组号: 03

上海大学计算机工程与科学学院

实 验 报 告

（数据结构 1）

学　　　期：2022-2023 年 1 季

组　　　长：　刘彦辰

学　　　号：　21121319

指导教师：　朱能军

成绩评定：__________（教师填写）

二〇二二年 12 月 13 日

小组信息			
姓名	学号	贡献比	签名
刘彦辰	21121319	40%	
李睿凤	21121906	30%	
车心宇	21121928	30%	

实验概述	
实验零	（熟悉上机环境、进度安排、评分制度；确定小组成员）
实验一	**约瑟夫问题变种**
实验二	?
实验三	?
实验四	?

I 任务分配

姓名	职责
刘彦辰	项目总体架构与测试；类 `arrayList` 设计与测试；界面设计； 两个问题的数组实现及算法分析与证明；程序性能分析与测试；部分报告撰写
李睿凤	类 `dChain_circle` 设计与测试；类 `timeCounter` 设计与测试；部分报告撰写
车心宇	两个问题的链表解决实现；类 `dChain_circle` 测试；部分报告撰写

II 项目目录结构以及类图

由于时间原因这部分省略。

项目已开源到 https://github.com/jamesnulliu/SHU_DS_znj_Project01 ，感兴趣的话大家可以自己去看一看。

III 数据结构

1. 双向链表实现循环链表

双向链表描述的线性表中，构成链表的每个节点中存储一个线性表元素，且存在两个指针域：一个指向其前驱的指针域prev，一个指向其后继的指针域next，形成双向链表。

头节点的前驱指针指向尾结点，尾结点的后继指针指向头节点，实现链表的循环。元素之间逻辑上的相邻关系在物理位置上并不一定相邻，而是通过指针联系在一起形成了一条链式结构。

假设有一线性表，n 个元素存储在 n 个节点中，每个节点通过指针链接。

元素结点

Debug 时出现提示内容为：

> Class template has already been defined.

的错误，即产生了定义不一致问题（而不是重定义问题）。

广泛地说，重定义的发生，在于所有参与编译的源文件中出现对某个对象的多次定义；编译时编译器无法明确该对象究竟使用何种定义。

当将一些内容写入头文件，我们要注意可能有多个源文件将 include 该头文件；因此绝大部分情况下如果不将对象的声明和定义分文件写（即声明写在头文件，定义写在源文件），将出现重定义问题。

两种特殊情况声明和定义不可以分文件写：

1. 内联函数（inline function)
   内联函数的函数体并非真正的函数定义，编译器运行到调用部分时会展开内联函数（直接将函数体替代至调用位置）。
2. 模板（template)
   模板类的定义也并非真正的定义，编译器在运行时才会实例化。

由于以上两种情况**没有真正定义**，就算被 include 进多个头文件，也不会出现重定义问题。

然而，当不同头文件的定义出现分歧（定义内容不一致），就会导致导致编译器无法确定究竟哪个定义是正确的，进而抛出诸如“Class template has already been defined. ”的错误提示。

删除

本次项目为满足需求，我们对 `erase` 函数进行重载，分别根据下标和指针删除线性表元素。

待删除元素的地址是随机的，无法通过寻址公式来定位。若传入下标小于等于链表节点的一半，从头节点开始向后遍历链表寻找指定节点，反之，从尾结点向前遍历查找；该删除节点的前驱节点的next指针指向将要删除的结点的后继结点，后继结点的 `prev` 指针指向将删除节点的前驱节点。若删除节点为头节点，将该链表的 `firstNode` 后移，若为尾结点则将 `lastNode` 前移，删除指定的节点，`listSize` 减一。

输出

我们定义了 public 函数 `output` 用来输出；因此 `operator<<` 只需调用 `output` 即可，没必要声明成友元函数。

然而，如果想要声明友元，我们需要注意 `operator<<` 不是类 `dChain_circle` 的成员函数（实际上，`operator<<` 可以算作类 `std::ostream` 的成员函数），因此 `operator<<` 不和类 `dChain_circle` 共享模板；其在 `dChain_circle` 内声明友元时需额外声明它的模板函数。

`std::cout` 是一个在 std 命名空间里定义的对象（也可以说是一个全局变量），在 namespace std 外部使用时需要指出 `cout` 所属的命名空间，所以需要用 scope operator `::` 指明其所属空间。

而在我们的项目中，`out` 是在函数内定义的一个 `std::ostream` 类对象，因此使用时不需要额外添加命名空间。

const 成员函数

`get_firstNode` 和 `get_lastNode` 等函数需要额外提供 const 版本。

在 C++ 中，我们应该只升高而非降低安全级别，参考以下例子：

```
T a;                    // Create a T object called {a}.
const T& ra = a;        // Bind {a} to a const reference, safty level up, valid.
const T* pa = &a;       // Store address of {a} to a pointer to const, safty level up, valid.
const T b;              // Create a const T object called {b}.
T& rb = b;              // Bind {b} to a non-const reference, safty level down, invalid.
T* pb = &b;             // Store address of {b} to a pointer to non-const, safty level down, invalid.
```

在类设计中，有以下规定：non-const 和 const 对象都可以调用 const 函数，而 const 对象不能调用 non-const 函数。

因此我们考考虑哪些函数应该设计两个版本，哪些函数只应该设计一个版本。

注意：在 const 成员函数中，this 指向对象为 const 对象，所以在该函数内调用的其他类内成员函数必须拥有 const 版本。

2. 数组实现的线性表

在数组描述的线性表中，数组的每一个位置都可以用来储存线性表的一个元素。第 i 个元素与其下标（index）的映射关系可以用以下公式表示：

$$\text{location}(i) = i - 1$$

假设有一数组`arr[0:n-1]`中共有 n 个元素：

插入

将元素`e`插入至位置`i`，我们需要将`arr[i:n-1]`中的元素依次复制到`arr[i+1:n]`，再将元素`arr[i]`修改为`e`。

此时数组中共有 $n+1$ 个元素。

插入的时间复杂度为$O(n)$。

删除

删除`arr[i]`，我们需要将`arr[i+1,n-1]`中的元素依次复制到`arr[i:n-2]`。

此时数组中共有 $n-1$ 个元素

删除的时间复杂度为$O(n)$。

动态扩容

在用指针申请内存时，由于C++的静态性，只能申请固定长度的连续内存。

若内存已满而想增加元素，则需要对数组进行扩容。

扩容的本质是申请一串更大的内存（习惯上，新内存大小应

为原内存的2倍），再将原内存内的数据迁移过去。

申请内存和其他操作的时间复杂度为 $O(1)$，而数据迁移的时间复杂度为 $O(n)$；因此总时间复杂度为 $O(n)$。

以下为对一维数组 `arr[0:n-1]` 扩容的伪代码：

Pesudocode for changeLength1D -----

```
let arr2 be new array[0:newLength-1]
k = min(newLength, oldLength)
for j from 0 to k-1
    arr2[j] = arr[j];
substitute arr for arr2
```

迭代器

为用户自定义容器类设计符合STL规范的迭代器（iterator）是有意义的行为。当迭代器成功实现，就可以方便地使用STL提供的各种算法。具体实现过程可以参考我们提供的**源码附件**，这里不多做展开。

IV 算法分析

1. 使用链表

在双链表下，我们可以更直观的完成项目目标。

问题分析

- **问题一**
  我们可以将每一轮的操作分为两步。（以X为例，`px`为X的位置）
  1. 向后数 K 份简历，拿走它。
     `px`往后移 $K-1$ 位，删除该节点。
  2. 走向下一份简历。
     `px`往后移动一位。
- **问题二**
  移动操作与问题一相同但是多出一步添加操作。（以X为例，`px`为X的位置，Y的操作不变）
  1. 向后数 K 份简历，并且移动到该位置。
     `px`往后移 $K-1$ 位。
  2. 在X后放一份简历，并拿走X面前这份简历。
     在`px`后插入一个节点，其序号为$max+1$，删除`px`指向的节点。
  3. 走向下一份简历。
     `px`往后移动一位。

为了减少`px`移动的次数，我们引入等效步长 $Step$ ，优化遍历方法，将移动次数限制在 $\frac{Length}{2}$ 内。

数学公式

下面用推导中用p表示所在位置，$Move$ 表示输入的步长，$Step$ 表示等价移动步长（正负表示方向），$Length$表示目前剩余简历数目。

$Step = Move \bmod Length - 1$

$$Step = \begin{cases} Step & Step \leq \frac{Length}{2} \\ Step - Length & Step > \frac{Length}{2} \end{cases}$$

伪代码

已知所有简历数据都存储在双向循环链表`chain`中，初始共有 $N(Length)$ 份简历，X 每次逆时针数 K 份，Y 每次顺时针数 M份。我们将上一节中的所有公式泛化地用 f 表示，则可将两个问题的程序用伪代码表示出来。

问题 1 的伪代码时间复杂度为$O(n^2)$

Pesudocode for circle_test1 -----

```
px = firstNode                // O(1)
py = lastNode                 // O(1)
loop                          // O(n)
```

```
    stepX = f(Lenth,K)      // O(1)
    stepY = f(Lenth,M)      // O(1)
    px move stepX           // O(n)
    py move -stepY          // O(n)
    t1=px                   // O(1)
    t2=py                   // O(1)
    if t1==t2
        erase t1;           // O(1)
    else
        erase t2 and t1;    // O(1)
    if lenth==0
        break loop
    px move 1               // O(1)
    py move -1              // O(1)
```

问题 2 的伪代码时间复杂度为$O(n^2)$。

Pesudocode for circle_test2 -----

```
px = firstNood              // O(1)
py = lastNood               // O(1)
loop                        // O(n)
    stepX = f(Lenth,K)      // O(1)
    stepY = f(Lenth,M)      // O(1)
    px move stepX           // O(n)
    py move -stepY          // O(n)
    t1=px;                  // O(1)
    t2=py；                 // O(1)
    if t1 == t2
        erase t1            // O(1)
    else
        erase t1 and t2     // O(1)
    if lenth == 0
        break loop
    Add Node After px       // O(1)
    px move 1               // O(1)
    py move -1              // O(1)
    if lenth==0
        break loop
```

2. 使用数组

与链表相比，**看似**本次项目的两题使用 ArrayList 实现是一个欠佳的选择。

ArrayList 的 get 操作比 Chain 快（ArrayList 的 get 时间复杂度为 $O(1)$，而 Chain 的时间复杂度为 $O(n)$）；

ArrayList 的 erase 操作效率比 Chain 低（ArrayList 的时间复杂度 $O(n)$，Chain 的时间复杂度为 $O(1)$）。

本次两题中 Chain 能直接用 node pointer 进行迭代，因此不存在调用 get 操作的情况；但两题都存在使用 erase 删除元素的情况。

然而，当数据增多，步长增大，我们可以发现 ArrayList 在某些情况下的效率远胜于 Chain。具体的测试数据和分析将留在 **V 性能分析** 一节详细解释。

数学公式

使用 Array List 实现本次的题目需要先对部分数学公式进行推导。

问题1和2的移动方式是相同的；也就是说，假设已知某一轮（**定义每轮的第一步是 X 与 Y 开始数简历，最后一步是X 与 Y 取走简历，下同**） X 面前的简历下标为 $xStart$，Y 面前的简历下标为 $yStart$，且当前轮共有 N 份简历，X 每次逆时针数 K 份，Y 每次顺时针数 M 份；

则 X 和 Y 最终将要取走的目标简历的下标 $xTake$ 和 $yTake$ 符合以下公式：

$$xTake = (N + xStart + K \bmod N - 1) \bmod N \tag{4.1}$$

$$yTake = (N + yStart - M \bmod N + 1) \bmod N \tag{4.2}$$

得到 $xTake$ 和 $yTake$ 之后，

对于问题1，下一轮的 N' 以及 $xStart'$ 和 $yStart'$ 符合以下公式：

$$N' = \begin{cases} N - 1 & xTake = yTake \\ N - 2 & xTake \neq yTake \end{cases} \tag{4.3}$$

$$xStart' = \begin{cases} (N' + xTake - 1) \bmod N' & \text{if } yTake < xTake \\ xTake & \text{if } yTake \geqslant xTake \end{cases} \tag{4.4}$$

$$yStart' = \begin{cases} (N' + yTake - 1) \bmod N' & \text{if } yTake \leqslant xTake \\ (N' + yTake - 2) \bmod N' & \text{if } yTake > xTake \end{cases} \tag{4.5}$$

对于问题2，下一轮的 N' 以及 $xStart'$ 和 $yStart'$ 符合以下公式：

$$N' = \begin{cases} 0 & \text{if } N = 1 \text{ and } xTake = yTake \\ N & \text{if } N > 1 \text{ and } xTake = yTake \\ N - 1 & \text{if } N > 1 \text{ and } xTake \neq yTake \end{cases} \tag{4.6}$$

$$xStart' = \begin{cases} (N' + xTake - 1) \bmod N' & \text{if } yTake < xTake \\ xTake & \text{if } yTake \geqslant xTake \end{cases} \tag{4.7}$$

$$yStart' = (N' + yTake - 1) \bmod N' \tag{4.8}$$

注：对于两个问题，N' 总是由 N 以常数之差衰减得到。

伪代码

已知所有简历数据都存放在数组`arr`中，初始共有 N 份简历，X 每次逆时针数 K 份，Y 每次顺时针数 M 份。

我们将上一节中的所有公式泛化地用`f`表示，则可将两个问题的程序用伪代码表示出来。

问题1的伪代码，总时间复杂度为 $O(n^2)$：

Pesudocode for array_test1 -----

```
xStart = 0
yStart = N-1
loop                                        // O(n)
    xTake = f(xTake, N, K, M)               // O(1)
    yTake = f(yTake, N, K, M)               // O(1)
    N = f(N, xTake, yTake)                  // O(1)
    if N == 0
        break loop
    xStart = f(N, xTake, yTake)             // O(1)
    yStart = f(N, xTake, yTake)             // O(1)
    if xTake == yTake
        erase arr[xTake]                    // O(n)
    else
        erase arr[max(xTake, yTake)]        // O(n)
        erase arr[min(xTake, yTake)]        // O(n)
```

问题2的伪代码，总时间复杂度为 $O(n^2)$：

Pesudocode for array_test2 -----

```
xStart = 0
yStart = N-1
loopCount = 0
loop                                          // O(n)
    xTake = f(xTake, N, K, M)                 // O(1)
    yTake = f(yTake, N, K, M)                 // O(1)
    N = f(N, xTake, yTake)                    // O(1)
    if N == 0
        break loop
    xStart = f(N, xTake, yTake)               // O(1)
    yStart = f(N, xTake, yTake)               // O(1)
    if xTake == yTake
        if (K + M) mod N == 1                 // constraint
            break loop due to endless loop
```

```
    else
        erase arr[yTake]                    // O(n)
    arr[xTake] = N + loopCount              // O(1)
    loopCount = loopCount + 1               // O(1)
```

值得注意的是，在问题2中，增加了当 $xTake = yTake \text{ and } (K+M) \bmod N = 1$ 时结束循环的约束条件（constraint）。我们将在下一节详细分析该条件。

对问题2无限循环的约束证明

如上节所述，当某轮的各参数满足

$$xTake = yTake \text{ and } (K+M) \bmod N = 1$$

接下类会出现无限循环（不相信的话读者可以自己手动算一算，若初始时 $N_0 = 5$，$K = 3$，$M = 2$，当 N 衰减为 2 时会陷入无限循环；更多的例子可以参考**附件 endless_loop.xlsx** 中的 Sheet3）。

上述约束条件最早由第 7 组的**邵宇宸**同学提出，其给出的解释利用了几何上的旋转对称性。我在下文将基于该条件出进一步的说明。

A. 代数证明

根据**公式 4.6**，任意处于无限循环中的状态 S_i 必须满足：

$$N > 1 \text{ and } xTake = yTake \Leftrightarrow N_{next} = N \tag{4.9}$$

注：$N > 1 \text{ and } xTake = yTake$ 是 $N' = N$ 的充要条件；即，只要状态 S_i 处于无限循环中，必然同时满足 $N_i > 1 \text{ and } xTake_i = yTake_i$ 和 $N_{i+1} = N_i$。

假设第 $k, k+1, k+2$ 个状态都处于无限循环中，令他们为状态 S, S', S''；则根据公式 4.9 的充要条件，状态 S, S', S'' 都满足 $N > 1 \text{ and } xTake = yTake$ 。

而我们的目的，是在已知 $N > 1 \text{ and } xTake = yTake$ 的条件下，通过 S, S', S'' 找出一个包含 K, M, N 的约束等式 $E_{constraint}(K, M, N)$；当且仅当 $E_{constraint}(K, M, N)$ 成立，程序进入无限循环的过程。

将**公式 4.9** 带入**公式 4.7** 和**公式 4.8**，计算得到状态 S' 的 $xStart'$ 和 $yStart'$：

$$xStart' = xTake \tag{4.10}$$

$$yStart' = \begin{cases} yTake - 1 & \text{if } yTake \neq 0 \\ N - 1 & \text{if } yTake = 0 \end{cases} \tag{4.11}$$

将**公式 4.10** 和**公式 4.11** 回代入**公式 4.1** 和**公式 4.2**，计算得到状态 S' 的 $xTake'$ 和 $yTake'$：

$$xTake' = (N + xTake + K \bmod N - 1) \bmod N \tag{4.12}$$

$$yTake' = \begin{cases} (N + yTake - M \bmod N) \bmod N & \text{if } yTake \neq 0 \\ (2N - M \bmod N) \bmod N & \text{if } yTake = 0 \end{cases} \tag{4.13}$$

由于 S' 是无限循环中的状态，欲有 $N'' = N'$，则必有：

$$xTake' = yTake' \tag{4.14}$$

合并**公式 4.12**，**公式 4.13**，**公式 4.14**，得到：

$$\begin{aligned} K \bmod N + M \bmod N = cN + 1, c \in \text{Integer} \\ \Rightarrow (K + M) \bmod N = 1 \end{aligned} \tag{4.15}$$

综上，**公式 4.15** 指明了我们要求的 $E_{constraint}(K, M, N)$，即：

$$(K + M) \bmod N = 1 \text{ when } N > 1 \text{ and } xTake_{before} = yTake_{before} \tag{4.16}$$

B. 几何证明

假设有第 $k, k+1, k+2$ 轮，分别以 S, S', S'' 表示。

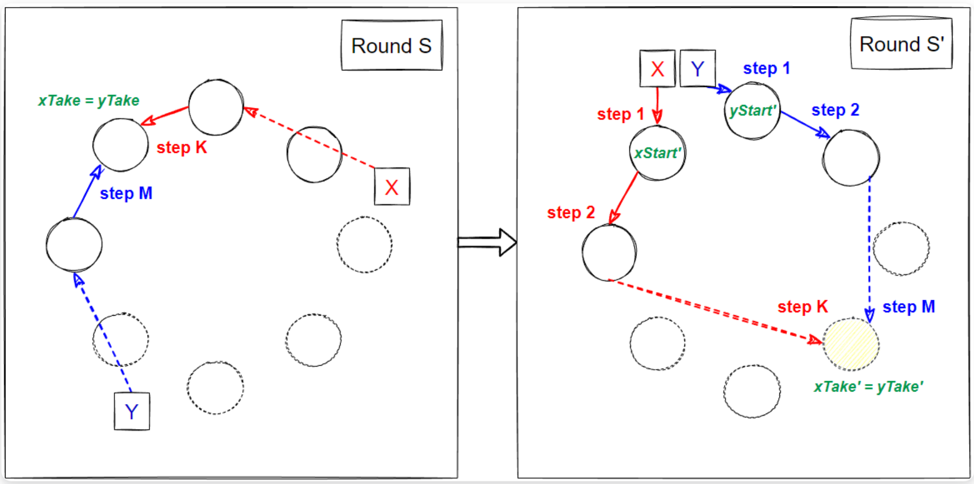

图 4-1

由**图 4-1** 可以看到，不管状态 S 中 $xStart$ 和 $yStart$ 的位置（这步与代数证明不相同），只要 $xTake = yTake$，根据题意必有 $N' = N$。

同理，欲使状态 S'' 中的 N'' 与状态 S' 中的 N' 相同，必要条件为 $xTake' = yTake'$。

若 $xTake' = yTake'$，由于圆的对称性，在取完简历后状态 S'' 将与状态 S' 等价（可以参考状态 S 取简历的情况）。

我们能够很容易地观察出，只要 $(K + M) \bmod N' = 1$，从状态 S' 开始，对之后的任意状态 S^*，有

$$N^* = N'$$

此时状态 S' 是第一个属于无限循环的状态（并没有对状态 S 进行明确判断）。

因此得到 $E_{constraint}(K, M, N)$ 为：

$$(K + M) \bmod N = 1 \text{ when } N > 1 \text{ and } xTake_{before} = yTake_{before}$$

泛化分析

将**等式 4.16** 作为约束条件有一个致命的缺陷，就是对无限循环过程的判断必须在运行中而非运行前：我们不得不在直到 N 衰减为 0 前，持续运行程序，并检查每个状态下约束条件是否成立。

举个简单的例子，当 $N = 100, K = 100, M = 99$ 时，我们目前的程序必须一直运行到 $N_{endless} = 2$ 时才检测出符合**等式 4.16**的无限循环情况。

那么可否推出一个更加泛化的公式，在程序运行的开始就判断出是否会出现无限循环呢？

首先列出**表4-1**，其中第一行表示初始 N_0 的取值，第一列表示 $(K + M) \bmod N_0$ 的取值；不同行列的交点中的数值表示循环结果，0 为取尽，非 0 代表进入无尽循环时 $N_{endless}$ 的值。（更多无尽循环情况的数据可以在**附件 endless_loop.xlsx** 中的 Sheet1, Sheet2 和 Sheet3）中找到。

k+m\n	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	2	0	0	2	0	0	2	0	0
4	0	3	0	3	3	0	3	0	0
5		2	4	2	2	2	2	2	2
6		0	0	5	0	0	0	0	0
7			2	2	6	2	2	6	3
8			0	0	0	7	0	0	0
9				0	2	2	8	2	2
10				3	3	0	0	9	0
11					0	0	5	5	10
12					0	0	0	0	0
13						3	6	2	0
14						0	0	0	0
15							0	0	0
16							0	0	0
17								4	2
18								0	0
19									2
20									0

表 4-1

注意表中底色为红色的焦点，其条件似乎与我们先前的约束条件（**等式 4.16**）极其相似：$N_0 = N_{endless}$ 而 $(K + M) \bmod N_0 = 1 \text{ when } N_0 > 1$；然而这里缺少一个很关键的 $xTake_{before} = yTake_{before}$ 条件。

解释这个问题非常简单：如**图 4-2** 尽管初始时不存在上一状态中的 $xTake_{before}$ 和 $yTake_{before}$，我们可以假设存在一个 BEFORE 状态。这样就与约束条件匹配了。

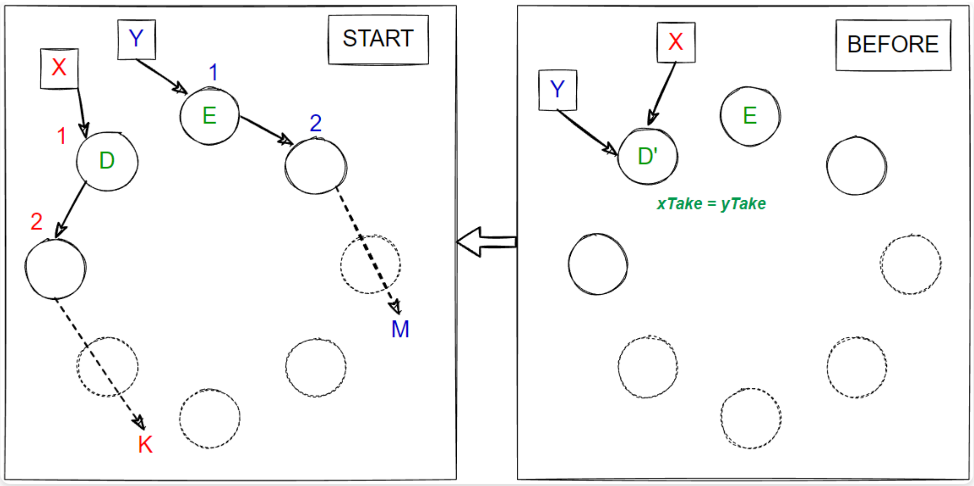

图 4-2

由于数据分布得过于分散，以我们小组目前的能力确实没有找到足够归纳成数学方程的规律。

作为补偿，在源码 source/testFunc/testArrayList.cpp 中我们提供了函数 outputResultofTest2 用来输出（结果保存至 .txt 文件）一定范围的 N 内所有可能的 $K+M$ 取值，使得最终结果进入无尽循环（无尽循环时的 $N_{endless}$ 也会一并输出）；另外，该函数还利用了STL提供的 hash map 来验证无尽循环的发生与 K, M 的单独取值无关，只与 $K+M$ 有关。

V 性能分析

1. 理论分析

在 **IV.2** 中我们提到，对于本次项目的两题，使用 ArrayList 和 Chain 在性能上会有所差异。

ArrayList 的 get 操作比 Chain 快（ArrayList 的 get 时间复杂度为 $O(1)$，而 Chain 的时间复杂度为 $O(n)$）；

ArrayList 的 erase 操作效率比 Chain 低（ArrayList 的时间复杂度 $O(n)$，Chain 的时间复杂度为 $O(1)$)。

本次两题中 Chain 能直接用 node pointer 进行迭代，因此不存在调用 get 操作的情况；但两题都存在使用 erase 删除元素的情况。

尽管 ArrayList 的 erase 操作相比 Chain 效率更低，但是当移动步数足够长（例如 $K \bmod N = 5000$ 且 $N = 10000$），Chain 中用于迭代的指针将不得不花费大量的效率在移动上——而这种效率的损失是无法进行优化的。

此时尽管 ArrayList 在 erase 操作中花费的更多的时间，但是结果上可能相比 Chain 效率快上很多。

当然，若移动步数很短而数据量很大（例如 $K \bmod N = 1$ 且 $N = 10000$），此时使用 Chain 一定是更优的选择。

2. 设计类 timeCounter

设计类 timeCounter 目的为计算某段程序的平均运行时间。

调用成员函数 `startCounting` 开始计时；调用 `endCounting` 结束计时，并将运行时间存储在类内成员 `resultList` 中。

如果在调用 `startCounting` 前就调用 `endCounting`，将抛出异常提示未开始计时。

成员函数 `calAverage` 利用类 arrayList 中设计的迭代器和STL算法计算其内部已有时间数据的平均值。

timeCounter 类的析构函数无需手动释放 arrayList 对象 `resultList` 中申请的空间，因为 arrayList 的析构函数会在 timeCounter 类对象被析构时被自动调用。

3. 运行时间测量

由于问题 2 涉及到无限循环条件的判断，我们只测量问题 1 两种方式的运行时间差异

IDE: Visual Studio 2022 Community

Platform Toolset: Visual Studio 2022 (v143)

C++ Language Standard: ISO C++20 Standard (/std:c++20)

Optimization: /O2 /Oi /Os

Basic Runtime Checks: Default

Configuration: Release

Platform: x64

Each case would be tested for 10 times and calculate average time use.

Result			
N	M+K	time(ms)	method
100000	3	297	chain
		374	array
	10	321	chain
		343	array
	20	325	chain
		332	array
	40	343	chain
		329	array
	250	691	chain
		331	array
	25000	33205	chain
		331	array

图 5-1

图 5-1 展示了部分时间测量结果（更多结果可以在**附件 endless_loop.xlsx** 中的 Sheet4 找到）。不难发现，当 $M+K$ 相对 N 较小，循环链表的运行效率更高；反之，数组运行效率更高。另外，可以发现数组的运行效率相对稳定，而链表的效率随 $M+K$ 的增大下降速度较快。

因此我们的结论是，当 $\frac{(M+K) \bmod N}{N}$ 较大，应选择 ArrayList 解决本次项目的两个问题