

# 离散时间模拟——银行业务模拟

## 一、实验要求

### 1. 实验题目

模拟银行业务，处理客户存钱和取钱请求，银行共有两个窗口，客户先在第一个窗口排队办理业务，当且仅当客户为取钱业务且申请额超出银行现存资金总额时，进入第二个窗口排队等待。当第一窗口处理完一单客户存钱请求时，则顺序判断第二窗口客户的取钱请求能否满足。

所有客户信息均需由程序随机产生，这些信息包括：到达时间、办理业务的类型、数额、办理时长。

输入：银行初始资金，银行营业时间

输出：客户在银行内逗留的平均时间

### 2. 拓展部分

- 设置多个第一种窗口，具体数目由输入决定，客户达到银行时，选择队伍最短的第一类窗口排队等待。
- 原问题中，当第二类窗口办理业务时，第一类窗口停止办理业务，直到第二类窗口业务办理完成后，第一类窗口才可继续办理。这不符合生活常识，所以更改为：所有窗口均可同时办理业务。
- 在处理客户到达时间时，考虑到实际情况中，银行营业存在忙碌、普通、清闲三种状态，对应的客户数目会有所不同，若当天为“忙碌”，则这一天到达的客户比较密集，客户数量多。
- 对于客户的业务金额、办理时长，在产生随机数时，使之粗略满足正态分布。
- 设置了可以输出事件表，显示每一条业务办理信息。并且时间输出表可以延迟输出，每 800 ms 输出一条，且是否需要延迟输出，延迟时长均可自行设置。
- 设置了 Debug 版本，所有客户的信息均由键盘输入。因为当客户的信息随机生成时，难以复现出 BUG 的客户信息序列。

## 二、设计思路

### 1. 核心思路

采用**事件表驱动**的思想，采用**链队列**存储事件表，各事件按照时间顺序排列。

事件分为五种类型：

- 类型 0：新客户到达，随机生成他的业务信息，选择最短窗口排队，生成下一客户到达事件
- 类型 1：某个一号窗口开始办理业务
  - 若为取钱，且银行资金总额足够，那么他开始办理业务，将他的离开事件加入事件表、
  - 若为取钱，但银行资金总额不足，那么该客户进入第二类窗口排队等待，将原本排在他后面的客户的“开始办理业务”事件加入事件表
  - 若为存钱，则开始办理业务，将他的离开事件加入事件表
- 类型 2：二号窗口开始办理业务，将该客户的离开事件加入事件表
- 类型 3：某个一号窗口的客户办理完业务，离开
  - 若离开事件晚于银行关门时间，则视为业务未办理完成
  - 若为存钱业务办理完成，则顺序检查第二类窗口客户的取钱请求能否满足，若可以，则将他们的“开始办理业务”事件加入事件表
  - 该客户离开后，将排在他后面的客户的“开始办理业务”事件加入事件表
- 类型 4：二号窗口客户办理完业务，离开

- 若晚于银行关门时间，则视为业务未办理完成

最后银行关门时，检查各窗口是否仍有人排队等待，若有，则视为业务未办理，客户离开。

## 2. 使用到的数据结构及其 ADT

本程序中使用了队列，存储事件，存储各窗口的客户，其 ADT 为：

---

ADT Queue{

    数据对象：  $D = \{ a_i \mid a_i \in \text{ElemSet}, i = 1, 2, \dots, n, n \geq 0 \}$

    数据关系：  $R = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, a_{i-1} \leq a_i, i = 2, \dots, n \}$

    基本操作：(本次实验使用到的)

        InitQueue(&Q); // 创造空队列

        QueueEmpty(Q); // 判断链表是否为空

        GetHead(Q, &e); // 返回队头元素

        EnQueue(&Q, e); // 插入元素 e 为 Q 的新队尾元素

        DeQueue(&Q, &e); // 删除 Q 的队头元素，并用 e 返回其值

        QueueTraverse(Q, visit()); // 遍历队列

}ADT Queue

---

## 3. 核心模块

OpenForDay(); // 初始化各窗口，初始化事件表，产生第一位客户到达事件

EventInsert(LNodeE\* ev); // 将事件按照时间顺序加入事件表

CustomerArrive(LNodeE\* q); // 客户达到，类型 0 事件

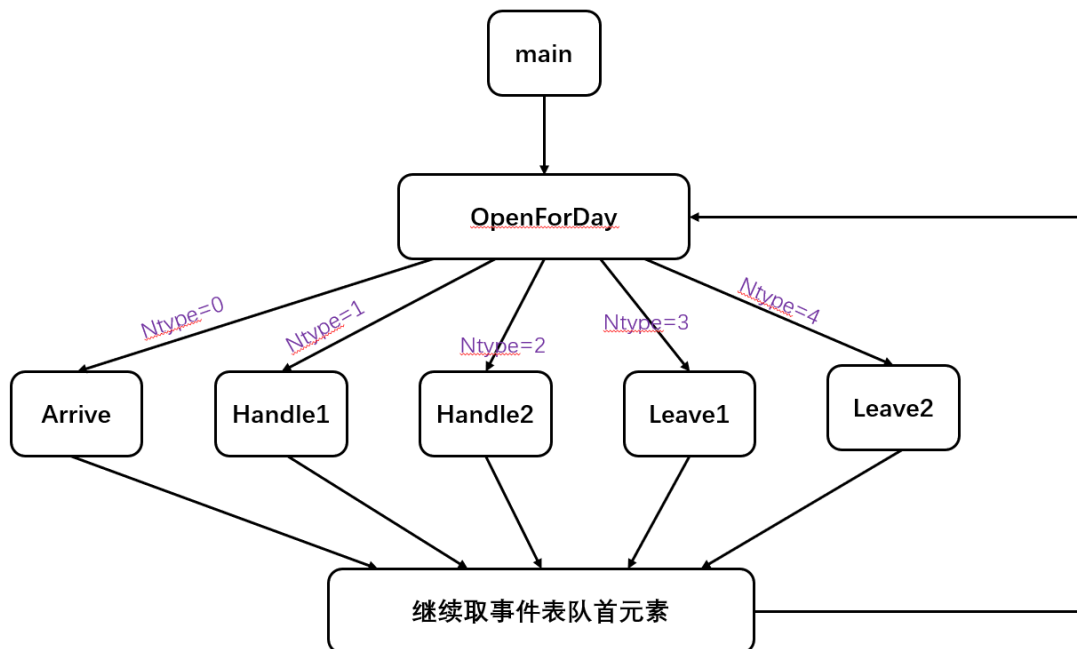
CustomerHandle1(LNodeE\* q); // 第一类窗口客户开始办理业务，类型 1 事件

CustomerHandle2(LNodeE\* q); // 第二类窗口客户开始办理业务，类型 2 事件

CustomerHandle1(LNodeE\* q); // 第一类窗口客户办理完业务，类型 3 事件

CustomerLeave2(LNodeE\* q); // 第二类窗口客户办理完业务，类型 4 事件

## 4. 模块流程关系



### 三、关键代码讲解

#### 1. EventInsert(LNodeE\* ev)

```

void EventInsert(LNodeE* ev)
{
    //将事件按时间顺序加入到事件表中，且时间相同时，离开事件先于到达事件
    LNodeE* q = EventList->next, * p = EventList;
    int start = time(0);
    while (q) {
        if (q->OccurTime > ev->OccurTime) {
            p->next = ev; ev->next = q;
            return;
        }
        if (q->OccurTime == ev->OccurTime && (ev->NType == 4 || ev->NType == 3))
        {
            p->next = ev; ev->next = q;
            return;
        }
        q = q->next;
        p = p->next;
    }
    p->next = ev;
    ev->next = NULL;
}
  
```

#### 2. CustomerArrive(LNodeE\* q)

```

void CustomerArrive(LNodeE* q)
{
    if (q->OccurTime >= closeTime) return;
    ++CustomerNum;
    LNodeQ* nexcus = new LNodeQ;
    nexcus->ArriveTime = q->OccurTime;
    nexcus->DurTime = GetRandomT();
    nexcus->mount = GetRandomM(); //生成当前客户的业务信息
    int now_window = GetShortestLine(); //选择排队人数最少的窗口
  
```

```

window1[now_window].tail->next = nexcus;
window1[now_window].tail = nexcus;
window1[now_window].tail->next = NULL;
if (window1[now_window].head->next == window1[now_window].tail) {
//如果当前窗口没有人排队，则该客户无需排队，直接开始办理业务
    LNodeE* new_ev = new LNodeE;
    new_ev->NType = 1;
    new_ev->next = NULL;    new_ev->now_customer = nexcus;
    new_ev->OccurTime = q->OccurTime;    new_ev->WinLocate = now_window;
    EventInsert(new_ev);
}
LNodeE* new_ev = new LNodeE; //产生下一客户到达事件
new_ev->next = NULL;    new_ev->now_customer = NULL;
new_ev->NType = 0;
new_ev->OccurTime = q->OccurTime + GetRandomT_arrive();
if (new_ev->OccurTime < CloseTime) EventInsert(new_ev);
}

```

### 3. CustomerHandle1(LNodeE\* q)

```

void CustomerHandle1(LNodeE* q)
//类型=1，某个一号窗口开始办理业务；
//若取钱，且足够，那么直接取，将其离开事件加入事件表。离开时再将下一人开始办理事件加入事件表；
//若取钱，但不够，将其加入2号窗口，从当前的窗口删除，并将他的下一个人办理业务加入事件表；
//若存钱，将其离开事件加入事件表
{
    if (q->now_customer->mount < 0) {
        if (q->now_customer->mount + ReMoney >= 0) { //取钱，够取
            ReMoney += q->now_customer->mount;
            LNodeE* new_ev = new LNodeE;
            new_ev->next = NULL;    new_ev->now_customer = q->now_customer;
            new_ev->NType = 3;
            new_ev->OccurTime = q->OccurTime + q->now_customer->DurTime;
            new_ev->WinLocate = q->WinLocate;
            EventInsert(new_ev);
        }
        else { //取钱不够取
            if (window1[q->winLocate].head->next->next) {
                LNodeE* new_ev = new LNodeE;
                new_ev->next = NULL;
                new_ev->now_customer = window1[q->winLocate].head->next;
                new_ev->NType = 1;    new_ev->OccurTime = q->OccurTime;
                new_ev->WinLocate = q->WinLocate;
                EventInsert(new_ev);
            }
            Deletewindow(1, q->winLocate); //在当前的一号窗口将其删除
            window2.tail->next = q->now_customer;
            window2.tail = q->now_customer;
            window2.tail->next = NULL;
        }
    }
    else { //存钱
        LNodeE* new_ev = new LNodeE;
        new_ev->next = NULL;    new_ev->now_customer = q->now_customer;
        new_ev->NType = 3;    new_ev->OccurTime = q->OccurTime + q->
now_customer->DurTime;
        new_ev->WinLocate = q->WinLocate;
    }
}

```

```

        EventInsert(new_ev);
    }
}

```

## 4. CustomerLeave1(LNodeE\* q)

```

void CustomerLeave1(LNodeE* q)
{
    //若离开事件晚于银行关门时间，则视为业务未办理完成
    //若为存钱业务办理完成，则顺序检查第二类窗口客户的取钱请求能否满足，若可以，则将他们的“开始办理
    业务”事件加入事件表
    //该客户离开后，将排在他后面的客户的“开始办理业务”事件加入事件表
    LNodeQ* tmp = window1[q->winLocate].head->next;
    if (q->OccurTime > CloseTime) {
        TotalTime += CloseTime - tmp->ArriveTime;
        if (q->now_customer->mount < 0) {
            ReMoney = ReMoney - q->now_customer->mount;
        }
        if (window1[q->winLocate].head->next->next) {
            //之后排在这个窗口的，直接加离开事件
            LNodeE* new_ev = new LNodeE;
            new_ev->next = NULL;    new_ev->now_customer = window1[q-
>winLocate].head->next;
            new_ev->NType = 3;    new_ev->OccurTime = q->OccurTime;
            new_ev->winLocate = q->winLocate;
            EventInsert(new_ev);
        }
        Deletewindow(1, q->winLocate);
    }
    else {
        TotalTime += q->OccurTime - tmp->ArriveTime;
        if (tmp->mount > 0) {
            ReMoney += tmp->mount;
            LastTime = LastTime > q->OccurTime ? LastTime : q->OccurTime;
            LNodeQ* tmp_win2 = window2.head->next; //遍历第二个窗口的客户
            LNodeQ* pre = window2.head; //跟随指针
            while (tmp_win2 && LastTime < CloseTime) {
                if (tmp_win2->mount + ReMoney >= 0) {
                    //假如当前客户的取钱请求可以满足，则将第二个窗口的办理时间加入事件表
                    LNodeE* en = new LNodeE;
                    en->NType = 2;    en->OccurTime = LastTime;
                    en->next = NULL;    en->now_customer = tmp_win2;
                    en->winLocate = 1;
                    if (en->OccurTime < CloseTime) {
                        EventInsert(en);
                        LastTime += tmp_win2->DurTime;
                        ReMoney += tmp_win2->mount;
                    }
                    pre->next = tmp_win2->next;
                    tmp_win2 = pre->next;
                    if (!tmp_win2) window2.tail = pre; //这里很重要
                }
                else {
                    pre = pre->next;
                    tmp_win2 = tmp_win2->next;
                }
            }
        }
    }
}

```

```

    }
    Deletewindow(1, q->winLocate);
    if (window1[q->winLocate].head->next) {
        //该客户离开后，将排在他后面的客户的“开始办理业务”事件加入事件表
        LNodeE* new_ev = new LNodeE;
        new_ev->next = NULL;    new_ev->now_customer = window1[q->winLocate].head->next;
        new_ev->NType = 1;    new_ev->OccurTime = q->OccurTime;
        new_ev->winLocate = q->winLocate;
        EventInsert(new_ev);
    }
}
free(q->now_customer);
}

```

## 5. CustomerHandle2(LNodeE\* q)

```

void CustomerHandle2(LNodeE* q)
//类型=2,二号窗口办理业务，取钱，将其离开事件加入事件表
{
    LNodeE* new_ev = new LNodeE;
    new_ev->next = NULL;    new_ev->now_customer = q->now_customer;
    new_ev->NType = 4;    new_ev->OccurTime = q->OccurTime + q->now_customer->DurTime;
    new_ev->winLocate = q->winLocate;
    EventInsert(new_ev);
}

```

## 6. CustomerLeave2(LNodeE\* q)

```

void CustomerLeave2(LNodeE* q)
{//二号窗口的客户办理完业务，离开
    LNodeQ* tmp = q->now_customer;
    if (q->OccurTime > CloseTime) {
        TotalTime += CloseTime - tmp->ArriveTime;
        ReMoney = ReMoney - q->now_customer->mount;
    }
    else {
        TotalTime += q->OccurTime - tmp->ArriveTime;
    }
    free(q->now_customer);
}

```

注：为方便阅读代码，以上代码在引用时均删去了输出部分，输出为打印事件表

## 7. GetRandomM()

```

int GetRandomM()
{//生成客户办理的金额，正数为存钱，负数为取钱，并使之粗略满足正态分布
    int a = 0;
    int b = rand() % 2;
    int N = (rand() % 111 + rand() % 71) % 10;
    while (a == 0) {
        if (N <= 1) a = rand() % 200 + 101;
        else if (N <= 5) a = rand() % 1800 + 200;
    }
}

```

```

        else if (N <= 7)    a = rand() % 3000 + 2000;
        else if (N <= 8)    a = rand() % 3000 + 5000;
        else a = rand() % 20001;
    }
    if (b == 1) return a;
    else return -a;
}

```

## 四、调试分析

### 1. 时间复杂度

- 设总客户数为  $n$ ，每个客户所能产生的事件最多为 4 件（到达，在一号窗口办理业务，在二号窗口办理业务，离开），最少为 2 件（到达，没有开始办理业务而离开），故事件总数  $\leq 4*n$ 。
- 事件表为链队列，操作只包括取队首元素，在队尾增加新元素，存取全部事件的过程中，这一模块的时间复杂度为  $O(n)$ 。
- 在 CustomerArrive 模块中，需要将事件的按时间顺序插入事件表，除此之外的操作都是常数时间的，所以时间复杂度为  $O(n)$ ；
- 在 CustomerHandle1 模块中，需要将事件的按时间顺序插入事件表，除此之外的操作都是常数时间的，所以时间复杂度为  $O(n)$ ；
- 在 CustomerLeave1 模块中，需要判断二号窗口的客户的取钱请求是否满足，若满足，则将其业务开始办理的事件的按时间顺序插入事件表，最坏情况下，需要将所有在第二窗口等待的客户的办理时间加入事件表，所以最坏情况下时间复杂度为  $O(n^2)$ ；
- 在 CustomerHandle2 模块中，需要将事件的按时间顺序插入事件表，除此之外的操作都是常数时间的，所以时间复杂度为  $O(n)$ ；
- 在 CustomerLeave1 模块中，只涉及常数时间的操作，所以时间复杂度为  $O(1)$ ；
- 综上，最坏情况下时间复杂度为  $O(n^3)$ ，但在实际过程中，时间复杂度达不到这个  $n^3$  量级，因为一方面，各类事件中，只有第一窗口客户离开事件才有可能达到  $O(n^2)$  的时间，但这需要两个条件，一是当前客户办理的是存钱业务，且存的钱足够多，能保证将所有在第二窗口排队的客户的取钱请求都满足，二是第二窗口排队的客户数量足够多，与总客户数量在同一量级，否则在遍历第二窗口队伍时，时间复杂度达不到  $O(n)$ 。但这两个条件同时满足属于较为极端的情况，在随机模拟的过程中并不会大量出现。考虑到在多数情况下，排到第二窗口的客户数量并不多，所以平均意义上，整个程序的时间复杂度接近于  $O(n^2)$ 。

### 2. 空间复杂度

设客户总数为  $n$ ，那么存储每一个客户的信息，需要的空间为  $O(n)$ ；存储每个客户产生的事件，需要的空间为  $O(n)$ 。

为节省空间，在每个事件中，并不会重复存储当前客户的业务信息，而是采用指针指向客户指针。此外，当一个客户办理完业务离开银行时，会立即释放掉他所占用的客户节点。

综上，本程序的空间复杂度为  $O(n)$ 。

### 3. 遇到的BUG以及处理方法

- 出现了所有客户均在排队，没有人正常办理业务的 BUG。问题在于当某个客户在第一类窗口开始办理业务时，如果他为取钱业务但不够取，该客户会排到第二类窗口，在处理这个地方时，需要两步操作，一是将该客户结点从当前窗口的队列删除，加入第二类窗口；二是如果该客户身后有人排队，那么需要将后面这个人的开始办理业务事件加入事件表。最开始处理时，忽略了考虑在他身后排队的客户，最终导致这个窗口只能排队而无人办理业务。

- 当一个存钱业务的客户办理完毕离开时，顺序检查第二类窗口的队伍时，访问到了队列之外的空间。最初以为是问题出在第二类窗口加入新结点时，需要置最后一个结点的 `next = NULL`，但问题并非在此。当顺序检查第二类窗口的客户时，使用了两个指针，一个指向当前检查的客户，一个作为跟随指针，当当前客户的取钱请求可以满足时，就把他从这个队列删除，防止重复检查。删除操作的代码为：

```
pre->next = tmp_win2->next;
tmp_win2 = pre->next;
```

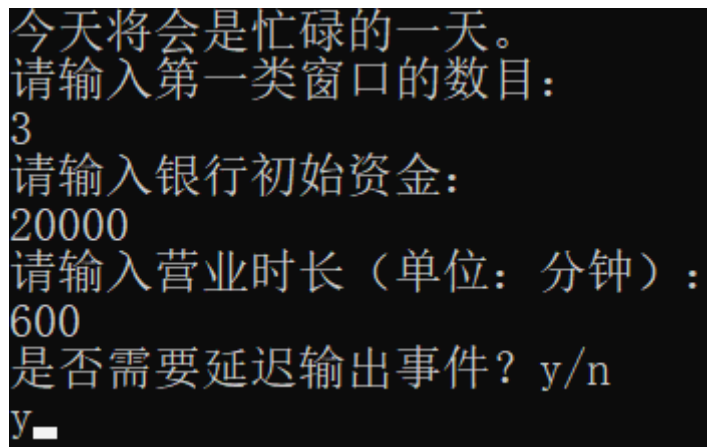
`tmp_win2` 指向当前检查的客户，`pre` 为跟随指针。BUG 出现在一种特定情况，那就是删除的结点为队尾结点时，那么 `pre->next` 就会是 `NULL`，此时如果将 `tmp_win2` 赋值为 `pre->next`，那么在下次循环时，就会访问到队列外的空间。所以，需要改为：

```
pre->next = tmp_win2->next;
tmp_win2 = pre->next;
if (!tmp_win2) window2.tail = pre;
```

- 出现了银行资金总额为负值的情况。问题出现在，在最初设计时，当一个客户办理完业务时，更新银行的金额，比如该客户为取钱 1000 元，那么当他顺利办理完毕业务后，才会将银行总金额 +1000。但这会导致在某些特定情况下，银行的总金额变为负值，那就是当一个客户办理完存钱业务后，会顺序检查第二类窗口，如果在第二类窗口排队的客户的取钱请求可以满足，那么就将他的开始办理业务加入事件表，这就意味着他所要取的那一部分钱“已经许给他了”，这笔钱虽然还在银行里，但本质上已经支出去了。但此时因为第二类窗口的客户还没办理完业务，所以这笔钱还算在银行总金额里，所以如果接下来有客户在第一类窗口取钱，仍可以将这笔钱取出，最终可能导致银行的总金额出现负值。所以调整为：当客户的办理业务为取钱时，当他离开时，更新银行钱数；当客户的办理业务为取钱时，无论是第一类窗口还是第二类窗口，当他开始办理取钱业务时，就更新银行钱数。
- 此外还有一些小的 BUG，比如变量名输入错误，事件类型输入错误，这类错误较小，但比较蠢，Debug 出来以后会令人非常生气...

## 五、代码测试

### 1. 输入界面



```
今天将会是忙碌的一天。
请输入第一类窗口的数目：
3
请输入银行初始资金：
20000
请输入营业时长（单位：分钟）：
600
是否需要延迟输出事件？ y/n
y_
```

第一行：会随机生成“忙碌”、“普通”、“清闲”三种状态，三种状态的客户数量由多到少。

第二行：键盘输入第一类窗口数量，上线设置为 100. 在实际测试中发现，当第一类窗口数量大余 10 个后，实际排队情况区别并不大。

第四行：键盘输入银行初始资金。



第六行：键盘输入营业时长，理论上限为整形数据类型的上限，但建议一天的营业时长实际不超过600分钟（10小时）

第八行：是否需要延迟输出事件，如果需要，请输入y，其他输入均视为不需要。当事件延迟输出时，每一条事件会以800ms的间隔输出。

## 2. 随机生成的事件表

```
事件0:
    6:00    银行开门，开始营业，营业时长为600分钟
    当前银行存有资金总额：20000元

事件1:
    6:00    一位客户到达银行，办理的业务是存钱1228元, 需要办理时长为：32分钟。
    该客户在第一类1号窗口排队等待
    当前银行存有资金总额：20000元

事件2:
    6:00    一位客户在第一类1号窗口开始办理存钱1228元业务
    当前银行存有资金总额：20000元

事件3:
    6:15    一位客户到达银行，办理的业务是存钱1361元, 需要办理时长为：17分钟。
    该客户在第一类2号窗口排队等待
    当前银行存有资金总额：20000元

事件4:
    6:15    一位客户在第一类2号窗口开始办理存钱1361元业务
    当前银行存有资金总额：20000元

事件5:
    6:30    一位客户到达银行，办理的业务是取钱1274元, 需要办理时长为：32分钟。
    该客户在第一类3号窗口排队等待
    当前银行存有资金总额：20000元

事件6:
    6:30    一位客户在第一类3号窗口开始办理取钱1274元业务
    当前银行存有资金总额：18726元

事件7:
    6:32    一位客户在第一类2号窗口办理完存钱1361元业务，离开银行，逗留时间：17分钟
    当前银行存有资金总额：20087元

事件8:
    6:32    一位客户在第一类1号窗口办理完存钱1228元业务，离开银行，逗留时间：32分钟
    当前银行存有资金总额：21315元
```

```
事件94:
11:51 一位客户在第一类1号窗口开始办理存钱266元业务
当前银行存有资金总额: 13367元

事件95:
11:57 一位客户在第一类2号窗口办理完存钱4071元业务, 离开银行, 逗留时间: 21分钟
当前银行存有资金总额: 17438元

事件96:
12:02 一位客户到达银行, 办理的业务是取钱1562元, 需要办理时长为: 19分钟。
该客户在第一类2号窗口排队等待
当前银行存有资金总额: 17438元

事件97:
12:02 一位客户在第一类2号窗口开始办理取钱1562元业务
当前银行存有资金总额: 15876元

事件98:
12:14 一位客户在第一类1号窗口办理完存钱266元业务, 离开银行, 逗留时间: 23分钟
当前银行存有资金总额: 16142元

事件99:
12:15 一位客户到达银行, 办理的业务是取钱2681元, 需要办理时长为: 32分钟。
该客户在第一类1号窗口排队等待
当前银行存有资金总额: 16142元

事件100:
12:15 一位客户在第一类1号窗口开始办理取钱2681元业务
当前银行存有资金总额: 13461元

事件101:
12:21 一位客户在第一类2号窗口办理完取钱1562元业务, 离开银行, 逗留时间: 19分钟
当前银行存有资金总额: 13461元

事件102:
12:26 一位客户到达银行, 办理的业务是存钱1230元, 需要办理时长为: 44分钟。
该客户在第一类2号窗口排队等待
当前银行存有资金总额: 13461元

事件103:
12:26 一位客户在第一类2号窗口开始办理存钱1230元业务
当前银行存有资金总额: 13461元

事件104:
12:33 一位客户到达银行, 办理的业务是存钱1445元, 需要办理时长为: 33分钟。
该客户在第一类3号窗口排队等待
当前银行存有资金总额: 13461元

事件155:
15:49 一位客户在第一类2号窗口办理完取钱6902元业务, 离开银行, 逗留时间: 36分钟
当前银行存有资金总额: 2571元

事件156:
15:54 一位客户到达银行, 办理的业务是存钱5187元, 需要办理时长为: 26分钟。
该客户在第一类1号窗口排队等待
当前银行存有资金总额: 2571元

事件157:
15:54 一位客户在第一类1号窗口开始办理存钱5187元业务
当前银行存有资金总额: 2571元

事件158:
16:00 一位客户未能在第一类1号窗口办理完存钱5187元业务, 离开银行, 逗留时间: 6分钟
当前银行存有资金总额: 2571元

事件159:
16:00 一位客户未能在第二类窗口开始办理取钱7575业务, 离开银行, 逗留时间: 15分钟
当前银行存有资金总额: 2571元

事件160:
16:00 营业结束. 共计53位顾客来办理业务, 在银行逗留的总时长为1249分钟, 平均逗留时间为23. 57分钟
当前银行存有资金总额: 2571元
```

会输出全部事件, 最后一条为计算平均逗留时间。

## 六、实验总结

1. 本次实验, 我写了三个版本的程序 (均在附录中), 一是完成题目基本要求的基础版本, 二是 Debug 版 (各用户的信息可以手动输入), 三是扩展版本, 添加了一些自己发散的部分。Debug 版只需要修改一下输入形式, 所用时间最短。在另外两个版本中, 基础版本所用时间远超过扩展版本, 包括代码书写时间和 Debug 时间, 尽管基础版本能实现的功能较少。几乎所有的 BUG 都出现在基础版本, 现在反思一下, 原因主要有:

- 在写代码之前，只是粗略想了一下程序的总体框架，并没有认真分析有多少模块，每个模块具体完成哪些任务，模块之间的联系是什么
  - 程序较大，自己写大作业的经验不多
2. 在基础版本的程序中，是事件和客户两类队列双驱动进行的，较为混乱，在扩展版本改为纯粹的事件表驱动，每次执行事件表的第一条事件。
  3. Debug 版本发挥了较大作用，因为有些 BUG 只出现在某类特定情况，而如果客户序列完全随机生成的话，出 BUG 的序列复现的难度较大。Debug 版本将所有的客户均由键盘输入，可以更清晰地看到出问题的环节具体在哪里。
  4. 所有的变量名称尽可能包含这个变量的信息，比如 CustomerNum，这一方面有助于后期调试时一眼就能看出不同变量表示什么，另一方面也减少了混淆变量名的情况。

## 七、附录

---

### 1. 离散事件模拟.cpp

完成了实验要求的基础内容

### 2. 离散事件模拟手动输入版 (debug版) .cpp

支持手动输入所有的客户信息，方便复现出 BUG 的客户序列

### 3. 离散事件模拟扩展.cpp

增加了自己发散的内容