

程序设计与算法综合训练

实验报告

实验名称： 银行业务模拟系统

专业班级： 人工智能二班

学号： WA2214014

姓名： 杨跃浙

目录

一、实验内容及要求	3
1.1 实验目的	3
1.2 实验内容	3
1.3 实验要求	3
1.4 实验任务	3
二、描述主要思路	4
2.1 问题要点:	4
2.2 描述主要思路	5
三、完善基本代码	6
3.1 原代码缺少的部分	6
3.2 代码补充	6
四、代码完善	9
4.1 修改原本代码中的纰漏	9
4.1.1 可能存在的问题	9
4.1.2 主要代码修改	10
4.1.3 调试	13
4.1.4 修改 clock 函数	14
4.1.5 调试	16
4.2 绝对时间	18
4.2.1 修改核心代码	18
4.2.2 调试	19
4.3 模拟一天工作时间	19
4.3.1 代码逻辑	19
4.3.2 代码调试	20
4.3.3 修改代码逻辑	21
4.3.4 调试过程	22
4.3.3 核心代码修改	32
4.4 模拟一周的工作	35
4.4.1 修改代码	35
4.4.2 代码效果	37
五、实验总结	37

一、实验内容及要求

1.1 实验目的

加深对队列数据结构的理解，掌握离散事件问题的模拟方法，强化学生的逻辑思维能力和动手能力，巩固良好的编程习惯，掌握工程软件设计的基本方法，为后续课程的学习打下坚实基础。

1.2 实验内容

问题描述：

假设某银行有四个窗口对外接待客户，从早晨银行开门起不断有客户进入银行。由于每个窗口在某个时刻只能接待一个客户，因此在客户人数众多时需在每个窗口前顺次排队，对于刚进入银行的客户，如果某个窗口的业务员正空闲，则可上前办理业务，反之，若四个窗口均有客户所占，他便会排在人数最少的队伍后面。现在需要编制程序以模拟银行的这种业务活动并计算一天中客户在银行逗留的平均时间。

1.3 实验要求

基本要求：

- (1) 初始化 (OpenForDay)，模拟银行开门时各数据结构—结构体的状态。
- (2) 事件驱动 (EventDriven)，对客户到达和离开事件做相应处理，即到达、排队、业务办理等。
- (3) 下班处理 (CloseForDay)，模拟银行关门时的动作，统计客户平均逗留时间。

1.4 实验任务

1. 请用自己的语言描述解决该问题的主要思路

1)...

2)...

2. 完成示例代码并解决其中的 bug

3. 输入和输出时间使用绝对时间（即日常生活使用的时间，比如对于 8:30，输入可以分开处理“时”和“分”，比如 8 30，也可以输入字符串“8:30”来处理，输出使用类似“8:30”的格式），

时间流逝单位为“分”

4. 计算某银行一周中的平均客户逗留时间

1) 上午 8:30 上班, 下午 5:00 下班

2) 周日不上班

3) 中午不休息

4) 中午休息 (12:00 至 14:00)

a. 一个值班窗口

b. 无值班窗口

二、描述主要思路

2.1 问题要点:

要点:

1. 队列

- 4 个服务窗口队列

2. 随机数

- 自定义随机范围
- 模拟客户到达时间和业务办理时长

3. 时间

- 编程需要相对时间 (上班时间为时间起点)
- 实际符合习惯是绝对时间

4. 自定义银行上班时长

- 上午 8:30 上班, 下午 5:00 下班
- 周日休息
- 中午不休息
- 中午休息

I. 无值班服务窗口

II. 有 1 个值班服务窗口

5. 客户结构体 (Customer)

- 到达时间: 相对时间或绝对时间
- 服务需要时长

6. 服务窗口结构体 (Window)

- 剩余服务时长
- 累计总等待时长
- 排队队列

7. 银行结构体 (Bank)

- 上下班时间
- 是否午休 (1/0)

- 值班窗口数
- 服务窗口队列数组
- 待进入客户队列
- 累计服务人数

2.2 描述主要思路

1.1) 初始化银行模拟环境: 在程序中定义一个 Bank 类, 包含窗口(Window)对象和队列(Queue)对象的数组, 每个窗口跟踪是否有客户正在接待, 每个队列管理等待该窗口的客户列表。使用变量来记录银行的开门时间和当前时间。

1.2) 定义类和数据结构, 使用数组来管理窗口和队列, 以及一个全局计时器变量。

2.1) 模拟客户到来和服务: 创建一个函数来模拟客户的到来。检查窗口的空闲状态, 并决定客户是直接被客户服务还是需要排队。如果排队, 则选择一个队伍, 这可以通过比较队列的长度来实现。

2.2) 编写一个函数, 遍历所有窗口状态, 找到空闲的窗口或最短的队列。可能会使用优先队列或其他数据结构以保持队列长度排序。

3.1) 处理客户业务: 为每个窗口分配一个计时器, 当客户服务完成时, 更新窗口状态并通知队列中的下一个客户。

3.2) 为每个窗口实现一个计时器(可能是一个整数或一个时间戳), 并编写代码处理客户的完成服务和窗口状态的更新。

4.1) 时间流逝处理: 使用一个循环来模拟时间的流逝, 每次循环代表一个时间单位的过去, 并更新全局计时器。

4.2) 通过循环结构和增加全局计时器的值来实现。

5.1) 数据收集与处理: 创建数据结构来记录每个客户的到达和离开时间, 以便计算他们的逗留时间。

5.2) 使用散列表或数组记录每个客户的到达时间和完成服务的时间。

6.1) 计算平均逗留时间: 在银行关闭时, 遍历所有服务过的客户, 累加他们的逗留时间并除以客户总数来计算平均值。

6.2) 编写一个函数, 在所有客户都服务完毕后调用, 计算总逗留时间并得出平均逗留时间。

7.1) 结束当日模拟: 在银行关闭时, 确保所有的客户都已经得到服务。对未完成服务的客户进行处理, 确保数据的完整性。

7.2) 在模拟的最后阶段, 确保处理所有的客户对象, 可能需要在循环结束后额外的逻辑来处理仍在队列中的客户。

三、完善基本代码

3.1 原代码缺少的部分

- 1) Queue 数据结构的实现：代码中多次提到了队列（Queue）及其相关操作，如“initQueue”、“enqueue”、“dequeue”和“getQueueHead”，但是并没有给出这些函数的定义和队列的数据结构。
- 2) NUM_WINDOWS 的定义：表示服务窗口数量的常量“NUM_WINDOWS”没有在代码中定义。
- 3) 服务时间和到达间隔的常量定义：“SIMULATION_DURATION”、“MAX_ARRIVAL_INTERVAL”、“MIN_SERVICE_TIME”、“MAX_SERVICE_TIME”等用于模拟的常量没有定义。
- 4) 随机数生成函数的实现：虽然代码中有调用“genRand”函数，但该函数的具体实现并未提供。
- 5) 时间种子设置函数“setSeed”的实现：代码中提供了“setSeed”函数的原型，但没有给出实现。
- 6) Node 结构的定义：在“printStatus”函数中提到了 Node 结构体，是队列的节点，但没有给出定义。
- 7) 全局计时器的实现：代码中没有显示全局计时器如何更新和使用。
- 8) genCustomers 函数的完整实现：“genCustomers”函数应该循环生成多个客户直到银行关门时间，但代码只展示了一个客户的生成。
- 10) main 函数中 Bank 对象的具体初始化：Bank bank; 语句只是声明了一个 Bank 类型的变量，但并没有具体初始化里面的每个服务窗口和队列。
- 11) 具体的实现细节、异常处理和边界条件的处理：队列的实现细节和具体的服务窗口逻辑。服务窗口队列为空时的情况等。

3.2 代码补充

主要补充代码：

1) 常量定义

```
#define NUM_WINDOWS 3
#define SIMULATION_DURATION 10
#define MAX_ARRIVAL_INTERVAL 10 // 最大到达间隔
#define MIN_SERVICE_TIME 5 // 最小服务时间
```

```
#define MAX_SERVICE_TIME 15 // 最大服务时间
```

2) 队列结构体和操作函数

```
typedef struct Node {  
    Customer data;  
    Node* next;  
} Node;
```

```
typedef struct Queue {  
    Node* head;  
    Node* tail;  
    int size;  
} Queue;
```

```
void initQueue(Queue* q) {  
    q->head = NULL;  
    q->tail = NULL;  
    q->size = 0;  
}
```

```
void enqueue(Queue* q, Customer customer) {  
    Node* newNode = new Node;  
    newNode->data = customer;  
    newNode->next = NULL;  
    if (q->head == NULL) {  
        q->head = newNode;  
        q->tail = newNode;  
    }  
    else {  
        q->tail->next = newNode;  
        q->tail = newNode;  
    }  
    q->size++;  
}
```

```
void dequeue(Queue* q) {  
    if (q->head != NULL) {  
        Node* temp = q->head;  
        q->head = q->head->next;  
        delete temp;  
        if (q->head == NULL) {  
            q->tail = NULL;  
        }  
        q->size--;  
    }  
}
```

```
}
```

```
Customer getQueueHead(Queue* q) {  
    if (q->head != NULL) {  
        return q->head->data;  
    }  
    else {  
        Customer emptyCustomer = { 0, 0, 0 };  
        return emptyCustomer; // 返回一个空的顾客结构体  
    }  
}
```

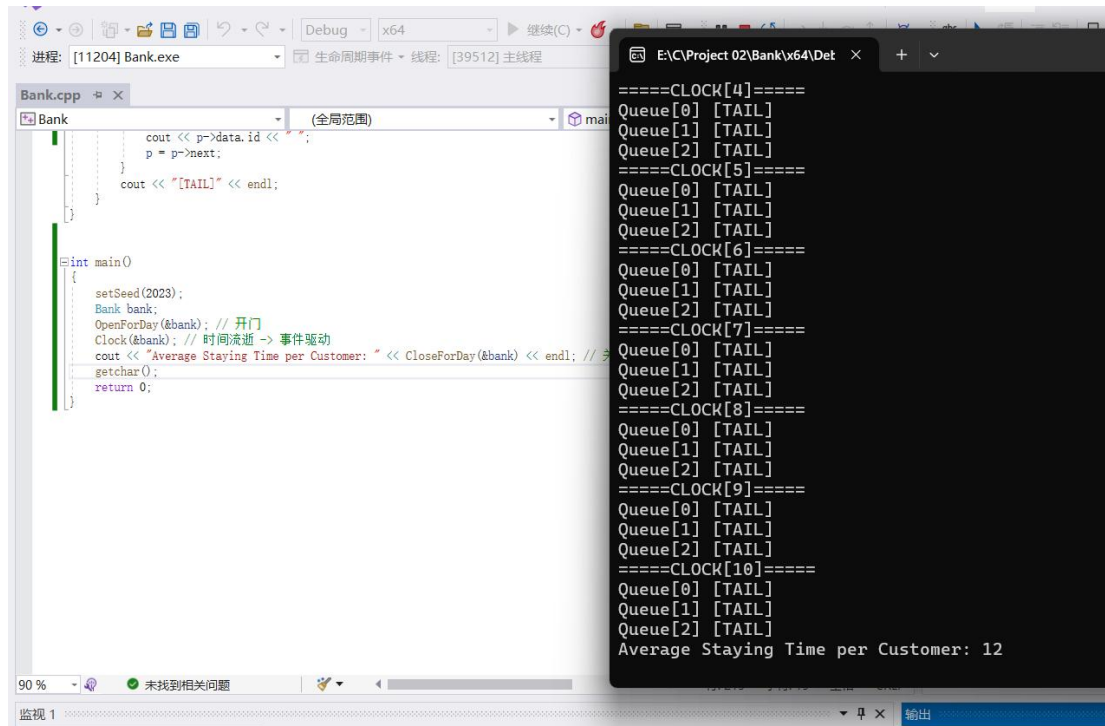
3) 缺少的函数

```
void setSeed(int seed = -1) { // -1 使用时间作为随机种子
```

```
    if (seed == -1) {  
        srand(time(NULL));  
    }  
    else {  
        srand(seed);  
    }  
}
```

```
int genRand(int min, int max) {  
    return min + rand() % (max - min + 1);  
}
```

完成代码补充之后，程序能够运行 但不完全符合预期输出



四、代码完善

4.1 修改原本代码中的纰漏

4.1.1 可能存在的问题

1. `genCustomers` 函数只生成了一个客户：

当前的 `genCustomers` 函数只生成了一个客户，而模拟需要多个客户。你可能需要在一个循环中生成多个客户，或者通过某种机制在整个模拟过程中不断地生成客户。

2. 客户的 `id` 没有递增：

在 `genCustomers` 函数中，客户的 `id` 被设置为 0，并且在之后没有更新。这可能导致所有客户都有相同的 `id`，使得追踪单个客户变得困难。应当有一个机制保证每个生成的客户都有唯一的 `id`。

3. `EventDriven` 函数中队列的选择逻辑有误：

在 `EventDriven` 函数中，寻找最短队列的逻辑可能有错误。变量 `min` 被初始化为 0，然后在循环中用于比较队列的大小。这可能导致永远选择第一个窗口，因为 `min` 没有被设置为有效的最大值或队列大小以开始比较。

4. `EventDriven` 函数中剩余服务时间的处理可能不正确：

当检查 `bank->windows[min].remaining_service_time > 0` 并增加总等待时间时，应当确保这个逻辑正确反映了当前窗口的状态和顾客的等待时间。

5. 没有处理客户离开后的 `id` 打印：

在 `printStatus` 函数中，打印队列状态时没有处理已经离开队列的客户。理论上，离开队列的客户的 `id` 不应再打印出来。

6. 服务窗口状态的更新可能有问题：

在 `Clock` 函数中，服务窗口的状态更新可能有逻辑上的问题。尤其是当 `cur_remaining_service_time` 为 0 时，应当确保正确地移除客户，并更新窗口状态。

7. 内存泄漏的风险：

`dequeue` 函数中正确删除了节点，但如果这是队列中的最后一个节点，`q->tail` 应当设置为 `NULL`。另外，应当确保在整个程序结束时释放所有分配的内存，以避免内存泄漏。

8. 顾客队列中的服务时间未被使用：

在 `Clock` 函数中，从顾客队列中获取的 `cur_customer` 的 `service_time` 在顾客到达其服务窗口前并未被使用。这应该在事件驱动环节被考虑进去。

9. 关闭银行时未考虑尚未完成服务的客户：

CloseForDay 函数计算平均逗留时间时，假设所有顾客都已完成服务。但如果有顾客在银行关闭时仍在排队或正在服务，这些顾客的逗留时间似乎没有被考虑进去。

10. 错误处理和边界条件的处理不足：

程序中缺少错误处理，例如队列操作失败、内存分配失败等情况。此外，边界条件（如队列为空时的操作）的处理也不足，这在健壮的程序设计中是很重要的。

11. 程序的稳健性和异常情况处理：

在实际的系统中，应当考虑各种异常情况，例如，客户到达但所有队列都满了，或者客户到达时间超出银行营业时间等情况。这些都需要在程序中被妥善处理。

12. 客户离开逻辑的实现可能有误：

在 Clock 函数中，当客户服务完成后应当离开，但目前的逻辑似乎是在减少服务时间之后立即移除，而不考虑客户的实际服务时间是否已经结束。

13. Clock 函数中的窗口遍历可能存在错误：

在遍历各个窗口并减少服务时间的代码段中，应该使用 bank->windows[i] 而不是 bank->windows->。

14. 逻辑上的服务时间统计可能不准确：

在 EventDriven 函数中，你增加了 total_wait_time，这应该代表客户在队列中的等待时间。然而，你在顾客入队时也增加了服务时间，这可能导致对总等待时间的双重计算。

15. CloseForDay 函数中的平均逗留时间计算：

此函数的计算假定所有客户都已经完成了服务，而实际上，应该只计算那些已经完成服务的客户的等待时间。此外，如果你希望包括在关门时仍在排队的客户，需要有一个机制来记录每个客户实际离开的时间。

16. genCustomers 函数中的顾客到达逻辑：

现在的逻辑是在银行开门时生成一个顾客，实际上，顾客可能在任何时间到达。应该有一个机制在整个模拟过程中定期检查并生成新的顾客。

4.1.2 主要代码修改

1. 使 genCustomers 函数在模拟时间内持续生成客户，并为每个客户分配一个唯一的 id。

```
void genCustomers(Bank* bank)
{
    // 生成的顾客应该分布在整个模拟时间内
    int customer_id = 1; // 开始的顾客 ID
    for (int t = bank->open_time; t < bank->close_time; t += genRand(1, MAX_ARRIVAL_INTERVAL)) {
        Customer customer = {
            customer_id++,
            t,

```

```

        genRand(MIN_SERVICE_TIME, MAX_SERVICE_TIME)
    };
    enqueue(&bank->customer_queue, customer);
    cout << "Customer " << customer.id << " Arrived at " << customer.arrival_time << " Estimated
Service Time is " << customer.service_time << endl;
}
}

```

2. 在 EventDriven 函数中修复选择最短队列的逻辑。

```

void EventDriven(Bank* bank, Customer* cur_customer, int now)
{
    int min_index = 0; // 用于跟踪最短队列的窗口索引
    int min_size = INT_MAX; // 设置一个较大的数作为比较基准
    for (int i = 0; i < NUM_WINDOWS; ++i) {
        if (bank->windows[i].queue.size < min_size) {
            min_index = i;
            min_size = bank->windows[i].queue.size;
        }
    }
    int min = min_index;
    // 顾客入队
    enqueue(&bank->windows[min].queue, *cur_customer);
    bank->num_to_service += 1; // 总顾客数+1
    // 统计时长
    if (bank->windows[min].remaining_service_time > 0) { // 1. 统计等待时长；要考虑到每个人的等待时
        长，所以要累加剩余服务时长
        bank->windows[min].total_wait_time += bank->windows[min].remaining_service_time;
    }
    bank->windows[min].total_wait_time += cur_customer->service_time; // 2. 统计服务时长
    if (bank->windows[min].remaining_service_time == 0) {
        bank->windows[min].cur_remaining_service_time = cur_customer->service_time; // 3. 总剩余服
        务时长为 0，说明该客户为当前队列“第一个”客户
    }
    bank->windows[min].remaining_service_time += cur_customer->service_time; // 4. 刷新该队列的剩余
    服务时长
}

```

3. 确保 CloseForDay 函数在计算平均逗留时间时，也考虑到了银行关闭时仍在服务的客户。在程序结束时确保所有内存被释放，避免内存泄漏。

```

double CloseForDay(Bank* bank)
{ // 计算客户平均逗留时长
    double total_time = 0;
    for (int i = 0; i < NUM_WINDOWS; ++i) {
        total_time += bank->windows[i].total_wait_time;
    }
}

```

```

    }
    for (int i = 0; i < NUM_WINDOWS; ++i) {
        // 如果队列中还有客户，则继续累加他们的服务时间
        Node* p = bank->windows[i].queue.head;
        while (p != NULL) {
            total_time += bank->close_time - p->data.arrival_time;
            p = p->next;
        }
    }
    while (bank->customer_queue.size > 0) {
        dequeue(&bank->customer_queue);
    }
    return total_time / bank->num_to_service; // 假设关门时没有办理完成的顾客继续办理直到完成
}

```

4. 在 Clock 函数中修复服务窗口状态更新的逻辑。

```

void Clock(Bank* bank) {
    int service_duration = bank->close_time - bank->open_time; // 银行开门总时长
    for (int now = 0; now < service_duration + 1; ++now) { // 时间流逝
        // 遍历每个服务窗口，更新服务状态
        for (int i = 0; i < NUM_WINDOWS; ++i) {
            // 若当前服务窗口有客户，则减少服务时间
            if (bank->windows[i].remaining_service_time > 0) {
                bank->windows[i].remaining_service_time -= 1;
            }

            // 若当前正在办理业务的客户的服务时间结束，移除该客户
            if (bank->windows[i].cur_remaining_service_time > 0) {
                bank->windows[i].cur_remaining_service_time -= 1;
                if (bank->windows[i].cur_remaining_service_time == 0) {
                    dequeue(&bank->windows[i].queue); // 客户服务完成，移除客户
                    // 如果队列中还有客户，立即为下一位客户提供服务
                    if (bank->windows[i].queue.size > 0) {
                        Customer next_customer = getQueueHead(&bank->windows[i].queue);
                        bank->windows[i].cur_remaining_service_time = next_customer.service_time;
                    }
                }
            }
        }
    }
    // 处理新到达的顾客
    while (bank->customer_queue.size > 0) {
        Customer cur_customer = getQueueHead(&bank->customer_queue);
        // 如果当前时刻有顾客到达，调用 EventDriven 函数进行处理
        if (cur_customer.arrival_time <= now) {

```

```

        dequeue(&bank->customer_queue); // 从顾客队列中移除该顾客
        EventDriven(bank, &cur_customer, now);
    }
    else {
        // 如果队头顾客尚未到达，则退出循环
        break;
    }
}
// 打印当前银行状态
printStatus(bank, now);
}
}

```

4.1.3 调试

```

j Bank
void printStatus(Bank* bank, int now)
{ // 打印当前队列状态
    cout << "====CLOCK[" << now << "]====" << endl;
    for (int i = 0; i < NUM_WINDOWS; ++i) {
        Node* p = bank->windows[i].queue.head;
        cout << "Queue[" << i << "] ";
        while (p != NULL) {
            cout << p->data.id << " ";
            p = p->next;
        }
        cout << "[TAIL]" << endl;
    }
}

int main()
{
    //setSeed(2023);
    Bank bank;
    OpenForDay(&bank); // 开门
    Clock(&bank); // 时间流逝 -> 事件驱动
    cout << "Average Staying Time per Customer: " << CloseForDay(&bank);
    getch();
    return 0;
}

```

```

Queue[0] 1 [TAIL]
Queue[1] [TAIL]
Queue[2] [TAIL]
====CLOCK[5]====
Queue[0] 1 [TAIL]
Queue[1] [TAIL]
Queue[2] [TAIL]
====CLOCK[6]====
Queue[0] 1 [TAIL]
Queue[1] [TAIL]
Queue[2] [TAIL]
====CLOCK[7]====
Queue[0] 1 [TAIL]
Queue[1] [TAIL]
Queue[2] [TAIL]
====CLOCK[8]====
Queue[0] 1 [TAIL]
Queue[1] 2 [TAIL]
Queue[2] [TAIL]
====CLOCK[9]====
Queue[0] 1 [TAIL]
Queue[1] 2 [TAIL]
Queue[2] 3 [TAIL]
====CLOCK[10]====
Queue[0] 1 [TAIL]
Queue[1] 2 [TAIL]
Queue[2] 3 [TAIL]
Average Staying Time per Customer: 17.3333

```

发现输出仍然有问题，但是修复了原本只有一个顾客的问题 但是当银行关门时，仍在办理业务的顾客被忽略了

```

=====CLOCK[75]=====
Queue[0] 14 [TAIL]
Queue[1] 15 [TAIL]
Queue[2] [TAIL]
=====CLOCK[76]=====
Queue[0] 14 [TAIL]
Queue[1] 15 [TAIL]
Queue[2] [TAIL]
=====CLOCK[77]=====
Queue[0] 14 [TAIL]
Queue[1] 15 [TAIL]
Queue[2] [TAIL]
=====CLOCK[78]=====
Queue[0] 14 [TAIL]
Queue[1] 15 [TAIL]
Queue[2] [TAIL]
=====CLOCK[79]=====
Queue[0] 16 [TAIL]
Queue[1] 15 [TAIL]
Queue[2] [TAIL]
=====CLOCK[80]=====
Queue[0] 16 [TAIL]
Queue[1] 15 [TAIL]
Queue[2] [TAIL]
=====CLOCK[81]=====
Queue[0] 16 [TAIL]
Queue[1] 15 [TAIL]
Queue[2] [TAIL]
=====CLOCK[82]=====
Queue[2] 19 [TAIL]
=====CLOCK[99]=====
Queue[0] 17 [TAIL]
Queue[1] 18 [TAIL]
Queue[2] 19 [TAIL]
=====CLOCK[100]=====
Queue[0] [TAIL]
Queue[1] 18 [TAIL]
Queue[2] 19 [TAIL]
Average Staying Time per Customer: 11.6842

```

基本是符合逻辑的，可能在处理最后的顾客是仍有一些问题

4.1.4 修改 clock 函数

为了在银行关门后仍然处理所有已经在排队的顾客，直到他们全部完成服务，并且避免在所有顾客都被服务完毕后输出多余的状态信息，我们可以在 Clock 函数中添加逻辑来判断是否所有服务窗口都空闲且没有顾客在排队，如果是，则结束循环。

```

void Clock(Bank* bank) {
    int now = 0; // 初始化当前时间
    // 继续运行，直到所有顾客都被服务完毕
    while (now < bank->close_time || !allWindowsIdle(bank) || bank->customer_queue.size > 0) {
        // 服务窗口更新，只在银行开放时间内接待新顾客
        for (int i = 0; i < NUM_WINDOWS; ++i) {
            if (bank->windows[i].remaining_service_time > 0) {
                bank->windows[i].remaining_service_time -= 1;
            }
            if (bank->windows[i].cur_remaining_service_time > 0) {
                bank->windows[i].cur_remaining_service_time -= 1;
                if (bank->windows[i].cur_remaining_service_time == 0 &&
                    bank->windows[i].queue.size > 0) {
                    Customer next_customer = getQueueHead(&bank->windows[i].queue);
                    bank->windows[i].cur_remaining_service_time = next_customer.service_time;
                    dequeue(&bank->windows[i].queue); // 从队列中移除已经开始服务的客户
                }
            }
        }
    }
}

```

```

    }

    // 在银行开放时间内接待新顾客
    if (now < bank->close_time) {
        while (bank->customer_queue.size > 0 &&
getQueueHead(&bank->customer_queue).arrival_time <= now) {
            Customer cur_customer = getQueueHead(&bank->customer_queue);
            dequeue(&bank->customer_queue); // 从顾客队列中移除该顾客
            EventDriven(bank, &cur_customer, now);
        }
    }

    // 检查是否所有窗口都空闲，且没有顾客在排队
    if (allWindowsIdle(bank) && bank->customer_queue.size == 0 && now >= bank->close_time) {
        break; // 如果是，跳出循环
    }

    // 打印当前状态
    printStatus(bank, now);

    now++; // 时间前进
}
}

bool allWindowsIdle(Bank* bank) {
    for (int i = 0; i < NUM_WINDOWS; ++i) {
        if (bank->windows[i].remaining_service_time > 0 || bank->windows[i].queue.size > 0) {
            return false; // 如果有窗口不空闲或有顾客在排队，返回 false
        }
    }
    return true; // 所有窗口空闲，没有顾客在排队
}

```

4.1.5 调试

```
=====CLOCK[106]=====
Queue[0] [TAIL]
Queue[1] 18 [TAIL]
Queue[2] 19 [TAIL]
=====CLOCK[107]=====
Queue[0] [TAIL]
Queue[1] 18 [TAIL]
Queue[2] [TAIL]
=====CLOCK[108]=====
Queue[0] [TAIL]
Queue[1] 18 [TAIL]
Queue[2] [TAIL]
=====CLOCK[109]=====
Queue[0] [TAIL]
Queue[1] 18 [TAIL]
Queue[2] [TAIL]
Average Staying Time per Customer: 11.3158
```

```
Customer 14 Arrived at 68 Estimated Service Time is 15
Customer 15 Arrived at 73 Estimated Service Time is 11
Customer 16 Arrived at 79 Estimated Service Time is 7
Customer 17 Arrived at 86 Estimated Service Time is 14
Customer 18 Arrived at 95 Estimated Service Time is 15
Customer 19 Arrived at 98 Estimated Service Time is 9
```

修改之后 clock 状态的输出是正确的


```

Customer 1 Arrived at 0 Estimated Service Time is 13
Customer 2 Arrived at 8 Estimated Service Time is 14
Customer 3 Arrived at 9 Estimated Service Time is 12
Customer 4 Arrived at 14 Estimated Service Time is 10
=====CLOCK[0]=====
Queue[0] 1 [TAIL]
=====CLOCK[1]=====
Queue[0] 1 [TAIL]
=====CLOCK[2]=====
Queue[0] 1 [TAIL]
=====CLOCK[3]=====
Queue[0] 1 [TAIL]
=====CLOCK[4]=====
Queue[0] 1 [TAIL]
=====CLOCK[5]=====
Queue[0] 1 [TAIL]
=====CLOCK[6]=====
Queue[0] 1 [TAIL]
=====CLOCK[7]=====
Queue[0] 1 [TAIL]
=====CLOCK[8]=====
Queue[0] 1 2 [TAIL]
=====CLOCK[9]=====
Queue[0] 1 2 3 [TAIL]
=====CLOCK[10]=====
Queue[0] 1 2 3 [TAIL]
=====CLOCK[11]=====
Queue[0] 1 2 3 [TAIL]
=====CLOCK[12]=====
Queue[0] 1 2 3 [TAIL]

```

```

=====CLOCK[10]=====
Queue[0] 1 2 3 [TAIL]
=====CLOCK[11]=====
Queue[0] 1 2 3 [TAIL]
=====CLOCK[12]=====
Queue[0] 1 2 3 [TAIL]
=====CLOCK[13]=====
Queue[0] 2 3 [TAIL]
=====CLOCK[14]=====
Queue[0] 2 3 4 [TAIL]
=====CLOCK[15]=====
Queue[0] 2 3 4 [TAIL]
=====CLOCK[16]=====
Queue[0] 2 3 4 [TAIL]
=====CLOCK[17]=====
Queue[0] 2 3 4 [TAIL]
=====CLOCK[18]=====
Queue[0] 2 3 4 [TAIL]
=====CLOCK[19]=====
Queue[0] 2 3 4 [TAIL]
=====CLOCK[20]=====
Queue[0] 2 3 4 [TAIL]
=====CLOCK[21]=====
Queue[0] 2 3 4 [TAIL]
=====CLOCK[22]=====
Queue[0] 2 3 4 [TAIL]
=====CLOCK[23]=====
Queue[0] 2 3 4 [TAIL]
=====CLOCK[24]=====
Queue[0] 2 3 4 [TAIL]
Queue[0] 4 [TAIL]
=====CLOCK[46]=====
Queue[0] 4 [TAIL]
=====CLOCK[47]=====
Queue[0] 4 [TAIL]
=====CLOCK[48]=====
Queue[0] 4 [TAIL]
=====CLOCK[49]=====
Queue[0] 4 [TAIL]
=====CLOCK[50]=====
Queue[0] 4 [TAIL]
=====CLOCK[51]=====
Queue[0] 4 [TAIL]
Average Staying Time per Customer: 24.25

```

减少窗口数量，发现计算的平均逗留时间也是正确的

4.2 绝对时间

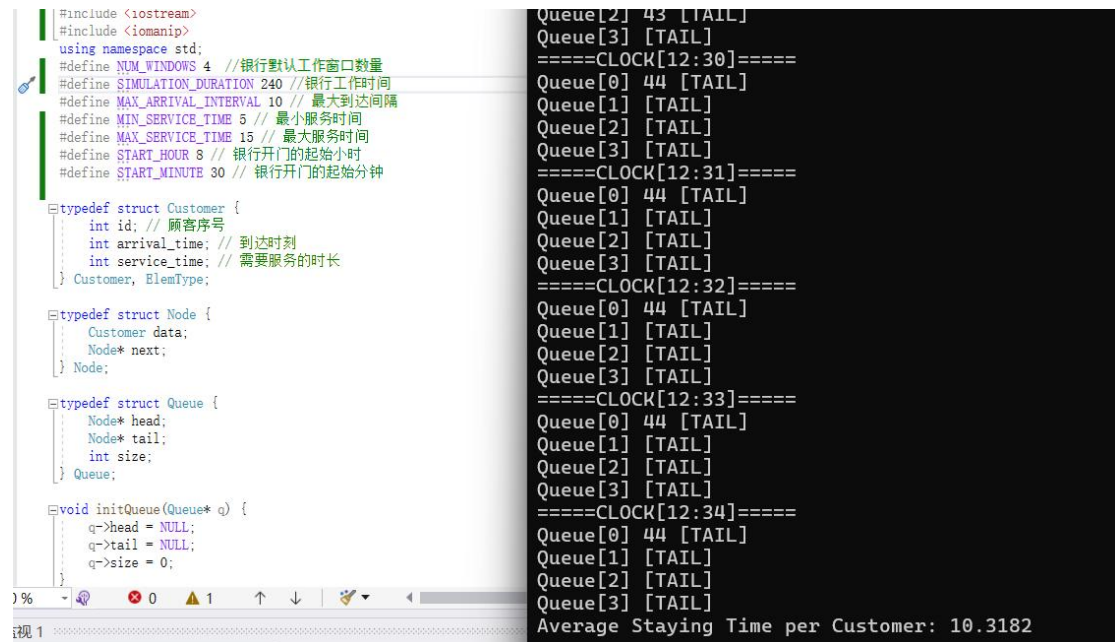
4.2.1 修改核心代码

```
#define START_HOUR 8 // 银行开门的起始小时
#define START_MINUTE 30 // 银行开门的起始分钟
void printStatus(Bank* bank, int now) {
    // 将模拟的分钟数转换为绝对时间
    int totalMinutes = now + START_HOUR * 60 + START_MINUTE; // 加上模拟开始的实际分钟数
    int hours = totalMinutes / 60; // 计算小时数
    int minutes = totalMinutes % 60; // 计算分钟数

    hours = hours % 24;

    // 输出当前时间，格式为 24 小时制
    cout << "====CLOCK[" << setw(2) << setfill('0') << hours << ":" << setw(2) << setfill('0') << minutes
    << "]" << endl;
    for (int i = 0; i < NUM_WINDOWS; ++i) {
        Node* p = bank->windows[i].queue.head;
        cout << "Queue[" << i << "]";
        while (p != NULL) {
            cout << p->data.id << " ";
            p = p->next;
        }
        cout << "[TAIL]" << endl;
    }
}
```

4.2.2 调试



```
#include <iostream>
#include <iomanip>
using namespace std;
#define NUM_WINDOWS 4 // 银行默认窗口数量
#define SIMULATION_DURATION 240 // 银行工作时间
#define MAX_ARRIVAL_INTERVAL 10 // 最大到达间隔
#define MIN_SERVICE_TIME 5 // 最小服务时间
#define MAX_SERVICE_TIME 15 // 最大服务时间
#define START_HOUR 8 // 银行开门的起始小时
#define START_MINUTE 30 // 银行开门的起始分钟

typedef struct Customer {
    int id; // 顾客序号
    int arrival_time; // 到达时刻
    int service_time; // 需要服务的时长
} Customer, ElemType;

typedef struct Node {
    Customer data;
    Node* next;
} Node;

typedef struct Queue {
    Node* head;
    Node* tail;
    int size;
} Queue;

void initQueue(Queue* q) {
    q->head = NULL;
    q->tail = NULL;
    q->size = 0;
}

Queue[2] 43 [TAIL]
Queue[3] [TAIL]
====CLOCK[12:30]====
Queue[0] 44 [TAIL]
Queue[1] [TAIL]
Queue[2] [TAIL]
Queue[3] [TAIL]
====CLOCK[12:31]====
Queue[0] 44 [TAIL]
Queue[1] [TAIL]
Queue[2] [TAIL]
Queue[3] [TAIL]
====CLOCK[12:32]====
Queue[0] 44 [TAIL]
Queue[1] [TAIL]
Queue[2] [TAIL]
Queue[3] [TAIL]
====CLOCK[12:33]====
Queue[0] 44 [TAIL]
Queue[1] [TAIL]
Queue[2] [TAIL]
Queue[3] [TAIL]
====CLOCK[12:34]====
Queue[0] 44 [TAIL]
Queue[1] [TAIL]
Queue[2] [TAIL]
Queue[3] [TAIL]
Average Staying Time per Customer: 10.3182
```

输出符合预期

4.3 模拟一天工作时间

4.3.1 代码逻辑

工作时间为：早上 8:30——12: 00 下午 14:00——17:00 增加一个常量代表中午休息时间（12:00——14:00）的工作窗口数量

在这段时间内 只有这些窗口工作

在午休时间只要打印工作的窗口的状态

午休时间的逻辑是

1. 如果在午休前在办理的 则等办理完之后关闭窗口
2. 午休期间工作窗口数量>0 则可以允许顾客进去银行并在这些窗口排队 =0 则拒绝这段时间的顾客进入 将这些顾客改为 14:00 进去
3. 午休结束后 所以窗口开放 允许所有在排队（不是正在办理的）重新排队

4.3.2 代码调试

```
Queue[0] [TAIL]
=====CLOCK[13:54]=====
Queue[0] [TAIL]
=====CLOCK[13:55]=====
Queue[0] [TAIL]
=====CLOCK[13:56]=====
Queue[0] [TAIL]
=====CLOCK[13:57]=====
Queue[0] [TAIL]
=====CLOCK[13:58]=====
Queue[0] [TAIL]
=====CLOCK[13:59]=====
Queue[0] [TAIL]
=====CLOCK[14:00]=====
Queue[0] 38 42 46 50 54 58 [TAIL]
Queue[1] 39 43 47 51 55 59 [TAIL]
Queue[2] 40 44 48 52 56 [TAIL]
Queue[3] 41 45 49 53 57 [TAIL]
=====CLOCK[14:01]=====
Queue[0] 38 42 46 50 54 58 [TAIL]
Queue[1] 39 43 47 51 55 59 [TAIL]
Queue[2] 40 44 48 52 56 [TAIL]
Queue[3] 41 45 49 53 57 [TAIL]
=====CLOCK[14:02]=====
```

代码调试发现 午休状态不能正常工作 还需要修改 clock 函数

```

=====CLOCK[13:53]=====
Queue[0] [TAIL]
=====CLOCK[13:54]=====
Queue[0] [TAIL]
=====CLOCK[13:55]=====
Queue[0] [TAIL]
=====CLOCK[13:56]=====
Queue[0] [TAIL]
=====CLOCK[13:57]=====
Queue[0] [TAIL]
=====CLOCK[13:58]=====
Queue[0] [TAIL]
=====CLOCK[13:59]=====
Queue[0] [TAIL]
=====CLOCK[14:00]=====
Queue[0] 38 42 46 50 54 58 [TAIL]
Queue[1] 39 43 47 51 55 59 [TAIL]
Queue[2] 40 44 48 52 56 [TAIL]
Queue[3] 41 45 49 53 57 [TAIL]
=====CLOCK[14:01]=====
Queue[0] 38 42 46 50 54 58 [TAIL]
Queue[1] 39 43 47 51 55 59 [TAIL]
Queue[2] 40 44 48 52 56 [TAIL]
Queue[3] 41 45 49 53 57 [TAIL]

```

仍然有问题 午休时间不能正常工作

4.3.3 修改代码逻辑

考虑到如果按照原本的方案，先让顾客排序，然后在该窗口等待之前的顾客办理完成之后再办理业务

那么我们在午休时间需要考虑的是：

1. 午休时间开始，如果有窗口仍然在办理业务，那么我们需要优先处理完该顾客的业务，然后关闭该窗口。
2. 需要将关闭窗口的所以等待顾客全部出队，并按照次序排到工作窗口处
3. 工作窗口还需要加入新来的顾客
4. 午休时间结束，工作窗口的排队顾客需要全部出队，按照次序排到新的窗口

该过程需要顾客多次重新排队，不仅代码难以实现，同时也不符合实际（不太可能有人傻傻的等一个中午吧）

所以，为了方便代码编写，同时更加符合实际情况，我对代码逻辑进行了修改

实际生活中，进入银行我们常常会被要求取号（顾客 ID），并按照号码顺序叫号，一旦有工作窗口空了，按照号码顺序就可以去办理业务，这样就能保证绝对的公平（时间顺序）同时顾客不可能傻傻等好久，所以顾客都有自己的容忍时间，一旦超过一定限度就会离开因此对应代码，我也做了对应修改

1. 对 Customer 结构体增加三项：容忍时间：代表着顾客最多可以忍受的排队等待时间，等待时间：代表着顾客实际的等待时间，是否被服务：代表着最终顾客有没有办理业务
2. 对 Bank 结构体增加三项：等待队列：需要等待的顾客排在等待队列中，对应现实中的大厅，顾客数组：用于记录今天所有来到银行的顾客，离开的顾客数组：用于记录今天到了银

行但是最后因为等待时间超过容忍时间而离开的顾客

3. 修改 Clock 函数：模拟过程中，如果有新的顾客到达且窗口已经满了，则让顾客进入等待队列，按照到达时间等待服务。同时，修改 Windows 类型，由于只要记录一个顾客，所以只要单一结构体就可以了，而不是队列。同时，为了记录每个顾客的等待时间，每次我们需要遍历一遍 WaitingQueue，更新每个顾客的等待时间，一旦等待时间超过该用户的容忍时间，把顾客是否被服务的标记变为 false，代表顾客离开。

4. 午休时间：午休时间时，仍在办理的窗口继续办理，一旦该顾客完成，则关闭该窗口，需要修改对应的正在服务的窗口数量。

5. 修改 PrintStatus 函数：该函数在要对应修改，如果等待队列有人，则需要打印等待队列，同时，为了区分午休时间和正常时间，我也在打印时间时做了对应处理。

6. 修改其他对应函数

4.3.4 调试过程

因为这个调试的次数比较多，这里就放了几次主要的错误

```
cout << "====CLOCK[17:06]====\n";
// 打印等待区
if (bank->waitQueue.empty() == false)
{
    cout << "Window[0] is serving Customer ID: 94\n";
    cout << "Window[1] is serving Customer ID: 93\n";
    cout << "====CLOCK[17:07]====\n";
    cout << "Window[0] is serving Customer ID: 94\n";
    cout << "Window[1] is empty\n";
    cout << "====CLOCK[17:08]====\n";
    cout << "Window[0] is serving Customer ID: 94\n";
    cout << "Window[1] is empty\n";
    cout << "====CLOCK[17:09]====\n";
    cout << "Window[0] is serving Customer ID: 94\n";
    cout << "Window[1] is empty\n";
    cout << "====CLOCK[17:10]====\n";
    cout << "Window[0] is serving Customer ID: 94\n";
    cout << "Window[1] is empty\n";
    cout << "====CLOCK[17:11]====\n";
    cout << "Window[0] is serving Customer ID: 94\n";
    cout << "Window[1] is empty\n";
    cout << "====CLOCK[17:12]====\n";
    cout << "Window[0] is serving Customer ID: 94\n";
    cout << "Window[1] is empty\n";
    cout << "====CLOCK[17:13]====\n";
    cout << "Window[0] is serving Customer ID: 94\n";
    cout << "Window[1] is empty\n";
    cout << "====CLOCK[17:14]====\n";
    cout << "Window[0] is empty\n";
    cout << "Window[1] is empty\n";
    cout << "Average Staying Time per Customer: 0\n";
}

int main()
{
    //setSeed(2023);
    Bank bank;
    OpenForDay(&bank);
    Clock(&bank);
    cout << "Ave\n";
    getchar();
    return 0;
}
```

一开始能正确处理服务，但是不能正确计算待在银行的时间


```

    cout << endl;

main()
//setSeed(2023);
Bank bank;
OpenForDay(&bank); //
Clock(&bank); // 时间
cout << "Average Stay
getchar();
return 0;

Waiting Queue: 4 7 11 15 27 42 46 59 74 82 83 94
=====CLOCK[109:21]=====
Window[0] is empty
Window[1] is empty
Window[2] is empty
Waiting Queue: 4 7 11 15 27 42 46 59 74 82 83 94
=====CLOCK[109:22]=====
Window[0] is empty
Window[1] is empty
Window[2] is empty
Waiting Queue: 4 7 11 15 27 42 46 59 74 82 83 94
=====CLOCK[109:23]=====
Window[0] is empty
Window[1] is empty
Window[2] is empty
Waiting Queue: 4 7 11 15 27 42 46 59 74 82 83 94
=====CLOCK[109:24]=====
Window[0] is empty
Window[1] is empty
Window[2] is empty
Waiting Queue: 4 7 11 15 27 42 46 59 74 82 83 94
=====CLOCK[109:25]=====
Window[0] is empty
Window[1] is empty
Window[2] is empty

```

修改逻辑之后发现陷入了死循环,发现是超过容忍时间的顾客一直在等待队列中没有正确出队

```

= p->next;

等待队列 (如果
(bank->waitingQ
queue(&bank->wa

stomers_served
turn total_time

turn 0.0; // 如

Status(Bank* ba
ars = now / 60;
utes = now % 6
< "=====CLOCK["
nt i = 0; i < N
nt << "Window["
(bank->windows
cout << "is s

se {
    cout << "is e
0 8

Window[1] is empty
=====CLOCK[15:24]=====
Window[0] is serving Customer ID: 76
Window[1] is serving Customer ID: 77
=====CLOCK[15:25]=====
Window[0] is serving Customer ID: 76
Window[1] is serving Customer ID: 77
=====CLOCK[15:26]=====
Window[0] is serving Customer ID: 76
Window[1] is serving Customer ID: 77
=====CLOCK[15:27]=====
Window[0] is serving Customer ID: 76
Window[1] is serving Customer ID: 77
=====CLOCK[15:28]=====
Window[0] is empty
Window[1] is serving Customer ID: 77
=====CLOCK[15:29]=====
Window[0] is empty
Window[1] is serving Customer ID: 77
=====CLOCK[15:30]=====
Window[0] is empty
Window[1] is serving Customer ID: 77
=====CLOCK[15:31]=====
Window[0] is empty
Window[1] is serving Customer ID: 77
=====CLOCK[15:32]=====
Window[0] is empty

```

修改好之后正确的输出服务过程

```

Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:43]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:44]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:45]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:46]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:47]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:48]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:49]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:50]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:51]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:52]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:53]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:54]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:55]=====
Window[0] is empty
Average Staying Time per Customer: 13
|

```

减少测试时间 发现最后算的时间没有计算等待时间 发现是Clock更新了等待时间之后该顾客又出队了 等待时间没有被正确计算 所以增加了一个今日顾客的数据（一开始没有）

```

Customer 1 Arrived at 08:30 Estimated Service Time is 14 Tolerance Time is 20
Customer 2 Arrived at 08:35 Estimated Service Time is 12 Tolerance Time is 11
=====CLOCK[08:30]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:31]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:32]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:33]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:34]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:35]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:36]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:37]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:38]=====
Window[0] is serving Customer ID: 1

```



```

Customer 1 Arrived at 08:30 Estimated Service Time is 14 Tolerance Time is 20
Customer 2 Arrived at 08:35 Estimated Service Time is 12 Tolerance Time is 11
=====CLOCK[08:30]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:31]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:32]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:33]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:34]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:35]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:36]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:37]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:38]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:39]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:40]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:40]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:41]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:42]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:43]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:44]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:45]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:46]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:47]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:48]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:49]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:50]=====

```

```
=====CLOCK[08:47]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:48]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:49]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:50]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:51]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:52]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:53]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:54]=====
Window[0] is serving Customer ID: 2
=====CLOCK[08:55]=====
Window[0] is empty
Average Staying Time per Customer: 17
```

修改之后可以正确计算等待时间了 为了测试之前遇到了问题,能不能让顾客有效了离开(超过容忍时间), 我把容忍时间改成 1 测试

```

Customer customer = {
    customer_id++,
    absolute_arrival_time,
    genRand(MIN_SERVICE_TIME, MAX_SERVICE_TIME),
    1, //tolerance_time,
    0, // 初始化等待时间为0
    true // 初始化被服务
};

```

```

Customer 1 Arrived at 08:30 Estimated Service Time is 14 Tolerance Time is 1
Customer 2 Arrived at 08:35 Estimated Service Time is 12 Tolerance Time is 1
=====CLOCK[08:30]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:31]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:32]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:33]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:34]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:35]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:36]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:37]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:38]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:39]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:40]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:41]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:42]=====
Window[0] is serving Customer ID: 1

```

```

Waiting Queue: 2
=====CLOCK[08:36]=====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
=====CLOCK[08:37]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:38]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:39]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:40]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:41]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:42]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:43]=====
Window[0] is empty
Average Staying Time per Customer: 14

```

发现好像还是计算了 2 号顾客的服务时间

[0]	{id=1 arrival_time=510 service_time=14 ...}
[1]	{id=2 arrival_time=515 service_time=12 ...}
id	2
arrival_time	515
service_time	12
tolerance_time	1
waiting_time	2
is_served	true
[原始视图]	{_Mypair=allocator }

调试发现原来是忘记更新了今日顾客数据中的是否被服务的属性（因为一开始没了，后来加了忘了改这个了）

[1]	{id=2 arrival_time=515 service_time=12 ...}
id	2
arrival_time	515
service_time	12
tolerance_time	1
waiting_time	2
is_served	false

修改之后调试是正确的

```
-----CLOCK[08:39]-----
Window[0] is serving Customer ID: 1
=====CLOCK[08:40]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:41]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:42]=====
Window[0] is serving Customer ID: 1
=====CLOCK[08:43]=====
Window[0] is empty
Average Staying Time per Customer: 8
```

发现计算的平均时间是正确的 但是不能看到有哪些顾客没有被服务，这显然是不合理的，为了记录离开的顾客，我又增加了一个离开的顾客数组（一开始也没有）

```
Window[0] is serving Customer ID: 1
====CLOCK[08:33]====
Window[0] is serving Customer ID: 1
====CLOCK[08:34]====
Window[0] is serving Customer ID: 1
====CLOCK[08:35]====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
====CLOCK[08:36]====
Window[0] is serving Customer ID: 1
Waiting Queue: 2
====CLOCK[08:37]====
Window[0] is serving Customer ID: 1
====CLOCK[08:38]====
Window[0] is serving Customer ID: 1
====CLOCK[08:39]====
Window[0] is serving Customer ID: 1
====CLOCK[08:40]====
Window[0] is serving Customer ID: 1
====CLOCK[08:41]====
Window[0] is serving Customer ID: 1
====CLOCK[08:42]====
Window[0] is serving Customer ID: 1
====CLOCK[08:43]====
Window[0] is empty
Average Staying Time per Customer: 8 minutes
Number of Leaving Customers: 1
ID of Leaving Customers: 2
|
```

是正确的

接下来就是修改午休时间

```

vit Window[0] is serving Customer ID: 47
oi =====CLOCK[12:34] Lunch Time =====
Window[0] is serving Customer ID: 47
=====CLOCK[12:35] Lunch Time =====
Window[0] is serving Customer ID: 47
=====CLOCK[12:36] Lunch Time =====
Window[0] is empty
=====CLOCK[12:37] Lunch Time =====
Window[0] is empty
=====CLOCK[12:38] Lunch Time =====
oi Window[0] is empty
=====CLOCK[12:39] Lunch Time =====
Window[0] is empty
=====CLOCK[12:40] Lunch Time =====
Window[0] is empty
=====CLOCK[12:41] Lunch Time =====
Window[0] is empty
=====CLOCK[12:42] Lunch Time =====
Window[0] is serving Customer ID: 51
=====CLOCK[12:43] Lunch Time =====
Window[0] is serving Customer ID: 51
=====CLOCK[12:44] Lunch Time =====
Window[0] is serving Customer ID: 51
=====CLOCK[12:45] Lunch Time =====
+ E Window[0] is serving Customer ID: 51
=====CLOCK[12:46] Lunch Time =====
Window[0] is serving Customer ID: 51
=====CLOCK[12:47] Lunch Time =====
Window[0] is serving Customer ID: 51
=====CLOCK[12:48] Lunch Time =====

```

这明显错了，午休时间莫名奇妙少了好多顾客 后来发现这其实就是我统计窗口的逻辑错了，为了保证在服务的窗口继续服务完，我需要打印的窗口数量明显不是简单的开着的窗口（但是我当时没意识到，所以之后的调试走了些弯路）


```

Window[0] is serving Customer ID: 42
Window[1] is empty
Window[2] is empty
=====CLOCK[11:59] Normal Hours =====
Window[0] is serving Customer ID: 42
Window[1] is serving Customer ID: 43
Window[2] is empty
=====CLOCK[12:00] Lunch Time =====
Window[0] is empty
Window[1] is serving Customer ID: 43
=====CLOCK[12:01] Lunch Time =====
Window[0] is serving Customer ID: 44
Window[1] is serving Customer ID: 43
=====CLOCK[12:02] Lunch Time =====
Window[0] is serving Customer ID: 44
Window[1] is serving Customer ID: 43
Waiting Queue: 45
=====CLOCK[12:03] Lunch Time =====
Window[0] is serving Customer ID: 44
Window[1] is serving Customer ID: 43
Waiting Queue: 45
=====CLOCK[12:04] Lunch Time =====
Window[0] is serving Customer ID: 44
Window[1] is serving Customer ID: 43
Waiting Queue: 45

```

为了避免太多无用了内容，我省略了中间的调试过程，最后的结果是这样的
 需要注意一点，像图中这种情况如果 1 号窗口一直在工作，我们 0 号窗口就算空闲了仍然需要打印，因为 0 号窗口才是正常上班的窗口，也就是说，如果 2 号窗口仍在服务，我们需要打印 0 号和 2 号。但这就意味了我们又要新开一个数组来记录那些仍在工作的窗口（银行到下班时间是类似），实在太麻烦了。。。于是我用了一个简单的方法，记录一下最大的窗口代码，把第一个到最大的那个都打印出来。也就是说如果 2 号窗口仍在服务，我们打印的是 0,1,2 号窗口。其中 1 号窗口是空的

```

Window[1] is empty
Window[2] is empty
=====CLOCK[17:13] Normal Hours =====
Window[0] is empty
Window[1] is empty
Window[2] is empty
Average Staying Time per Customer: 13.1277 minutes
Number of Leaving Customers: 8
ID of Leaving Customers: 47 50 51 52 58 57 60 65

```

最终的计算也是没有问题的，大量午休时间到达的顾客最终离开了，这很符合现实情况

4.3.3 核心代码修改

这其实需要修改好多地方，所以就放一些重要的地方

```
typedef struct Customer {
    int id;
    int arrival_time;
    int service_time;
    int tolerance_time;
    int waiting_time = 0; // 新增等待时间
    bool is_served = true; // 新增是否被服务标记
} Customer;

typedef struct Bank {
    int open_time, close_time;
    Window windows[NUM_WINDOWS];
    Queue waitingQueue;
    Queue customer_queue;
    int num_to_service;
    vector<Customer> daily_customers; // 新增顾客数组
    vector<int> leave_customers;
} Bank;

void Clock(Bank* bank) {
    int now = START_HOUR * 60 + START_MINUTE;
    int lunch_start_time = LUNCH_START_HOUR * 60;
    int lunch_end_time = LUNCH_END_HOUR * 60;
    bool is_lunch_time = false;

    while (now < bank->close_time || !allWindowsIdle(bank) || bank->waitingQueue.size > 0) {
        is_lunch_time = now >= lunch_start_time && now < lunch_end_time;
        // 处理等待队列中顾客的等待时间和容忍时间
        Node* current = bank->waitingQueue.head;
        Node* prev = nullptr;
        while (current != nullptr) {
            if (current->data.is_served) {
                current->data.waiting_time++;
                // 更新 daily_customers 中对应顾客的等待时间
                for (auto& customer : bank->daily_customers) {
                    if (customer.id == current->data.id) {
                        customer.waiting_time = current->data.waiting_time;
                        break;
                    }
                }
            }
            if (current->data.waiting_time > current->data.tolerance_time) {
                // 如果等待时间超过容忍时间，顾客离开
            }
        }
    }
}
```



```

        bank->leave_customers.push_back(current->data.id);
        current->data.is_served = false;
        for (auto& customer : bank->daily_customers) {
            if (customer.id == current->data.id) {
                customer.is_served = current->data.is_served;
                break;
            }
        }
        Node* toDelete = current;
        if (prev != nullptr) {
            prev->next = current->next;
        }
        else {
            bank->waitingQueue.head = current->next;
        }
        if (current == bank->waitingQueue.tail) {
            bank->waitingQueue.tail = prev;
        }
        current = current->next;
        delete toDelete;
        bank->waitingQueue.size--;
        continue;
    }
}
prev = current;
current = current->next;
}
int active_windows = is_lunch_time ? NUM_WINDOWS_LUNCH : NUM_WINDOWS;
int working_during_lunch = 0;
if (is_lunch_time) {
    // 如果是午休时间，需要检查哪些窗口可以继续工作
    for (int i = active_windows; i < NUM_WINDOWS; i++) {
        if (bank->windows[i].cur_remaining_service_time > 0) {
            working_during_lunch=i;
        }
    }
}
// 处理新到达的顾客
while (bank->customer_queue.size > 0 && getQueueHead(&bank->customer_queue).arrival_time
<= now) {
    Customer customer = getQueueHead(&bank->customer_queue);
    bool assigned = false;
    for (int i = 0; i < active_windows; ++i) {
        if (bank->windows[i].cur_customer_id == 0) { // 检查是否有空窗口

```

```

        bank->windows[i].cur_customer_id = customer.id;
        bank->windows[i].cur_remaining_service_time = customer.service_time;
        dequeue(&bank->customer_queue); // 从顾客队列中移除该顾客
        assigned = true;
        break;
    }
}
if (!assigned) {
    enqueue(&bank->waitingQueue, customer); // 如果所有窗口都忙，将顾客加入等待队
列

    dequeue(&bank->customer_queue); // 从顾客队列中移除
}
}

// 更新所有窗口中顾客的服务时间
for (int i = 0; i < NUM_WINDOWS; ++i) {
    if (bank->windows[i].cur_remaining_service_time > 0) {
        bank->windows[i].cur_remaining_service_time--;
        if (bank->windows[i].cur_remaining_service_time == 0) {
            bank->windows[i].cur_customer_id = 0; // 完成服务
        }
    }
}

// 检查是否有顾客可以从等待队列移动到服务窗口
transferFromWaitingQueueToWindow(bank, active_windows);
printStats(bank, now, active_windows+working_during_lunch);
now++; // 时间前进一分钟
}
}

```

代码里面已经加了一些注释了，便于理解。因为代码确实比较长，有点难理解，有时候我自己也容易看迷糊

哦对 附上常量

```

#define NUM_WINDOWS 3 //银行默认工作窗口数量
#define SIMULATION_DURATION 510 //银行工作时间
#define MAX_ARRIVAL_INTERVAL 10 // 最大到达间隔
#define MIN_SERVICE_TIME 5 // 最小服务时间
#define MAX_SERVICE_TIME 15 // 最大服务时间
#define START_HOUR 8 // 开门的起始小时
#define START_MINUTE 30 // 开门的起始分钟
#define LUNCH_START_HOUR 12 // 午休开始的小时
#define LUNCH_END_HOUR 14 // 午休结束的小时
#define END_HOUR 17 // 关门的小时
#define NUM_WINDOWS_LUNCH 1 // 午休时间工作的窗口数量

```

4.4 模拟一周的工作

4.4.1 修改代码

这个其实就只要改一下主函数就行了，加个循环，然后打印一下时间，PPT 里面说的是周日休息，那就算六天的就好了

其他还需要修改的是增加星期常量和全局变量计算这周的平均时间和总共离开的顾客数量，修改后的主函数如下：

```
int main()
{
    Bank bank;
    for (int i = 1; i < 7; i++)
    {
        cout << "-----Today is " << daysOfWeek[i - 1]
        << "-----" << endl;

        setSeed(2023+i);
        OpenForDay(&bank); // 开门
        Clock(&bank); // 时间流逝 -> 事件驱动
        CloseForDay(&bank); // 关门
    }
    cout << "-----Today is " << daysOfWeek[6] <<
    "-----" << endl;

    cout << "Average Staying Time per Customer(This Week): " << staytime/6 << " minutes" << endl;
    cout << "Number of Leaving Customers(This Week): " << leave << endl;
    getchar();
    return 0;
}
```

4.4.2 代码效果

```
Window[2] is serving Customer ID: 94
====CLOCK[17:09] Normal Hours ====
Window[0] is empty
Window[1] is empty
Window[2] is serving Customer ID: 94
====CLOCK[17:10] Normal Hours ====
Window[0] is empty
Window[1] is empty
Window[2] is empty
Average Staying Time per Customer: 11.3404 minutes
Number of Leaving Customers: 8
ID of Leaving Customers: 43 46 49 47 51 53 55 60
-----Today is Sunday-----
Average Staying Time per Customer(This Week): 11.6275 minutes
Number of Leaving Customers(This Week): 38
```

是正确的 这个结果比较长，以附件的形式放在这了（如果找不到，在代码文件夹里面找，



超链接换了位置就不生效了)

五、实验总结

这次实验相对来讲难度并不大，但是因为要考虑到午休时间的问题，导致代码编写及其繁琐，要用户不断的出队入队，显然也是不合理的，于是我修改了代码的逻辑，增加了等待队列，来模拟现实生活中的银行大厅，同时为顾客增加了容忍时间，来模拟现实中的顾客。模拟银行业务系统，虽然编写上没有很多的技术难题，但是对于一个工程来说，任何一点一点 bug 都是致命的。这要求代码的编写需要遵循现实情况，使代码有很强的健壮性，如午休是窗口的处理，不能让办一半的顾客直接离开，也不能让顾客重复的排队，在下班时间也不能让办一半的顾客离开，这些显然是不合理的。所以模拟现实中的银行业务系统，要求我们注重代码细节，他不一定需要很新的方法，但是确需要我们考虑的很完善，要考虑到每种特殊的情况，在之后的专业学习中，会带着这次实验学到的继续努力。