离散事件模拟

姓名：吕凯盛
学号：20081590
完成时间：2021 年 10 月 26 日

一．实验要求

[问题描述]客户业务分为两种。第一种是申请从银行得到一笔资金，即取款或借款。第二种是向银行投入一笔资金，即存款或还款。银行有两个服务窗口，相应地有两个队列。客户到达银行后先排第一个队。处理每个客户业务时，如果居于第一种，且申请额超出银行现存资金总额顺得不到满足，则立刻排入第二个队等候，直至满足时才离开银行；否则业务处理完后立刻离开银行。每接待完一个第二种业务的客户，则顺序检查相处理(如果可能)第二个队列中的客广，对能满足的申请者予以满足，不能满足者重新排列第二个队列的队尾。注意，在此检查过程中，一旦银行资金总额少于或等于刚才第一个队列中最后一个客户(第二种业务)被接待之前的数额，或者本次已将第二个队列检查或处理了一遍，就停止被盗(因为此时已不可能还有能满足者)转而继续接待第一个队列的客户。任何时刻都只开一个窗口。假设检查不需要时间。营业时间结束时所有存户立即离开银行。

写一个上述银行业务的事件驱动模拟系统，通过模拟方法求出客户在银行内逗留的平

均时间。

[基本要求]利用动态储存结构实现模拟。

[测试数据]一天营业开始时银行拥有的款额为 10000(元)．营业时间为 600(分钟)。其他模拟参量自定。注意测定两种极端的情况：一是两个到达事件之间的间隔时间很短，而客户的交易时间很长，另一个恰好相反，设置两个到达事件的间隔时间很长，而客户的交易时间很短。

二．设计思路

[模块分析]

大作业程序包含了四个模块：1.结构体声明。2.事件链表类的实现。3.队列类的实现。4.模拟器主程序的实现。

[结构体]

在文件中是 structure.h，其中包含了事件结构体，顾客结构体，随机数(到达时间，处理时间，交易额)结构体，以及随机数生成函数。

[事件链表类]

在文件中是 eventlist.h，其中声明了 eventlist 类，属性为头结点，可以实现的功能为：1.构造函数。2.析构函数。3.打印函数。4.判断事件链表是否为空。5.提取链表首节点。6.插入函数（按照 occurtime 的顺序从小到大插入）。

```
class eventlist
{
public:
    eventlist();
    ~eventlist();
    bool isempty();
    void show();
    littleevent pop();
    void inserfun(littleevent toadd);
    void inserfun(int occurtime, int ntype);
private:
    event* head;
};
```

[队列类]

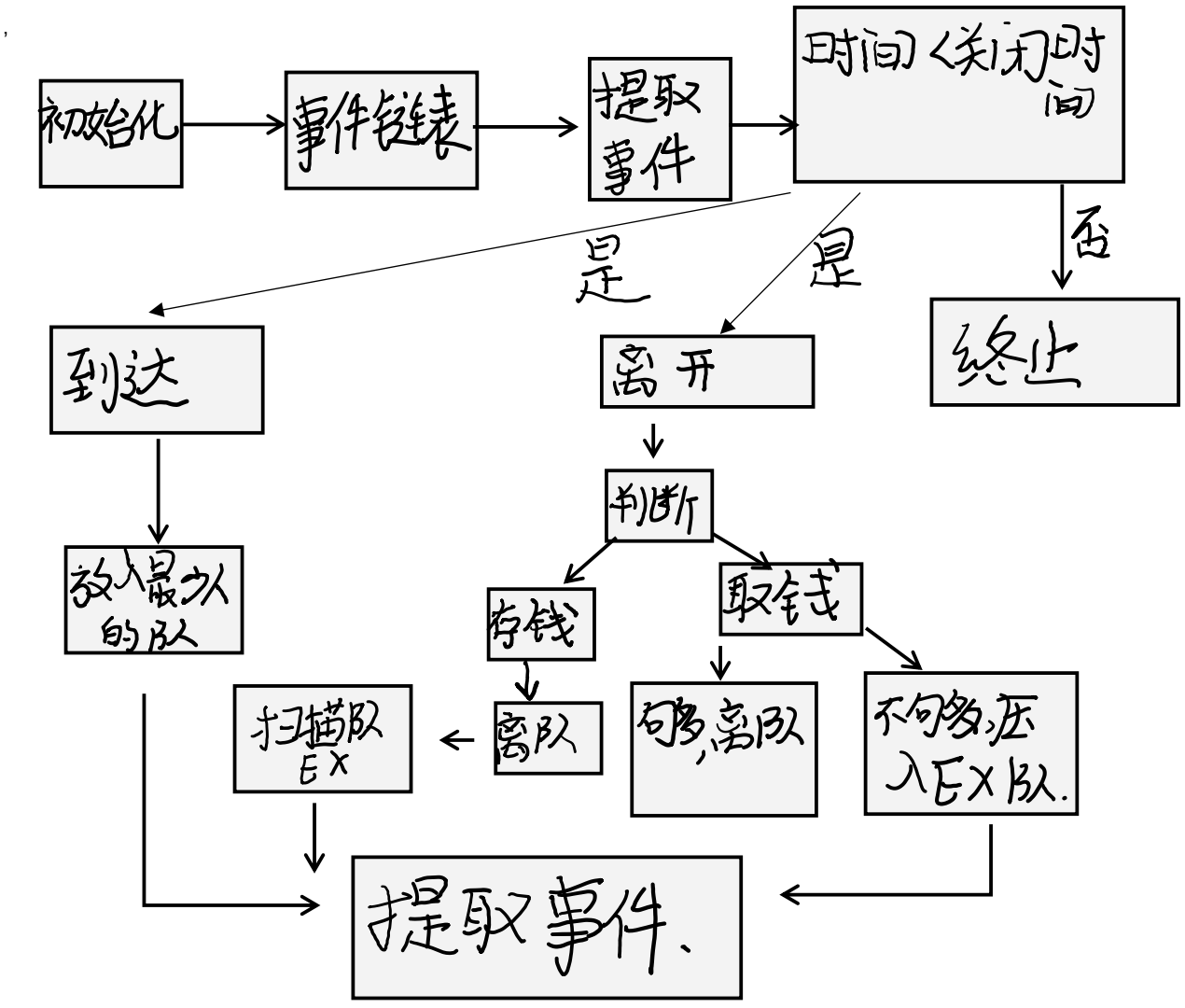
在文件中是 queue.h，其中声明了 queue 类。其属性为:头结点，尾结点，队列长度可以实现的功能为：1.构造

```
class customerqueue {
private:
    customer* head;
    customer* rear;
    int lenth;
public:
    int howmuch();
    customerqueue();
    void display(int mode );
    void addpeople(int arrivaltime, int durationtime, int amount);
    bool isempty();
    void leave();
    int howlong();
    ~customerqueue();
    void scanmode_2(int& custom, int& totaltime, int& bankmoney, int lastmoney, int& nowtime);
    int time();
    int getarr();
    void clearallpeople(int& totaltime, int& custom, int now);
```

函数。2.析构函数。3.展示函数。4.判断是否为空。5.提取队列首的结点。6.离队函数。7. 特殊查找函数。8.清理函数。9.访问函数。10.查找最少人队列函数。

[模拟器主程序]

首先将整个程序初始化。之后从事件链表提取出事件。判断事件的时间是否大于关门事件，若大于，终止程序，并且进行清场。如果不大于，则进行判断。若为到达事件，则生成一个离开事件插入事件链表，并且生成一个顾客类插入到等待队列。若为离开事件，对其进行判断。（1）如果是存钱，则直接离队，并且开始扫描等待的队列。（2）如果是取钱，若钱够，则离队。若钱不够，则离队，并压入等待队列。然后继续提取下一个事件，进入循环。



三．关键代码讲解

```
void eventlist::inserfun(littleevent toadd) {
    event* temp = new event;
    temp->occurtime = toadd.occurtime;
    temp->ntype = toadd.ntype;
    event* current, * prev;
    current = prev = head;
    while (current && current->occurtime < temp->occurtime) {
        prev = current;
        current = current->next;
    }
    if (current == head) {
        temp->next = current;
        head = temp;
    }
    else {
        temp->next = current;
        prev->next = temp;
    }
}
```

[事件链表插入函数]

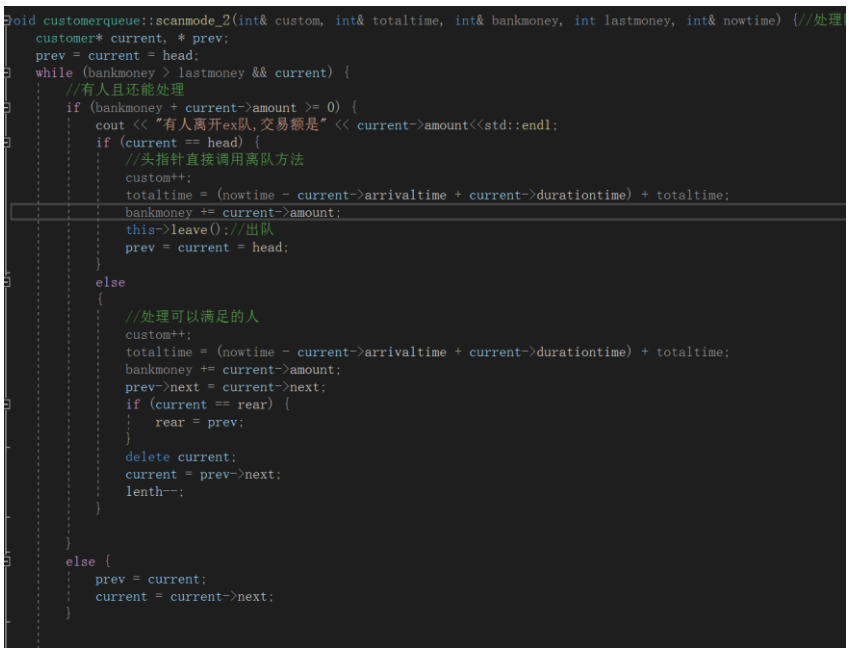
1.将 toadd（要插入的结点）复制给 temp。

2.然后将 current 移动到恰好大于 temp 的时候。

3.若为头结点则采用头插法代替头结点，反之则插到 current 和 prev 之间。

[等待队伍搜索函数]

[part1]



```
void customerqueue::scanmode_2(int& custom, int& totaltime, int& bankmoney, int lastmoney, int& nowtime) {//处理
    customer* current, * prev;
    prev = current = head;
    while (bankmoney > lastmoney && current) {
        //有人且还能处理
        if (bankmoney + current->amount >= 0) {
            cout << "有人离开ex队,交易额是" << current->amount<<std::endl;
            if (current == head) {
                //头指针直接调用离队方法
                custom++;
                totaltime = (nowtime - current->arrivaltime + current->durationtime) + totaltime;
                bankmoney += current->amount;
                this->leave();//出队
                prev = current = head;
            }
            else
            {
                //处理可以满足的人
                custom++;
                totaltime = (nowtime - current->arrivaltime + current->durationtime) + totaltime;
                bankmoney += current->amount;
                prev->next = current->next;
                if (current == rear) {
                    rear = prev;
                }
                delete current;
                current = prev->next;
                lenth--;
            }

        }
        else {
            prev = current;
            current = current->next;
        }
```

1.首先进行循环判断，若当前的钱大于上一次存钱的钱（如果小于，则绝对不可能满足）且等待队伍中还有人，则进行搜索。

2.若当前银行的钱可以满足队中取钱的要求，则将其离队，并且同步银行的数据。这段代码是从

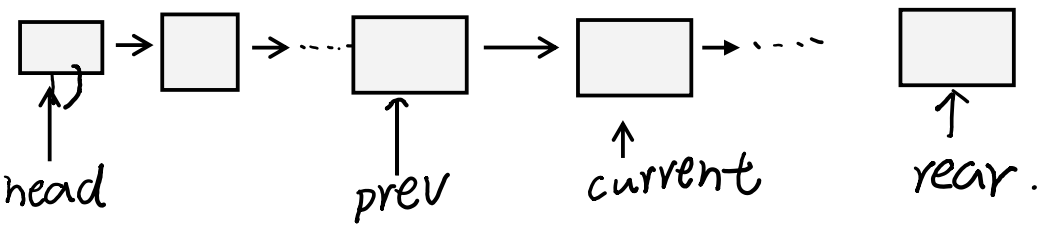
if(bankmoney+current->amount>=0)到 lenth--为止。

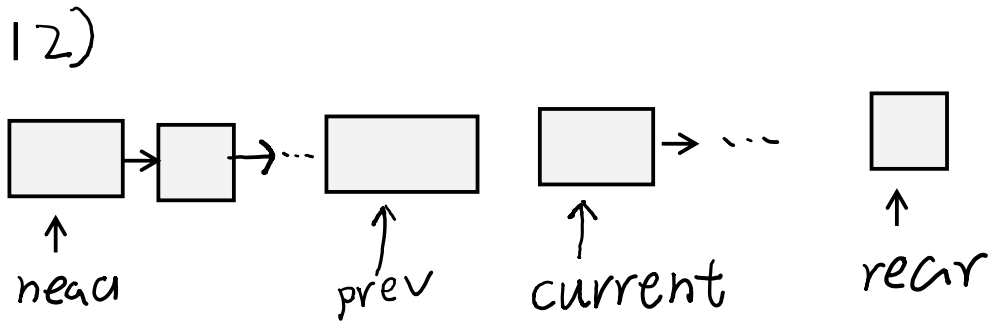
[part2]

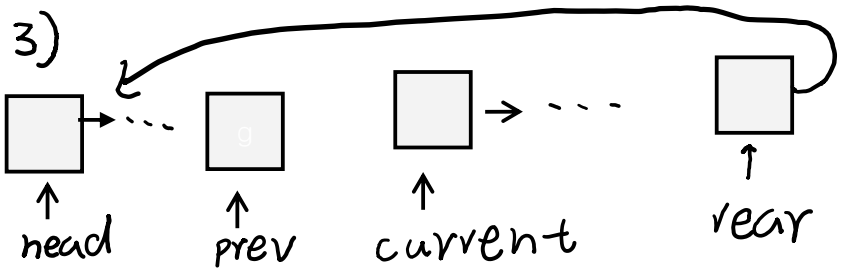
处理无法满足的人。

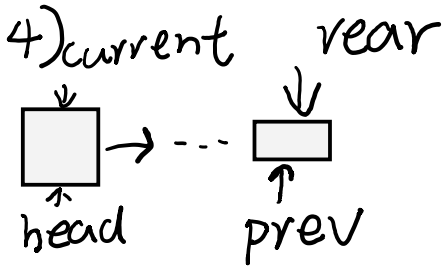
```
//搜索完，把无法满足的置于队尾
if (current != nullptr && current != prev) {
    prev->next = nullptr;
    rear->next = head;
    head = current;
    rear = prev;

}
```

1. 首先进行判断。如果 current 在队尾或者队首，则不用进行调换。
2. 将前面没办法满足的放到队尾（这一部分人比较难满足）代码逻辑如下图所示









首先断开 prev 和 current 的连接，再将 rear 的 next 指向 head。最后将 head 设为 current，将 rear 设为 prev。

[模拟器主程序]

[part1]初始化

声明需要用到的所有变量，并且把

```
//初始化
srand((unsigned)time(NULL));
eventlist evlist;
customerqueue *handle_queue, waitqueue;//handle处理正在做的，wait处理大额取款
handle_queue = new customerqueue[handle_num];
littleevent happen = { 0,-1 }, detect = { 0,0 };//happen是一个中间变量,detect用于提取出的事件,并对其分析
evlist.inserfun(happen);
detect = evlist.pop();
int now;
randomtime ran;//储存随机数
```

happen={0，1}插入到链表，表明在时间 0 的时候有人来办理业务。

[part2]

模拟部分。

```
while (detect.occurtime < closetime) {
    if (detect.ntype == -1) {//入队事件
        //cout << totalmoney<<"\t";
        random(ran);
        happen.ntype =-1, happen.occurtime = ran.intertime + detect.occurtime;
        evlist.inserfun(happen);//下一个来的人
        handle_queue[now=findmin(handle_queue,handle_num)].addpeople(detect.occurtime, ran.duration, ran.amount);//把这个人插到队伍后面
        evlist.inserfun(detect.occurtime + ran.duration, now);//要处理的人
        if (detect.occurtime < evlist.lasttime[detect.ntype]) {
            //到的时间小于lasttime
            evlist.inserfun(evlist.lasttime[detect.ntype]+ ran.duration, now);
        }
        else
        {
            evlist.inserfun(detect.occurtime-evlist.lasttime[detect.ntype] + ran.duration, now);
        }
        cout << now << "队来了人,交易额是" << ran.amount << "\n";//log
    }
```

当现在时间小于 closetime 时，循环维持。

1. 若事件类型为-1，此事件为入队事件。利用 random 函数生成下一个人到达的时间，这个人所需的时间，这个人处理的金额。将其插入到最少人的队伍。
2. 若事件类型不为-1，事件类型为 i 的时候，表明在第 i 条队.

   若类的 amount>=0 表明进行存款服务，将这个顾客进行离队。同步银行的数据。

然后对大额取款队列进行搜索，满足可以满足的人。搜索完将 totalmoney 同步到 lastmoney 。

```
else if (detect.ntype != -1)
{
    if (handle_queue[detect.ntype].howmuch() >= 0) {//存款直接过去
        totalmoney += handle_queue[detect.ntype].howmuch();
        totaltime += handle_queue[detect.ntype].time()+detect.occurtime- handle_queue[detect.ntype].getarr();
        custom++;
        cout << "队" << detect.ntype << "的人离队了，交易额是" << handle_queue[detect.ntype].howmuch() << endl;//log
        handle_queue[detect.ntype].leave();
        waitqueue.scanmode_2(custom, totaltime, totalmoney, lastmoney, detect.occurtime);//处理第二条队
        lastmoney = totalmoney;

    }
    else if (handle_queue[detect.ntype].howmuch() < 0) {//取款的情况
        if (handle_queue[detect.ntype].howmuch() * (-1) <= totalmoney)//可取
        {

            totalmoney += handle_queue[detect.ntype].howmuch();
            lastmoney = totalmoney;
            custom++;
            totaltime += handle_queue[detect.ntype].time() + detect.occurtime - handle_queue[detect.ntype].getarr();
            cout << "队" << detect.ntype << "的人离队了，交易额是" << handle_queue[detect.ntype].howmuch()<< endl;//log
            handle_queue[detect.ntype].leave();//处理完这个人

        }
        else if (handle_queue[detect.ntype].howmuch() * (-1) > totalmoney) {//取款额大于银行的钱
            //压到第二条队,且第detect.ntype条队的人occurtime全部减去detect.duration
            waitqueue.addpeople(handle_queue[detect.ntype].getarr(), handle_queue[detect.ntype].time(), handle_queue[detect.ntype].l
            cout << "队" << detect.ntype << "的人由于钱不够进入ex队,交易额是" << handle_queue[detect.ntype].howmuch() << endl;
            handle_queue[detect.ntype].leave();
            waitqueue.display(1);
        }
    }
}
```

若类的 amount<0,表明进行取款业务，若 amount+toalmoney>=0 则可以取钱，直接取出，并且离队。同步相关数据。若

amount+toalmoney<0,则不够取钱，将这个人离队，插入到 waitqueue 中，等待。

3. 结束相关操作后，再次读取事件链表回到循环

```
//处理完上面的情况，继续读取时间链表。
detect = evlist.pop();
```

4. 跳出循环后，表明

```
for (int i = 0; i < handle_num; i++) {
    handle_queue[i].clearallpeople(totaltime, custom, detect.occurtime);
}

waitqueue.clearallpeople(totaltime, custom, closetime);
```

银行已经关门，进行清场，对 handle_queue 和 waitqueue 的人清理。

清理函数为右图所示，读取每一个的到达事件，用 now 减去，即得到所消耗时间，将其加到 totaltime。并同步顾客数。

```
void customerqueue::clearallpeople(int& totaltime, int& custom, int now) {
    customer* current, * temp;
    current = temp = head;
    while (current != NULL) {
        temp = current;
        current = current->next;
        totaltime += now - temp->arrivaltime;
        custom++;
        delete temp;
    }
}
```

四．调试分析

[part1]时空复杂度分析

代码分为几个重要部分：1.事件链表。2.lenth 条队列。3.等待队列。

下面针对几种情况进行讨论：

参数 mxdur 是最大处理时间，mxint 是最大间隔时间，mxmon 是最大取钱数目,minus 是取钱的概率，lenth 是第一类窗口的数量，totalmoney 是银行总钱数。

1. 标准参数

参数为：mxdur = 30, mxint = 12, mxmon = 1000, minus_percent = 0.5，lenth=1，totalmoney=10000。此时取钱的概率是 50%，因此总钱数总体会维持在 10000 上下，故此时 waitqueue 近似可以忽略，需要进行操作的只有 eventlist 和 handlequeue。Eventlist 有 add 和 pop 操作，handlequeue 有 add 和 leave 操作。假设有 N 个人来，那么此时的时间复杂度为：O(N)。

进一步发现当 madur=O（mxint）时，lenth 对时空复杂度影响不大。

2. 参数为：mxdur = 100, mxint = 4, mxmon = 1000, minus_percent = 0.5，lenth=1，totalmoney=10000。此时表明有很多人来，而处理速度很慢。此时 waitqueue 依然可以忽略。需要进行操作的只有 eventlist 和 handlequeue。假设有 N 个人来，此时的操作复杂度主要集

中在 eventlist 的插入操作中。此时时间复杂度为 O(N^2),空间复杂度为 O(N),此时若开放新的窗口，可以大大降低 eventlist 的时空复杂度.

3. 参数为：mxdur = 4, mxint = 100, mxmon = 1000， minus_percent = 0.5，lenth=1，totalmoney=10000。此时几乎没有人来，而处理速度很快。假设有 N 个人来，此时事件链表和 queue 几乎没有堆积，时间复杂度为 O(N),空间复杂度小于 O(N)。

4. 参数为：mxdur = 30, mxint = 12, mxmon = 1000， minus_percent = 0.8，lenth=1，totalmoney=10000。此时几乎都堆积在 waitqueue 中，无法处理。而 eventlist 和 handle_queue 的人非常少。若有 N 人来，时间复杂度为 O(N^2),空间复杂度为 O(N).

进一步发现，当 minus_percent 相当大的时候，无论 mxdur，mxint，lenth 取何值（当然不能太离谱），都无法有效的降低复杂度。只有当 total money 很小的时候及时补充才能有效降低复杂度。

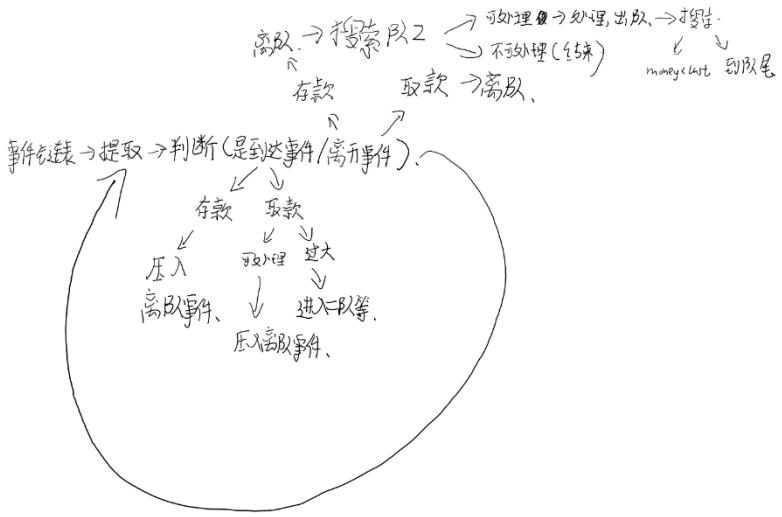
[part2]实验中遇到的问题及解决

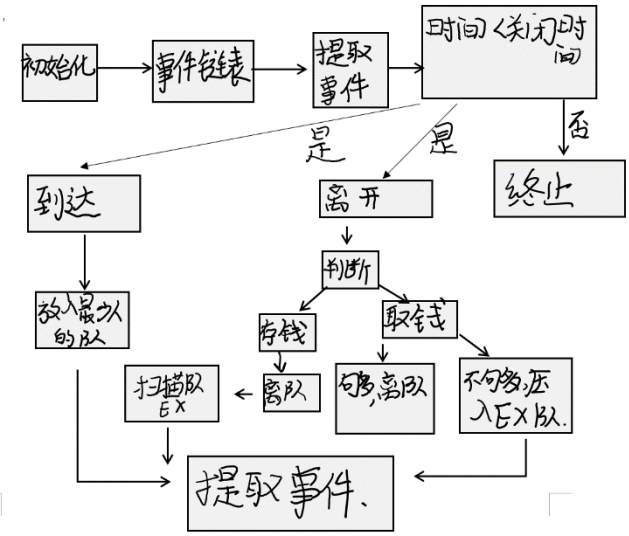
1. 大额取款的时间轴如何表示，以及不会造成乱轴。

   解决：本程序对 waitqueue 中的人搜索时，外界时间轴并未移动。搜索完以后外界时间轴才继续移动。这样，相当于是外界进行业务的时候，waitqueue 的时间轴也在移动。这一定程度上提高了处理速度。当然也有不足之处，waitqueue 取东西的时候依然使用的是上一次的 totalmoney，这可能会造成时间上的浪费。

2. 构建了错误的事件驱动模型。

一开始构建了右图所示的事件驱动模型，导致时间轴的问题一直无法解决，最后修改为下面的事件驱动模型。（检查是瞬间发生的）



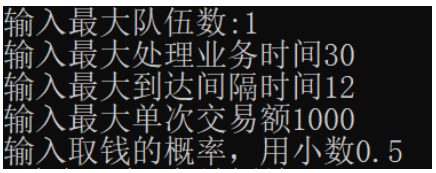


五．代码测试

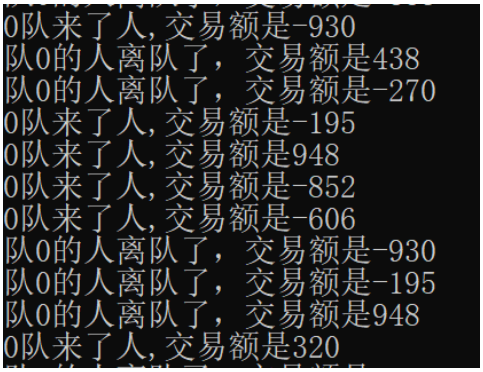
采用上面的参数进行测试。

1. mxdur = 30, mxint = 12, mxmon = 1000， minus_percent = 0.5， lenth=1，totalmoney=10000。

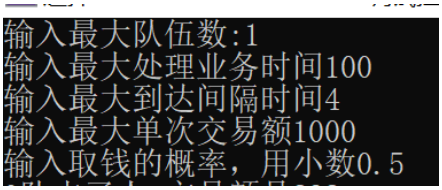
结果为

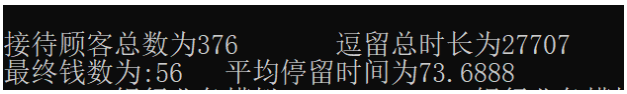


还输出了顾客的移动行为。(此处的队 0 即指队 1).



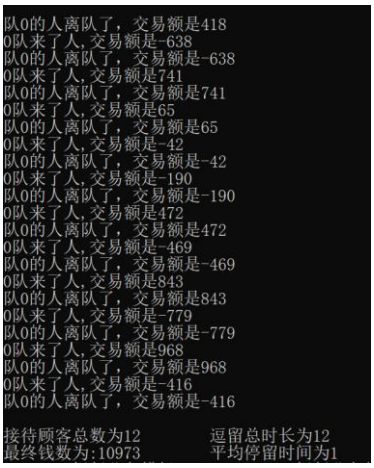
2.参数为：mxdur = 100, mxint = 4, mxmon = 1000， minus_percent = 0.5，lenth=1，totalmoney=1 0000。结果和客户行为如下图，





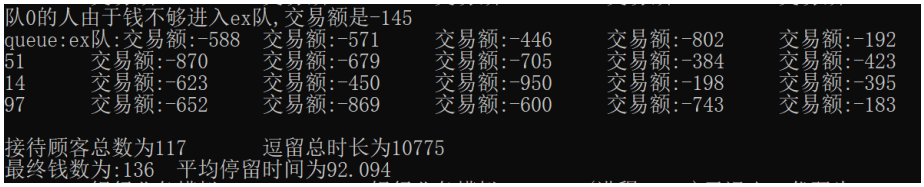
3. 参数为：mxdur = 4, mxint = 100, mxmon = 1000， minus_percent = 0.5，lenth=1，totalmoney=10000。

结果如右图



4. 参数为：mxdur = 30, mxint = 12, mxmon = 1000，minus_percent = 0.8，lenth=1，totalmoney=10000。

结果如右图



下面是对于情况 2 的优化

参数为：mxdur = 100, mxint = 4, mxmon = 1000， minus_percent = 0.5，lenth=10，

totalmoney=1 0000。

结果为，显然，效率大大提高了。

下面是对情况 4 的优化，设定最大忍耐时间是 2*maxduration，超出这个时间，waitqueue 自动离开。运行后，结果为同样大大提高了效率。

```
else if (nowtime - current->arrivaltime >= 2 * maxduration) {
        custom++;
        totaltime = (nowtime - current->arrivaltime) + totaltime;
        if (current == head) {
            this->leave();//出队
            prev = current = head;
        }
        else {
            prev->next = current->next;
            if (current == rear) {
                rear = prev;
            }
            delete current;
            current = prev->next;
            lenth--;
        }
    }*/
```

六．实验总结

经过这次大作业，我对于队列和链表的算法有了比较深的了解，并且自己编写了一个比较大的程序，提高了编程能力。更重要的是，我在做这次大作业的时候，认识到现实世界的时间可以通过计算机进行模拟，锻炼了自己的，建模思维，计算思维。

七．文件清单

1. structure.h
2. queue.h
3. eventlist.h
4. 银行业务模拟.cpp
5. 银行业务模拟.exe