实习报告：区块链大数据分析

题目：设计并实现一个区块链数据处理程序，以便对区块链中包含的大量交易记录进行分析，进而发现一些规律。

班级：2101班 姓名：吴国正 学号：2021201543 完成日期：2023.1.20

一、需求分析

1. 初始数据以文件形式存放。
2. 默认初始的数据集有标题行，插入的数据集没有标题行。
3. 文件名和功能选择均由用户从键盘输入。通过命令行的方式进行交互。
4. 以文件的形式输出，需要用户指明特定路径，否则按照默认路径输出。
5. 输出的结果是：数据查询和数据分析的结果、时间，数据初始化和数据插入的时间。
6. 测试数据：题目提供的大数据集（测试初始化和插入数据的时间和正确性），本人精简的数据集（验证数据分析和数据查询的准确性）

二、概要设计

对于数据的初始化，我们分为两个部分：a) 读取存放区块的文件。b) 读取存放交易信息的文件。程序中设计下列n个抽象数据类型。

1. 定义“链表结点“类型

```
class Node{
        数据成员：指向后继结点的指针、数据域。
        基本操作：
            Node(const Block &data, Node *next = NULL)
            初始条件：调用构造函数。
                操作结果：数据域(data)，指向后继结点的指针（next).
            insertAfter(Node *p)
                初始条件：指针p不为空。
            操作结果：在本节点之后插入一个同类节点。
            deleteAfter()
                初始条件：当前结点不是尾结点，否则返回nullptr。
                操作结果：删除本结点的后继结点，并返回地址。
            nextNode()
                初始条件：无。
                操作结果：获取后继结点的地址。
}
```

2. 定义“链表”类型

```
class LinkList{
        数据成员：表头表尾指针、当前位置和前一位置指针、元素个数、当前元素位置号。
        基本操作：
            newNode(Block item, Node *ptrNext = NULL)
                初始条件：无。
                操作结果：生成新结点。
```

freeNode(Node *p)
初始条件：p不为空指针。
操作结果：释放结点。
LinkList()
初始条件：调用构造函数。
操作结果：初始化size和position为0。
~LinkList()
初始条件：调用析构函数。
操作结果：调用clear()函数。
clear()
初始条件：无。
操作结果：清空链表。
getSize()
初始条件：无。
操作结果：返回链表中元素个数。
get_front()
初始条件：无。
操作结果：返回链表头指针。
isEmpty()
初始条件：无。
操作结果：链表是否为空。
reset(int pos = 0)
初始条件：pos非负且不超元素个数。
操作结果：初始化游标位置。
next()
初始条件：无。
操作结果：使游标移到下一个结点。
endOfList()
初始条件：无。
操作结果：游标是否到了结尾。
currentPosition()
初始条件：无。
操作结果：返回游标当前位置。
insertFront(Block &item)
初始条件：无。
操作结果：在表头插入结点。
insertRear(Block &item)
初始条件：无。
操作结果：在表尾插入结点。
insertAt(Block &item)
初始条件：无。
操作结果：在当前节点之前插入结点。
insertAfter(Block &item)
初始条件：无。
操作结果：在当前节点之后插入结点。
deleteFront()
初始条件：无。
操作结果：删除头结点。
deleteCurrent()
初始条件：无。
操作结果：删除当前结点。
data()
初始条件：无。
操作结果：返回当前结点数据引用。
getPtr()
初始条件：无。
操作结果：返回当前结点数据地址。
getRearPtr()

```
            初始条件：无。
            操作结果：返回尾结点数据地址。

}
```

3. 定义“平衡二叉树结点”类型

```
template<class K, class V> //k为key,V为value.
struct AVLTreeNode{
    数据成员：左右孩子、父亲结点、键值、平衡因子。
    基本操作：
        AVLTreeNode(const pair<K, V>& Kv)
            初始条件：调用构造函数。
            操作结果：_left(nullptr), _right(nullptr), _parent(nullptr), _Kv(Kv) , _bf(0)。

}
```

4. 定义“平衡二叉树”类型

```
class AVLTree{
    typedef AVLTreeNode<K, V> Node;
    数据成员：根结点的指针。
    基本操作：
        Destroy(Node *root)
            初始条件：根结点不为空。
            操作结果：递归释放各个结点。
        AVLTree()
            初始条件：调用构造函数。
            操作结果：将根节点置为nullptr。
         ~AVLTree()
            初始条件：无。
            操作结果：调用Destroy(Node *root),根节点置为nullptr。
        copy(Node* cp)
            初始条件：cp不为空。
            操作结果：复制构造结点，如果节点为空，返回空，否则返回新的节点。
        AVLTree(const AVLTree<K, V>& job)
            初始条件：拷贝构造函数，job地址与this不同。
            操作结果：深拷贝构造。
        operator=(AVLTree<K, V> tmp)
            初始条件：&tmp!=this。
            操作结果：将当前树置为目标树。
        operator[](const K& key)
            初始条件：无。
            操作结果：调用Insert()函数，如果树中有key结点，否则创建key结点，返回目标节点的引用。
        Insert(const pair<K, V>& kv)
            初始条件：树已存在。
            操作结果：返回目标节点的指针，并且给出是否真的插入。
        RotateRL(Node* parent)
            初始条件：无。
            操作结果：右左旋转。
        RotateLR(Node* parent)
            初始条件：无。
            操作结果：左右旋转。
        RotateL(Node* parent)
            初始条件：无。
```

```
            操作结果：左单旋。
        RotateR(Node* parent)
            初始条件：无。
            操作结果：右单旋。
        height(Node* root)
            初始条件：无。
            操作结果：求高度。
        Find(const K& key)
            初始条件：无。
            操作结果：如果以key为键值的目标节点存在，则返回节点指针，否则返回nullptr。
        isexited(const K& key)
            初始条件：无。
            操作结果：以key为键值的目标节点存在返回true，否则返回false。
            Inorder(vector<void * > &vec)
            初始条件：无。
            操作结果：调用_Inorder()。
        IsAVLTree()
            初始条件：无。
            操作结果：调用_IsAVLTree()。
         _Inorder(Node* root,vector<void *> &vec)
            初始条件：根节点不为空。
            操作结果：将递归遍历的结果存在vec中。
         _IsAVLTree(Node* root)
            初始条件：根节点不为空。
            操作结果：检查是否是平衡树。
        getRoot()
            初始条件：无。
            操作结果：返回根节点指针。
}
```

5. 定义“交易”类型

```
class Transaction{
    数据成员：交易编号、交易所属区块ID、交易转出方、交易金额、交易转入方。
    基本操作：
        Transaction()
            初始条件：调用构造函数。
            操作结果：将交易编号、交易所属区块ID、交易金额置0。
            print()
        初始条件：交易内容不为空。
            操作结果：打印本次交易信息。
}
```

6. 定义“区块”类型：

```
class Block{
    数据成员：区块编号、哈希、unix时间戳、交易树、区块交易时间。
    基本操作：
            Block()
        初始条件：调用构造函数。
        操作结果：将交易所属区块ID、时间戳、区块生成时间置0。
    print()
        初始条件：区块内容不为空。
        操作结果：打印区块内容（不包括交易）。
}
```

7. 定义“加载Block”类型

```
class LoadBlock{
        数据成员：文件名、文件流、区块数量。
        基本操作：
            LoadBlock()
                初始条件：调用默认构造函数。
            操作结果：将size置0。
                LoadBlock(string name)
                初始条件：调用构造函数。
            操作结果：filename(name),size(0)，打开指定文件。
            ~LoadBlock()
                初始条件：调用析构函数。
            操作结果：关闭文件。
            InitalLinkedlist(LinkList& l,AVLTree<int,Block*>& avltree)
                初始条件：无。
            操作结果：初始化区块链l，并以区块ID为key,区块在链表中的地址为value，建立索引树。
}
```

8. 定义“加载Transaction”类型

```
class LoadTrans{
        数据成员：文件名、文件流、交易数量、成功插入数量、失败插入数量。
        基本操作：
                string2double(string str)；
                初始条件：str是数值。
                操作结果：返回相应的double。
                LoadTrans()
                初始条件：默认构造函数。
                操作结果：构造对象。
                LoadTrans(string name)
                初始条件：构造函数。
                操作结果：构造对象。
                 ~LoadTrans()
                初始条件：析构函数。
                操作结果：关闭文件，析构对象。
        getSize()
                初始条件：无。
                操作结果：返回交易数。
            InitalLinkedlist(LinkList &l,AVLTree<int,Block*> avltree)
                初始条件：无。
                操作结果：利用AVL索引树插入交易（避免每次遍历链表，提速）。
            Inital(LinkList &l)
                初始条件：无。
                操作结果：基础方法遍历链表插入交易。
            Insert(LinkList &l,string name,AVLTree<int,Block*> avltree)
                初始条件：无。
                操作结果：插入新的交易。
}
```

9. 定义“用户”类型

```
class Account{
        数据成员：账户ID、账户余额
```

```
                        、以时间为键值的AVL树，value是一个double向量，
                            存放在这个时间戳上所有的交易金额（>0 表示是接收方，<0 表示是转出方）
                、以时间为键值的AVL树，value是这个时间戳上所有交易金额的总和。
        基本操作：
            func()
            初始条件：无。
            操作结果：保存交易树上中序遍历的结果。
            search1(long long left,long long right,int type)
            初始条件：left<=right。
            操作结果：实现2.1功能:type==0为转入和转出,< 0为转出,> 0为转入。
                                返回当前账户在指定时间段内的所有交易。
            search2(long long edge)
            初始条件：无。
            操作结果：实现2.2功能：查询某个账号在某个时刻的金额（允许有负数）。
}
```

10. 定义“用户树”类型

```
class AccountTree{
        数据成员：账户数量、以账户ID为key，Account为value的AVL树。
        基本操作：
            string2double(string str)
                初始条件：str表示数值。
                操作结果：返回相应的double。
            initalvec()
                初始条件：无
                操作结果：执行AccountTree下所有账户的func()函数。
            mycomp(pair<long long,double> a,pair<long long,double> b)
                初始条件：为static函数。
                操作结果：用于排序比较交易金额绝对值的大小。
            mycomp2(pair<string,double> a,pair<string,double> b)
                初始条件：为static函数。
                操作结果：用于比较账户余额的大小。
            AccountTree()
                初始条件：默认构造函数。
                操作结果：size=0。
            getsize()
                初始条件：无。
                操作结果：返回账户数量
            getAccount(string Id)
                初始条件：无。
                操作结果：返回ID这个Account的引用，若没有则将此账户加进去，返回引用。
            load(string filename, AVLTree<int,Block*> *blockAVLptr)
                初始条件：filename文件存在。
                操作结果：从文件filename中构建账户树。
            loadFromlinkedlist(LinkList& l)
                初始条件：区块链l存在。
                操作结果：从已有的区块链中构建账户树。
            InsertFromFile(string filename, AVLTree<int,Block*> *blockAVLptr)
                初始条件：filename文件存在。
                操作结果：从用户指定的文件中插入新的交易。
            search1(string ID,long long left,long long right,int type,int k=10)
                初始条件：ID存在。
                操作结果：查找指定账号在一个时间段内的所有转入或转出记录，返回总记录数，
                                交易金额最大的前k条记录（k 为一个正整数，由查询输入，默认为10）。
            search2(string ID,long long edge)
```

```
            初始条件：ID存在。
            操作结果：查询某个账号在某个时刻的金额（允许有负数）。
        search3(long long timepoint,int k=50)
            初始条件：时间合法。
            操作结果：在某个时刻的福布斯富豪榜!输出在该时刻最有钱的前k个用户，
                                    k默认值50，可以由用户修改k值。
        isexited(string ID)
            初始条件：ID存在。
            操作结果：查询某个账号是否存在。
        getAllAccount()
            初始条件：无。
            操作结果：按照字典序递增的顺序得到所有账户的ID。

}
```

11. 定义“图的弧”类型

```
typedef struct ArcBox{
    数据成员：弧的头尾顶点、弧头相同的链域、弧尾相同的链域、弧的权重。
}
```

12. 定义“图的顶点”类型

```
class VexNode{
    数据成员：账户ID、指向该顶点的第一条入弧、指向该顶点的第一条出弧、出度、入度。
    基本操作：
        operator==(VexNode &T)
            初始条件：T存在。
            操作结果：判断两个顶点的ID是否相等。
}
```

13. 定义“图”类型

```
class OLGraph{
#define INFINITY 99999999999.0 //表示无穷
    数据成员：表头向量、有向图当前顶点数和弧数、总入度和总出度、记录所有new的AcrBox地址的vector
                、以顶点账户ID为key和账户在表头向量的下标为value的AVL索引树、
        以交易的from+to为key和以弧指针为value的AVL索引树。
    基本操作：
        string2double(string str)
                初始条件：str代表的是一个数值。
            操作结果：将字符串转化为双精度浮点数。
        mycompout(VexNode a,VexNode b)
                初始条件：两顶点存在。
            操作结果：以出度的大小为标准对顶点进行比较。
            mycompin(VexNode a,VexNode b)
                初始条件：两顶点存在。
            操作结果：以入度的大小为标准对顶点进行比较。
            insertPoint(string ID)
                初始条件：ID不为空。
            操作结果：在图中插入ID顶点。
            insertArc(string v1,string v2,double _money)
                初始条件：v1、v2不为空，_money非负。
            操作结果：插入弧。
```

```
        Insert(vector<string> vec_id,vector<simtrans> vecArc)
            初始条件：无。
        操作结果：循环调用insertPoint()和insertArc()函数，完成多个点和弧的插入。
        ~OLGraph()
            初始条件：调用析构函数。
        操作结果：将新建的弧全部delete。
        LocateVex(string ID)
            初始条件：无。
        操作结果：定位ID顶点，不存在则返回-1。
        create(AccountTree *atree,string filename)
            初始条件：文件存在，交易树存在。
        操作结果：利用交易文件进行初始化。
    create2(AccountTree *atree,LinkList &l)
        初始条件：区块链存在，交易树存在。
        操作结果：利用已有的区块链进行图的初始化（更快）。
    InsertTransFromFile(string filename)
        初始条件：文件存在。
        操作结果：将用户新加入的交易信息插入到图中
        getVexnum()
        初始条件：无。
        操作结果：返回顶点数。
    getArcnum()
        初始条件：无。
        操作结果：返回弧数。
    av_out_degree()
        初始条件：无。
        操作结果：返回平均出度。
    av_in_degree()
        初始条件：无。
        操作结果：返回平均入度。
    arcs(int a,int b)
        初始条件：a和b代表的顶点存在。
        操作结果：求a,b两点之间是否弧a->b，并返回权重。
    getTop(int type,int k=5)
        初始条件：无。
        操作结果：type>=0出度，<0入度，显示出度 / 入度最高的前k个帐号（默认k为5）。
    TopologicalSort()
        初始条件：无。
        操作结果：判断是否有环存在。
    Short1(int start,vector<double> &dis,vector<int> &pre)
        初始条件：start在范围内。
        操作结果：未经优化的Dijkstra算法。
    Short2(int start,vector<double> &dis,vector<int> &pre)
        初始条件：start在范围内。
        操作结果：优化的Dijkstra算法。
}
```

三、详细设计

1. “用户”类型

```
class Account
{
```

```
private:
    static bool mycomp(pair<long long,double> a,pair<long long,double> b){
        double a_am=fabs(a.second);
        double b_am=fabs(b.second);
        return a_am>b_am;
    }
public:
    string ID;
    double amount;
    Account(){amount=0;};
    Account(string id):ID(id){amount=0;};
    AVLTree<long long, vector<double>> tree;//以时间为键值的AVL树，value是一个double向量，存放在这个时间戳上所有的交易金额。
    vector<void *> vec; //将tree中的Node指针拿出来
    vector< pair<long long,double>> orderedvec;//以时间为键值的AVL树，value是这个时间戳上所有交易金额的总和。
public:
    void func(){//保存交易树上中序遍历的结果。
        tree.Inorder(vec);
    }

public:
    vector< pair<long long,double> > search1(long long left,long long right,int type);//type==0为转入和转出,< 0为转出,> 0为转入。返回当前账户在指定时间段内的所有交易。
    double search2(long long edge);//查询账号在某个时刻的金额（允许有负数）
};
```

部分成员函数代码如下：

Account::search1()

```
vector<pair<long long, double> > Account::search1(long long left, long long right, int type) {
    vector<pair<long long, double> > result;
    int len = vec.size();
    for (int i = 0; i < len; i++) {// 因为一个账户在一个时间的所有交易存在vector中，这里我们将这些交易额相加，以便排序。
        pair<long long, vector<double>> *ptr;
        ptr = (pair<long long, vector<double>> *) vec[i];
        int transSizeofOnetime = ptr->second.size();//这一时间的交易数目
        for (int j = 0; j < transSizeofOnetime; j++) {
            pair<long long, double> buf(ptr->first, ptr->second[j]);
            orderedvec.push_back(buf);
        }
    }
    sort(orderedvec.begin(), orderedvec.end(), mycomp);//按交易额绝对值对交易进行排序

    len=orderedvec.size();
    for(int i=0;i<len;i++){
        if(orderedvec[i].first>=left&&orderedvec[i].first<=right){
            if(type==0){//全部转入和转出
                result.push_back(orderedvec[i]);
            }
            if(type<0&&orderedvec[i].second<0){//转出
                result.push_back(orderedvec[i]);
            }
            if(type>0&&orderedvec[i].second>0){//转入
                result.push_back(orderedvec[i]);
            }
        }
```

```
    }
    return result;
}
```

Account::search2()

```
double Account::search2(long long edge) {
    double result = 0.0;
    int len = vec.size();
    for (int i = 0; i < len; i++) {
        pair<long long, vector<double>> *tmp = ((pair<long long, vector<double>> *) vec[i]);
        if (tmp->first <= edge) {//指定时间edge之前
            for (int j = 0; j < tmp->second.size(); j++) {
                result += tmp->second[j];
            }
        }
    }
    return result;
}
```

2. 用户树类型

```
class AccountTree
{
private:
int size; //账户数量
public:
AVLTree<string,Account> IDtree;
private:
double string2double(string str) {
    stringstream ss;
    ss << str;
    double result;
    ss >> result;
    return result;
}
void initalvec(){//对树中每个账户进行func操作
    vector<void *> vecc;
    IDtree.Inorder(vecc);
    int len=vecc.size();
    for(int i=0;i<len;i++){
        ((pair<string,Account>*)vecc[i])->second.func();
    }
}
static bool mycomp(pair<long long,double> a,pair<long long,double> b){
    double a_am=fabs(a.second);
    double b_am=fabs(b.second);
    return a_am>b_am;
}
static bool mycomp2(pair<string,double> a,pair<string,double> b){return a.second>b.second;}
public:
AccountTree(){size=0;}
int getsize(){return size;}
Account& getAccount(string Id){return IDtree[Id];};
void load(string filename, AVLTree<int,Block*> *blockAVLptr);//为了更容易获取账户某笔交易发生的时间，我们利用在加载时生成的AVL<BlockID,Block *> 来加快访问
```

```
void loadFromlinkedlist(LinkList& l);//从已有的区块链中加载数据
void InsertFromFile(string filename, AVLTree<int,Block*> *blockAVLptr);//从用户指定的文件中插入新的交易
vector< pair<long long,double>> search1(string ID,long long left,long long right,int type,int k=10);
double search2(string ID,long long edge);
vector< pair<string,double>> search3(long long timepoint,int k=50);
bool isexited(string ID){//查询某个账号是否存在
    if(IDtree.Find(ID)== nullptr)
        return false;
    else
        return true;
}
vector<string> getAllAccount(){//返回所有账户ID
    vector<string> result;
    vector<void *> inorder_vec;
    IDtree.Inorder(inorder_vec);
    for (int i = 0; i < inorder_vec.size(); i++) {
        pair<string, Account> *ptr;
        ptr = (pair<string, Account> *) inorder_vec[i];
        result.push_back(ptr->first);
    }
    return result;
}
};
```

部分成员函数代码如下：

AccountTree::loadFromlinkedlist()

```
void AccountTree::loadFromlinkedlist(LinkList& l)//从已有的区块链中加载数据
{
    double _amount;
    int Blockid;
    l.reset();//将区块链复位
    while(!l.endOfList())//遍历区块链，直到末尾
        {
            Block b;
            b=l.data();
            vector<void *> result;
            b.transTree.Inorder(result);
            for(int i=0;i<result.size();i++) {
                pair<long long, Transaction> *ptr;
                ptr = (pair<long long, Transaction> *) result[i];
                _amount=ptr->second.amount;
                Blockid=ptr->second.blockID;
                if (!isexited(ptr->second.from))//如果转出方是新的账户，则插入
                    size++;
                if (!isexited(ptr->second.to))//如果转入方是新的账户，则插入
                    size++;
                Account tmp_acc;
                tmp_acc=IDtree[ptr->second.from];
                tmp_acc.tree[b.block_timestamp].push_back(-_amount);
                tmp_acc.ID=ptr->second.from;
                tmp_acc.amount+=(-_amount);

                tmp_acc=IDtree[ptr->second.to];
                tmp_acc.tree[b.block_timestamp].push_back(_amount);
                tmp_acc.ID=ptr->second.to;
```

```
                tmp_acc.amount+=(_amount);
            }
            l.next();//移动区块链指针
        }
    initalvec();//加载好后对树中各个账户进行func操作。
}
```

AccountTree::InsertFromFile()

```
void AccountTree::InsertFromFile(string filename, AVLTree<int,Block*> *blockAVLptr) {//从用户指定的文件中插入新的交易
    ifstream file;
    file.open(filename);
    if (!file.is_open()) {
        cout << "InsertTransactions open error\n";
        exit(0);
    }
    string tmp;
    string from, to;
    double _amount;
    int Blockid;
    while (file.peek() != EOF) {
        getline(file, tmp, ',');
        getline(file, tmp, ',');
        Blockid = atoi(tmp.c_str());
        getline(file, from, ',');
        getline(file, tmp, ',');
        _amount = string2double(tmp);
        getline(file, to);

        if (!isexited(from))
            size++;
        if (!isexited(to))
            size++;

        Account tmp_acc;
        tmp_acc=IDtree[from];
        tmp_acc.tree[(blockAVLptr->Find(Blockid)->_Kv.second)->block_timestamp].push_back(-_amount);
        tmp_acc.ID=from;
        tmp_acc.amount+=(-_amount);

        tmp_acc=IDtree[to];
        tmp_acc.tree[(blockAVLptr->Find(Blockid)->_Kv.second)->block_timestamp].push_back(_amount);
        tmp_acc.ID=to;
        tmp_acc.amount+=(_amount);

    }
    file.close();
    initalvec();
}
```

AccountTree::search1()

```
vector<pair<long long, double> > AccountTree::search1(string ID, long long left, long long right, int type, int k) {
    vector<pair<long long, double> > tmp;
    vector<pair<long long, double> > result;
```

```
    tmp = IDtree.Find(ID)->_Kv.second.search1(left, right, type);//根据账户ID调用对应账户的search1.
    int tmp_size = tmp.size();
    if (k <= tmp_size) {//如果外部输入的k<=tmp_size
        for (int i = 0; i < k; i++) {// 取出最大的前k个
            result.push_back(tmp[i]);
        }
    } else {//外部的k大于tmp_size
        for (int i = 0; i < tmp_size; i++) {
            result.push_back(tmp[i]);
        }
    }
    return result;
}
```

AccountTree::search2()

```
double AccountTree::search2(std::string ID, long long edge) {
    return IDtree[ID].search2(edge);//调用对应ID的search2
}
```

AccountTree::search3()

```
vector<pair<string, double>> AccountTree::search3(long long timepoint, int k) {
    vector<pair<string, double>> tmp_result, result;//分别为无序的和有序的所有账户的值
    vector<void *> inorder_vec;//中序遍历结果
    IDtree.Inorder(inorder_vec);
    for (int i = 0; i < inorder_vec.size(); i++) {
        pair<string, Account> *ptr;
        ptr = (pair<string, Account> *) inorder_vec[i];
        double tmp = ptr->second.search2(timepoint);
        pair<string, double> buf(ptr->first, tmp);
        tmp_result.push_back(buf);
    }
    int llen = tmp_result.size();//所有
    sort(tmp_result.begin(), tmp_result.end(), mycomp2);
    int len = k < llen ? k : llen;
    for (int i = 0; i < len; i++) {
        result.push_back(tmp_result[i]);
    }
    return result;
}
```

2. 图类型

```
class OLGraph
{
#define INFINITY 99999999999.0
private:
vector<VexNode > xlist;//表头向量
int vexnum,arcnum;//有向图的当前顶点数和弧数
double All_outdegree,All_indegree;
vector<ArcBox *> trash_can;//将所有new的ArcBox地址记录下来，方便删除
AVLTree<string,int> indextree;//用于查找ID在xlist中的下标
AVLTree<string ,ArcBox *> Arctree; //用于检查是否重复
private:
```

```
double string2double(string str) {
    stringstream ss;
    ss << str;
    double result;
    ss >> result;
    return result;
}
static bool mycompout(VexNode a,VexNode b){//以出度的大小为标准对顶点进行比较。
    return a.out_degree>b.out_degree;
}
static bool mycompin(VexNode a,VexNode b){//以入度的大小为标准对顶点进行比较。
    return a.in_degree>b.in_degree;
}

bool insertPoint(string ID);//插入ID点,true表示成功插入，false表示已有ID点
bool insertArc(string v1,string v2,double _money);//插入弧
void Insert(vector<string> vec_id,vector<simtrans> vecArc);
public:
OLGraph(){
    vexnum=arcnum=0;
    All_outdegree=All_indegree=0;
}
~OLGraph(){
    int len=trash_can.size();
    for(int i=0;i<len;i++){
        delete trash_can[i];
    }
}
int LocateVex(string ID){//定位ID顶点，不存在则返回-1
    if(indextree.isexited(ID)){
        return indextree[ID];
    }
    else{
        return -1;
    }
}
void create(AccountTree *atree,string filename);//利用交易文件进行初始化
void create2(AccountTree *atree,LinkList &l);//利用已有的区块链进行图的初始化
void InsertTransFromFile(string filename);//将用户新加入的交易信息插入到图中
int getVexnum(){return vexnum;}
int getArcnum(){return arcnum;}
double av_out_degree(){
    return All_outdegree/(double)xlist.size();
}
double av_in_degree(){
    return All_indegree/(double)xlist.size();
}
vector<pair<string,int>> getTop(int type,int k=5);//type>=0出度，<0入度，显示出度 / 入度最高的前k个帐号
bool TopologicalSort();
double arcs(int a,int b);//求a,b两点之间是否弧a->b，并返回权重。
void Short1(int start,vector<double> &dis,vector<int> &pre);//未经优化的Dijkstra算法
void Short2(int start,vector<double> &dis,vector<int> &pre);
};
```

部分成员函数代码如下：

OLGraph::create2()

```
void OLGraph::create2(AccountTree *atree, LinkList &l)//利用已有的区块链进行图的初始化
{
    vector<simtrans> vecArc;
    string tmp,from,to;
    double mon;
    l.reset();//将区块链游标置于首位
    while(!l.endOfList())
        {
            Block b;
            b=l.data();
            vector<void *> result;
            b.transTree.lnorder(result);
            for(int i=0;i<result.size();i++) {
                pair<long long, Transaction> *ptr;
                ptr = (pair<long long, Transaction> *) result[i];
                from=ptr->second.from;
                to=ptr->second.to;
                mon=ptr->second.amount;
                simtrans sim(from,to,mon);
                vecArc.push_back(sim);
            }
            l.next();
        }
    Insert(atree->getAllAccount(),vecArc);
}
```

OLGraph::InserTransFromFile()

```
void OLGraph::InsertTransFromFile(string filename)//将用户新加入的交易信息插入到图中
{
    vector<simtrans> vecArc;
    ifstream file;
    file.open(filename);
    if(!file.is_open()){
        cout<<"Diagraph: trans.csv open fail\n";
        exit(0);
    }
    string tmp,from,to;
    double mon;
    while(file.peek()!=EOF){
        getline(file,tmp,',');
        getline(file,tmp,',');
        getline(file,from,',');
        getline(file,tmp,',');
        mon= string2double(tmp);
        getline(file,to);
        simtrans sim(from,to,mon);
        vecArc.push_back(sim);
    }
    int _arcnum=vecArc.size();
    //插入弧
    for(int i=0;i<_arcnum;i++){
        insertArc(vecArc[i].from,vecArc[i].to,vecArc[i].moneyy);
    }
```

```
    arcnum+=_arcnum;
    file.close();
}
```

OLGraph::getTop()

```
vector<pair<string,int>> OLGraph::getTop(int type,int k){//type>=0出度，<0入度
    vector<VexNode> tmp(xlist);
    vector<pair<string,int>> result;
    if(type<0){//入度
        sort(tmp.begin(),tmp.end(), mycompin);
        if(k<xlist.size()) {
            for (int i = 0; i < k; i++) {
                pair<string, int> p(tmp[i].data, tmp[i].in_degree);
                result.push_back(p);
            }
        }
        else{
            for(int i=0;i<xlist.size();i++) {
                pair<string, int> p(tmp[i].data, tmp[i].in_degree);
                result.push_back(p);
            }
        }
    }
    else {//出度
        sort(tmp.begin(), tmp.end(), mycompout);
        if (k < xlist.size()) {
            for (int i = 0; i < k; i++) {
                pair<string, int> p(tmp[i].data, tmp[i].out_degree);
                result.push_back(p);
            }
        }
        else {
            for (int i = 0; i < xlist.size(); i++) {
                pair<string, int> p(tmp[i].data, tmp[i].out_degree);
                result.push_back(p);
            }
        }
    }
    return result;
}
```

OLGraph::TopologicalSort()

```
bool OLGraph::TopologicalSort(){
    vector<VexNode> xxlist(xlist);
    ArcBox * p;
    stack<int> S;
    for(int i=0;i<vexnum;i++){
        if(xxlist[i].in_degree==0)
            S.push(i);
    }
    int count=0;
    while(!S.empty()){
        int tmp=S.top();
        S.pop();
```

```
        count++;
        for(p=xxlist[tmp].firstout;p;p=p->tlink){
            int k=p->headvex;
            if(!(--xxlist[k].in_degree))//k点入度减1后是否为0
                S.push(k);
        }//for
    }//while
    if(count<vexnum)return false;//有回路
    else
        return true;
}
```

OLGraph::Short1()

```
void OLGraph::Short1(int start,vector<double> &dis,vector<int> &pre)
{
    bool visit[vexnum];//用来标记点是否被当做基本点
    for(int i=0;i<vexnum;i++){
        dis.push_back(arcs(start,i));//求各个点到目标点距离
        visit[i]=false;
        pre.push_back(0xfffff);//初始化每个点的大小为一个大数
    }
    visit[start] = true;//将起始点本身初始化
    dis[start] = 0;//第一次直接把起点标记为基点  起点到起点的距离为0
    pre[start] = start;//起点的前一个点设置为本身
    int i, j;
    int tempv = start;//tempv来存当前这轮迭代的基点
    for (i = 1; i < vexnum; i++)//n-1次循环
        {
            for (j = 0; j < vexnum; j++)//暴力搜索图中所有的点
                {
                    if (!visit[j] && dis[tempv] + arcs(tempv,j) < dis[j])
                        //如果j没被选为基点过 并且 基点到起点的距离加基点到j点边的距离
                        //小于j点目前到起点的距离，那么就更新
                    {
                        dis[j] = dis[tempv] + arcs(tempv,j);
                        pre[j] = tempv;
                        //更新dis数组和前驱数组
                    }
                }
            int temp =INFINITY;//找下一次迭代的新基点，就是选没当过基点，并且距离
            //起点距离最小的点
            for (j = 0; j < vexnum; j++)//暴力搜索每一个点
                {
                    if (!visit[j])//如果没当过基点
                    {
                        if (dis[j] < temp)//不断找距离起点最小点
                        {
                            temp = dis[j];
                            tempv = j;//更新temp值并记录这个点的下标
                        }
                    }
                }
            visit[tempv] = true;//OK这个tempv点是目前离起点最近的并且没当过基点的点
            //标记成true
        }
}
```

OLGraph::Short2()

```
void OLGraph::Short2(int start,vector<double> &dis,vector<int> &pre)
{
    struct node{
    int p,w;
    node(int a,int b):p(a),w(b){}
    bool operator< (const node& b) const
    {
        return w>b.w;
    }
};
    priority_queue<node>*sup=new priority_queue<node>;
    bool visit[vexnum];
    for(int i=0;i<vexnum;i++){
        visit[i]=false;
        dis.push_back(99999999);
        pre.push_back(start);
    }
    dis[start]=0;
    pre[start]=start;
    sup->push(node(start,0));
    while(!sup->empty())
        {
            node front=sup->top();
            sup->pop();
            int tempv=front.p;
            if(visit[tempv]){//是否没有当过基本点
                continue;
            }
            visit[tempv]=true;
            ArcBox *pp;
            for(pp=xlist[tempv].firstout;pp;pp=pp->tlink)//利用十字链表的存储结构可以避免暴力搜索出弧的头结点
                {
                    int p=pp->headvex;
                    if(!visit[p]&&(dis[tempv]+pp->money<dis[p]))//更新距离
                    {
                        dis[p]=dis[tempv]+pp->money;
                        pre[p]=tempv;
                        sup->push(node(p,dis[p]));
                    }
                }
        }
    delete sup;
}
```

四、调试分析

1. 算法优化

a. 加载交易

这里是将磁盘读入的transaction直接插入到区块链l相应的区块上。但是这里有一个操作是根据BlockID来找到区块中对应的区块，目前想到了两种方法：1. 基础方法：遍历链表查找时间复杂度为O(n^2)。2. 建立一个AVL树索引，key为BlockID，value为这个block在区块链中的地址。时间复杂度为O(n*$\log_2$n)。下图为实验中对两种办法的对比，与理论一致。可以发现速度得到了明显的提升。

```
E:\Clionfile\cmake-build-debug\Clionfile.exe
l.size: 2129
lt.fail: 0
lt.success: 1048575
lt.size: 1048575
15.387s

进程已结束,退出代码0
```

```
E:\Clionfile\cmake-build-debug\Clionfile.exe
l.size: 2129
lt.fail: 0
lt.success: 1048575
lt.size: 1048575
4.113s
```

b. 有向图的存储结构选择比较

共有账户877,501个。如果采取邻接矩阵存取，利用率为1/877501，不太适合。用邻接表的话，需要额外建立逆邻接表，所以我们直接选择十字链表。

c. LocateVex的实现问题

但在实现中遇到了一个问题，就是LocateVex(v)函数，这个函数是用来确定顶点v在表头向量xlist中的位置。我最初的实现如下，但在运行过程中发现运行时间太长，甚至于无法得出结果。通过输出各部分运行时间调试，我发现是oldLocateVex()的问题，我使用了std::find()函数来实现查找功能，但忽略了他的时间复杂度是O(n),这对于百万级别的数据来说简直是灾难，所以我建立了一棵AVL树，key为ID，value为在xlist中的下标，这样时间复杂度就变成了O($\log_2$n)，这样就可以轻松应对百万级的数据了，改进后的图初始化只用了不到6s，性能得到了极大的提升。

```
int oldLocateVex(string ID){//定位ID顶点，不存在则返回-1
        //cout<<"E: "<<ID<<endl;
        VexNode tmp={ID, nullptr, nullptr};
        vector<VexNode>::iterator iter=std::find(xlist.begin(),xlist.end(),tmp);
        int flag=iter-xlist.begin();
        if(flag==xlist.size())//没有这个点
            return -1;
        else
            return flag;
    }
```

```
Load account: 5.895s
```

d. 求有向图的最短路径问题

- 我一开始想到了书上所学的Dijkstra算法，并将邻接矩阵改进成为十字链表，但最后效果并不好，究其原因，是因为时间复杂度为$O(n^2)$，对于百万级的数据处理时间是无法想象的。
- 接下来要做的就是想办法对已有的基础算法进行优化，我们可以发现，书上基础的Dijkstra算法的时间复杂度由三部分控制，首先是对空路径的初始化，这里是两个for循环（图1）。这个问题可以通过改变路径的记录方式来优化。其次，是找离当前所知v0点最近的v点，这里形成了for循环嵌（图2）。受到我们课内所学的堆排序的启发，我们可以利用小根堆来将所有已知和v0点的距离的点放进堆里，这样我们每次就可以直接取出后最小的v点，避免了暴力搜索。最后，是更新当前最短路径及距离。在原算法中（图3），我们采取了暴力搜索的方法，对图中所有的顶点都进行了检查，但经过思考我们可以发现，只需要检查我们上一步找到的v点的出弧的弧头就可以了，只有这些点才有可能需要更新，结合我们十字链表存储方式的便捷性，可以方便地直接找到这些弧头，避免了暴力搜索。综上，我们可以大致将计算出改进后算法的时间复杂度为$O(e*\log_2 e)$，其中e为图中的弧个数。
- 在这里有一个问题需要注意，涉及到程序的内存管理。对于一些数据量很大的局部变量，我们应该尽量使用堆空间，避免因为栈溢出而导致的程序崩溃。

算法 7.15 为用 C 语言描述的迪杰斯特拉算法。

```
void ShortestPath_DIJ( MGraph G, int v0, PathMatrix &P, ShortPathTable &D ) {
  // 用 Dijkstra 算法求有向网 G 的 v0 顶点到其余顶点 v 的最短路径 P[v]及其带权长度 D[v].
  // 若 P[v][w]为 TRUE,则 w 是从 v0 到 v 当前求得最短路径上的顶点。
  // final[v]为 TRUE 当且仅当 v∈S,即已经求得从 v0 到 v 的最短路径。
  for (v = 0; v<G.vexnum; ++v) {
    final[v] = FALSE;  D[v] = G.arcs[v0][v];
    for (w = 0; w<G.vexnum; ++w)  P[v][w] = FALSE;     // 设空路径
    if (D[v] < INFINITY) {P[v][v0] = TRUE;  P[v][v] = TRUE; }
  }// for
  D[v0] = 0;  final[v0] = TRUE;                  // 初始化,v0 顶点属于 S 集
  //开始主循环,每次求得 v0 到某个 v 顶点的最短路径,并加 v 到 S 集
  for (i = 1; i<G.vexnum; ++i) {                 // 其余 G.vexnum - 1 个顶点
    min = INFINITY;                              // 当前所知离 v0 顶点的最近距离
    for (w = 0; w<G.vexnum; ++w)
      if (!final[w])                             // w 顶点在 V - S 中
        if (D[w]<min) {v = w;  min = D[w]; }     // w 顶点离 v0 顶点更近
    final[v] = TRUE;                             // 离 v0 顶点最近的 v 加入 S 集
    for (w = 0; w<G.vexnum; ++w)  // 更新当前最短路径及距离
      if (!final[w] && (min + G.arcs[v][w]<D[w])) {  // 修改 D[w]和 P[w], w∈V - S
        D[w] = min + G.arcs[v][w];
        P[w] = P[v];  P[w][w] = TRUE;  // P[w] = P[v] + [w]
      }// if
  }// for
}// ShortestPath_DIJ
```

算法 7.15 为用 C 语言描述的迪杰斯特拉算法。

```
void ShortestPath_DIJ( MGraph G, int v0, PathMatrix &P, ShortPathTable &D ) {
  // 用 Dijkstra 算法求有向网 G 的 v0 顶点到其余顶点 v 的最短路径 P[v]及其带权长度 D[v]。
  // 若 P[v][w]为 TRUE,则 w 是从 v0 到 v 当前求得最短路径上的顶点。
  // final[v]为 TRUE 当且仅当 v∈S,即已经求得从 v0 到 v 的最短路径。
  for (v = 0; v<G.vexnum; ++v) {
    final[v] = FALSE;  D[v] = G.arcs[v0][v];
    for (w = 0; w<G.vexnum; ++w)  P[v][w] = FALSE;     // 设空路径
    if (D[v] < INFINITY) {P[v][v0] = TRUE;  P[v][v] = TRUE; }
  }// for
  D[v0] = 0;  final[v0] = TRUE;                        // 初始化,v0 顶点属于 S 集
  //开始主循环,每次求得 v0 到某个 v 顶点的最短路径,并加 v 到 S 集
  for (i = 1; i<G.vexnum; ++i) {                       // 其余 G.vexnum-1 个顶点
    min = INFINITY;                                    // 当前所知离 v0 顶点的最近距离
    for (w = 0; w<G.vexnum; ++w)
      if (!final[w])                                   // w 顶点在 V-S 中
        if (D[w]<min) {v = w;  min = D[w]; }           // w 顶点离 v0 顶点更近
    final[v] = TRUE;                                   // 离 v0 顶点最近的 v 加入 S 集
    for (w = 0; w<G.vexnum; ++w)  // 更新当前最短路径及距离
      if (!final[w] && (min+G.arcs[v][w]<D[w])) {  // 修改 D[w]和 P[w], w∈V-S
        D[w] = min + G.arcs[v][w];
        P[w] = P[v];  P[w][w] = TRUE;  // P[w] = P[v]+[w]
      }// if
  }// for
}// ShortestPath_DIJ
```

算法 7.15 为用 C 语言描述的迪杰斯特拉算法。

```
void ShortestPath_DIJ( MGraph G, int v0, PathMatrix &P, ShortPathTable &D ) {
  // 用 Dijkstra 算法求有向网 G 的 v0 顶点到其余顶点 v 的最短路径 P[v]及其带权长度 D[v]。
  // 若 P[v][w]为 TRUE,则 w 是从 v0 到 v 当前求得最短路径上的顶点。
  // final[v]为 TRUE 当且仅当 v∈S,即已经求得从 v0 到 v 的最短路径。
  for (v = 0; v<G.vexnum; ++v) {
    final[v] = FALSE;  D[v] = G.arcs[v0][v];
    for (w = 0; w<G.vexnum; ++w)  P[v][w] = FALSE;     // 设空路径
    if (D[v] < INFINITY) {P[v][v0] = TRUE;  P[v][v] = TRUE; }
  }// for
  D[v0] = 0;  final[v0] = TRUE;                    // 初始化,v0 顶点属于 S 集
  //开始主循环,每次求得 v0 到某个 v 顶点的最短路径,并加 v 到 S 集
  for (i = 1; i<G.vexnum; ++i) {                   // 其余 G.vexnum-1 个顶点
    min = INFINITY;                                // 当前所知离 v0 顶点的最近距离
    for (w = 0; w<G.vexnum; ++w)
      if (!final[w])                               // w 顶点在 V-S 中
        if (D[w]<min) {v = w;  min = D[w]; }       // w 顶点离 v0 顶点更近
    final[v] = TRUE;                               // 离 v0 顶点最近的 v 加入 S 集
    for (w = 0; w<G.vexnum; ++w)  // 更新当前最短路径及距离
      if (!final[w] && (min+G.arcs[v][w]<D[w])) {  // 修改 D[w]和 P[w], w∈V-S
        D[w] = min + G.arcs[v][w];
        P[w] = P[v];  P[w][w] = TRUE;  // P[w] = P[v]+[w]
      }// if
  }// for
}// ShortestPath_DIJ
```

e. 求福布斯富豪榜

其实可以利用平衡二叉树的特性，以金额绝对值为键值，直接选取前k个。避免二次排序，或者利用堆排序。

f. 窗口的内存泄漏问题

由于在使用过程中会打开多个子窗口或者对话框，我们必须考虑内存泄漏问题，经过设置关闭窗口之后就释放内存后，解决了这一问题。

2. 经验体会

a. 库中的函数不一定就是最好的，我们一定要明白其中的原理，做到知其然知其所以然。

b. 我们要尽量避免重复地读取磁盘，这会严重影响速度。

c. 书上的知识不是绝对的，我们要大胆去思考算法中可以改进的地方。

d. 这个实验不仅使我全面深刻地复习了这学期所学的大部分内容，更重要的是新学了很多东西，例如Git、堆等，极大地提升了我的自学能力。

3. 主要算法的复杂度分析

a. 最短路径问题

点数（n），边数（m）

优化前时间复杂度是O(n^2)。优化后时间复杂度是O(e*$\log_2$e)，空间复杂度是O(e).

b. 拓补排序

时间复杂度是O(n^2)。

c. 排序算法

时间复杂度是O(n*$\log_2$n)。

d. AVL树查找（插入）

时间复杂度是O($\log_2$n)。

五、附录

源程序文件名清单：

LinkedList.h //链表类头文件

LinkedList.cpp //链表类源文件

AVL.h //AVL树头文件

Struct.h //基本抽象数据类型（BLock,Transaction)头文件

Account.h //抽象数据类型（账户Account，账户树AccountTree)头文件

Account.cpp //抽象数据类型（账户Account，账户树AccountTree)源文件

LoadBlock.h //抽象数据类型（加载区块）头文件

LoadBlock.cpp //抽象数据类型（加载区块）源文件

LoadTrans.h //抽象数据类型（加载交易）头文件

LoadTrans.cpp //抽象数据类型（加载交易）源文件

Diagraph.h //图头文件

Diagraph.cpp //图源文件

Handle.h //将以上接口进行后端打包，作为与QT界面程序连接的接口。

Handle.cpp //源文件

main.cpp //主函数（程序）