实验一: 银行业务模拟实验报告

舒文炫 PB18000029

2021 年 12 月 18 日

目录

Chapter 1

实验要求

银行业务模拟是栈和队列的一个应用场景，基础部分描述如下：客户业务分为取款和存款，客户到达时会先排队，然后对每个客户业务进行处理，如果银行现存资金可以满足客户需求，客户在处理完后立即离开，否则排入另一个队列，等待银行得到资金，就循环遍历这个队列去寻找能否满足其中的客户，直到满足不了为止。这样一直进行，直到银行关门，给出客户在银行的逗留时间。

附加部分：对第一种队列，考虑开放多个窗口，每次顾客到来会选择队列最短的窗口

我所完成的除了以上基础部分和附加部分外，还增添了一些其他的设置，可以方便的自定义模拟的各种参数，研究不同参数下得到结果的不同，也可以通过查看详细信息得到每一步模拟发生的情况和当前的状态，在人机交互方面比较友好。

Chapter 2

设计思路

为了实现更好的文件管理和功能的模块化，我将这个实验代码拆分成实现不同功能的各个文件，这也方便进行 debug。

首先是对所需要的参数的声明，根据实验要求，这里需要银行总资金以及营业时间，客户业务需要资金的范围，客户业务处理时间范围，第一种队列的窗口数目，为了方便管理，我将其统一放到头文件 setting.h 中，并给出这些参数的默认值，如果不自定义参数，这些参数将按照默认值来，同时这个头文件还能实现重设参数的功能，方便用户在修改了参数后，如果还想回到默认值，可以很方便的回去。

然后是数据结构的定义，根据实验要求，是对离散事件模拟，需要事件，每个事件包含发生时间，事件类型，指向下一个事件的指针，这样按照事件发生时间排列，可以构成一个事件链表。然后客户也需要定义，客户会在对应窗口排队，需要包含其到达时间，交易金额，交易用时，指向下一个客户的指针，构成客户队列。然后对每个窗口也需要定义，给出窗口长度，以及指向客户队头和队尾的指针，方便插入和删除操作。这些设计我将其放到 customer.h 头文件中

上面是最底层的设计，后面的实现都要基于这些参数和数据结构。

然后是设计事件的操作，包括创建事件，插入事件到事件表，这里需要进行比较，因为事件根据其发生时间构成有序表，还有打印事件表, 删除事件, 还有比较特殊的需要考虑，当银行可以服务第二种队列的客户时，这时根据我的理解其他的客户都不能被服务，也就是被阻塞了，那么此时，这些客户的离开时间就会被改变，所以这些事件需要被重新构造。这些操作放到头文件 eventop.h 中

然后是对客户队列的操作，包括客户入队列，需要先找到当前最短队列，将客户插入，以及在对应窗口客户服务完成后出队列，还有当银行获得资金时，需要按序扫描第二种队列，寻找可以满足的客户，服务他，让他离开，这里扫描停止当且仅当所有客户都扫过一遍或者当前资金已经不够时。

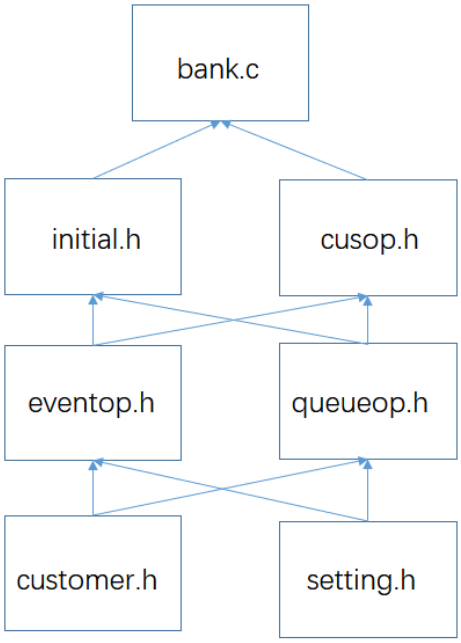
上面两个操作对底层数据结构进行一层封装，后面需要进行这些操作，只要调用这两个头文件。

然后设计客户模块，需要处理客户到达事件和客户离开事件，以及当达到银行的营业时间时，将剩余客户赶走 (free 其空间)，不过此时这些客户的等待时间也要算上。这些操作我放在 cusop.h 中

然后是银行初始化模块，需要在模拟程序启动后，对银行进行初始化，按照所给参数构建各种窗口，队列，这个文件我放在 initial.h 中

最后是顶级模块，调用 cusop.h 和 initial.h 文件，进行银行业务模拟，由于所有操作以及封装完成，这个模块会显得十分简洁, 框架清晰。

上面的文字用一张图表示就是



箭头指向表示文件之间的依赖关系

Chapter 3

关键代码讲解

这一部分将具体展示我的代码

3.1 底层的参数部分

```
#ifndef _SETTINGS_H_
#define _SETTINGS_H_

int total=10000;
int closetime=600;
int windownum=2;
int arrive_interval=10;
int arrive_min=1;
int transaction_interval=6;
int transaction_min=5;
int transaction_amount=500;
int transaction_minamount=-350;
int details=0;
int totaltime;
int customernum;
int amountnow;
int distime;
int lastdisoccur;
```

这是相关参数的定义，基本上是按照对应英文来定义的 total 是总资金，closetime 是银行关门时间，windownum 是第一种窗口的数量，arrive_interval 和 arrive_min 共同作用表示到达事件发生的间隔为 [arrive_interval-arrive_min,arrive_interval+arrive_min], 后面的 transaction_min，transaction_interval 共同作用，表示交易时间的范围，transaction_amount，transaction_minamount 共同作用表示交易金额的范围，和前面到达事件发生间隔是相同的计算。details 表示你是否想看每一步模拟的细节，amountnow 是银行当前资金，customernum 是当前服务客户数量，totaltime 表示客户等待总共用时，后面两个参数的用处会在后面说明，这里大部分情况下用不上这两个参数，是在处理第二种队列里面排队的客户时使用的。

```
int reset(){

    total=10000;
    closetime=600;
    windownum=2;
    arrive_interval=10;
    arrive_min=1;
    transaction_interval=6;
    transaction_min=5;
    transaction_amount=500;
    transaction_minamount=-350;
    details=0;

}
#endif // _SETTINGS_H_
```

这是重设参数函数，也就是将上面的全局变量重新用默认值赋值

3.2 数据结构部分

```
#ifndef _CUSTOMER_H_
#define _CUSTOMER_H_

typedef struct event{//事件表
    int occur_time;
    int event_type;
    int flag;
    struct event *next;

}Event,*pEvent;

typedef struct cus_queue{//在各个窗口的队列
    int arrive_time;
    int duartion;
    int amount;
    int flag;
    struct cus_queue *next;
}cus_queue;


typedef struct windows{//每个窗口
    int queuelen;
    struct cus_queue *head,*tail;

}windows,*pwindows;

pEvent pev,ev;

pwindows pw;
#endif
```

可以看到这里事件我用结构体 Event 表示，指向事件的指针定义为 pEvent，基础包含的事件发生时间 occur_time 和事件种类 event_type，这里事件种类主要分为离开事件，到达几号窗口事件，用对应的整型数表示，flag 参数在后面用到是再说明其作用。

客户我用结构体 cus_queue 表示，包含到达时间 arrive_time，交易金额 amount, 交易用时 duration, 指向下一个客户的指针，然后窗口用结构体 windows 表示，包含该窗口的队列长度 queuelen 和客户队列的头尾指针，这样做方便进行最短队列的查找，客户的插入和删除。

后面还接着定义了全局变量 pev，pw 就是我的事件表和窗口，ev 方便我从事件表中取事件

3.3 事件的操作

```
#include"customer.h"
#include<malloc.h>
#ifndef _EVENT_H_
#define _EVENT_H_


pEvent init_event();//初始化事件表
int print_event(pEvent p);//打印事件表
int orderinsert(pEvent p,pEvent a);//按序插入事件
pEvent makeevent(int a,int b);//创建事件
pEvent deevent(pEvent p);//删除事件
int reconductevent(pEvent p,int a);//重建事件表
int resetflag(pEvent p);//重设事件的flag
```

上面是我声明的一系列操作，具体功能在注释里写上了, 下面是具体实现

```
pEvent init_event(){
    pEvent q;
    q=(pEvent)malloc(sizeof(Event));
    q->event_type=0;
    q->occur_time=-1;
    q->flag=0;
    q->next=NULL;
    return q;

}
pEvent makeevent(int a,int b){
    pEvent p;
    p=(pEvent)malloc(sizeof(Event));
    p->event_type=b;
    p->occur_time=a;
    p->flag=0;
    p->next=NULL;

    return p;
}
```

初始化事件表，这里是创建了事件的头指针，同时将头指针的 event_type 复用做事件表的长度，创建事件传入参数 a，b 分别作为事件发生时间和类型

```
int print_event(pEvent p){
    pEvent q;
    q=p;
    while(q!=NULL){
        printf("%d %d->",q->event_type,q->occur_time);
        q=q->next;
    }
    printf("\n");
    return 0;
}
```

打印事件表也就是从头到尾遍历事件表并输出到命令行界面

```
int orderinsert(pEvent p,pEvent a){
    pEvent q,r;
    q=p;

    while(q->next!=NULL){
        if(q->next->occur_time<a->occur_time){
            q=q->next;
        }
        else{
            a->next=q->next;
            q->next=a;
            p->event_type++;
            break;
        }
    }
    if(q->next==NULL){
            event* event::next
            q->next=a;
            p->event_type++;
    }
    return 0;
}
```

这里按照事件发生顺序插入所创建的事件，也就是遍历事件表并比较要插入的事件发生时间以及当前指针所指的事件发生时间，插入完成之后需要将事件表头节点的 event_type 加一，表示表长 +1

```
pEvent deevent(pEvent p){
    pEvent q;
    q=p;
    if(p->next!=NULL){
        q=p->next;
        p->next=p->next->next;
        p->event_type--;

    }
    return q;


}
```

删除事件就很容易了，直接删掉头结点的下一个结点，并将表长-1 即可

resetflag 函数就是将事件里面的 flag 域全部重设为 0，这个用处就是稍微简化下面的代码，这里就不放出来了

```
int reconductevent(pEvent p,int a){
    pEvent q,s;

    q=p;
    while(q->next!=NULL){
        if(q->next->event_type!=0&&q->next->flag!=1){
            p->event_type--;
            s=q->next;

            q->next=q->next->next;
            s->next=NULL;
            s->occur_time=s->occur_time+a;
            s->flag=1;
            orderinsert(p,s);
        }
        else
            q=q->next;
    }
    resetflag(p);
}
```

该函数的作用体现在当开始处理第二种队列里面的客户时，当前在第一种队列里面的客户的离开事件都会受到影响，就是被推迟，推迟的时间为 a，但是客户到达的时间不会被这个影响，那么这样一来整个事件表就需要重新构建，保证其为有序的状态，我这里就是去判断该事件是否为离开事件，是就先将其拿出来，离开时间 +a 然后再按序插入到这个事件表中，同时将该事件 flag 置 1，表示该事件已经被处理过了，下次遇到这种事件时就不要再处理了。事件表重建完后，，将所有的事件的 flag 重设，方便下一次重建操作

3.4 客户队列的操作

```
#include"customer.h"
#include<malloc.h>
#ifndef _QUEUEOP_H_
#define _QUEUEOP_H_

cus_queue* init_queue();//初始化客户队列
int print_queue(pwindows p,int a);//打印队列
int minqueue(pwindows p);//求最短队列
int enqueue(pwindows p,int i,int a,int b,int c);//客户入队列
cus_queue* makecus(int a,int b,int c);//创建客户
cus_queue* dequeue(pwindows p,int i);//删除对应窗口的客户
cus_queue* gethead(pwindows p,int i);//得到对应窗口队列的头
int checkqueue(pwindows p,int a);//检查第二种队列
```

这里初始化，创建，打印之类的操作都很容易，和事件里面差不多，没什么好说的, 就不用来凑字数了

```
int minqueue(pwindows p){
    int i,j;
    j=0;
    for(i=0;i<windownum;i++){
        if(p[i].queuelen<p[j].queuelen){
            j=i;
            break;
        }
    }
    return j;
}
```

这里求最短队列十分的容易，因为我维护了每个窗口长度这样的一个数据，所有只需要遍历窗口，比较，得到窗口长度最小的窗口序号就可以了

```
int enqueue(pwindows p,int i,int a,int b,int c){
    cus_queue* q;
    q=makecus(a,b,c);
    p[i].tail->next=q;
    p[i].tail=q;
    p[i].queuelen++;
    return 0;
}
```

这里插入也十分的容易，因为我维护了队列尾指针这样的数据，只需要建立事件，插入到尾指针指定位置然后更新一下尾指针和队列长度就可以了

```
cus_queue* dequeue(pwindows p,int i){
    cus_queue* q;
    q=p[i].head->next;
    p[i].head->next=p[i].head->next->next;
    if(p[i].head->next==NULL){
        p[i].tail=p[i].head;
    }
    q->next=NULL;
    p[i].queuelen--;

    return q;
}
```

删除就是直接把第一个结点删掉，然后队长-1 就行

```c
int checkqueue(pwindows p,int a){//a为银行收到资金前的剩余资金
    cus_queue* q,*qaq;
    cus_queue* r;
    distime=0;
    if(p[windownum].head->next!=NULL){//windownum对应的队列就是第二种队列，此时中断第一种队列的服务
        do{
            if(amountnow>=-p[windownum].head->next->amount){//当目前资金还可以满足第二种队列的客户时
                r=dequeue(pw,windownum);
                distime=distime+r->duartion;//distime是处理第二种队列里面客户导致的延迟时间
                amountnow=amountnow+r->amount;
                totaltime=totaltime+ev->occur_time+distime-r->arrive_time;//处理完该用户，总等待时间增加
                free(r);

            }
            else{//当前资金无法满足当前客户，循环查看后面的客户
                if(p[windownum].queuelen>1){
                        printf("%d ",p[windownum].head->next->amount);

                    qaq=p[windownum].head->next;
                    qaq->flag=1;//访问过的客户flag置1 ，防止重复访问，同时将该客户插入到队尾
                    p[windownum].head->next=p[windownum].head->next->next;
                    qaq->next=p[windownum].tail->next;
                    p[windownum].tail->next=qaq;
                    p[windownum].tail=p[windownum].tail->next;
                }
                else{
                qaq=p[windownum].head->next;
                    qaq->flag=1;
                }
            }//队列全部访问了一遍或者银行当前资金少于a就停止循环，此时无法满足后面的客户了
        }while(p[windownum].head->next!=NULL&&p[windownum].head->next->flag!=1&&amountnow>a);
```

```c
    if(p[windownum].head->next!=NULL){//重设flag
        q=p[windownum].head->next;
        while(q!=NULL){
            q->flag=0;
            q=q->next;
        }
    }
    lastdisoccur=ev->occur_time;//上一次中断发生时间
    }


    return 0;
}
```

这里我个人认为比较巧妙的是就是 checkqueue 函数的设计，按照题目要求，当银行获得资金时，需要扫描第二种队列看是否可以满足，因为第二种队列里面的客户，是很早就来了但是银行资金不足无法服务，这时需要优先处理这类客户，第一种队列的客户服务需要中止，那么那些第一种队列的客户的离开时间就会受到影响，需要进行额外的处理，这个 lastdisoccur 变量就是为了处理这种情况。这里设置 flag 是为了方便判断该队列是否已经全部访问了一遍，其他大部分内容，我已经在注释中给出

3.5 银行的初始化各种操作

```c
int initializebank();//初始化银行，主要是给出模拟交互界面
int openforday();//银行开门营业


int initializebank(){//对银行进行初始化设定
    int i;
    printf("欢迎进入银行业务模拟系统，输入0跳过初始化，使用系统默认设定，输入1进入设定\n");
    scanf("%d",&i);
    printf("是否查看详情,是输入1，否输入0\n");
    scanf("%d",&details);
    if(i==0){
        printf("进入银行\n");
    }
    else{
        printf("输入银行总额以及营业时间\n");
        scanf("%d %d",&total,&closetime);
        printf("输入第一种窗口个数\n");
        scanf("%d",&windownum);
        printf("输入顾客到达时间间隔范围和最短时间间隔\n");
        scanf("%d %d",&arrive_interval,&arrive_min);
        printf("输入顾客交易时间间隔范围和最短时间间隔\n");
        scanf("%d %d",&transaction_interval,&transaction_min);
        printf("输入顾客交易的金额范围以及最小交易金额数\n");
        scanf("%d %d",&transaction_amount,&transaction_minamount);

        printf("初始化完成,进入银行\n");

    }
    return 0;
}
```

首先是 initializebank 函数，就是给出一个银行业务模拟系统的交互界面，你可以选择进行各种参数的设置，是否需要查看每一步的具体情况之类的操作

```
int openforday(){
    int i;
    totaltime=0;
    customernum=0;
    distime=0;
    lastdisoccur=0;
    pev=init_event();
    ev=makeevent(0,0);
    orderinsert(pev,ev);
    srand((unsigned)time(NULL));
    pw=(pwindows)malloc((windownum+1)*sizeof(windows));
    for(i=0;i<windownum+1;i++){
        pw[i].queuelen=0;
        pw[i].head=init_queue();
        pw[i].tail=pw[i].head;


    }


}
```

然后是 openforday 函数，该函数调用后将会生成队列窗口，将总时间，总人数置 0，为后面模拟做准备, 这一块逻辑上没什么难点

3.6 客户事件处理

```
int cus_arrived();//处理用户到来事件
int cus_departure();//处理用户离开事件
int closethebank();//银行关门，这里需要对银行剩余的没服务的客户进行清理，所以也算对用户操作
```

声明的三个操作

```
int cus_arrived(){
    int inter_time;
    int dur_time;
    int cus_amount;
    int t,i;
    pEvent a,b;
    customernum++;
    //当前客户到来时产生随机数用来得到下一个客户交易时间，交易金额
    dur_time=rand()%transaction_interval+transaction_min;
    inter_time=rand()%arrive_interval+arrive_min;
    cus_amount=rand()%transaction_amount+transaction_minamount;
    t=ev->occur_time+inter_time;


    if(t<closetime){//下一个客户在银行关门前到来就将下一个客户的到来事件加入事件表
        a=makeevent(t,0);
        orderinsert(pev,a);
    }
        i=minqueue(pw);//将当前客户插入到队长最短的队列的窗口
        enqueue(pw,i,ev->occur_time,dur_time,cus_amount);

    if(pw[i].queuelen==1){//如果当前队列的长度为1，那么就可以产生该客户的离开事件了
        if(lastdisoccur+distime<ev->occur_time){
            b=makeevent(ev->occur_time+dur_time,i+1);
            orderinsert(pev,b);
        }
        else{
            b=makeevent(distime+lastdisoccur+dur_time,i+1);
            orderinsert(pev,b);
        }


    }
    return 0;
}
```

客户到来事件的处理，主要部分注释里面写的应该比较清楚，这里我说一下 lastdisoccur 的作用，因为在处理第二种队列的客户时，第一种队列正在等待离开的客户会被影响，虽然客户的到来并不会被影响，不过在第一种队列长度只有 1 时，需要生成这个新到来的客户的离开事件，这时就会被影响了，那么就不能用常规的办法生成这个离开事件，不然时间是不正确的。lastdisoccur 就是上一次银行获得资金的时间，这时发生中断，distime 是在中断持续的时间，这样的话，如果新客户在中断结束之后到来，那就是正常的，如果在中断过程中到来，也就是时间大于 lastdisoccur 小于 lastdisoccur+distime，那你就得等到 lastdisoccur+distime 之后才能办理，离开时间会变为 lastdisoccur+distime+durtime. 由于事件表有

序，大于 lastdisoccur 这个条件一定会满足，这里只需要比较这个到达时间和 lastdisoccur+distime 的大小即可。

```
int cus_departure(){
    int i,j;
    pEvent qwq;
    cus_queue* cusqwq;
    i=ev->event_type;
    cusqwq=dequeue(pw,i-1);
    int disflag;
    disflag=0;
    if(ev->occur_time<=closetime){//离开事件在银行关门之前
        if(cusqwq->amount<=0){//客户想要取款
            if(amountnow+cusqwq->amount<0)//银行没有足够资金满足，将客户插入到第二种队列
                enqueue(pw,windownum,cusqwq->arrive_time,cusqwq->duartion,cusqwq->amount);
            else{//银行可以满足，服务该客户
                amountnow=amountnow+cusqwq->amount;
                totaltime=totaltime+ev->occur_time-cusqwq->arrive_time;
            }
        }
        else{//客户想要存款
            j=amountnow;//先保存当前银行资金
            amountnow=amountnow+cusqwq->amount;
            totaltime=totaltime+ev->occur_time-cusqwq->arrive_time;
            checkqueue(pw,j);//银行开始服务第二种队列的客户，直到资金小于j，这时会得到中断持续时间和中断开始时间
            printf("@@@@@%d %d\n",distime,lastdisoccur);
            if(distime!=0)//发生了中断，重建事件表，主要是对已经存在的离开事件进行推迟
                disflag=1;
                reconductevent(pev,distime);
        }
            free(cusqwq);
            if(pw[i-1].queuelen!=0){//产生当前窗口下一个客户的离开事件
                cusqwq=gethead(pw,i-1);
                if(disflag==1){//发生了中断，下一个客户的离开时间会被推迟
                    qwq=makeevent(lastdisoccur+distime+cusqwq->duartion,1);
                }
                else //没中断，下一个客户可以正常离开
                    qwq=makeevent(ev->occur_time+cusqwq->duartion,i);
                orderinsert(pev,qwq);
            }

    }
    else{//离开事件在银行关门后，不做任何处理，直接把等待时间加上
        totaltime=totaltime+closetime-cusqwq->arrive_time;
```

客户离开事件的处理，主要部分看注释就可以了，这里同样的需要使用到 lastdisoccur 和 distime，因为发生了中断后，当前事件表中的离开事件全部需要后延，但是客户到来事件不受影响，所以需要进行事件表的重建，而不是简单的给所有事件加上相同的延迟就可以了，对于当前窗口下一个客户的离开事件的产生，处理方式与上面客户到来且队列长度只有 1 的情况相同，这里不再赘述。

```
int closethebank(){
    cus_queue* qwq;
    int i;
    for(i=0;i<windownum+1;i++){


    while(pw[i].head->next!=NULL){
        qwq=dequeue(pw,i);
        totaltime=totaltime+closetime-qwq->arrive_time;
        free(qwq);
    }
    }
    return 0;

}
```

银行关闭事件，这时需要把所有正在等待的客户全部清理，将他们的等待时间都加到总等待时间里面，清空所有队列和事件表。

3.7 顶层模块，银行业务模拟综合

```
int main(){
    int i,j,k;
    int flag=1;
    int re=1;
    float average;
    while(flag){
        if(re)
            initializebank();
        openforday();
        amountnow=total;

        while(pev->event_type!=0){
                    if(details){
                        printf("------\n");
                        print_event(pev);
                        print_queue(pw,windownum);
                        printf("总时间:%d 现在的金钱:%d 顾客总数:%d\n",totaltime,amountnow,customernum);
                        printf("------\n");
                    }
                ev=deevent(pev);
                if(ev->event_type==0){
                    cus_arrived();
                }
                else{
                    cus_departure();

                }
                free(ev);


        }
        closethebank();
```

```
        ,
        closethebank();
                    if(details){
                        printf("------\n");
                        print_event(pev);
                        print_queue(pw,windownum);
                        printf("总时间:%d 现在的金钱:%d 顾客总数:%d\n",totaltime,amountnow,customernum);
                        printf("------\n");
                    }
        average=(float)totaltime/customernum;
        printf("平均逗留时间为%f\n",average);
    printf("继续模拟请输入1，否则输入0\n");
    scanf("%d",&flag);
    if(flag==0)
        break;
    printf("继续使用刚才的配置输入0，调整配置输入1\n");
    scanf("%d",&re);
    if(re)
        reset();
    }

    return 0;
}
```

可以看到这一模块只需要调用前面定义的函数，每次取出一个事件，根据事件的类型进行不同的处理，框架十分的清楚，应该也不用过多描述了。

Chapter 4

调试分析

4.1 时空复杂度分析

这里问题规模是客户的数量

先看对事件的一系列操作初始化事件表，创建事件自不用过多关注，事件复杂的都是 O(1), 打印事件表需要遍历整个链表，时间复杂度为 O(n)，每次有序的插入事件到事件表，由于也是直接遍历，需要 O(n) 的时间，删除事件比较容易，只要 O(1). 重建事件表这一块，我使用了 flag 域来辅助判断该事件是否被处理过，而且 flag 域也只在这里有用，所以相当于增添了一些空间的开销，时间复杂度，由于需要对每个离开事件重新设置发生时间，然后再有序的插入，所有时间复杂度会达到 $O(n^2)$.

再来看客户队列的操作，初始化以及创建，也是很简单不多赘述，打印队列相当于把所有客户遍历一遍，所以时间复杂度 O(n), 寻找最小队列，只需要 O(windownum) 时间，插入和删除客户都是 O(1) 时间，因为我维护了头尾指针变量，相当于用一个空间来换取遍历所花时间。主要的 checkqueue 函数，这里我也使用了 flag 域辅助判断队列是否已经访问过一遍，所以这里相当与有额外的空间花销，时间复杂度仍是 O(n) 的

初始化模块 initial.h 没什么好说的，这里跳过。客户操作模块，用户到来事件的处理主要时间开销在插入新事件到事件表，所以时间复杂度为 O(n), 用户离开事件的处理在重建事件表，但是这个在很多情况下并不常发生，不过按最坏的来，不会超过 $O(n^2)$ 就是了。银行关闭事件，将剩余的所以客户清除，剩余多少客户这个不好估计，最坏的是一个也没走掉，所以是 O(n)

顶层模块，就是调用这些函数，综合去看，知道了下层的实现，也就很好分析，这里就不再赘述了

4.2 实验中遇到的问题以及解决方案

- 本实验一个难点在于多出来第二种队列，会影响第一种队列的客户的离开，一开始想法很简单，就是让所以人离开变迟，那我就直接用一个全局变量 delay 记录这个延迟然后最后算时间的时候加上就可以了，但是这样算出来是不对的，只有离开事件会被推迟，到达事件不会，这时会使得事件表的有序性被破坏，实际事件的发生顺序被改变的话，得到的结果自然不会是想要的结果，解决方案就是设计了重构事件表函数
- 重构事件表里面也存在问题，一开始我就直接把每个离开事件拿出来，加上延迟，重新插进去，但是这样无法判断，下一个取出的事件是否被处理过，后果就是使得某一个事件一直被处理，然后函数就爆了，解决方案比较简单粗暴，我加入 flag 域去标明该事件有没有被处理过
- 在循环遍历第二个队列时最开始也是遇到了和上面相同的问题，我开始是想直接使用一个指针来指示第一个结点，当我再次遍历到这个指针时，说明这个队列我已经完全访问了一遍，但是如果一开始就把第一个结点删去了，这个指针就没了，那么循环永远无法结束，或者考虑使用第一个没被删除的结点，这样写起来会麻烦一点，解决方案就是给每个客户加入 flag 域，访问过就置 1，当遇到 flag 为 1 的结点，就停止，这样就不会问题了

Chapter 5

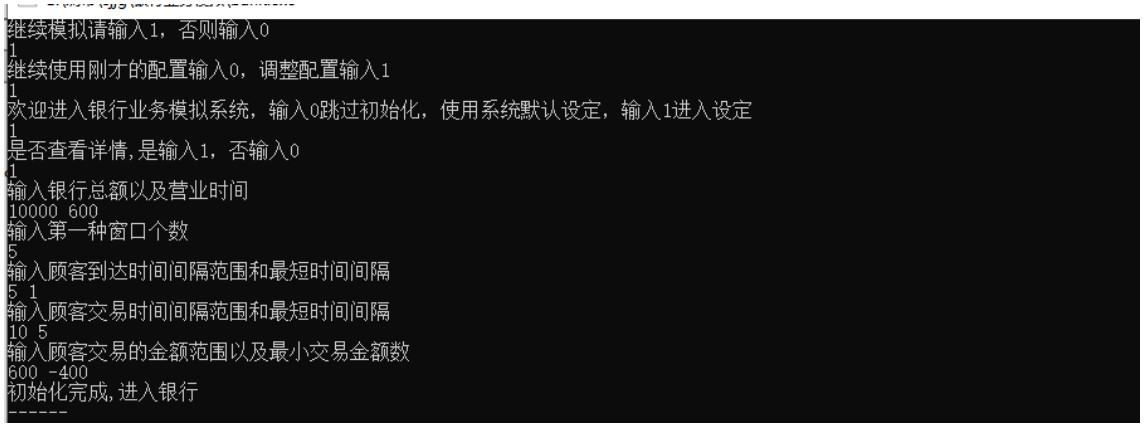
代码测试

下面我对我的代码进行测试，运行 bank.exe 程序，将会显示这样的页面

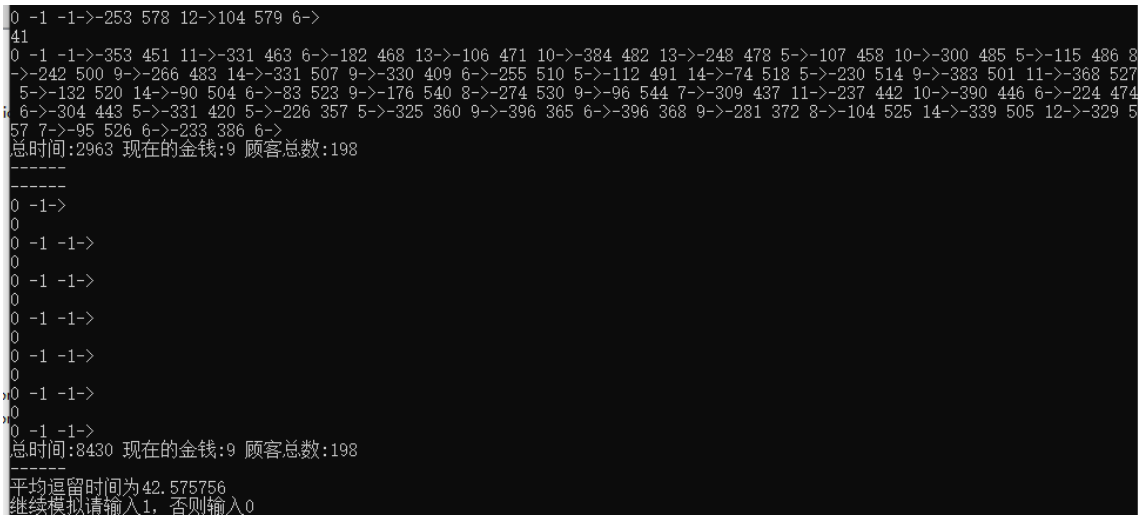
第一次我先使用默认设置，模拟得到下面的结果

默认的设置在客交易时间和到来间隔这些参数方面取得比较均衡，同时客户存取钱的概率差不多，所以可以看到这里平均等待时间基本上很小，和客户交易时间差不多

下面我修改了设置

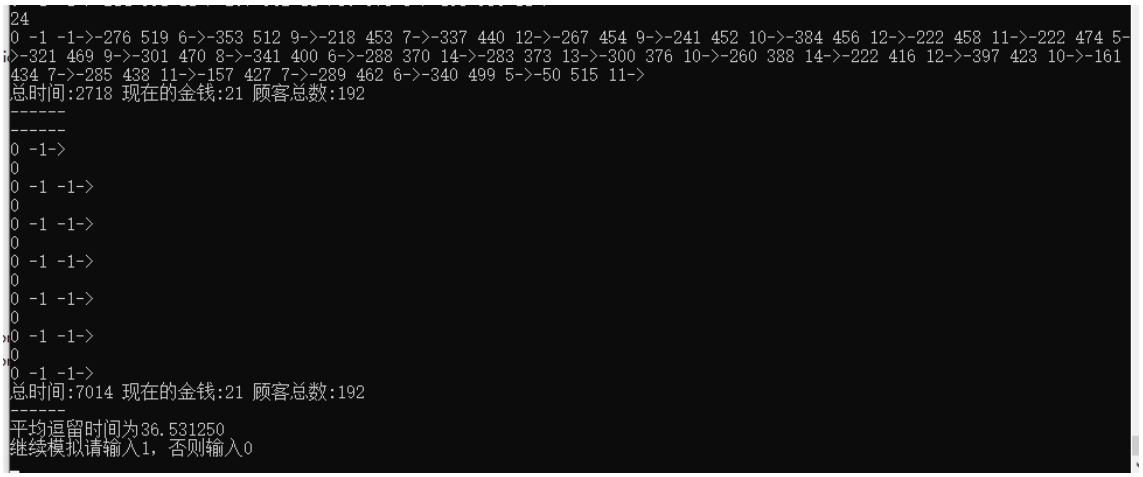


这个设置我故意使得交易金额偏向取钱，这样可以看到如下结果



在银行关门前，第二种窗口下的客户非常的多，这也导致总的等待时间非常的长，远远大于我设置的用户交易时间

再以这个相同的配置跑一次模拟，得到如下结果



可以看到和前一次模拟的结果基本没有区别，平均等待时间仍然很长。

限于篇幅，模拟测试部分大概在这里展示这么多，助教有兴趣可以自行设置不同的参数跑一跑

Chapter 6

实验总结

本次实验我更加深入的学习了队列的使用，在某些细节处也体会到了使用空间来换取时间的妙处，在设计实验时，将所要实现的目标进行拆分，分层封装，这样的做法可以为我以后做更复杂的项目打下基础。不过我的代码也存在一些不足，使用的都是很基本的数据结构，解决问题也是使用比较粗暴的方法，应该会存在一些地方，可以优化我的时间复杂度，比如在重建事件表那一块，每次调用有序插入可能并不是很有必要的。不过总而言之，本次实验的收获还是很多的。

Chapter 7

附录

关于程序的运行，基本上在代码测试里面展示过了，助教如果想运行可以参考测试部分的展示, 个人感觉这个人机交互做的还是比较平易近人的，就是没做图形化就是了 orz。

提交的文件清单

- bank.c 主程序
- initial.h 初始化头文件
- cusop.h 用户事件处理头文件
- eventop.h 事件操作头文件
- queueop.h 用户队列操作头文件
- settings.h 参数配置头文件
- customer.h 数据结构定义头文件
- bank.exe 银行业务模拟程序