

课程设计

|  |  |  |  |
| --- | --- | --- | --- |
| 学院名称： | 信息产业学院 | | |
| 课程名称： | 数据结构 | | |
| 设计名称： | 银行排队系统 | | |
| 专业班级： | 211060104 | | |
| 成员1： | 21106010413 | 成员2： | 21106010401 |
| 成员3： | 21106010403 | 成员4： | 21106010428 |
| 学生姓名： | 刘钟泽、师玉娜、任嘉钰、赵昊峰 | | |
| 指导教师： | 高俊 | | |

2023年6月7日

# 项目分析

## 项目背景

排队是现实生活中最为常见的一个场景。作为学生，在食堂打饭的时候我们需要排队；领取演出的票的时候，我们需要排队；在银行的时候，我们需要排队办理业务。除了上述场景外，排队还广泛应用于公共交通、景点门票等领域。例如，公交车站有时会采用线性排队或者两端开口排队，以尽量提高乘客上车的效率。在景点门票领取方面，一些景点为了避免排队拥堵，采用了线上预订和取票的方式，让游客在刚到达时就能够愉快地体验景区。

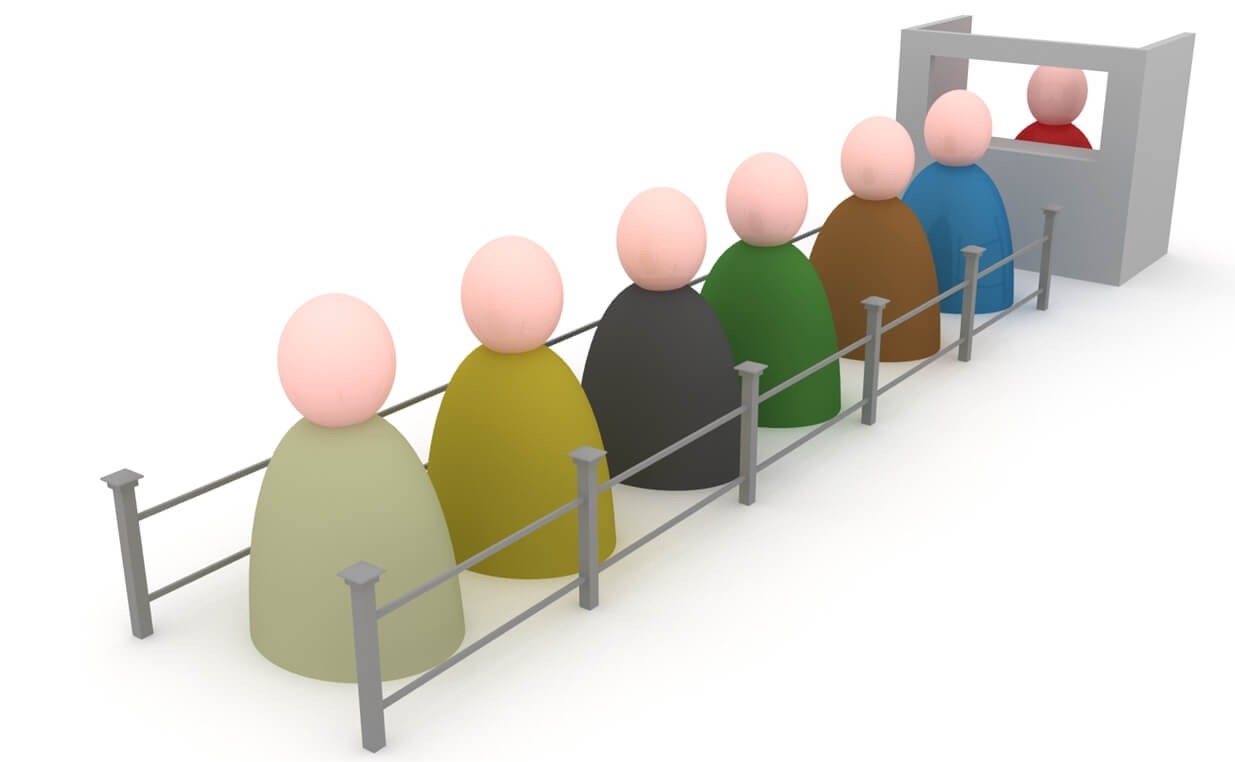


图 1.1‑1排队图

排队的重要性也是毋庸置疑的，大家自觉的排队实际上也能够规避很多潜在的安全隐患。在公共场合，如果没有人自觉排队，可能会造成拥挤和混乱，引发事故。此外，排队是现代文明的一种很重要的表现，并体现了人与人之间的相互关怀，营造出很好的社会氛围。国家甚至制定了排队日，旨在希望人们在排队的一点一滴中形成一种习惯和意识，让更多的人不自觉地履行这一公民的基本义务。

在实际排队过程中，除了人的自身素质因素外，还涉及到许多复杂的数学模型。比如，M/M/1排队模型是最基本和最常用的排队模型，其 中M表示顾客到达服从泊松分布，M也表示服务时间服从指数分布，1则表示只有一个服务器。当然，在实际场景中，各种因素都会对排队 产生影响，如顾客的到达间隔不稳定，需要考虑开放多个服务窗口等等。因此，研究排队问题具有极高的现实应用价值。

无论是理论研究还是应用探讨，排队都是一个值得深入探究的领域。实际上，有许多专业的排队研究机构和排队管理软件在帮助我们进一步提高排队的效率和质量，为我们的日常生活提供更加便捷的服务。毫无疑问，人们应该认识到排队的重要性，自觉地排队，从而构建更加和谐、安全的社会环境。

因此我们的研究很有意义。

## 项目需求分析

针对于银行业务这一项目，本项目在实现的过程中，应当能够满足以下的需求：

* **功能完善**

系统应当能够满足基本需求，即能够正确的按照完成顺序输出顾客的编号。

* **执行效率高**

针对数据量比较大的情况，本系统也应该具有在较短时间内求解出正确答案的能力。

* **健壮性**

当用户输入的数据非法时，系统应当能够识别并处理错误，而非直接崩溃退出。

## 项目要求

### 功能要求

某银行有A，B两个业务窗口，且处理业务的速度不一样，其中A窗口处理速度是B窗口的2倍——即当A窗口每处理完2个顾客是，B窗口处理完1个顾客。给定到达银行的顾客序列，请按照业务完成的顺序输出顾客序列。假定不考虑顾客信后到达的时间间隔，并且当不同窗口同时处理完2个顾客时，A窗口的顾客优先输出。

### 输入格式

输入为一行正整数，其中第一数字N（N<=1000）为顾客总数，后面跟着N位顾客的编号。编号为奇数的顾客需要到A窗口办理业务，为偶数的顾客则去B窗口。数字间以空格分隔。

### 输出格式

按照业务处理完成的顺序输出顾客的编号。数字键以空格分隔，但是最后一个编号不能有多余的空格。

### 项目示例

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 输入 | 输出 | 说明 |
| 1 | 8 2 1 3 9 4 11 13 15 | 1 3 2 9 11 4 13 15 | 正常测试，A窗口人多 |
| 2 | 8 2 1 3 9 4 11 12 16 | 1 3 2 9 11 4 12 16 | 正常测试，B窗口人多 |
| 3 | 1 6 | 6 | 最小N |

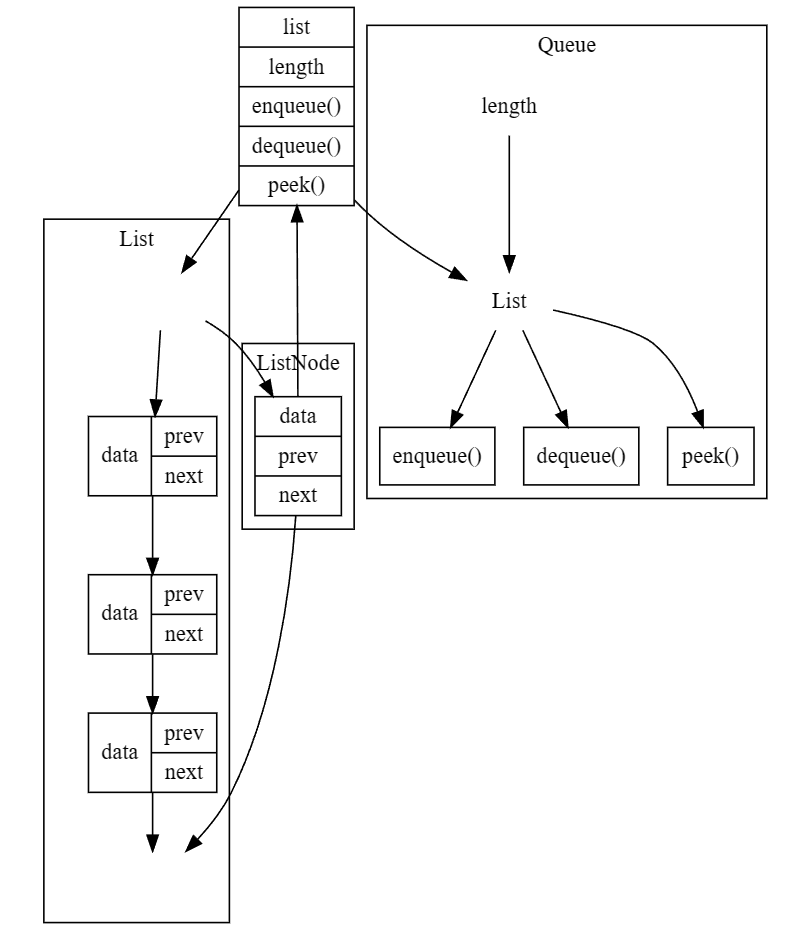
# 项目设计

## 数据结构设计

如上述功能分析所示，本项目需要实现一种先进先出的数据结构，也即队列。因此，本项目需要使用队列这一数据结构，从而满足所需要项目需求。

## 类设计

本项目中使用到队列（Queue）这一数据结构，而队列的底层数据类型则采用了链表进行实实现。经典的链表一般包括两个抽象数据类型（ADT）——链表结点类（ListNode）与链表类（List），而两个类之间的耦合关系可以采用嵌套、继承等多种关系。

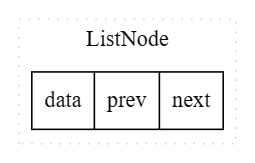


为了实现代码的复用性，本系统实现了一个**链表**。采用struct描述链表结点类（ListNode），这样使得链表结点类（List）可以直接访问链表结点而不需要定义友元关系。本系统实现的链表结构各种操作的时间复杂度如下：

* 插入操作：O(n)
* 删除操作：O(n)
* 查询操作：O(n)
* 遍历操作：O(n)

### 结点类（ListNode）

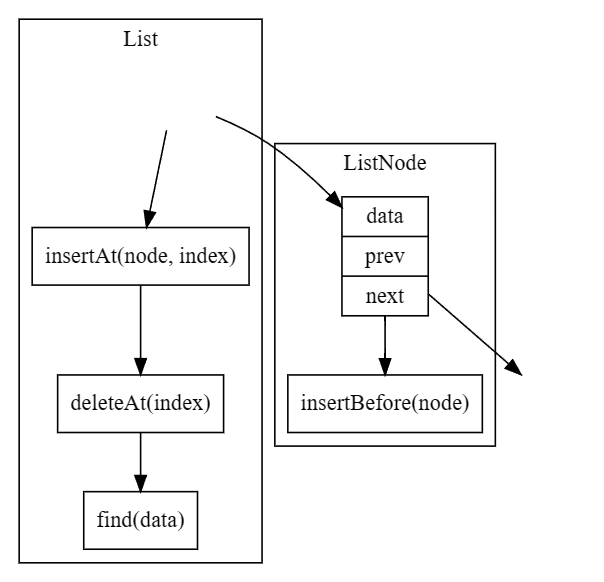
链表的结点中存储的数据有链表数据、后继结点和前驱结点。其UML图如下所示：



### 双向链表类（List）

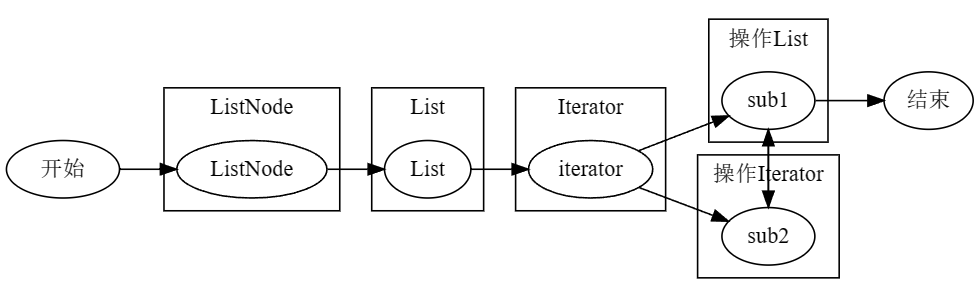
链表的实现原理大同小异，不同之处在于：是否带头结点、是否带尾结点、每一个结点是否带前驱结点等。为了使得链表中各种操作的时间复杂度都尽可能低，因此这里选择了带头结点和尾结点的双向链表来实现链表中的各种操作。也即：选择了牺牲空间来达到降低时间复杂度的效果。

链表的UML图如下所示：



为了便于对链表进行遍历、插入、删除、查找等操作，增加了一个iterator类。iterator类内部存储一个链表节点指针，同时，通过运算符重载相应的自增、自减、判等等操作。

iterator类、ListNode类和List类的关系如下图所示：



其中，List类中的主要函数如下所示：

* inline int **size**()const

返回链表中结点的个数，不包括头结点。

* inline bool **empty**()const

判断链表是否为空，也即链表中结点的个数是否为0。

* void **push\_back**(**Type** data)

在链表尾部插入新的数据，也即新增一个结点并且加入到链表末端。

* void **push\_front**(**Type** data)

在链表头部插入新的数据，也即新增一个结点并且加入到链表的头部。

* **iterator** **find**(const **Type**& data)const

在链表中查找值为data的元素是否存在，返回该位置的迭代器，若查找失败返回空指针对应的迭代器。

* **Type** **remove**(**iterator** index)

移除迭代器所处位置的元素，返回移除位置元素的值。

* void **erase\_last**()

移除链表末端的元素，即最后的结点。

* void **erase\_first**()

移除链表首端的元素，即第一个结点。

* void **clear**()

清空链表，即删除链表中所有的结点。

* **iterator** **begin**()

返回链表第一个元素所在位置的迭代器。

* **iterator** **end**()

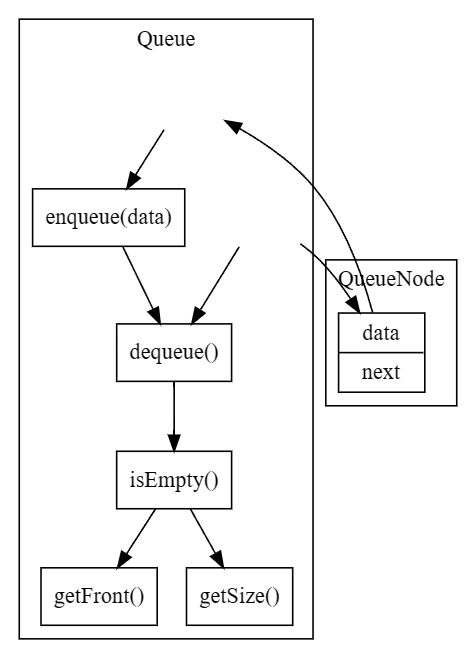
返回链表尾结点后的空结点的迭代器。

### 队列类（Queue）

队列是计算机程序中常用的数据结构，常常用于计算机模拟现实事务，如：排队等。队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，因此是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，被称为空队列。

队列的数据元素又称为队列元素。在队列中插入一个队列元素称为入队，从队列中删除一个队列元素称为出队。因为队列只允许在一端插入，在另一端删除，所以只有最早进入队列的元素才能最先从队列中删除，故队列又称为先进先出（FIFO—first in first out）线性表。

队列的包括了顺序队列和循环队列，实现存储的底层数据可以通过向量或者链表完成。本项目中的队列以链表作为底层数据结构，实现了顺序队列。其UNL图如下所示：



队列中主要函数如下所示：

* inline bool **empty**()const

判断队列是否为空，也即队列内部链表是否为空。

* inline int **size**()const

返回队列中链表节点的个数。

* void **push**(const **Type**& i)

在队列尾部加入一个元素，也即入队。

* void **pop**()

删除队列头部的元素，也即出队。

* const **Type**& **front**()const

获取队首的元素值。

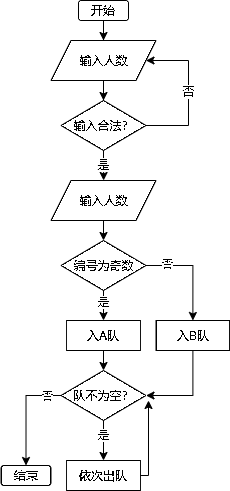
通过上述函数操作，即完成了一个队列所需要的最基本操作。其中，队的各个操作的时间复杂度如下所示：

* 入队操作：O(1)
* 出队操作：O(1)
* 读取操作：O(1)

# 项目实现

## 项目主体功能

### 项目主体功能流程图



### 项目主体功能代码

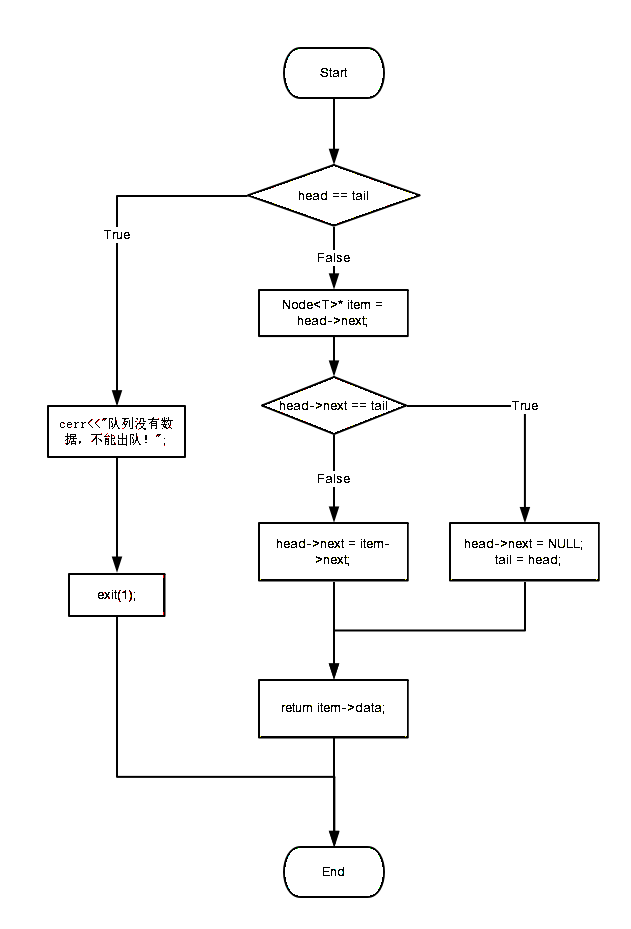
数据结构设计

该项目明显是对队列的一个应用实现典例，此题采用了链式队列进行操作，构建三个链式队列A，B，C用以解决问题。

类结构设计

模板类以及具体链式队列的实现类，以链表结点，头尾指针作为数据成员

为实现封装性：私有数据成员



struct Link\_node //队列结点，包含数据以及指针

{

T data;

Link\_node\* next;

};

Link\_node\* rear, \* front; //队尾，队头指针

模板类以及链式队列类（List\_Queue）

#pragma once

#include<iostream>

using namespace std;

template<class T>

class Queue //队列模板类创建

{

public:

virtual void clear() = 0; //将队列清空

virtual int size()const = 0; //求队列长度

virtual void In\_Queue(T& x) = 0; //入队列

virtual T Out\_Queue() = 0; //出队列

virtual T Read\_Queue()const = 0; //读取队头元素

virtual bool Is\_empty()const = 0; //判断队列是否为空

};

template<class T>

class Link\_Queue :public Queue<T>

{

public:

Link\_Queue() { front = NULL; rear = NULL; } //构造函数将队头，队尾指针置空

~Link\_Queue() { clear(); } //析构函数清除队列

void clear(); //将队列清空

int size()const; //求队列长度

void In\_Queue(T& x); //入队列

T Out\_Queue(); //出队列

T Read\_Queue()const; //读取队头元素

bool Is\_empty()const; //判断队列是否为空

private:

struct Link\_node //队列结点，包含数据以及指针

{

T data;

Link\_node\* next;

};

Link\_node\* rear, \* front; //队尾，队头指针

};

队列比较类（Compare\_Queue）

class Compare\_Queue //比较队列类

{

private:

Link\_Queue<int>Queue\_A; //定义三个int型队列

Link\_Queue<int>Queue\_B;

Link\_Queue<int>Queue\_C;

public:

Compare\_Queue() {}; //构造函数

~Compare\_Queue() {}; //析构函数

void Handling(); //办理业务实现函数

};

成员与操作设计

具体链式队列函数实现：

template<class T>

void Link\_Queue<T>::clear() //清空队列

{

Link\_node\* p;

while (front != NULL) //当队列头指针不为空（即队列不为空时）进行头指针后移逐个删除队列元素

{

p = front;

front = front->next;

delete p;

}

rear = NULL; //将尾指针置空

}

template<class T>

int Link\_Queue<T>::size()const

{

Link\_node\* p = front;

int count = 0;

while (p != NULL) //当头指针不为空时依次后移。count++，进行计数

{

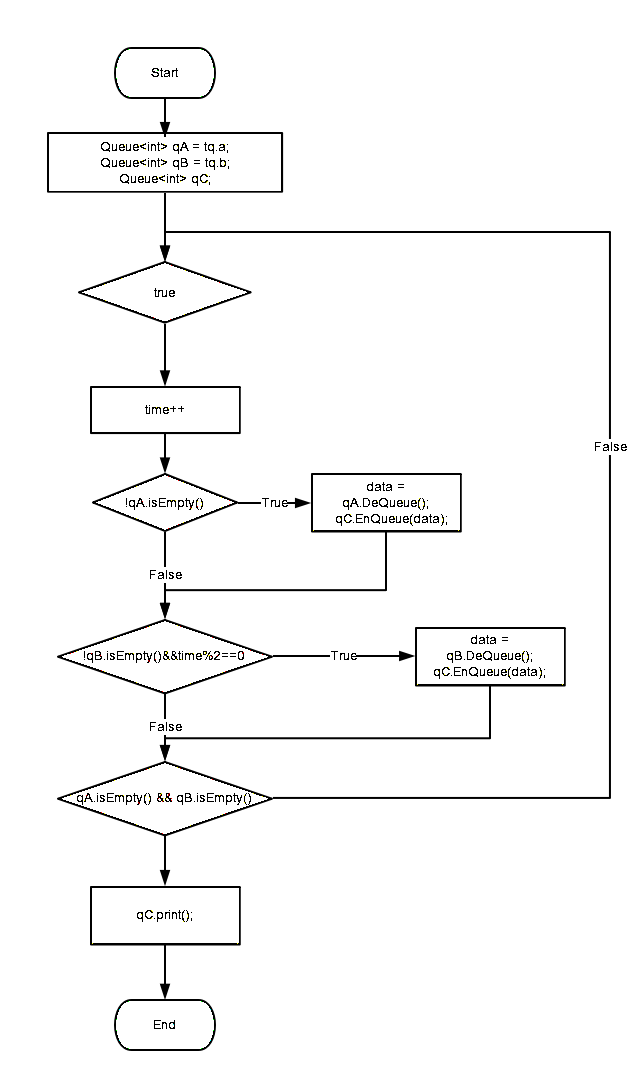
count++;

p = p->next;

}

return count;

}



template<class T>

void Link\_Queue<T>::In\_Queue(T& x)

{

Link\_node\* p = new Link\_node;

if (p == NULL)

{

cout << "存储分配错误！" << endl; //检查开辟新结点是否成功

exit(1);

}

if (rear == NULL) //如果此队列中无元素则进行插入第一个元素操作

{

p->data = x;

rear = p;

front = p;

rear->next = NULL;

}

else //如果队列中有元素则进行在队尾插入操作

{

p->data = x;

rear->next = p;

rear = p;

rear->next = NULL;

}

}

template<class T>

T Link\_Queue<T>::Out\_Queue()

{

Link\_node\* p=front;

T x;

while (front != NULL) //判断队列中是否有元素

{

if (p == NULL)

{

cout << "存储分配错误！" << endl; //判断建立结点是否成功

exit(1);

}

else //如果成功返回头指针数据值，并将头指针后移

{

p = front;

x = p->data;

front = front->next;

if (front == NULL) //原来只有一个元素出队后队列为空

{

rear == NULL; //修改队尾指针

}

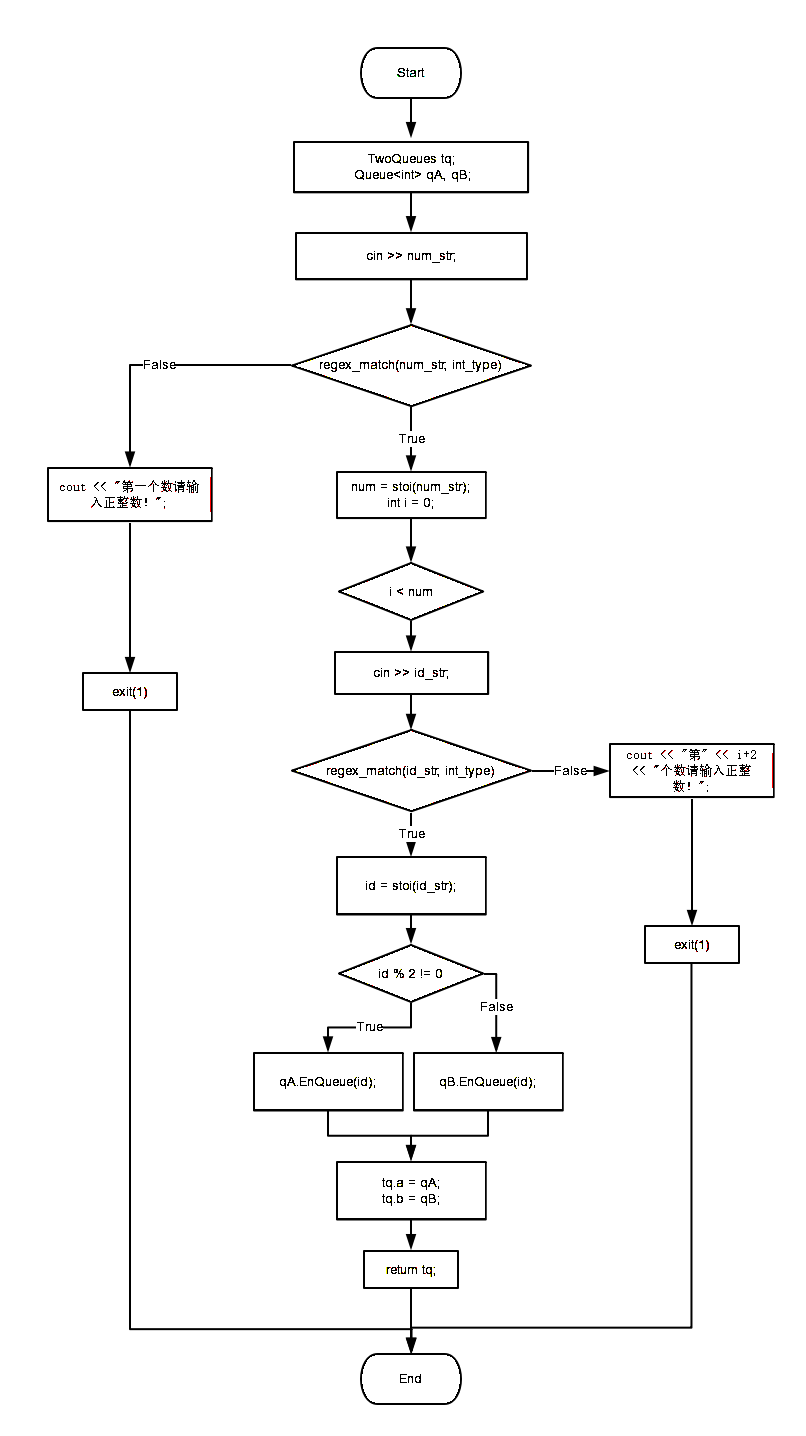
delete p;

return x;

}

}

}



template<class T>

T Link\_Queue<T>::Read\_Queue()const

{

if (front != NULL) //如果队列中有元素，返回队头元素的值

{

return front->data;

}

else

{

cout << "队列中无元素，不可读取队列首元" << endl;

}

}

template<class T>

bool Link\_Queue<T>::Is\_empty()const

{

if (rear == NULL||front==NULL) //头指针或者尾指针为空则队列为空

{

return true;

}

else

{

return false;

}

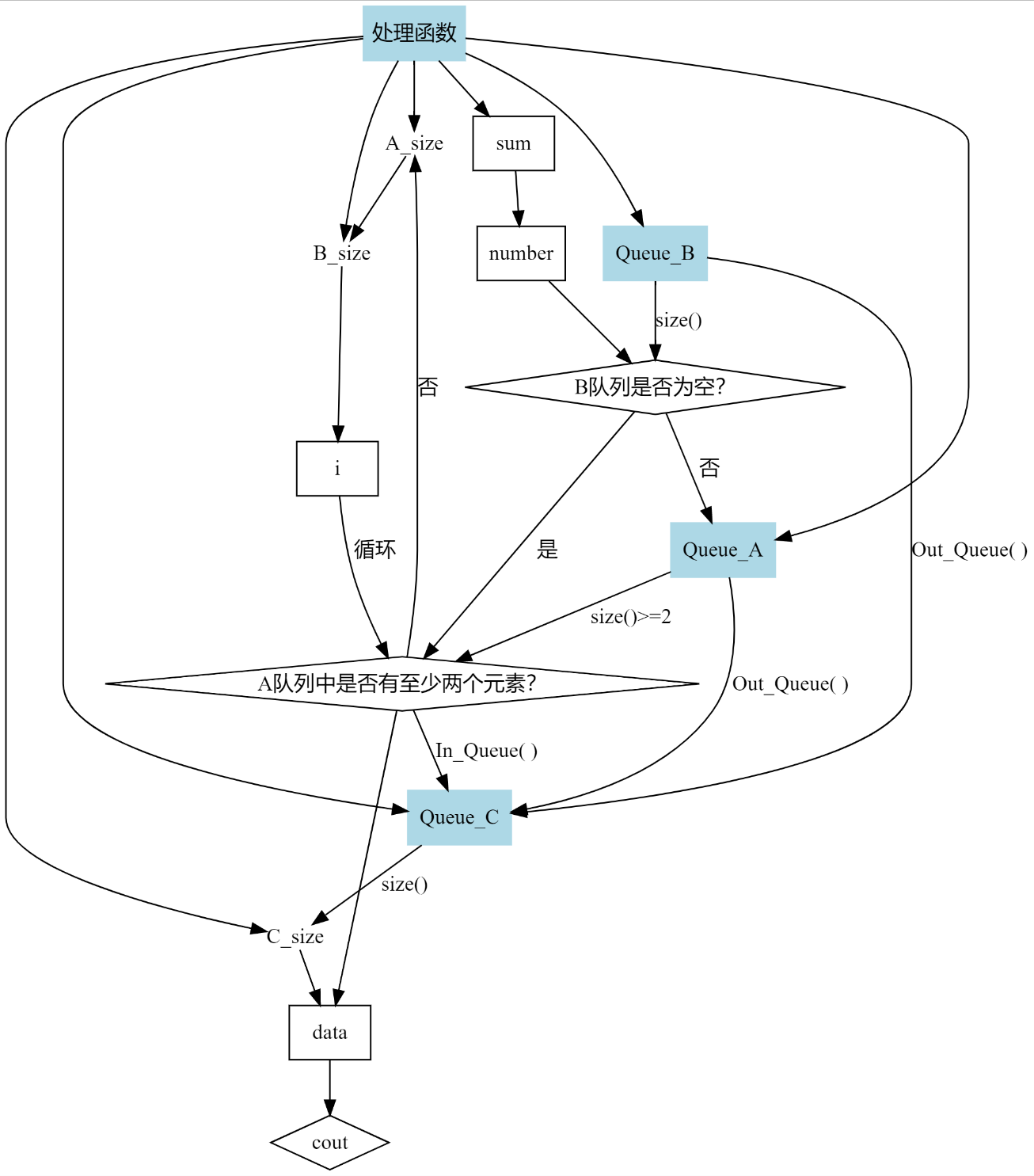
}

# 实现

## 银行业务功能的实现

### 银行业务功能流程简介

银行业务功能是将输入的编号判断奇、偶以后分别输入到A、B中，根据题目所给规则输出队列到C中再进行输出C中元素



### 银行业务功能核心代码

void Compare\_Queue::Handling()

{

int sum,number=0,data; //总的排队人数sum， 不同人的编号number，用于暂时存储数据

int A\_size,B\_size,C\_size; //记录A,B,C队列的长度

cin >> sum;

for (int i = 0; i < sum; i++) //循环输入总人数以及人员编号

{

cin >> number;

if ((number % 2) == 0) //人员编号为偶数进入B队列

{

Queue\_B.In\_Queue(number);

}

else //人员编号为奇数进入A队列

{

Queue\_A.In\_Queue(number);

}

}

while ((Queue\_A.size() >= 2) && (Queue\_B.Is\_empty() == false)) //当A队列还有至少两个元素，且B队列不为空时

{

data = Queue\_A.Out\_Queue(); //输出A队列中的两个元素，B队列中的一个元素

Queue\_C.In\_Queue(data);

data = Queue\_A.Out\_Queue();

Queue\_C.In\_Queue(data);

data = Queue\_B.Out\_Queue();

Queue\_C.In\_Queue(data);

}

if ((Queue\_A.size() < 2) && (Queue\_B.Is\_empty() == false)) //当A队列中元素少于2，B队列不为空时

{

B\_size = Queue\_B.size();

A\_size = Queue\_A.size();

for (int i = 0; i < A\_size; i++) //先将A队列元素全部输出，再将B队列元素全部输出

{

data = Queue\_A.Out\_Queue();

Queue\_C.In\_Queue(data);

}

for (int i = 0; i < B\_size; i++)

{

data = Queue\_B.Out\_Queue();

Queue\_C.In\_Queue(data);

}

}

if (Queue\_B.Is\_empty() == true) //如果B队列中元素为空

{

A\_size = Queue\_A.size();

for (int i = 0; i < A\_size; i++) //将A中元素全部放入C中

{

data = Queue\_A.Out\_Queue();

Queue\_C.In\_Queue(data);

}

}

if (Queue\_C.Is\_empty() == false) //如果C中有元素

{

C\_size = Queue\_C.size();

for (int i = 0; i < C\_size; i++) //将C中元素全部输出

{

data=Queue\_C.Out\_Queue();

cout << data;

if (C\_size - 1 != i) //保证C输出元素最后一个后无括号

{

cout << " ";

}

}

}

else //C中无元素

{

cout << "无人在排队" << endl;

}

}

### 银行业务功能截屏示例

正常的银行业务功能展示，未涉及边界情况。

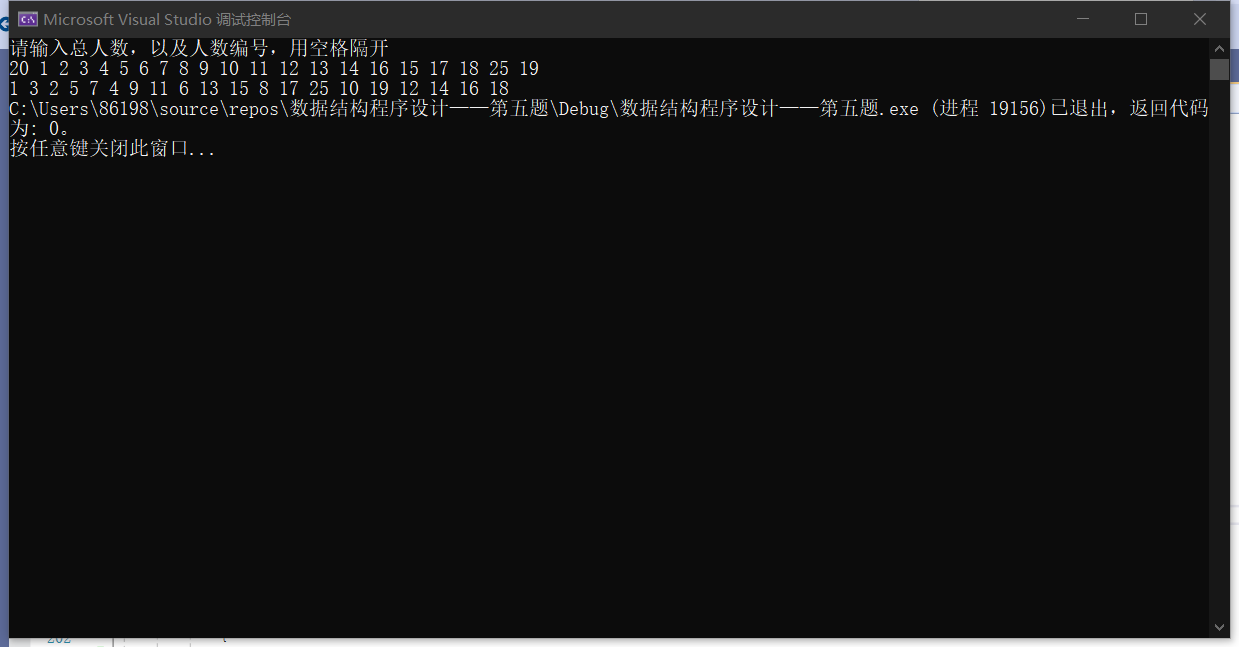
测试一：



测试二：



测试三：

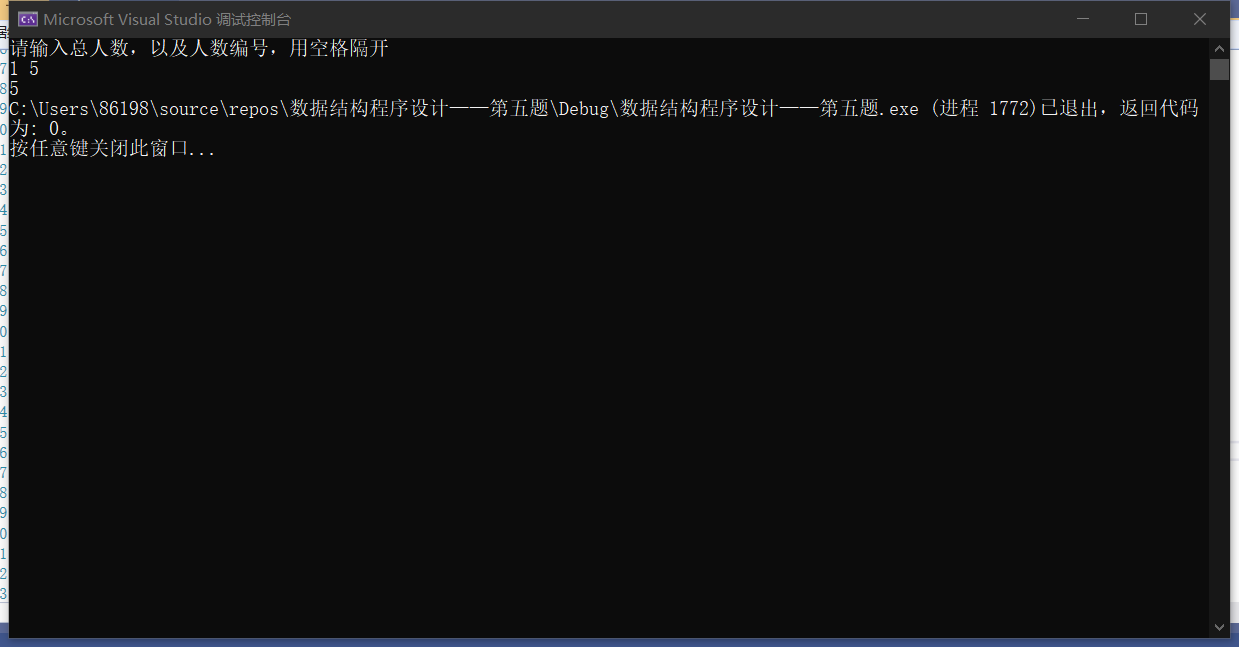


### 银行业务功能边界测试

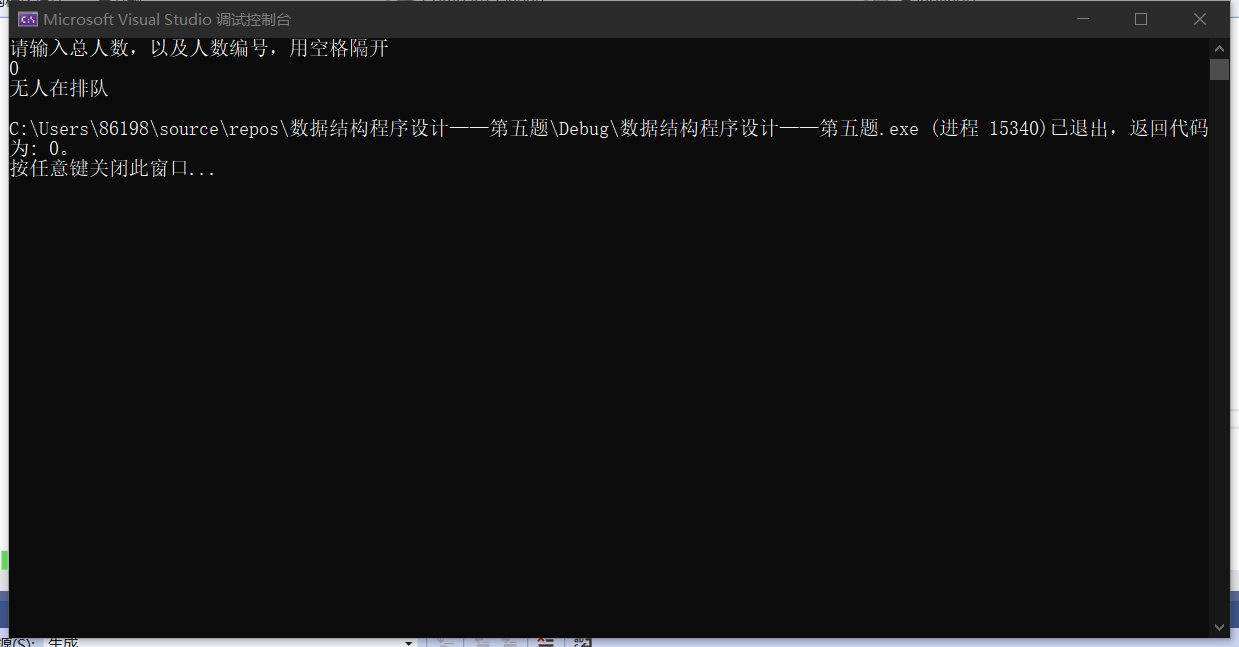
测试一（只有一个队列有元素）：



测试二：



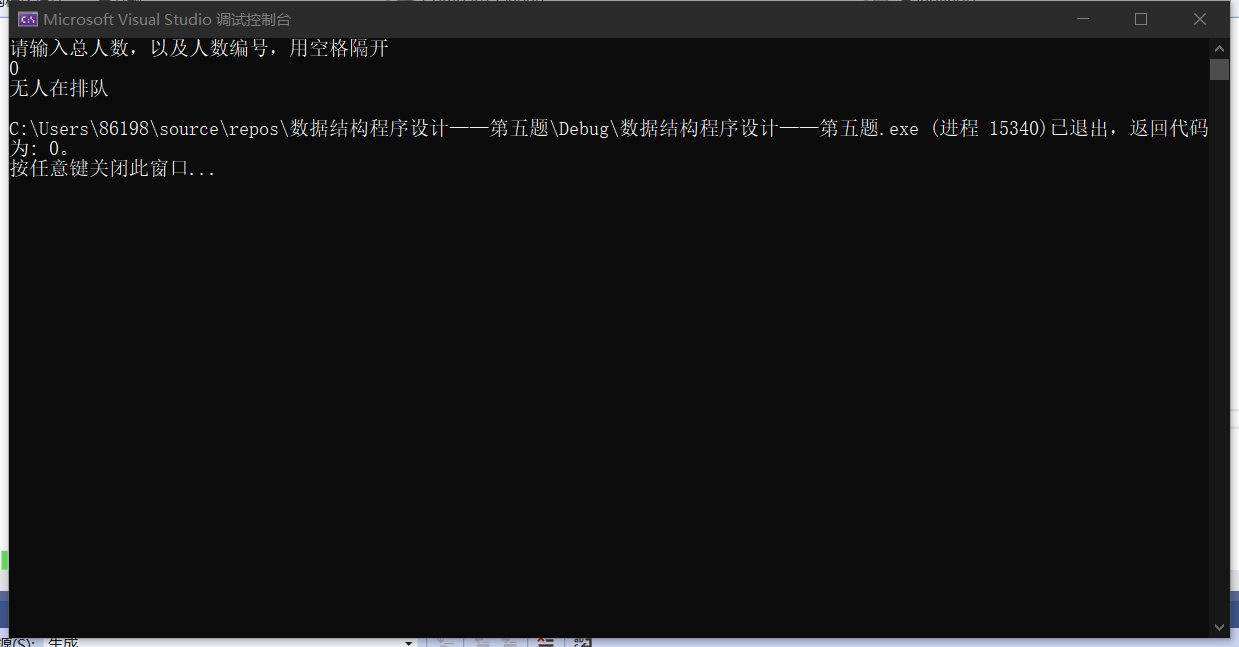
测试三（两队列均无元素）：



# 设计小结

在这次银行业务系统的编程中我掌握了队列的基本类实现，用模板类先进行声明再在子类中进行具体实现，在这次的银行业务设计中，我完成了基本建要求，也在一些地方做出了改进。

改进一、我进行了输入0时的情况的边界考虑



改进二、进行了无限制排队人数，采用链式栈，具有通用性。

当然除了改进以外我也在这个程序中学到了不少：我加深了对于模板类的理解，还巩固了队列的知识，最重要的是更加理解了封装的概念，对类的封装有了更加深刻的认识。希望以后能在编程中学到更多。