银行排队系统的设计

数据结构 2024-2025 第一次讨论课

第四组

2024 年 10 月 18 日

第四组成员：
张嘉麟 2352595 魏余昊 2353242 周灵菲 2352570
杜泉睿 2352351 魏嘉泽 2353940 刘艺 2351305

一、研究问题

1.背景与要求

在银行或类似的服务场所，排队系统是一个常见的应用场景。该系统的目标是按先到先服务的原则为客户提供服务，即先进先出的策略，非常适合使用队列数据结构来实现。

银行的柜台需要按照客户到达的顺序进行服务，但有时也会有优先服务的客户(如 VIP 客户)。因此，系统需要管理两个队列:

1）普通客户队列:按照客户到达的顺序进行服务，遵循 FIFO(First In First Out)原则。

2）VIP 客户队列:优先于普通客户队列中的客户进行服务。

2.问题分析

队列操作：两个队列分别处理不同优先级的客户。普通客户使用 FIFO 原则，从队首取出进行服务。VIP 客户优先于普通客户得到服务，且同样遵循 FIFO。

关键在于如何设计系统，使得当有 VIP 客户时，VIP 队列先于普通客户队列得到服务。

3.实现流程

1）初始化队列：创建两个队列数据结构，一个用于存储普通客户，另一个用于存储 VIP 客户。可以使用链表实现的队列来处理客户的入队和出队操作。

2）客户到达：普通客户到达时，将其加入普通客户队列。VIP 客户到达时，将其加入 VIP 客户队列。

3）处理服务请求：当系统准备服务时，首先检查 VIP 客户队列是否为空。如果 VIP 客户队列不为空，优先从 VIP 客户队列中取出队首元素，进行服务。如果 VIP 客户队列为空，则处理普通客户队列，按照 FIFO 原则提供服务。如果两个队列都为空，则系统暂时没有客户需要服务。

4）系统终止条件：当没有客户等待时，系统进入空闲状态，可以继续接收新客户或者终止。

4.难点解读

1）优先级管理：实现中最大的难点是如何有效管理 VIP 客户优先于普通客户。解决办法是引入两个队列，并在服务时优先检查 VIP 队列是否有客户待处理，这要求系统在服务时每次都要判断两个队列的状态。

2）边界情况处理：处理队列为空的情况，确保在服务客户时检查队列状态，避免在队列为空时执行无效的出队操作。当 VIP 和普通客户队列同时为空时，系统应当正确处理并返回合适的提示。

5.题目要求外的新增功能

重新排队功能：普通客户和 VIP 客户可以在不同条件下重新排队，保持队列的顺序完整性。

新增叫号功能：通过增加叫号功能，可以让系统在 VIP 客户全部处理完后，继续处理普通客户，保证服务的顺序合理。

混合数据结构管理：系统采用了队列和链表的混合结构进行客户管理，提升了数据存储和处理的灵活性。

处理并发队列：系统能够同时处理两个队列，确保服务流程不冲突，且高效运行。

新增的异常处理功能：处理因 VIP 客户加入或普通客户重排队造成的队列异常情况，保证服务流程的稳定性。

这些新增功能提升了排队系统的实用性和灵活性，使其能更好地应对实际场景中的复杂需

求。

二、代码框架

2.1 BankQueueManager.h

功能：定义 BankQueueManager 类，包含管理客户队列所需的数据成员和成员函数。
数据成员：
静态变量 vipCount 和 normalCount：分别记录 VIP 客户和普通客户的数量。
normalQueue：普通客户队列。
vipQueue：VIP 客户队列。
allNormalCustomers：所有普通客户的列表。
allVipCustomers：所有 VIP 客户的列表。
成员函数：
assignQueueNumber(Customer* customer, bool isVIP)：为客户分配队列号码。
isDuplicateRegistration(const std::string& queueNumber)：检查队列号码是否重复注册。
findCustomerByQueueNumber(const std::string& queueNumber)：根据队列号码查找客户。
reEnQueueCustomer(Customer* customer, bool isVIP, int position)：将客户重新入队。
enqueueNormal(Customer* customer)：将普通客户入队。
enqueueVIP(Customer* customer)：将 VIP 客户入队。
serveCustomer()：服务下一个客户。
getNormalQueue()：获取普通队列。
getVipQueue()：获取 VIP 队列。
~BankQueueManager()：析构函数，释放内存。
removeFirstNormalCustomer()：移除普通队列中的第一个客户。
removeFirstVipCustomer()：移除 VIP 队列中的第一个客户。

2.2 Customer.h

功能： 定义 Customer 类，包含客户的信息和操作客户信息的方法。
数据成员：
静态变量 normalCount 和 vipCount：分别记录普通客户和 VIP 客户的数量。
id：客户 ID。
name：客户姓名。
priority：客户优先级（VIP 客户为 1，普通客户为 0）。
queueNumber：客户队列号码。
成员函数：
Customer(std::string id, std::string name, int priority)：构造函数。
getId() const：获取客户 ID。
getName() const：获取客户姓名。
getPriority() const：获取客户优先级。
getQueueNumber() const：获取客户队列号码。
setId(const std::string& id)：设置客户 ID。
setName(const std::string& name)：设置客户姓名。
setPriority(int priority)：设置客户优先级。

setQueueNumber(const std::string& queueNumber): 设置客户队列号码。
assignQueueNumber(): 分配队列号码。

2.3 Node.h

功能： 定义 Node 类，作为链表节点，用于构建客户队列。
数据成员：
data: 指向 Customer 对象的指针。
next: 指向下一个节点的指针。
prev: 指向上一个节点的指针。

2.4 Queue.h

功能： 定义 Queue 类，提供队列的基本操作。
数据成员：
head: 指向队列头节点的指针。
tail: 指向队列尾节点的指针。
成员函数：
enqueue(Customer* customer): 将客户入队。
dequeue(): 将客户出队。
isEmpty() const: 检查队列是否为空。
front() const: 获取队列头部的客户。
size() const: 获取队列大小。
insert(Customer* customer, int position): 在指定位置插入客户。
remove(Customer* customer): 移除指定客户。
begin() const: 获取队列头部节点。
end() const: 获取队列尾部节点。
~Queue(): 析构函数，释放内存。

2.5 BankQueueManager.cpp

功能： 实现 BankQueueManager 类的成员函数。

2.6 Customer.cpp

功能： 实现 Customer 类的成员函数。

2.7. Node.cpp

功能： 实现 Node 类的构造函数和析构函数。

2.8 Queue.cpp

功能： 实现 Queue 类的成员函数。

2.9 main.cpp

功能： 系统的主入口，提供用户界面和交互逻辑。

```
*********************************************************
*          银行信息管理系统菜单                          *
*     1 新增普通客户      2 新增VIP客户                  *
*     3 过号返回          4 显示队列                     *
*     5 叫号              0 退出系统                     *
*********************************************************
选择菜单项<0~4>: 1
请输入普通客户姓名: saf
普通客户 saf 已加入队列, 排队号码为: B3
```

三、逻辑结构

1.逻辑结构介绍

直观上看来，本系统应当采用树状结构，用一棵树的两个分支分别储存普通用户和 vip 用户的信息。但从实际编写的角度看，使用线性结构中的队列分别保存普通用户和 vip 用户的信息，不仅可以使代码编写更加简便，还可以使得代码的可阅读性更强。因此，本系统的逻辑结构为线性结构，并且以线性结构中的队列为主，链表等其他结构为辅。

2.相关线性结构

（1）队列

本系统中最明显的线性结构为队列，用于按照先进先出（FIFO）的原则存储和管理数据元素。在本系统中，Queue 类被用来管理 Customer 对象，分别通过 normalQueue 和 vipQueue 成员变量来管理普通客户和 VIP 客户的队列。具体的结构使用方法如下：

```
class Queue {
private:
    Node* head;
    Node* tail;
public:
    Queue();
    void enqueue(Customer* customer);
    Customer* dequeue();
    bool isEmpty() const;
    Customer* front() const;
    int size() const;
    void insert(Customer* customer, int position);
    void remove(Customer* customer);
    Node* begin() const;
    Node* end() const;
    ~Queue();
```

```
};
```

（2）链表

虽然本系统直观看来可以直接调用<queue>类实现相应的队列功能，但考虑到需要在队列内部进行相应的操作（比如过号激活的插入功能），本系统实则基于双向链表实现 Queue 的相关功能。这从 Node 类的定义（包含 next 和 prev 指针）以及 Queue 类的方法（如 enqueue 和 dequeue）可以看出。双向链表是一种线性结构，其中每个节点都包含指向其前一个和后一个节点的指针。

（3）数组变体

在 BankQueueManager 类中，allNormalCustomers 和 allVipCustomers 成员变量使用了 std::vector<Customer*> 类型。虽然 std::vector 在底层实现上通常是一个动态数组，但它提供了数组式的随机访问和动态大小调整的能力，因此在这里可以看作是数组的一种变体或扩展。不过，从逻辑结构的角度来看，std::vector 仍然保留了线性结构的特点，即元素之间按照索引顺序排列。

3.队列的优势与不足

由于本系统本身就是用于实现排队功能，与队列“先进先出”的特性完美契合，因此要完整实现本系统，只需对相应队列的优先级进行相关限定即可实现本系统所期望实现的功能。

然而，由于真正的队列不允许在队列中间插入元素，如果需要将过号顾客的信息重新插入到队列中（无论是普通队列还是 VIP 队列），只能通过重新建立一个链表（使用队列存储无法实现激活功能）储存过号客户的信息并对该链表的优先级进行限定，这会加大时间复杂度。

因此，本系统实际使用了链表编写队列的功能，打包后使用，这样虽然编写时会相对复杂，但最终既能实现队列的功能，又能解决在队列中插入元素的问题。

四、存储结构

存储结构基本介绍：

此排队系统基于几个核心的类：Customer、Node、Queue 和 BankQueueManager，共同构成了链式存储结构。

1） **链表的基本单元 Node：**Node 对象是队列中的结点，每个结点包含一个指向 Customer 对象的指针和指向前后结点的指针，多个结点共同构成双向链表。

```
class Node {
    friend class Queue;
private:
    Customer* data;       // 结点存储的客户数据
    Node* prev;           // 前一个结点指针
    Node* next;           // 后一个结点指针
    // 构造函数
    Node(Customer* data) : data(data), prev(nullptr), next(nullptr) {}
};
```

2）链表中客户内容存储 Customer：每个 Customer 对象代表一个客户，包含客户的 ID、姓名、优先级和队列编号。Customer 类有两个静态成员变量 normalCount 和 vipCount，用于追踪普通客户和 VIP 客户的编号。

```
class Customer {
public:
    string id;       // 客户 ID
    string name;     // 客户姓名
    int priority;    // 优先级（对于 VIP 客户来说更高）
};
```

链表存储结构的组织：

1）系统使用两个独立的队列来区分 VIP 客户和普通客户，这样可以确保 VIP 客户可以优先被服务。

2）每个客户在加入队列时都会被分配一个唯一的队列编号，这个编号是基于其加入的队列类型（VIP 或普通）和在该队列中的顺序。队列编号的生成是通过 BankQueueManager 类的 assignQueueNumber 方法实现的，该方法根据客户是否为 VIP 来决定编号的前缀（A/B），并使用静态变量来确保编号的唯一性。

链表存储的优点：

1）结点操作灵活：

在链表的头部或尾部进行插入和删除操作的时间复杂度为 O(1)，这意味着无论队列多长，添加或移除客户的操作都能在常数时间内完成。且由于链表是双向的，结点可以灵活地在队列中的任何位置插入或删除，这为实现复杂的队列管理功能（如优先级调整、过号重新排队等）提供了可能。

2）可扩展性好：

链表结构可以轻松扩展以支持额外的功能，如插入、删除、搜索特定结点等，而不需要对基础数据结构进行大规模的修改。

3）适应性：

链表可以很好地适应客户数量的变化，无论是在高峰时段还是低谷时段，系统都能有效地管理客户队列。

4）灵活的客户优先级处理：

可以在链表的任意位置插入 VIP 客户，保持优先服务的原则。

链表存储的不足：

1）内存使用上：

链表中的每个节点都包含额外的指针，在客户数量非常多的情况下这可能会大大增加内存的使用量。另外，动态分配的节点可能导致内存碎片，尤其是在频繁地创建和删除节点时。这可能影响系统的长期性能。

2）数据访问模式上：

链表不利于随机访问，因为访问链表中的任何元素都需要从头开始遍历，直到到达目标节点。这可能影响某些操作的性能，如查找特定客户。

3）发生异常的安全性上：

在异常发生时，需要确保链表的完整性不被破坏，例如在操作失败时能够回滚到稳定状

态，保证指针不丢失等。

五、算法思想

1. 客户类（Customer）的设计

Customer`类用于表示银行中的客户。

（1）属性：

id：客户的唯一标识。

name：客户的姓名。

priority：客户的优先级（用于区分 VIP 和非 VIP 客户）。

queueNumber：客户的队列号码。

（2）关键方法：

assignQueueNumber()：根据客户的优先级分配队列号码。VIP 客户队列号码以"A"开头，普通客户队列号码以"B"开头，后跟四位数字。

2. 节点类（Node）的设计

Node 类用于实现队列中的节点。

（1）属性：

data：指向`Customer`对象的指针。

next：指向队列中下一个节点的指针。

prev：指向前一个节点的指针。

（2）目的：实现队列中的节点，支持在双向链表中快速定位前一个和后一个节点。

3. 队列类（Queue）的设计

Queue 类实现了双向链表队列的基本操作。

（1）关键方法：

enqueue()：将客户添加到队列的末尾。

dequeue()：从队列的头部移除并返回客户，实现队列的先进先出（FIFO）特性。

isEmpty()：检查队列是否为空，用于判断队列中是否有客户。

front()：返回队列头部客户的指针，允许访问但不移除客户。

size()：返回队列中的客户数量，显示队列长度。

Insert()：在队列的指定位置插入客户，支持队列的动态调整。

remove()：从队列中移除指定的客户，支持队列的动态调整。

4. 银行队列管理类（BankQueueManager）的设计

`BankQueueManager`类负责管理银行的排队系统，包括普通客户队列和 VIP 客户队列。

（1）属性：

normalQueue ：普通客户队列。

vipQueue：VIP 客户队列。

allNormalCustomers：存储所有普通客户的列表。

allVipCustomers：存储所有 VIP 客户的列表。

vipCount 和 normalCount：分别记录 VIP 和普通客户的数量，用于生成唯一的队列号码。

（2）关键方法：

assignQueueNumber()：根据客户类型分配队列号码。

isDuplicateRegistration()：检查队列中是否存在重复的队列号码，确保每个客户的队列号码唯一。

findCustomerByQueueNumber()：根据队列号码查找客户，支持对客户的快速定位。

reEnQueueCustomer()：将客户重新加入队列，支持过号返回功能。

enqueueNormal()和 enqueueVIP()：分别将客户加入普通队列和 VIP 队列，实现客户分类管理。

serveCustomer()：服务下一个客户，优先服务 VIP 队列中的客户，实现优先级调度。

getNormalQueue()和`getVipQueue()：分别返回普通队列和 VIP 队列的指针。

析构函数：释放队列中所有客户对象占用的内存。

5. 主程序（main）的设计

（1）主程序提供了一个简单的文本菜单，允许用户执行以下操作：

新增普通客户或 VIP 客户到队列。

过号返回，即重新排队。

显示当前队列的状态。

叫号，即服务队列中的下一个客户。

退出程序。

（2）算法流程：

初始化 BankQueueManager 对象。显示菜单，接收用户输入。根据用户的选择执行相应的操作，例如新增客户到队列、重新排队、显示队列、服务客户等。循环执行，直到用户选择退出程序。实现了用户与系统之间的交互。

6. 内存管理

（1）客户对象的创建和销毁

当客户被添加到队列中时，是通过 new 关键字动态创建 Customer 对象。

当客户得到服务并从队列中移除后，在 serveCustomer 方法中，相应的 Customer 对象需要被销毁并释放内存。

（2）队列析构函数中内存管理

在 Queue 类的析构函数中，确保队列中所有剩余的客户对象都被删除。这是通过反复调用 dequeue 方法直到队列为空来实现的，每次 dequeue 都会返回一个客户对象，然后使用 delete 释放这个对象。

（3）避免内存泄漏

在客户被服务后，即从队列中移除时，相应的客户对象也被删除，以避免内存泄漏，确保系统的稳定性和性能。

7. 总结

通过以上算法思想，代码实现了银行排队管理系统的基本功能，能够有效地管理客户队列，确保服务的公平性和效率。系统的设计考虑了实际应用中的各种情况，如客户优先级、队列管理、用户交互和内存管理，使其成为一个实用的排队管理系统。

五、问题解决

1.多次释放链表内存的修订

```
#include "Node.h"

Node::Node(Customer* data) : data(data), next(nullptr), prev(nullptr) {}

//Node::~Node() { delete data; }
Node::~Node() { }
```

原本 node.cpp 和 customer.cpp 里面都进行了对于链表的内存释放，删除 node.cpp 中的内存释放函数后程序可正常运行。

2. VIP 和普通客户的优先级处理

问题: 在处理 VIP 和普通客户的队列时，如何确保优先服务 VIP 客户，并在 VIP 客户服务结束后继续处理普通客户?

解决方案: 我们使用了两个独立的动态链表，一个用于存储 VIP 客户，另一个用于普通客户。系统在处理叫号时，首先检查 VIP 队列，如果 VIP 队列不为空则优先处理 VIP 客户，处理完毕后再转到普通客户队列。这种队列分离的方式确保了优先级的正确处理。

3. 重新排队功能的实现

问题: 某些客户可能由于等待时间过长或其他原因，需要重新排队，这就要求我们系统能够支持重新排队功能。

解决方案: 为实现这一功能，我们在客户的队列号码基础上增加了一个重新排队操作。具体实现时，可以将客户重新插入到普通客户队列的末尾，从而保证队列的公平性。

4. 队列的动态管理

问题: 如何高效地管理两个动态链表，确保队列在客户不断进入和叫号的过程中保持稳定?

解决方案: 通过使用链表的头尾指针管理队列，插入和删除操作始终保持在 O(1)的复杂度，使得队列的动态变化更加高效。我们为 VIP 客户和普通客户分别维护了两个独立的链表，从而简化了每个客户类别的管理。

5. 输入数据校验

问题: 在输入客户信息或操作指令时，可能会遇到非法输入，如客户信息格式不正确、操作指令不在预期范围内。

解决方案: 我们增加了输入校验的机制，对于客户信息，系统会检查数据格式的有效性，对于操作指令，系统会在处理前验证输入的合法性，若发现非法输入，系统会输出“输入错误”并提示用户重新输入。

6. 多线程并发处理（可选扩展功能）

问题: 如果需要支持并发的多个窗口处理，如何在多个线程之间同步队列资源?

解决方案: 可以考虑使用线程锁或信号量来同步对共享队列的访问，确保多个线程在操作 VIP 和普通客户队列时不会发生冲突。这种扩展功能可以进一步提升系统的并发处理能力。

六、小组成员分工

第一阶段（上课讨论阶段）任务分工：

流程图制作：魏余昊
PPT 汇总：周灵菲
逻辑结构：杜泉睿
存储结构：魏嘉泽
算法思想：刘艺
小组成员分工、PPT 报告：张嘉麟

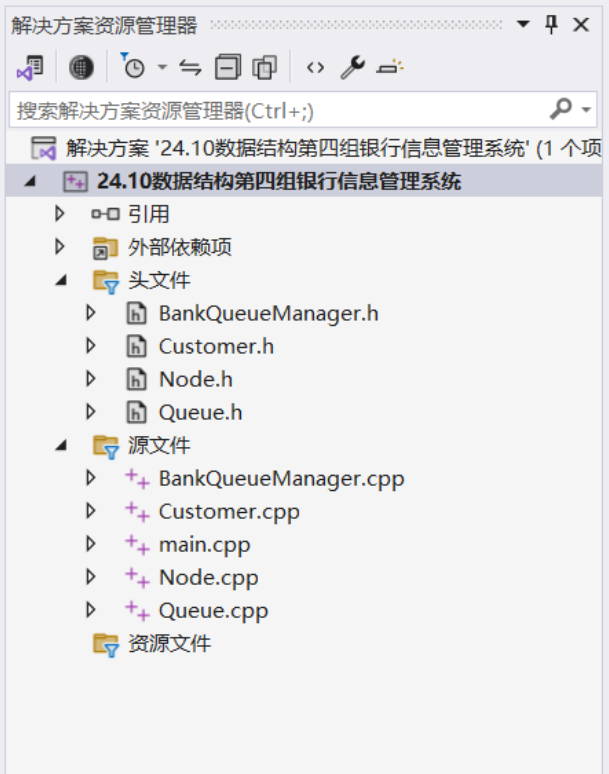
第二阶段（代码撰写阶段）任务分工：

逻辑结构储存结构的图表 负责：魏余昊 根据大纲修改
PPT 修改负责：周灵菲 根据大纲修改
Customer 文件负责：魏嘉泽 每个客人信息的信息存储功能
Queue 文件负责：杜泉睿 链表的相关功能
Bankqueuemanager 文件负责：刘艺 本系统核心功能，调用 Customer 和 Queue 文件来实现对挂号系统的总控
代码大纲撰写、小组成员分工、代码联排和 Debug：张嘉麟

第三阶段（报告撰写阶段）任务分工：

研究问题：魏余昊
代码框架+源代码：周灵菲
逻辑结构：杜泉睿
存储结构：魏嘉泽
算法思想：刘艺
小组成员分工、报告汇总和修改：张嘉麟

六、源代码（共 9 个文件，包含 4 个头文件，5 个 cpp 文件）



BankQueueManager.h

```
#ifndef BANKQUEUEMANAGER_H
#define BANKQUEUEMANAGER_H

#include "Queue.h"
#include <vector>
#include <string>

class BankQueueManager {
private:
    static int vipCount; // VIP 客户计数
    static int normalCount; // 普通客户计数
    Queue normalQueue; // 普通队列
    Queue vipQueue; // VIP 队列
    std::vector<Customer*> allNormalCustomers; // 所有普通客户
    std::vector<Customer*> allVipCustomers; // 所有 VIP 客户

    void assignQueueNumber(Customer* customer, bool isVIP); // 分配队列号码
    bool isDuplicateRegistration(const std::string& queueNumber); // 检查是否重复注册

public:
    BankQueueManager(); // 构造函数
```

```
    void reEnQueueCustomer(Customer* customer, bool isVIP, int position); // 重新入队客户
    void enqueueNormal(Customer* customer); // 入队普通客户
    void enqueueVIP(Customer* customer); // 入队 VIP 客户
    Customer* serveCustomer(); // 服务下一个客户
    Queue* getNormalQueue(); // 获取普通队列
    Queue* getVipQueue(); // 获取 VIP 队列
    ~BankQueueManager(); // 析构函数
    Customer* findCustomerByQueueNumber(const std::string& queueNumber); // 根据队列号码查找客户
    Customer* removeFirstNormalCustomer(); // 移除普通队列中的第一个客户
    Customer* removeFirstVipCustomer(); // 移除 vip 队列中的第一个客户

};
```

Customer.h

```
#ifndef CUSTOMER_H
#define CUSTOMER_H
#include <iomanip>
#include <string>
using namespace std;
class Customer {
private:
    static int normalCount;
    static int vipCount;
    std::string id;
    std::string name;
    int priority;
    std::string queueNumber;

    void assignQueueNumber();

public:
    Customer(std::string id, std::string name, int priority);

    std::string getId() const;
    std::string getName() const;
    int getPriority() const;
    std::string getQueueNumber() const;

    void setId(const std::string& id);
    void setName(const std::string& name);
    void setPriority(int priority);
    void setQueueNumber(const std::string& queueNumber);
```

```
};

#endif // CUSTOMER_H
```

Node.h

```
#pragma once
#ifndef NODE_H
#define NODE_H

#include "Customer.h"

class Node {
public:
    Customer* data;
    Node* next;
    Node* prev;

    Node(Customer* data);
    ~Node();
};

#endif // NODE_H
```

Queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

#include "Node.h"

class Queue {
private:
    Node* head;
    Node* tail;

public:
    Queue();
    void enqueue(Customer* customer);
    Customer* dequeue();
    bool isEmpty() const;
    Customer* front() const;
    int size() const;
    void insert(Customer* customer, int position);
    void remove(Customer* customer);
    Node* begin() const;
```

```
    Node* end() const;

    // Optional: to clear the queue when not needed
    ~Queue();
};

#endif // QUEUE_H
```

BankQueueManager.cpp

```
#include "BankQueueManager.h"
#include <sstream>
#include <iomanip>

int BankQueueManager::vipCount = 0; //  初始化 VIP 客户计数
int BankQueueManager::normalCount = 0; //  初始化普通客户计数

BankQueueManager::BankQueueManager() {}

void BankQueueManager::assignQueueNumber(Customer* customer, bool isVIP) {
    std::stringstream ss;
    if (isVIP) {
        ss << "A" << std::setw(4) << std::setfill('0') << ++vipCount; // VIP 客户队列编号格
式
    }
    else {
        ss << "B" << std::setw(4) << std::setfill('0') << ++normalCount; //  普通客户队列
编号格式
    }
    customer->setQueueNumber(ss.str()); //  设置客户的队列编号
}

bool BankQueueManager::isDuplicateRegistration(const std::string& queueNumber) {
    return findCustomerByQueueNumber(queueNumber) != nullptr; //  检查是否重复注册
}

Customer* BankQueueManager::findCustomerByQueueNumber(const std::string&
queueNumber) {
    for (Node* node = normalQueue.begin(); node != nullptr; node = node->next) {
        if (node->data->getQueueNumber() == queueNumber) return node->data; //  在
普通队列中查找
    }
    for (Node* node = vipQueue.begin(); node != nullptr; node = node->next) {
        if (node->data->getQueueNumber() == queueNumber) return node->data; //  在
VIP 队列中查找
```

```
    }
    return nullptr; // 没有找到客户
}

void BankQueueManager::reEnQueueCustomer(Customer* customer, bool isVIP, int
position) {
    (isVIP ? vipQueue : normalQueue).insert(customer, position); // 重新入队
}

Queue* BankQueueManager::getNormalQueue() { return &normalQueue; } // 获取普通队
列
Queue* BankQueueManager::getVipQueue() { return &vipQueue; } // 获取 VIP 队列

void BankQueueManager::enqueueNormal(Customer* customer) {
    allNormalCustomers.push_back(customer); // 记录所有普通客户
    normalQueue.enqueue(customer); // 入队普通客户
}

void BankQueueManager::enqueueVIP(Customer* customer) {
    allVipCustomers.push_back(customer); // 记录所有 VIP 客户
    vipQueue.enqueue(customer); // 入队 VIP 客户
}

Customer* BankQueueManager::serveCustomer() {
    if (!vipQueue.isEmpty()) {
        return vipQueue.dequeue(); // 优先服务 VIP 客户
    }
    if (!normalQueue.isEmpty()) {
        return normalQueue.dequeue(); // 服务普通客户
    }
    return nullptr; // 没有客户排队
}

BankQueueManager::~BankQueueManager() {
    while (!normalQueue.isEmpty()) {
        Customer* customer = normalQueue.dequeue();
        delete customer; // 释放内存
    }
    while (!vipQueue.isEmpty()) {
        Customer* customer = vipQueue.dequeue();
        delete customer; // 释放内存
    }
}
```

```
Customer* BankQueueManager::removeFirstNormalCustomer() {
    return normalQueue.dequeue(); // 移除并返回队列中的第一个客户
}


Customer* BankQueueManager::removeFirstVipCustomer() {
    return vipQueue.dequeue(); // 移除并返回队列中的第一个客户
}
```

Customer.cpp

```
#include "Customer.h"
#include <iomanip>
#include <sstream> // 添加这一行
int Customer::normalCount = 0;
int Customer::vipCount = 0;

Customer::Customer(std::string id, std::string name, int priority)
    : id(id), name(name), priority(priority) {
    assignQueueNumber();
}

std::string Customer::getId() const { return id; }
std::string Customer::getName() const { return name; }
int Customer::getPriority() const { return priority; }
std::string Customer::getQueueNumber() const { return queueNumber; }

void Customer::setId(const std::string& id) { this->id = id; }
void Customer::setName(const std::string& name) { this->name = name; }
void Customer::setPriority(int priority) { this->priority = priority; }
void Customer::setQueueNumber(const std::string& queueNumber) { this->queueNumber
= queueNumber; }


void Customer::assignQueueNumber() {
    std::ostringstream oss; // 使用 ostringstream 生成格式化的字符串
    if (priority == 1) { // VIP
        vipCount++;
        oss << "A" << std::setw(4) << std::setfill('0') << vipCount; // 生成 A0001 这样的
格式
    }
    else { // 普通客户
        normalCount++;
        oss << "B" << std::setw(4) << std::setfill('0') << normalCount; // 生成 B0001 这样
的格式
```

```
    }
    queueNumber = oss.str(); //  将格式化的字符串赋值给队列编号
}
```

main.cpp

```
#include "BankQueueManager.h"
#include <iostream>
#include <string>

using namespace std;

void menu() {
    cout
        << "*****************************************************\n"
        << "*                银行信息管理系统菜单                 *\n"
        << "*       1  新增普通客户        2  新增 VIP 客户       *\n"
        << "*       3  过号返回            4  显示队列            *\n"
        << "*       5  叫号                0  退出系统            *\n"
        << "*****************************************************\n"
        << "选择菜单项<0~5>: ";
}

int main() {
    //BankQueueManager bankQueueManager; //  创建  BankQueueManager  对象
    BankQueueManager manager;

    bool exitProgram = false;
    do {
        menu();
        int choice;
        cin >> choice;

        switch (choice) {
        case 1: {
            string name;
            cout << "请输入普通客户姓名: ";
            cin.ignore();
            getline(cin, name);
            Customer* normalCustomer = new Customer("", name, 0);
            manager.enqueueNormal(normalCustomer);
            cout << "普通客户  " << name << "  已加入队列，排队号码为: "
                 << normalCustomer->getQueueNumber() << "\n";
            break;
        }
```

```
            case 2: {
                string name;
                cout << "请输入 VIP 客户姓名: ";
                cin.ignore();
                getline(cin, name);
                Customer* vipCustomer = new Customer("", name, 1);
                manager.enqueueVIP(vipCustomer);
                cout << "VIP 客户 " << name << " 已加入队列，排队号码为: "
                    << vipCustomer->getQueueNumber() << "\n";
                break;
            }
            case 3: {
                string queueNumber;
                cout << "请输入过号客户的排队号码: ";
                cin >> queueNumber;
                Customer* customer =
manager.findCustomerByQueueNumber(queueNumber);
                if (customer) {
                    manager.reEnQueueCustomer(customer, customer->getPriority() == 1,
1);
                    cout << "客户已重新加入队列。\n";
                }
                else {
                    cout << "找不到该排队号码的客户。\n";
                }
                break;
            }
            case 4: {
                cout << "普通队列:\n";
                for (Node* node = manager.getNormalQueue()->begin(); node != nullptr;
node = node->next) {
                    cout << "队列号码: " << node->data->getQueueNumber() << ", 客户
姓名: " << node->data->getName() << endl;
                }
                cout << "VIP 队列:\n";
                for (Node* node = manager.getVipQueue()->begin(); node != nullptr; node =
node->next) {
                    cout << "队列号码: " << node->data->getQueueNumber() << ", 客户
姓名: " << node->data->getName() << endl;
                }
                break;
            }
            case 5: // 移除队列最前面的客户
            {
```

```
                Customer* removedCustomer = manager.removeFirstVipCustomer();
                if (removedCustomer) {
                    cout << "请 " << removedCustomer->getId() << "号病友前往看诊台"
<< endl;
                    delete removedCustomer; // 释放内存
                }
                else {
                    Customer* removedCustomer = manager.removeFirstNormalCustomer();
                    if (removedCustomer) {
                        cout << "请 " << removedCustomer->getId() << "号病友前往看诊
台" << endl;
                        delete removedCustomer; // 释放内存
                    }
                    else {
                        cout << "没有人在等待就医！" << endl;
                    }
                }
            }
            break;
            case 0:
                exitProgram = true;
                break;
            default:
                cout << "无效的选择，请重新输入。\n";
                break;
            }
        } while (!exitProgram);

        return 0;
}
```

node.cpp

```
#include "Node.h"

Node::Node(Customer* data) : data(data), next(nullptr), prev(nullptr) {}

//Node::~Node() { delete data; }
Node::~Node() {    }
```

Queue.cpp

```
#include "Queue.h"

Queue::Queue() : head(nullptr), tail(nullptr) {}

void Queue::enqueue(Customer* customer) {
	Node* newNode = new Node(customer);
	if (!tail) {
		head = tail = newNode;
	}
	else {
		tail->next = newNode;
		newNode->prev = tail;
		tail = newNode;
	}
}

Customer* Queue::dequeue() {
	if (!head) return nullptr;
	Node* temp = head;
	Customer* customer = temp->data;
	head = head->next;
	if (head) {
		head->prev = nullptr;
	}
	else {
		tail = nullptr;
	}
	delete temp;
	return customer;
}

bool Queue::isEmpty() const {
	return head == nullptr;
}

Customer* Queue::front() const {
	return head ? head->data : nullptr;
}

int Queue::size() const {
	int count = 0;
	Node* current = head;
```

```
    while (current) {
        count++;
        current = current->next;
    }
    return count;
}

void Queue::insert(Customer* customer, int position) {
    if (position < 0) return;

    Node* newNode = new Node(customer);
    if (position == 0) {
        newNode->next = head;
        if (head) {
            head->prev = newNode;
        }
        else {
            tail = newNode;
        }
        head = newNode;
        return;
    }

    Node* current = head;
    int index = 0;
    while (current && index < position - 1) {
        current = current->next;
        index++;
    }

    if (!current) return;

    newNode->next = current->next;
    newNode->prev = current;

    if (current->next) {
        current->next->prev = newNode;
    }
    else {
        tail = newNode;
    }

    current->next = newNode;
}
```

```
void Queue::remove(Customer* customer) {
    Node* current = head;
    while (current && current->data != customer) {
        current = current->next;
    }

    if (!current) return;

    if (current->prev) {
        current->prev->next = current->next;
    }
    else {
        head = current->next;
    }

    if (current->next) {
        current->next->prev = current->prev;
    }
    else {
        tail = current->prev;
    }

    delete current;
}

Node* Queue::begin() const {
    return head;
}

Node* Queue::end() const {
    return tail;
}

Queue::~Queue() {
    while (!isEmpty()) {
        Customer* customer = dequeue();
        delete customer; // Prevent memory leaks
    }
}
```