



数据结构课程设计文档

题目：银行业务

姓名： 赵卓冰

学号： 2252750

专业： 软件工程

年级： 2023 级

指导教师： 张颖

2024 年 11 月 5 日

开发环境与编译运行环境

项目概述

项目功能

- 1 输入说明
- 2 输出说明
- 3 关键规则

界面展示

- 1 Windows界面
- 2 linux界面

系统设计

- 1 数据结构
 - 1.1 顾客编号列表
 - 1.2 窗口类 `Window`
 - 1.3 队列类 `Queue`

核心算法

- 1 流程图

类与方法说明

- 1 类: `Vector<T>`
- 2 类: `Queue<T>`
- 3 类: `Window`
- 4 输入与输出函数

代码实现

总结和心得体会

- 1 队列的应用
- 2 窗口处理速度的模拟

1. 开发环境与编译运行环境

本项目可以在不同的开发环境和编译运行环境上运行

- Window操作系统：
 - 版本: Windows 10 x64
 - IDE: Visual Studio 2022 (Debug模式)
 - 编译器: MSVC 14.39.33519
- Linux操作系统：
 - 版本: Ubuntu 20.04.6 LTS
 - IDE: VS Code
 - 编译器: gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)

2. 项目概述

本项目旨在模拟一个银行中两个业务窗口的工作情况。两个窗口分别为A窗口和B窗口，其中A窗口的处理速度是B窗口的两倍，即每当A窗口处理完2个顾客时，B窗口处理完1个顾客。给定一批顾客到达银行，并根据顾客编号决定其选择的窗口（奇数编号的顾客选择A窗口，偶数编号的顾客选择B窗口）。系统根据各个顾客办理业务的顺序输出顾客的编号。

3. 项目功能

3.1. 输入说明

- 输入数据为一行正整数，其中第一数字N表示顾客总数，接下来是N个顾客的编号。
- 编号为奇数的顾客选择A窗口办理业务，编号为偶数编号的顾客选择B窗口办理业务。
- 输入数据格式：顾客总数 N 和顾客编号，数字间以空格分隔。

例如：

```
1 | 8 2 1 3 9 4 11 13 15
```

3.2. 输出说明

- 按照业务完成的顺序输出顾客的编号。
- 顾客编号之间以空格分隔，最后一个顾客编号后不加空格。

例如：

```
1 | 1 3 2 9 11 4 13 15
```

3.3. 关键规则

1. A窗口处理速度是B窗口的两倍。当A窗口每处理2个顾客时，B窗口处理1个顾客。
2. 当不同窗口同时处理完顾客时，A窗口的顾客优先输出。
3. 顾客到达银行后，按顺序分别加入到A窗口或B窗口的队列中，之后按照处理顺序离开。

4. 界面展示

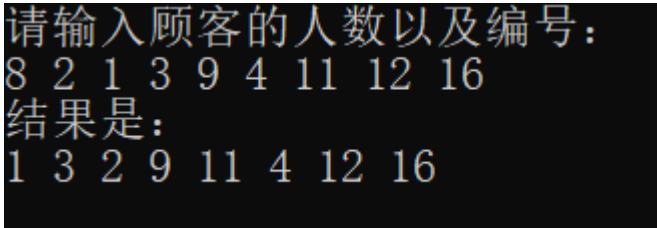
4.1. Windows界面

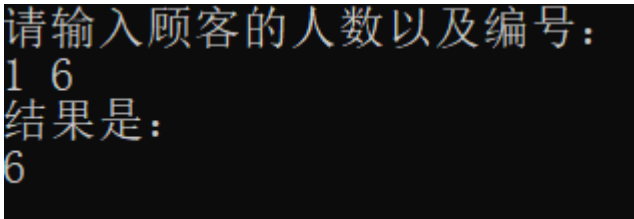
- 测试用例1

•

```
请输入顾客的人数以及编号：
8 2 1 3 9 4 11 13 15
结果是：
1 3 2 9 11 4 13 15
```

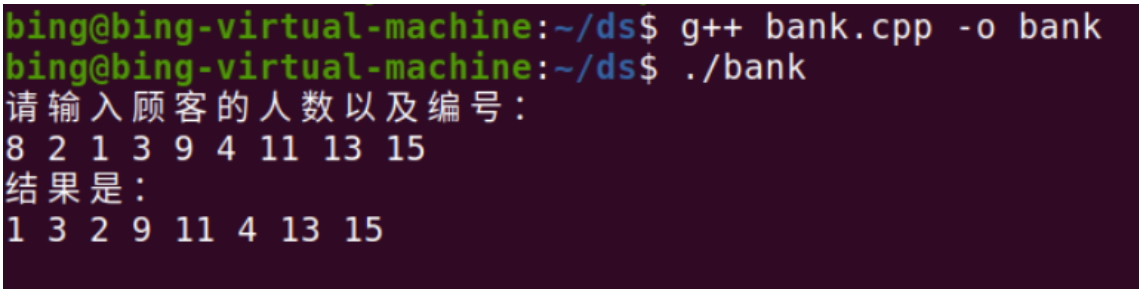
- 测试用例2

- 

```
请输入顾客的人数以及编号：
8 2 1 3 9 4 11 12 16
结果是：
1 3 2 9 11 4 12 16
```
- 测试用例3
- 

```
请输入顾客的人数以及编号：
1 6
结果是：
6
```

4.2. linux界面

- 

```
bing@bing-virtual-machine:~/ds$ g++ bank.cpp -o bank
bing@bing-virtual-machine:~/ds$ ./bank
请输入顾客的人数以及编号：
8 2 1 3 9 4 11 13 15
结果是：
1 3 2 9 11 4 13 15
```

5. 系统设计

5.1. 数据结构

5.1.1. 顾客编号列表

使用一个自定义的 `vector` 类存储顾客的编号。 `vector` 类实现了动态数组的功能，支持插入、删除、扩展容量等操作。

5.1.2. 窗口类 `window`

每个窗口有一个处理速度属性，并且有一个队列存储排队的顾客。窗口通过 `push` 方法接收顾客，通过 `pop` 方法按照窗口的处理速度处理顾客并将其移除。

5.1.3. 队列类 `Queue`

队列使用循环队列实现，支持高效的插入和删除操作。每个窗口内的顾客排队由队列来管理。

6. 核心算法

1. **顾客输入**：首先输入顾客数量和编号，根据编号的奇偶性将顾客分配到不同的窗口（奇数编号的顾客到A窗口，偶数编号的顾客到B窗口）。
2. **业务处理**：窗口A和B分别按照其处理速度（A窗口每次处理2个顾客，B窗口每次处理1个顾客）处理队列中的顾客。每次处理时，窗口A的顾客优先于窗口B的顾客。
3. **输出顾客编号**：根据顾客办理完业务的顺序输出顾客编号。

6.1. 流程图



7. 类与方法说明

7.1. 类： `Vector<T>`

- 成员变量：
 - `T* data`：动态数组，存储元素。
 - `size_t capacity`：当前分配的内存容量。
 - `size_t size`：当前存储的元素个数。
- 成员方法：
 - `PushBack(const T& value)`：向末尾插入一个元素。
 - `PopBack()`：删除末尾元素。
 - `Reverse()`：反转元素顺序。
 - `Clear()`：清空所有元素。

7.2. 类： `Queue<T>`

- 成员变量：
 - `T* data`：动态数组，存储队列元素。
 - `size_t front_index`：队列头部元素的索引。
 - `size_t back_index`：队列尾部元素的索引。
 - `size_t count`：当前队列中的元素个数。
 - `size_t capacity`：队列的容量。
- 成员方法：

- `Push(const T& value)`：向队尾插入元素。
- `Pop()`：删除队首元素。
- `Front()`：返回队首元素。
- `Back()`：返回队尾元素。

7.3. 类：window

- 成员变量：
 - `int speed`：窗口处理顾客的速度。
 - `Queue<int> clients`：队列，存储排队的顾客。
- 成员方法：
 - `Push(const int person)`：将顾客加入队列。
 - `Pop(Vector<int>& leave_order)`：处理顾客并将其编号添加到离开顺序列表中。

7.4. 输入与输出函数

- `Input(Vector<int>& people)`：输入顾客总数和顾客编号，存储在 `vector` 中。
- `Output(const Vector<int>& leave_order)`：输出顾客办理完业务后的离开顺序。

8. 代码实现

```
1  /*赵卓冰 2252750*/
2  #define _CRT_SECURE_NO_WARNINGS
3  #include <iostream>
4
5  // 条件编译，解决window和linux的差异
6  #ifdef _WIN32
7  #include <conio.h>
8  #endif
9
10 const int SPEED_A = 2; // A窗口的业务速度
11 const int SPEED_B = 1; // B窗口的业务速度
12
13 using namespace std;
14
15 // 向量类
16 template<typename T>
17 class Vector {
18 private:
19     T* data; // 数据指针，存储元素
20     size_t capacity; // 分配的内存容量
21     size_t size; // 当前存储的元素数量
22
23     // 扩展容量
24     void ExpandCapacity() {
25         size_t new_capacity = (capacity == 0) ? 1 : capacity * 2;
26         T* new_data = new T[new_capacity];
```

```

27     for (size_t i = 0; i < size; ++i) {
28         new_data[i] = data[i];
29     }
30     delete[] data;
31     data = new_data;
32     capacity = new_capacity;
33 }
34
35 public:
36     // 默认构造函数
37     Vector() : data(nullptr), capacity(0), size(0) {}
38
39     // 给定capacity的构造函数
40     Vector(size_t n) : capacity(n), size(n) {
41         data = new T[capacity];
42         for (size_t i = 0; i < size; ++i) {
43             data[i] = T();
44         }
45     }
46
47     // 给定capacity和初始值的构造函数
48     Vector(size_t n, T& value) : capacity(n), size(n) {
49         data = new T[capacity];
50         for (size_t i = 0; i < size; ++i) {
51             data[i] = value;
52         }
53     }
54
55     // 拷贝构造函数，允许初始化 Vector 的元素是其他 Vector 对象
56     Vector(const Vector& other) : capacity(other.capacity),
57     size(other.size) {
58         data = new T[capacity];
59         for (size_t i = 0; i < size; ++i) {
60             data[i] = other.data[i];
61         }
62     }
63
64     // 析构函数
65     ~Vector() {
66         delete[] data;
67     }
68
69     // 返回size
70     size_t Size() const {
71         return size;
72     }
73
74     bool Empty() {
75         return (size == 0);
76     }
77
78     void PushBack(const T& value) {

```

```

78     if (size == capacity) {
79         ExpandCapacity();
80     }
81     data[size++] = value;
82 }
83
84 void PopBack() {
85     if (size == 0) {
86         throw out_of_range("Vector is empty");
87     }
88     else {
89         --size;
90     }
91 }
92
93 void Reverse() {
94     int start = 0, end = size - 1;
95     while (start < end) {
96         swap(data[start], data[end]);
97         ++start, --end;
98     }
99 }
100
101 void clear() {
102     size = 0;
103 }
104
105 // 访问指定位置的元素（不带边界检查）
106 T& operator[](size_t index) {
107     return data[index];
108 }
109
110 const T& operator[](size_t index) const{
111     return data[index];
112 }
113 };
114
115 // 队列类
116 template <typename T>
117 class Queue {
118 private:
119     T* data;
120     size_t front_index; // 队列头部元素的索引
121     size_t back_index;  // 下一个可插入位置的索引，也就是尾部的后一个位置的
索引
122
123
124     size_t count; //当前队列中元素的数量
125     size_t capacity; // 队列的容量
126
127     void Resize() {
128         size_t new_capacity = capacity * 2; // 扩大容量为原来的两倍

```



```

129     T* new_data = new T[new_capacity];
130     for (size_t i = 0; i < count; ++i) {
131         new_data[i] = data[(front_index + i) % capacity];
132     }
133     delete[] data;
134     data = new_data;
135     capacity = new_capacity;
136     front_index = 0;
137     back_index = count;
138 }
139
140 public:
141     // 构造函数
142     Queue() : capacity(4), front_index(0), back_index(0), count(0) {
143         data = new T[capacity]; // 初始化容量为4
144     }
145
146     // 析构函数
147     ~Queue() {
148         delete[] data;
149     }
150
151     // 检查队列是否为空
152     bool Empty() const {
153         return count == 0;
154     }
155
156     // 返回队列中元素个数
157     size_t Size() const {
158         return count;
159     }
160
161     // 返回队首元素
162     T& Front() const {
163         if (Empty()) {
164             throw out_of_range("Queue is empty");
165         }
166         else {
167             return data[front_index];
168         }
169     }
170
171     // 返回队尾元素
172     T& Back() const {
173         if (Empty()) {
174             throw out_of_range("Queue is empty");
175         }
176         else {
177             return data[(back_index + capacity - 1) % capacity];
178         }
179     }
180

```

```
181 // 向队尾添加元素
182 void Push(const T& value) {
183     if (count == capacity) {
184         Resize();
185     }
186     data[back_index] = value;
187     back_index = (back_index + 1) % capacity; // 循环更新back索引
188     ++count;
189 }
190
191 // 删除队首元素
192 void Pop() {
193     if (Empty()) {
194         throw out_of_range("Queue is empty!");
195     }
196     else {
197         front_index = (front_index + 1) % capacity; // 循环更新
198         front_index索引
199         --count;
200     }
201 };
202
203 // 窗口类
204 class Window {
205 private:
206     int speed; // 处理业务的速度
207     Queue<int> clients; // 排队的客户
208
209 public:
210     Window(const int speed_value){
211         speed = speed_value;
212     }
213
214     bool Empty() {
215         return clients.Empty();
216     }
217
218     // person加入队列
219     void Push(const int person) {
220         clients.Push(person);
221     }
222
223     // 办理完业务的人离开队列
224     void Pop(Vector<int>& leave_order) {
225         for (int i = 0; i < speed; ++i) {
226             if (!Empty()) {
227                 leave_order.PushBack(clients.Front());
228                 clients.Pop();
229             }
230         }
231     }
```

```
232 };
233
234 // 输入函数
235 void Input(Vector<int>& people) {
236     cout << "请输入顾客的人数以及编号: " << endl;
237     int num; // 顾客的人数
238     cin >> num;
239     for (int i = 0; i < num; ++i) {
240         int number;
241         cin >> number;
242         people.PushBack(number);
243     }
244 }
245
246 // 输出函数
247 void Output(const Vector<int>& leave_order) {
248     cout << "结果是: " << endl;
249     for (size_t i = 0; i < leave_order.Size(); ++i) {
250         cout << leave_order[i];
251         if (i != leave_order.Size() - 1) {
252             cout << ' ';
253         }
254         else {
255             cout << endl;
256         }
257     }
258 }
259
260 int main() {
261     Vector<int> people; // 存储客户的向量
262     Input(people); // 输入客户向量
263
264     Window window_A(SPEED_A), window_B(SPEED_B); // 初始化两个窗口
265
266     for (size_t i = 0; i < people.Size(); ++i) {
267         // 如果是奇数
268         if (1 == people[i] % 2) {
269             window_A.Push(people[i]); // 加入A窗口队列
270         }
271         else {
272             window_B.Push(people[i]); // 加入B窗口队列
273         }
274     }
275
276     Vector<int> leave_order; // 存储客户离开顺序的向量
277
278     // 当A窗口非空或B窗口非空时
279     while (!window_A.Empty() || !window_B.Empty()) {
280         window_A.Pop(leave_order); // A窗口中顾客离开
281         window_B.Pop(leave_order); // B窗口中顾客离开
282     }
283 }
```

```
284     Output(leave_order); // 输出结果
285
286     cout<< endl << "按回车键退出程序..." << endl;
287     #ifdef _WIN32
288         while (_getch() != '\r')
289             continue;
290     #endif
291
292     return 0;
293 }
```

9. 总结和心得体会

在完成这个银行窗口业务处理模拟项目的过程中，我深刻体会到了数据结构和算法设计的重要性。本项目虽然表面上是一个简单的模拟程序，但通过对顾客流转过程的细致模拟，实际上需要我们对队列、窗口调度、以及并发处理进行较为全面的思考。这些问题在实际的生产环境中也是频繁出现的，例如在银行、医院、客服中心等场景中，都有类似的排队和处理问题。

9.1. 队列的应用

项目中使用了 `Queue` 类来模拟每个窗口的顾客排队情况。通过队列的先进先出（FIFO）特性，我们能够非常自然地管理每个窗口顾客的顺序。特别是在模拟窗口A和窗口B的业务处理速度时，队列帮助我们保持了顾客的顺序，同时也便于按照处理速度合理安排顾客的离开。

9.2. 窗口处理速度的模拟

窗口A和窗口B有不同的处理速度，窗口A的处理速度是窗口B的两倍。这个问题要求我们在程序设计中要合理地处理不同窗口的顾客优先级。通过在 `window` 类中实现不同的处理速度，利用 `Pop` 方法按窗口速度处理顾客，使得每个窗口的顾客可以按预期的顺序离开银行。这让我更深入地理解了如何根据不同的需求调整算法逻辑和优先级。

总的来说，这个项目不仅帮助我掌握了队列和动态数组的实现技巧，还提升了我对实际问题进行抽象建模的能力。在以后的学习中，我会继续保持对数据结构和算法的深入研究，力求在不同场景中运用自如。