToCard Tail; // 仅用于当计数 M = 1 时，不再遍历一遍找到头节点的前面一个节点。
6. }Link;
```

链表的基本操作包括初始化、遍历、插入、删除等，本实验中这几种操作都有涉及，分别体现在记录牌堆、查找特定牌的位置、摸牌、出牌过程。

核心代码位于附录中，这里仅展示插入的操作，思路是创建一个新的元素节点，将其的 Next 指针指向插入位置前的节点的 Next，再将插入位置前的 Next 指针指向它，注意顺序不能调换。

```
1. // 在 Ptr 后插入新节点
2. void InsertToLink(LinkDeck linkDeck, PtrToCard Ptr, char data[3])
3. {
4.     PtrToCard Temp = (PtrToCard)malloc(sizeof(Card));
5.     strcpy(Temp->Data, data);
6.     Temp->Next = Ptr->Next;
7.     Ptr->Next = Temp;
8. }
```

4.1.3 链队

队列的核心思想就是先进先出，只允许在队首 Front 出队，在队尾 Rear 入队。在具体的实现方式上，本实验采用链表实现，这样可以和链表共用同一种元素节点（即都包括 Data 和 Next），其中队列的 Next 指针和链表类似，指向下一个元素节点。结构体定义如下：

```
1. // 以队列结构储存的牌堆
2. typedef struct
3. {
4.     PtrToCard Front; // 队首元素
5.     PtrToCard Rear; // 队尾元素
6. }Quene;
```

本实验中队列的主要操作包括入队，先打出的牌记录在队首。具体实现见第六部分。

4.1.4 链栈

栈的核心思想是先进后出，后进先出，即只允许在栈顶进行入栈和出栈操作。在具体的实现方式上，本实验也采用链表实现，理由同上。区别是链栈中，节点的 Next 指针方向为从栈顶指向栈底。结构体定义如下：

```
1. // 以链栈结构储存的牌堆
2. typedef struct
```

```
3. {
4.     PtrToCard Top; // 栈顶元素
5. }Stack;
```

本实验中栈的主要操作包括入栈和出栈，分别对应将打出的牌放入弃牌堆和从弃牌堆摸牌的操作，此时弃牌堆符合一般认知，即“从下到上”累计打出的牌。具体实现见第六部分。

4.2 高层算法设计

4.2.1 洗牌算法

主要的洗牌算法有 Fisher-Yates Shuffle 算法和 Knuth-Durstenfeld Shuffle 算法等。

Fisher-Yates Shuffle 算法是一种对有限序列进行洗牌的算法。该算法获取序列中所有元素的列表，并通过从列表中随机抽取元素来连续确定洗牌序列中的下一个元素，直到没有元素剩余。可以证明，该算法产生无偏排列：每个排列的可能性均等。其步骤如下：

1. 写下从 1 到 N 的数字。
2. 在 1 和剩余未划掉的数字之间选取一个随机数 k 。
3. 从前开始数，划掉第 k 个还未划掉的数字，并将其写在一个单独列表的末尾。
4. 从步骤 2 开始重复，直到所有数字均被划掉。
5. 步骤 3 中写下的数字序列现在是原始数字的随机排列。

Knuth-Durstenfeld Shuffle 算法是在该算法基础上的优化，Fisher-Yates Shuffle 算法中，每次需要遍历找出剩余的数字，而 Knuth-Durstenfeld Shuffle 算法将每次被划掉的数字移动到原表尾，即将每次选中的数字和最后一个未划掉的数字交换位置，将时间复杂度从 $O(n^2)$降至 $O(n)$。其伪代码如下：

```
1. // Knuth-Durstenfeld Shuffle 算法洗牌
2. 函数 Knuth_Durstenfeld_Shuffle(二维字符数组 UnshuffledCards)
3.     初始化随机数种子
4.
5.     // 从牌堆的最后一张牌开始遍历
6.     对于 i 从 53 到 1：
7.         // 从 [0, i] 范围内随机选取一个索引 j
8.         生成一个随机数 j，使得 0 ≤ j ≤ i
9.
10.        交换 UnshuffledCards[i] 和 UnshuffledCards[j]
11. 结束函数
```

本实验中，采用效率更高的 Knuth-Durstenfeld Shuffle 算法。

4.3 具体实验内容实现及运行结果

4.3.1 初始化扑克牌序列并放入顺序表

初始化牌堆的运行结果如下：

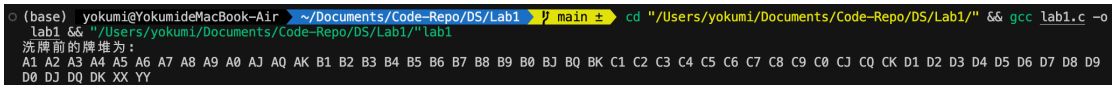

图 3 要求一运行结果

4.3.2 洗牌

采用 Knuth-Durstenfeld Shuffle 算法得到洗牌后的顺序表如下，随机数种子为 0：

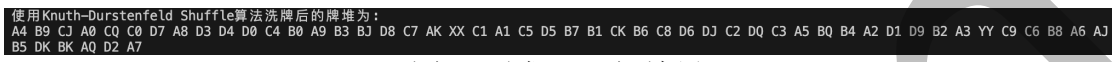

图 4 要求二运行结果

4.3.3 插入循环链表

源代码中通过 Insert2CircularLinkedList()函数，实现将洗牌后的元素依次从顺序表中取出，插入到一个循环链表 LinkDeck 中，从头节点开始打印插入完成的循环列表运行结果如下：

图 5 要求三运行结果

对比结果可见，元素顺序插入无误。

4.3.4 定位

本人设计了 FindCardinDeck(LinkDeck linkDeck, char card[3])函数，通过遍历链表来查找任何一张牌在链表中的位置（从牌堆顶开始数，牌堆顶为第一张牌），同时也设计了 FindXYinDeck(LinkDeck linkDeck)完成要求四中查找第一次出现的大王或小王牌的特定功能，运行结果如下：

图 6 要求四运行结果

通过和上面链表的内容进行对比，结果无误。源代码中在 FindXYinDeck()函数中同时将循环链表的头节点更新为第一次查找到的大王牌或小王牌，以便继续进行之后的操作。

4.3.5 报数+出牌

该步骤的思想是一个很经典的约瑟夫环问题：有 n 个人围成一个圆圈，从第一个人开始报数，每次报到第 m 个的人出局，然后从下一个人重新开始报数。这个过程持续进行，直到所有人都被淘汰。问题是，给定 n 和 m，确定最终存活下来的人的位置。

本实验可用循环链表实现。循环链表的元素节点成环，且便于插入、删除元

素。

通过链队储存打出的 A 开头的牌，并运行程序，得到运行结果如下：（计数上限 M = 7 的情况下）

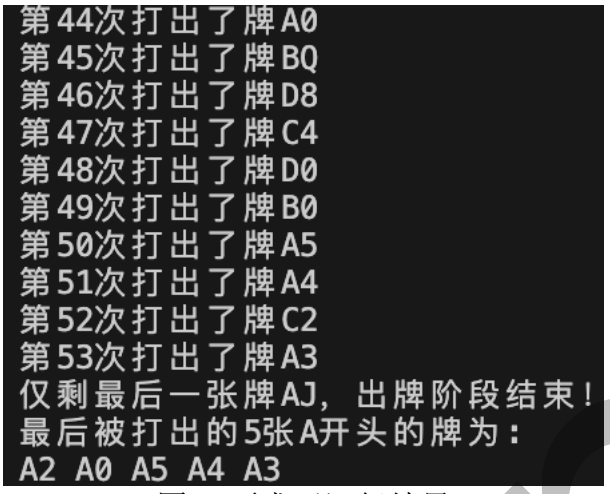

图 7 要求五运行结果

可以通过模拟过程进行验证，结果无误。

4.3.6 附加要求：增设摸牌机制

本实验增加在摸到大小王时，可以取回最多 5 张最近打出的牌插入到循环链表的尾部的机制。特别注意的是，不能将大王或小王牌，否则可能产生死循环的情况。

为处理这种情况，本实验用栈存放打出的牌，将先打出的牌置于栈底，最后打出的牌置于栈顶，出牌和摸牌都在栈顶进行操作，符合题目要求和一般认知。其中大王和小王牌都不入栈。其中摸牌部分的实现思想包括插入链表（尾插）和出栈，储存打出的牌即为入栈操作。

运行结果如下：（计数上限 M = 7 的情况下，同上）

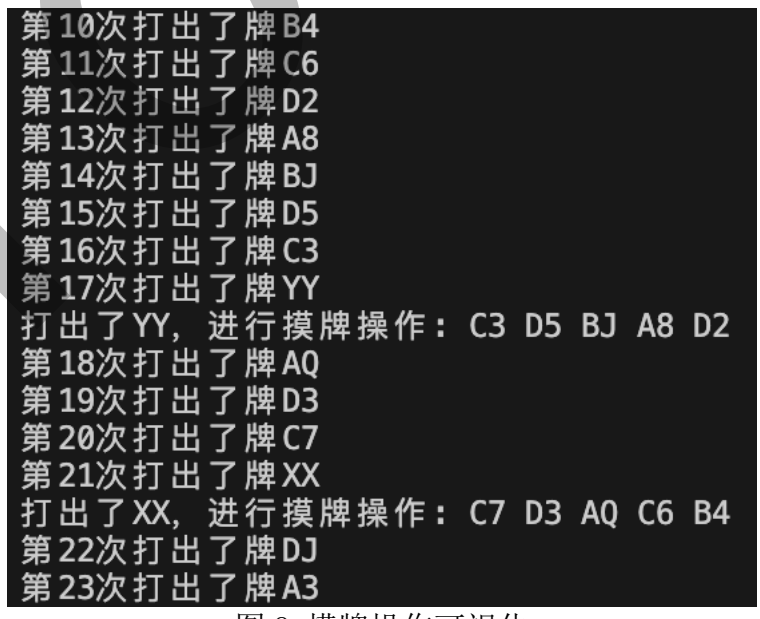

图 8 摸牌操作可视化

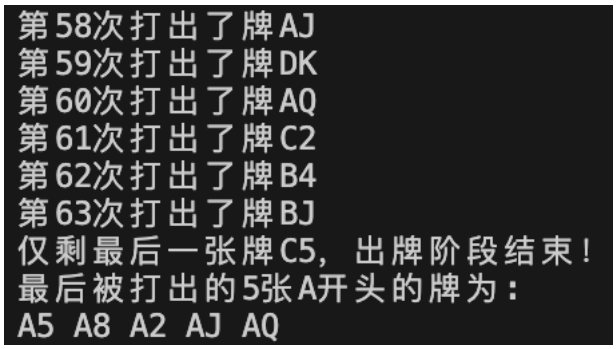

图 9 附加要求运行结果

通过模拟过程验证摸牌和最终运行结果均无误。

五、实验分析和总结（10 分）

5.1 工作思路及效果分析

本实验基于扑克牌场景，通过顺序表、链表、队列、栈等多种数据结构，模拟了洗牌、出牌、摸牌等过程，有效地结合了链表、栈和队列的特点，充分展现了每种数据结构在不同场景下的优势。通过实际编程实现，也锻炼了对这些结构的理解和操作能力，包括链表的初始化、插入、查找、删除，栈的初始化、入栈及出栈，队列的初始化、入队及出队。

本实验操作高效，比如通过循环链表模拟了约瑟夫环问题，链表节点插入和删除操作灵活，使得问题的解决变得直观高效。

通过学习 Knuth-Durstenfeld 算法并应用到洗牌过程中，保证了洗牌的随机性，且时间复杂度低。

5.2 遇到困难与心得

本实验中，由于包含链表结构，且队列和栈也基于链表来实现，涉及大量的指针操作，插入、删除等操作涉及的指针变换和指针内存释放均已严格检查，避免内存泄露或产生空指针、野指针的情况。

不过在编写代码的过程中，遇到了一个之前从未见过的报错信息如下：

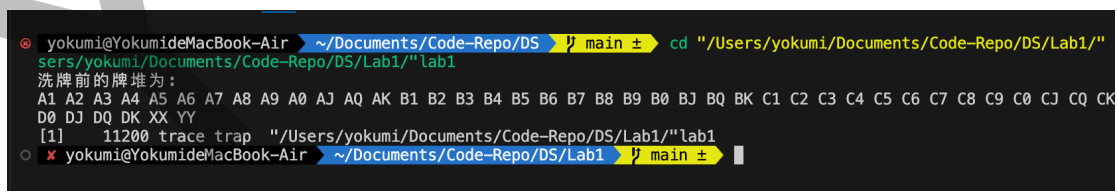

图 10 trace trap 报错

查阅搜索引擎后也几乎没有这方面的类似信息，于是尝试分别在 Windows 系统和 Linux 系统下分别运行都没有产生类似问题，猜测可能是 Mac 的 arm 处理器架构差异导致。

于是进行调试，操作如下：

```
1. cc -g lab1.c
```

```
2. $ lldb a.out
3. (lldb) target create "a.out"
4. (lldb) run
```

运行后程序仍在该位置中断，如下：

```
洗牌前的牌堆为：
A1 A2 A3 A4 A5 A6 A7 A8 A9 A0 AJ AQ AK B1 B2 B3 B4 B5 B6 B7 B8 B9 B0 BJ BQ BK C1 C2 C3 C4 C5 C6 C7 C8 C9 C0 CJ CQ
D0 DJ DQ DK XX YY
Process 13278 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BREAKPOINT (code=1, subcode=0x19ae15e60)
    frame #0: 0x000000019ae15e60 libsystem_c.dylib`__chk_fail_overlap + 24
libsystem_c.dylib`:
->  0x19ae15e60 <+24>: brk    #0x1

libsystem_c.dylib`:
    0x19ae15e64 <+0>:  pacibsp
    0x19ae15e68 <+4>:  stp    x22, x21, [sp, #-0x30]!
    0x19ae15e6c <+8>:  stp    x20, x19, [sp, #0x10]
Target 0: (a.out) stopped.
```

图 11 调试过程

再次通过以下命令查看堆栈回溯：

```
1. (lldb) bt
```

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BREAKPOINT (code=1, subcode=0x19ae15e60)
  * frame #0: 0x000000019ae15e60 libsystem_c.dylib`__chk_fail_overlap + 24
    frame #1: 0x000000019ada69e0 libsystem_c.dylib`__chk_overlap + 48
    frame #2: 0x000000019adae6b0 libsystem_c.dylib`__strcpy_chk + 84
    frame #3: 0x0000000100003ba8 a.out`Knuth_Durstenfeld_Shuffle(UnshuffledCards=0x000000016fdfe936) at lab1.c:65:9
    frame #4: 0x0000000100003e30 a.out`main at lab1.c:119:5
    frame #5: 0x000000019ab820e0 dyld`start + 2360
```

图 12 堆栈回溯

发现是大概率是 strcpy()的问题，此处位于我的洗牌算法交换两张牌的位置处。原来是 strcpy()会将字符串复制到其自身之上，即自己和自己复制，造成内存重叠，而经过实测，arm 架构显然不允许内存重叠这种安全隐患。解决方法是改成 memcpy()，问题解决。也算是又踩了一个不容易踩到的坑，收获颇丰。

六、程序源代码（10 分）

```
 1. #include <stdio.h>
 2. #include <string.h>
 3. #include <time.h>
 4. #include <stdlib.h>
 5.
 6. typedef struct PokerCard *PtrToCard; // 指向某张牌的指针
 7.
 8. // 记录牌堆信息的结构体节点
 9. typedef struct PokerCard
10. {
11.     /*
12.     * char Data[3]，Data[0]表示花色，Data[1]表示数值，Data[2]表示结束符'\0'，若两位前两位都为 X(Y)则表示大王(小王)
13.     * PtrToCard Next，指向下一张牌的指针
```

```
14.     */
15.     char Data[3];
16.     PtrToCard Next;
17. }Card;
18.
19. // 以链表结构储存的牌堆
20. typedef struct
21. {
22.     PtrToCard Head;
23.     PtrToCard Tail; // 仅用于当计数 M = 1 时，不再遍历一遍找到头节点的前面一个节点。
24. }Link;
25.
26. // 以链栈结构储存的牌堆
27. typedef struct
28. {
29.     PtrToCard Top; // 栈顶元素
30. }Stack;
31.
32. // 以队列结构储存的牌堆
33. typedef struct
34. {
35.     PtrToCard Front; // 队首元素
36.     PtrToCard Rear; // 队尾元素
37. }Quene;
38.
39. typedef Link *LinkDeck;
40. typedef Stack *StackDeck;
41. typedef Quene *QueneDeck;
42.
43. // 入队
44. void EnQuene(QueneDeck queneDeck, char data[3])
45. {
46.     if (queneDeck->Front == NULL)
47.     {
48.         queneDeck->Front = (PtrToCard)malloc(sizeof(Card));
49.         strcpy(queneDeck->Front->Data, data);
50.         queneDeck->Rear = queneDeck->Front;
51.         queneDeck->Front->Next = NULL;
52.     }
53.     else
54.     {
55.         queneDeck->Rear->Next = (PtrToCard)malloc(sizeof(Card));
56.         queneDeck->Rear = queneDeck->Rear->Next;
57.         strcpy(queneDeck->Rear->Data, data);
```

```
58.         queneDeck->Rear->Next = NULL;
59.     }
60. }
61.
62. // 队首元素出队
63. void DeQuene(QueneDeck queneDeck)
64. {
65.     if (queneDeck->Front == NULL)
66.     {
67.         printf("队列为空，无法出队\n");
68.         return;
69.     }
70.
71.     PtrToCard Ptr = queneDeck->Front;
72.     if (queneDeck->Front == queneDeck->Rear) // 队列只有一张牌
73.     {
74.         queneDeck->Front = queneDeck->Rear = NULL;
75.     }
76.     else
77.     {
78.         queneDeck->Front = queneDeck->Front->Next;
79.     }
80.     free(Ptr);
81. }
82.
83. // 从队首开始打印队列元素
84. void PrintQuene(QueneDeck queneDeck)
85. {
86.     PtrToCard Ptr = queneDeck->Front;
87.     if (Ptr == NULL)
88.     {
89.         printf("队列为空，无法打印\n");
90.         return;
91.     }
92.     do {
93.         printf("%s ", Ptr->Data);
94.         Ptr = Ptr->Next;
95.     } while (Ptr != NULL);
96.     printf("\n");
97. }
98.
99. // 打印队列中最后五个元素
100. void PrintQuene_5(QueneDeck queneDeck)
101. {
```

```
102.     PtrToCard Ptr = queneDeck->Front;
103.     int count = 0;
104.     if (Ptr == NULL)
105.     {
106.         printf("队列为空，无法打印\n");
107.         return;
108.     }
109.     do {
110.         count++;
111.         Ptr = Ptr->Next;
112.     } while (Ptr != NULL);
113.     Ptr = queneDeck->Front;
114.     if (count >= 5)
115.     {
116.         int i = 0;
117.         do {
118.             i++;
119.             Ptr = Ptr->Next;
120.         } while (i < count - 5);
121. 
122.         do {
123.             printf("%s ", Ptr->Data);
124.             Ptr = Ptr->Next;
125.         } while (Ptr != NULL);
126.     }
127.     else
128.     {
129.         do {
130.             printf("%s ", Ptr->Data);
131.             Ptr = Ptr->Next;
132.         } while (Ptr != NULL);
133.     }
134.     printf("\n");
135. }
136. 
137. // 入栈
138. void EnStack(StackDeck stackDeck, char data[3])
139. {
140.     if (stackDeck->Top == NULL)
141.     {
142.         stackDeck->Top = (PtrToCard)malloc(sizeof(Card));
143.         strcpy(stackDeck->Top->Data, data);
144.         stackDeck->Top->Next = NULL;
145.     }
```

```
146.     else
147.     {
148.         PtrToCard Ptr = (PtrToCard)malloc(sizeof(Card));
149.         strcpy(Ptr->Data, data);
150.         Ptr->Next = stackDeck->Top;
151.         stackDeck->Top = Ptr;
152.     }
153. }
154.
155. // 栈顶元素出栈
156. void DeStack(StackDeck stackDeck)
157. {
158.     if (stackDeck->Top == NULL)
159.     {
160.         printf("栈为空，无法出栈\n");
161.         return;
162.     }
163.
164.     PtrToCard Ptr = stackDeck->Top;
165.     stackDeck->Top = Ptr->Next;
166.     free(Ptr);
167. }
168.
169.
170. // 初始化未打乱的牌堆
171. void Generate_UnshuffledCards(char (*UnshuffledCards)[3])
172. {
173.     int k = 0;
174.     char values[13] = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '0', 'J', 'Q', 'K'};
175.     for (int i = 0; i < 4; i++) // 遍历花色
176.     {
177.         for (int j = 0; j < 13; j++) // 遍历数值
178.         {
179.             UnshuffledCards[k][0] = (char)((int)'A' + i);
180.             UnshuffledCards[k][1] = values[j];
181.             UnshuffledCards[k][2] = '\0'; // 结束符
182.             k++;
183.         }
184.     }
185.
186.     // 单独处理大王和小王
187.     strcpy(UnshuffledCards[52], "XX");
188.     strcpy(UnshuffledCards[53], "YY");
189.     return;
```

```
}

// 打印顺序表储存的牌堆
void Print_Cards(char (*UnshuffledCards)[3])
{
    for (int i = 0; i < 54; i++)
    {
        printf("%s ", UnshuffledCards[i]);
    }
    printf("\n");
}

// Knuth-Durstenfeld Shuffle 算法洗牌
void Knuth_Durstenfeld_Shuffle(char (*UnshuffledCards)[3])
{
    srand(0);
    for (int i = 54 - 1; i > 0; i--)
    {
        // 从 [0, i] 中随机选取一个索引
        int j = rand() % (i + 1);

        // 交换 UnshuffledCards[i] 和 UnshuffledCards[j]
        char temp[3];
        memcpy(temp, UnshuffledCards[i], sizeof(temp));
        memcpy(UnshuffledCards[i], UnshuffledCards[j], sizeof(temp));
        memcpy(UnshuffledCards[j], temp, sizeof(temp));
    }
}

// 将洗牌后的元素从顺序表中取出插入循环链表
LinkDeck Insert2CircularLinkedList(char (*UnshuffledCards)[3])
{
    LinkDeck linkDeck = (LinkDeck)malloc(sizeof(Link));
    linkDeck->Head = (PtrToCard)malloc(sizeof(Card));
    if (linkDeck->Head == NULL) {
        fprintf(stderr, "内存分配失败!\n");
        exit(1);
    }
    strcpy(linkDeck->Head->Data, UnshuffledCards[0]);
    linkDeck->Head->Next = NULL;
    PtrToCard Ptr = linkDeck->Head;
    for (int i = 1; i < 54; i++)
    {
        Ptr->Next = (PtrToCard)malloc(sizeof(Card));
```

```
234.         if (Ptr->Next == NULL) {
235.             fprintf(stderr, "内存分配失败!\n");
236.             exit(1);
237.         }
238.         Ptr = Ptr->Next;
239.         strcpy(Ptr->Data, UnshuffledCards[i]);
240.         Ptr->Next = NULL;
241.     }
242.     linkDeck->Tail = Ptr;
243.     linkDeck->Tail->Next = linkDeck->Head;
244.     return linkDeck;
245. }
246.
247. // 打印循环链表
248. void PrintCircularLinkedList(LinkDeck linkDeck)
249. {
250.     PtrToCard Ptr = linkDeck->Head;
251.     do {
252.         printf("%s ", Ptr->Data);
253.         Ptr = Ptr->Next;
254.     } while (Ptr != linkDeck->Head);
255.
256.     printf("\n");
257. }
258.
259. // 在 Ptr 后插入新节点
260. void InsertToLink(LinkDeck linkDeck, PtrToCard Ptr, char data[3])
261. {
262.     PtrToCard Temp = (PtrToCard)malloc(sizeof(Card));
263.     strcpy(Temp->Data, data);
264.     Temp->Next = Ptr->Next;
265.     Ptr->Next = Temp;
266. }
267.
268. // 根据牌面查找牌堆中任意一张牌的地址，未查找到则返回 NULL
269. PtrToCard FindCardinDeck(LinkDeck linkDeck, char card[3])
270. {
271.     PtrToCard Ptr = linkDeck->Head;
272.     do {
273.         if (strcmp(Ptr->Data, card) == 0){
274.             return Ptr;
275.         }
276.         Ptr = Ptr->Next;
277.     } while (Ptr != linkDeck->Head);
```

```
278.     return NULL;
279. }
280.
281. // 查找牌堆中出现的第一张大王或小王牌
282. void FindXYinDeck(LinkDeck linkDeck)
283. {
284.     PtrToCard PrePtr = linkDeck->Tail;
285.     PtrToCard Ptr = linkDeck->Head;
286.     int count = 1;
287.     do {
288.         if (strcmp(Ptr->Data, "XX") == 0)
289.         {
290.             printf("定位到第一张出现的大王或小王牌位于从牌堆顶开始的第%d 张，此牌为大王牌
\n", count);
291.             linkDeck->Head = Ptr;
292.             linkDeck->Tail = PrePtr;
293.             return;
294.         }
295.         else if (strcmp(Ptr->Data, "YY") == 0)
296.         {
297.             printf("定位到第一张出现的大王或小王牌位于从牌堆顶开始的第%d 张，此牌为小王牌
\n", count);
298.             linkDeck->Head = Ptr;
299.             linkDeck->Tail = PrePtr;
300.             return;
301.         }
302.         count++;
303.         PrePtr = Ptr;
304.         Ptr = Ptr->Next;
305.     } while (Ptr != linkDeck->Head);
306.     return;
307. }
308.
309. // 模拟报数+出牌过程
310. void PlayingCards(LinkDeck linkDeck, int M, QueneDeck queneDeckA, StackDeck stackDeck)
311. {
312.     if (linkDeck->Head == NULL)
313.     {
314.         printf("链表为空，无法进行出牌操作。\n");
315.         return;
316.     }
317.
318.     int count = 1, i = 1;
319.     PtrToCard prePtr = linkDeck->Tail;
```

```
320.     PtrToCard curPtr = linkDeck->Head;
321.
322.     while (curPtr->Next != NULL) // 直至只剩最后一张牌
323.     {
324.         if (count < M)
325.         {
326.             prePtr = curPtr;
327.             curPtr = curPtr->Next;
328.             count++;
329.         }
330.         else // 出牌
331.         {
332.             PtrToCard temp = curPtr;
333.             if (curPtr->Next == prePtr) //处理将要打出倒数第二张牌时，最后一张牌的 Next 指
针会指向自己的特殊情况
334.             {
335.                 prePtr->Next = NULL;
336.             }
337.             else
338.             {
339.                 prePtr->Next = curPtr->Next;
340.             }
341.             count = 1;
342.
343.             printf("第%d 次打出了牌%s\n", i, curPtr->Data);
344.             i++;
345.
346.             // 附加要求：当打出大小王时，可以取回 5 张，这里将这 5 张插入打出去牌的位置处，从
弃牌堆顶摸一张插一张。
347.             if (strcmp(curPtr->Data, "XX") == 0 || strcmp(curPtr->Data, "YY") == 0)
348.             {
349.                 int j = 0;
350.                 printf("打出了%s，进行摸牌操作: ", curPtr->Data);
351.                 PtrToCard Temp = prePtr;
352.                 while (j < 5 && stackDeck->Top != NULL)
353.                 {
354.                     // 该方法原先设计时是在任意位置后插入
355.                     InsertToLink(linkDeck, linkDeck->Tail, stackDeck->Top->Data);
356.                     // 如果在尾部插入，需要更新尾指针
357.                     if (Temp == linkDeck->Tail) {
358.                         linkDeck->Tail = Temp->Next;
359.                     }
360.                     printf("%s ", stackDeck->Top->Data);
361.                     DeStack(stackDeck);
```

```
362.                     // 尾插则不需要该语句
363.                     // Temp = Temp->Next;
364.                     j++;
365.                 }
366.                 curPtr->Next = prePtr->Next;
367.                 printf("\n");
368.             }
369.
370.             if (curPtr == linkDeck->Head)
371.             {
372.                 linkDeck->Head = curPtr->Next;
373.             }
374.
375.             // 将打出的牌置于弃牌堆(除了大小王)
376.             if (strcmp(curPtr->Data, "XX") != 0&& strcmp(curPtr->Data, "YY") != 0)
377.             {
378.                 EnStack(stackDeck, curPtr->Data);
379.             }
380.
381.             if (curPtr->Data[0] == 'A')
382.             {
383.                 EnQuene(queneDeckA, curPtr->Data);
384.             }
385.
386.             curPtr = curPtr->Next;
387.             free(temp);
388.         }
389.     }
390.
391.     printf("仅剩最后一张牌%s，出牌阶段结束！\n", curPtr->Data);
392. }
393.
394. int main(void){
395.
396.     // 定义顺序表（二维数组）储存牌堆并进行初始化
397.     char UnshuffledCards[54][3];
398.     Generate_UnshuffledCards(UnshuffledCards);
399.
400.     printf("洗牌前的牌堆为:\n");
401.     Print_Cards(UnshuffledCards);
402.     Knuth_Durstenfeld_Shuffle(UnshuffledCards);
403.     printf("使用 Knuth-Durstenfeld Shuffle 算法洗牌后的牌堆为:\n");
404.     Print_Cards(UnshuffledCards);
405.
```

```
406.     // 初始化循环链表用于储存手牌
407.     LinkDeck linkdeck = Insert2CircularLinkedList(UnshuffledCards);
408.     printf("循环链表中的牌:\n");
409.     PrintCircularLinkedList(linkdeck);
410.
411.     // 初始化链队用于储存打出的 A 开头的牌
412.     QueneDeck quenedeck_A = (QueneDeck)malloc(sizeof(Quene));
413.
414.     // 初始化链栈用于储存打出的所有牌
415.     StackDeck stackdeck = (StackDeck)malloc(sizeof(Stack));
416.
417.     // 定位：在循环链表中定位第一个出现的大王或小王牌，并将它作为牌堆顶
418.     FindXYinDeck(linkdeck);
419.     if (linkdeck->Head == NULL)
420.     {
421.         printf("未找到大王牌或小王牌\n");
422.     }
423.
424.     // 报数+出牌
425.     int M;
426.     printf("请指定计数上限 M: ");
427.     scanf("%d", &M);
428.     PlayingCards(linkdeck, M, quenedeck_A, stackdeck);
429.
430.     printf("最后被打出的 5 张 A 开头的牌为:\n");
431.     PrintQuene_5(quenedeck_A);
432.     return 0;
433. }
434.
```