组号:3

图片包含 游戏机, 画

描述已自动生成

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：2024-2025年冬季

组 长： 胡淇铭

学 号： 23122788

指导教师： 朱能军

成绩评定： （教师填写）

二〇二二年X月X日

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **小组信息** | | | | |
| 登记序号 | 姓名 | 学号 | 贡献比 | 签名 |
| 1 | 胡淇铭 | 23122788 | 35 |  |
| 2 | 李泽勋 | 23122798 | 33 |  |
| 3 | 谢静宜 | 23122816 | 32 |  |

|  |  |
| --- | --- |
| **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 | 面试安排程序设计 |
| 实验二 | (*实验题目*) |
| 实验三 | (*实验题目*) |
| 实验四 | (*实验题目*) |

实验一

一、**实验题目**

**（一）面试安排（1）**

1、问题描述

某IT公司招聘新员工，已收到N份简历，人力资源部X和Y负责挑选简历安排面试。他们把N份简历排成一个圆圈，按逆时针方向编号为1～N。开始时X站在1号简历前，按逆时针方向数到第K份简历，选中；Y站在N号简历前，按顺时针方向数到第M份简历，选中。两人同时取走所选简历后，分别按逆时针和顺时针走到下一份简历前，然后X和Y再重复上述方法取简历，直到取走全部简历，如果两人选中同一份简历，则只输出一个编号。

2、基本要求

要求输入3个数N、K和M，按取走简历的顺序（先甲后已）输出简历编号。

3、测试数据

输入样例：

10 4 3

输出样例：

4, 8; 9, 5; 3, 1; 2, 6; 10; 7

**（二）面试安排（2）**

1、问题描述

某IT公司招聘新员工，已收到N份简历，人力资源部X和Y负责挑选简历安排面试，Z负责补充新的简历。他们把N份简历排成一个圆圈，按逆时针方向编号为1～N。开始时X站在1号简历前，按逆时针方向数到第K份简历，选中；Y站在N号简历前，按顺时针方向数到第M份简历，选中。两人同时取走所选简历后，分别按逆时针和顺时针走到下一份简历前，此时（除非所有简历都已取完或者刚刚X和Y取走的是同一份简历），Z会拿来1份新简历（编号从N+1开始递加），插到X前面，然后X和Y再重复上述方法取简历，直到取走全部简历，如果两人选中同一份简历，则只输出一个编号。

2、基本要求

要求输入3个数N、K和M，按取走简历的顺序（先甲后已）输出简历编号。

3、测试数据

输入样例：

5 1 1

输出样例：

1, 5; 6, 4; 7, 3; 8, 2

二、**实验内容**

题目首先要求了一个循环链表结构，在这个链表中，顺序元素按照逆时针存储；同时，要求对链表中每一个结点的前驱/后继节点进行访问，于是，在这里我们采用了双向循环链表结构。接下来，简析题目要求：

**（一）面试安排(1):** 在从1到N编号，逆时针放置的简历中，X按逆时针取第K份简历，Y按顺时针取第M份简历，重复操作直到所有简历被选取。

**（二）面试安排(2):** 与（1）类似，每次X和Y选完简历后，若未选中同一份，Z插入一份新的简历到X前面。重复操作直到所有简历被选取。

三、**解决方案**

1. **算法设计**
   1. **数据结构**

在这里，我们采用了双向循环链表解决这两道问题。我们将双向循环链表分为三层，由内而外分别是：数据层，结点层，链表层。



**图1 双向循环链表的设计**

我们希望通过这样的设计，使得每一层完成不同的功能：

数据层：存放数据；

结点层：结点的实现与操作；

链表层：链表的实现与操作；

这样的设计使得逻辑清晰，可读性较高且易于维护，符合封装的思想。

**结点层(DblNode)设计：**

**数据成员设计：**

int data; *// 数据域*DblNode \*prior; *// 指向前驱结点的指针域*DblNode \*next; *// 指向后继结点的指针域*

**函数成员：**

|  |  |  |
| --- | --- | --- |
| 序号 | 函数名 | 作用 |
| 构造函数 | | |
| 1 | DblNode(); | 默认构造函数 |
| 2 | DblNode(int item,DblNode \*priorlink = nullptr,DblNode \*nextlink = nullptr); | 含参构造函数 |

**链表层(DoublyLinkedList)设计**

**数据成员设计：**

public:  
 DblNode \*head; *// 链表的头指针* int length; *// 链表的长度*

**函数成员：**

|  |  |  |
| --- | --- | --- |
| 序号 | 函数名 | 作用 |
| 构造函数、析构函数 | | |
| 1 | DoublyLinkedList(); | 默认构造函数 |
| 2 | DoublyLinkedList(int arr[], int n); | 含参构造函数 |
| 3 | ~DoublyLinkedList(); | 析构函数 |
| 功能函数 | | |
| 4 | bool isEmpty() const; | 判断链表是否为空 |
| 5 | int getLength() const; | 返回链表的长度 |
| 6 | void DeleteNode(DblNode \*node); | 删除结点 |
| 7 | void InsertBefore(DblNode \*node, int value); | 在输入结点前插入结点 |
| 8 | void InsertAfter(DblNode \*node, int value); | 在输入结点后插入结点 |

* 1. **算法思想**

|  |  |  |
| --- | --- | --- |
| 序号 | 函数名 | 作用 |
| 实现算法的功能函数 | | |
| 1 | void MoveForward\_X(DblNode\*& X, int N, int K) | 移动X |
| 2 | void MoveForward\_Y(DblNode\*& X, int N, int M) | 移动Y |
| 3 | void Task1(DoublyLinkedList &resumes, int K, int M, int N) | 实现题目一 |
| 4 | void Task2(DoublyLinkedList &resumes, int K, int M, int N) | 实现题目二 |

* + 1. **操作实现**

void MoveForward\_X(DblNode\*& X, int N, int K) {

    int Moves = (K - 1) % N;

    if (Moves <= N / 2){

        for (int i = 0; i < Moves; ++i)

            X = X->next;

    }

    else{

        for (int i = 0; i < N - Moves; ++i)

            X = X->prior;

    }

}

void MoveForward\_Y(DblNode\*& X, int N, int M) {

    int Moves = (M - 1) % N;

    if (Moves <= N / 2){

        for (int i = 0; i < Moves; ++i)

            X = X->prior;

    }

    else{

        for (int i = 0; i < N - Moves; ++i)

            X = X->next;

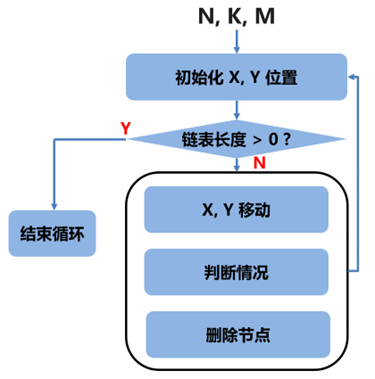
    }

}

MoveForward\_X 和 MoveForward\_Y通过双向循环链表实现节点的移动。首先，通过取模运算 (K - 1) % 来计算实际需要移动的步数，这样可以处理步数超过链表长度的情况，避免多余的循环。接着，根据移动步数 Moves 的大小，算法判断是向前（next）还是向后（prior）移动，确保在链表中找到最短路径。这种优化方式能够有效减少不必要的遍历，提高效率，尤其是在链表长度较大的情况下，避免了大量的无效操作。

* + 1. **算法实现**

**（一）面试安排(1)：**

****

**图2 面试安排(1)算法思想示意**

void Task1(DoublyLinkedList &resumes, int K, int M, int N) {

    DblNode \*X = resumes.head;   // X 的起始位置

    DblNode \*Y = resumes.head->prior; // Y 的起始位置

    while (resumes.getLength() > 0) {

        MoveForward\_X(X, N, K); // 移动X

        MoveForward\_Y(Y, N, M); // 移动Y

Task1 函数通过双向循环链表中的两个指针 X 和 Y，在每一轮循环中，X 向前移动 K 步，Y 向后移动 M 步，直到链表中的所有节点都被处理完。每当 X 和 Y 相遇时，输出当前节点 X 的数据；如果 X 和 Y 不相遇，则输出 X 和 Y 的数据。

  DblNode\* nextX = X->next;    //保存X下一个节点，防止指针空挂

  DblNode\* priorY = Y->prior;  //保存Y上一个节点，防止指针空挂

在处理节点时，首先保存 X 的下一个节点 nextX 和 Y 的前一个节点 priorY，以防止在删除节点后，指针指向空节点。

···else

        {

            cout << X->data << ", " << Y->data;

            if (nextX == Y){

                nextX = nextX->next;

                priorY = priorY->prior;

            }

            resumes.DeleteNode(Y);

        }

        resumes.DeleteNode(X);

        X = nextX;

        Y = priorY;

在删除节点时，先输出节点的数据，再通过 DeleteNode 删除 X 和 Y 指向的节点，随后更新指针 X 和 Y 指向新的有效节点，继续遍历链表，直到链表为空。

在这里，我们尤其要注意一个情况：**X的next为Y，Y的prior为X**。

图3 X与Y相邻情况

在这个情况下，在删除操作中，X与Y所在的结点会被删除，导致指针空挂的情况，算法失效。

            if (nextX == Y){

                nextX = nextX->next;

                priorY = priorY->prior;

            }

为了解决这个问题，在发生这个情况的时候，我们让存放X和Y的指针各自再走一步，走向下一个结点。显然，这样的操作是不会对结果产生影响的：题目中指明了“走向下一份”是在“拿走简历”之后。

**（二）面试安排(2)：**

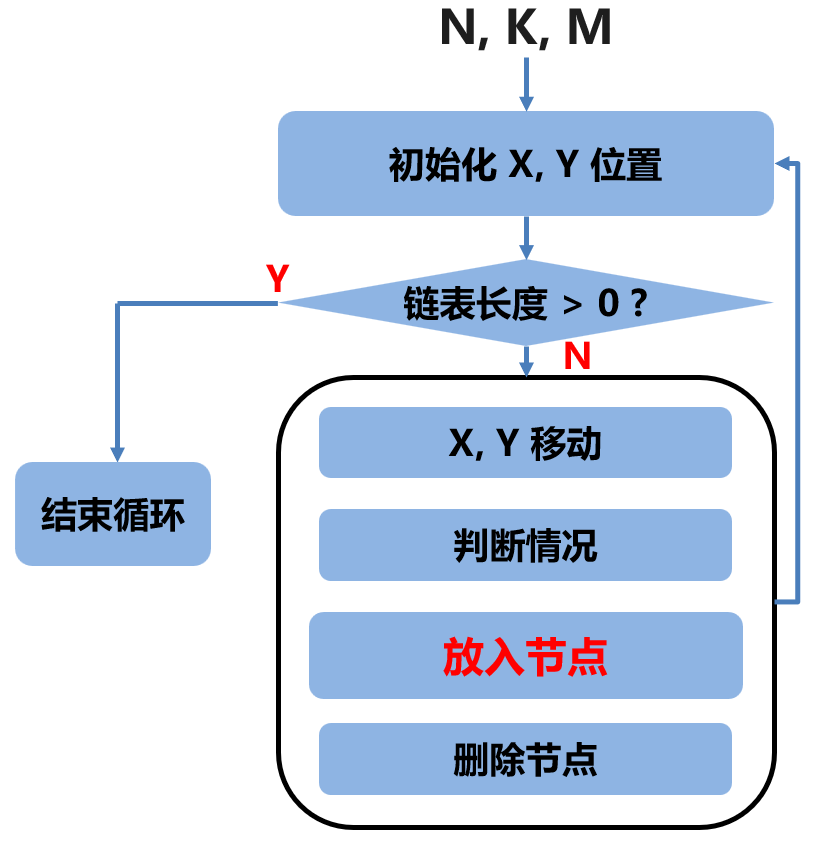


图4 面试安排(2)算法思想示意

void Task2(DoublyLinkedList &resumes, int K, int M, int N) {

    DblNode \*X = resumes.head;   // X 的起始位置

    DblNode \*Y = resumes.head->prior; // Y 的起始位置

    while (resumes.getLength() > 0) {

        MoveForward\_X(X, N, K); // 移动X

        MoveForward\_Y(Y, N, M); // 移动Y

        DblNode\* priorY = Y->prior;  //保存Y上一个节点，防止指针空挂

        ···

Task2 函数与 Task1 类似，同样通过两个指针 X 和 Y 在双向循环链表中进行遍历。

···else

        {

            cout << X->data << ", " << Y->data;

            if(resumes.getLength()>2) resumes.InsertAfter(X,++N);

            DblNode \*nextX = X->next; //保存X下一个节点，防止指针空挂

            resumes.DeleteNode(X);

            resumes.DeleteNode(Y);

            X = nextX;

        }

        Y = priorY;

        if (resumes.getLength() > 0) cout << "; "; // 格式化输出

然后，保存 X 的下一个节点 nextX，删除 X 和 Y 指向的节点，并更新指针 X 和 Y 指向新的有效节点。循环继续，直到链表为空。

            if(resumes.getLength()>2) resumes.InsertAfter(X,++N);

在这里，我们要注意到，**当链表长度小于2的时候，不会再进行放的操作，这是因为代码逻辑实现与现实不一致：**代码的放操作在删除之前，而现实在之后。

1. **源程序代码**

void ···InsertAfter(DblNode \*node, int value)

{···

    DblNode \*nextNode = node->next;

    node->next = newNode;

    newNode->prior = node;

    newNode->next = nextNode;

nextNode->prior = newNode;

···}

void ···InsertBefore(DblNode \*node, int value)

{···

    DblNode \*prevNode = node->prior;

    prevNode->next = newNode;

    newNode->prior = prevNode;

    newNode->next = node;

    node->prior = newNode;

    if (node == head) {head = newNode;}

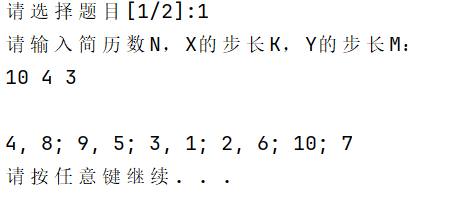
···}

int main() {  
 int choice;  
 do {  
 cout << "请选择题目[1/2]:";  
 cin >> choice;  
  
 if (choice != 1 && choice != 2) {  
 cout << "无效的选择，请输入 1 或 2" << endl;  
 }  
 } while (choice != 1 && choice != 2);  
  
 int N, K, M;  
 cout << "请输入简历数N，X的步长K，Y的步长M：" << endl;  
 cin >> N >> K >> M;  
  
 if (N <= 0) {  
 cout << "没有简历" << endl;  
 return 0;  
 }  
 *// 初始化双向循环链表，填入简历编号* int \*resumeArray = new int[N];  
 for (int i = 0; i < N; i++) {  
 resumeArray[i] = i + 1;  
 }  
 DoublyLinkedList resumes(resumeArray, N);  
 delete[] resumeArray;  
 cout << endl;  
 switch (choice) {  
 case 1:  
 Task1(resumes, K, M, N);  
 break;  
 case 2:  
 Task2(resumes, K, M, N);  
 break;  
 default:  
 cout << "没有这道题目" ;  
 }  
 system("PAUSE");  
 return 0;  
}

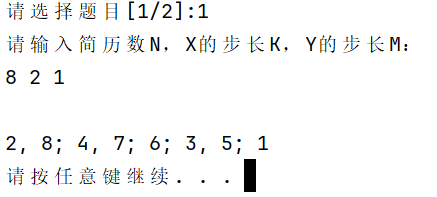
其余重要函数已经给出。

1. **实验结果**
2. **面试安排(1)：**

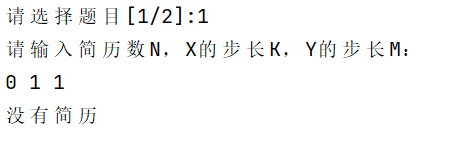
样例输入：



步数相差1：

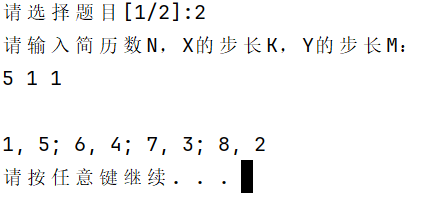


简历数量为0：

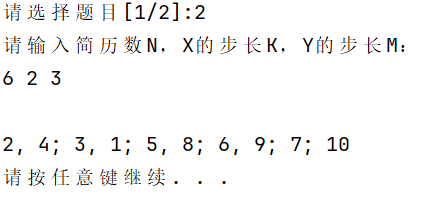


1. **面试安排(2)：**

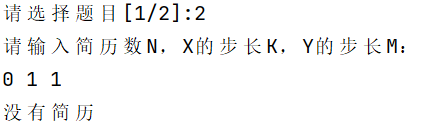
样例输入：



步数相差1：



简历数量为0：



1. **算法分析**

void MoveForward\_X(DblNode\*& X, int N, int K) {···}

void MoveForward\_Y(DblNode\*& X, int N, int M) {···}

这两个函数 MoveForward\_X 和 MoveForward\_Y 的时间复杂度为 O(N)。首先，Moves = (K - 1) % N 或 Moves = (M - 1) % N 使得循环的次数始终控制在 [0, N-1] 范围内。当 Moves <= N / 2 时，循环需要执行 Moves 次操作；否则，循环执行的次数为 N - Moves。因此，无论是哪种情况，循环的最大执行次数接近 N / 2，即最坏情况下时间复杂度为 O(N)。

以直接移动X为例进行对比：

for (int i = 0; i < k-1; i++) {

    X = X->next;

}

此时时间复杂度为O（K+M），相比之下显然取模算法对移动算法进行了优化，提高算法的运行效率和性能。

void Task1(DoublyLinkedList &resumes, int K, int M, int N) {···}

void Task2(DoublyLinkedList &resumes, int K, int M, int N) {···}

在每次 while循环中，首先调用 MoveForward\_X(X, N, K) 和 MoveForward\_Y(Y, N, M)，这两个函数在最坏情况下都需要遍历链表中的一部分，因此它们的时间复杂度是 O(N)。接着，删除链表中的节点 X 和 Y 的操作是 O(1)，但删除节点后，链表长度会减少。在每次循环中，两个节点会被删除，直到链表为空。由于每次循环都删除两个节点，所以循环的总次数为 N / 2 次，即约 N 次。

在每次循环中，由于MoveForward\_X 和 MoveForward\_Y 的时间复杂度为 O(N)，而整个循环会执行大约 N 次，因此**总体的时间复杂度是 O(N) \* O(N)，即 O(N²)。**

这个程序的空间复杂度主要由两个部分组成：输入数据存储和链表结构的存储。首先，在 main 函数中，创建了一个大小为 N 的动态数组 resumeArray 来存储简历编号，所需空间为 O(N)。然后，这些数据被用来初始化一个双向链表 resumes，每个链表节点包含数据和前后指针，因此链表本身的空间复杂度也是 O(N)。程序中的局部变量如 X, Y, nextX, priorY 等只是常数数量的指针或整数，它们占用的空间是 O(1)，不会随输入规模的变化而增长。虽然在 Task2 中可能插入一个新节点，但这只是链表内部的操作，也不会影响空间复杂度。因此，整个程序的**空间复杂度由链表的存储空间决定，最终是 O(N)**，其中 N 是链表的节点数。

5、总结与心得（*主要描述实验过程中存在的问题、原因、解决方法、收获、对实验内容的其他应用思考等*）