一、调试成功程序及说明

1、

题目：顺序结构与链式结构实现线性表基本操作

算法思想：见代码注释

附源程序。

顺序结构：

#include<iostream>

#define TYPE int

#define MAXSIZE 1000

**using** **namespace** std;

**class** SqList

{

**public**:

//1.数组

TYPE \* v;

//2.数组元素个数

**int** length;

};

//初始化函数

**void** InitSqList(SqList &L)

{

L.v=**new** TYPE[MAXSIZE];

L.length=0;

**if**(!(L.v))

{

cout<<"初始化失败"<<endl;

**return**;

}

cout<<"初始化成功"<<endl;

**return**;

}

//销毁函数

**void** DestroySqList(SqList &L)

{

**if**(L.v!=**NULL**)

{

**delete** L.v;

L.v=**NULL**;

cout<<"线性表销毁成功"<<endl;

}

**else**

{

cout<<"线性表不存在 无法销毁"<<endl;

}

}

//清空函数

**void** ClearSqList(SqList &L)

{

L.length=0;

{

cout<<"线性表清空成功"<<endl;

}

}

//求线性表长度函数

**int** SqListLength(SqList &L)

{

**return** L.length;

}

//判断线性表是否为空

**bool** SqListEmpty(SqList &L)

{

**if**(!L.length)

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

//按位置取值函数 num表示位置 e表示接收数组值的变量

**void** GetElem(SqList &L,**int** num,TYPE &e)

{

**if**(num<1||num>L.length+1)

{

cout<<"您的下标输入有误"<<endl;

**return**;

}

**else**

{

e=L.v[num-1];

}

}

//查找对应元素 若找到返回位置，若未找到返回0

**int** LocateElem(SqList &L,TYPE e)

{

**for**(**int** i=0;i<L.length;i++)

{

**if**(L.v[i]==e)

{

**return** i+1;

}

}

cout<<"未找到对应元素 返回0"<<endl;

**return** 0;

}

//查找前驱动元素函数 pre\_e表示接受前驱值的变量

**void** PriorElem(SqList &L,TYPE cur\_e,TYPE &pre\_e)

{

**int** index=-1;

**for**(**int** i=0;i<L.length;i++)

{

**if**(cur\_e==L.v[i]&&i!=0)

{

index=i;

}

}

**if**(index==-1)

{

cout<<"操作失败"<<endl;

**return**;

}

**else**

{

pre\_e=L.v[index-1];

}

}

//查找后继元素函数 next\_e表示接受后继元素的变量

**void** NextElem(SqList &L,TYPE cur\_e,TYPE &next\_e)

{

**int** index=-1;

**for**(**int** i=0;i<L.length;i++)

{

**if**(cur\_e==L.v[i]&&i!=L.length-1)

{

index=i;

}

}

**if**(index==-1)

{

cout<<"操作失败"<<endl;

**return**;

}

**else**

{

next\_e=L.v[index+1];

}

}

//遍历输出函数

**void** SqListTraverse(SqList &L)

{

**for**(**int** i=0;i<L.length;i++)

{

cout<<L.v[i]<<" ";

}

cout<<endl;

}

//按位置替换元素函数 num表示位置 e用来接收旧值

**void** SetElem(SqList &L,**int** num,TYPE &e)

{

**if**(num<1||num>L.length+1)

{

cout<<"输入有误"<<endl;

**return**;

}

**else**

{

**int** oldvalue=L.v[num-1];

L.v[num-1]=e;

e=oldvalue;

}

}

//插入元素函数 num表示想要插入的位置 e表示想要插入的元素

**void** InsertElem(SqList &L,**int** num,TYPE e)

{

**if**(L.length==MAXSIZE)

{

cout<<"存储空间已满 无法插入"<<endl;

}

**else** **if**(num<1||num>L.length+1)

{

cout<<"输入有误"<<endl;

**return**;

}

**else** **if**(num==L.length+1)

{

L.v[L.length]=e;

L.length++;

}

**else**

{

**for**(**int** i=L.length-1;i>=num-1;i--)

{

L.v[i+1]=L.v[i];

}

L.v[num-1]=e;

L.length++;

}

}

//按位置删除元素函数 num表示位置 e表示接受删除值的变量

**void** DeleteElem(SqList &L,**int** num,TYPE &e)

{

**if**(num<1||num>L.length)

{

cout<<"输入有误"<<endl;

**return**;

}

**else** **if**(num==L.length)

{

L.length--;

}

**else**

{

**for**(**int** i=num;i<=L.length-1;i++)

{

L.v[i-1]=L.v[i];

}

L.length--;

}

}

**int** main()

{

SqList L;

**return** 0;

}

链式结构：

#include<iostream>

#define TYPE int

**using** **namespace** std;

//声明节点对象

**class** LinkNode

{

**public**:

//数据域 存储类型自定

TYPE data;

//指针域 指向下一个节点 因此为LinkNode类型的指针

LinkNode \* next;

};

//初始化函数 传入链表指针

**void** InitLinkList(LinkNode\* &L)

{

//申请头节点空间 申请时可以只写类型名 不用在后面附具体值

L=**new** LinkNode;

//头节点指针暂时置空

L->next=**NULL**;

//L指向不为空时 可认为初始化成功

**if**(!L)

{

cout<<"初始化失败"<<endl;

**return**;

}

**else**

{

cout<<"初始化成功"<<endl;

**return**;

}

}

//判断链表是否为空 判断方法是看头节点的指针域是否为空

**bool** LinkListEmpty(LinkNode\* &L)

{

**if**(L->next==**NULL**)

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

//销毁单链表 算法：从头节点开始逐个释放节点内存并后移链表指针 直至链表指针为空

**void** DestroyLinkList(LinkNode\* &L)

{

//声明辅助指针 用来指向被销毁的节点

LinkNode \*p;

//当L尚未移到末节点的指针域即NULL时 执行释放循环

**while**(L!=**NULL**)

{

//1.将销毁辅助指针指向当前头节点（即当前L指向的节点）

p=L;

//2.先将L指向下一节点再进行销毁 否则将丢失L当前所指节点的信息 无法寻找之后的节点

L=L->next;

//注意这个操作 用于让当前指针指向下一个节点 非常重要

//3.L指向下一节点后，可以将p当前所指的节点内容释放

**delete** p;

//然后往复循环 最终L将末节点的指针域即NULL 释放最后一个节点后 循环结束

}

**if**(L==**NULL**)

{

cout<<"链表销毁成功"<<endl;

}

}

//清空链表函数 释放所有数据节点 保留链表指针和头节点

//算法：释放所有数据节点 头节点指针域置空

**void** ClearLinkList(LinkNode\* &L)

{

//声明两个辅助指针 p指向当前需释放节点 q指向需释放节点的下一节点

LinkNode \*p,\*q;

//p首先指向头节点的下一节点 即首元节点

p=L->next;

**while**(p!=**NULL**)//循环终止条件为p，q均为NULL

{

//由于p指向的节点释放后丢失指向下一节点的指针，所以需要在释放p之前提前用q锁定p所指节点的下一个节点

q=p->next;

**delete** p;

//p后移到下一节点

p=q;

}

//所有数据节点释放完成后 将头节点指针域置空

L->next=**NULL**;

cout<<"链表清空成功"<<endl;

}

//求单链表长度函数

//算法：对所有数据节点依次计数

**int** LinkListLength(LinkNode\* &L)

{

//申请辅助指针指向当前计数节点

LinkNode \*p;

//p指向第一个数据节点

p=L->next;

//计数器

**int** count=0;

//循环后移计数 结束条件是p指向为空

**while**(p!=**NULL**)

{

count++;

p=p->next;

}

**return** count;

}

//按位置取值函数 num表示位置 e表示接收数据的变量

//算法：对数据节点逐个计数 直至计数器和传入的顺序数相等 返回数据

**void** GetElem(LinkNode\* &L,**int** num,TYPE &e)

{

//判空

**if**(LinkListEmpty(L))

{

cout<<"链表为空 无法查找"<<endl;

**return**;

}

**else**

{

//初始化

LinkNode \*p=L->next;

**int** count=1;

**while**(p!=**NULL**&&count<num)

{

p=p->next;

count++;

}

//退出该循环有三种条件：

//1.p=NULL 对应num值大于链表长度 直至最后都找不到

//2.count>num 对应num值小于count的初始值1 不符合查找要求

//3.count==num 对应查找成功

**if**(p==**NULL**||count>num)

{

cout<<"输入有误 查找失败"<<endl;

}

**else**

{

e=p->data;

}

}

}

//按值查找函数 e为需要查找的值 返回值为位置 若未找到返回0

//算法：依次遍历数据节点进行比对

**int** LocateElem(LinkNode\* &L,TYPE e)

{

**if**(LinkListEmpty(L))

{

cout<<"链表为空 无法查找"<<endl;

**return** 0;

}

**else**

{

//初始化

LinkNode \*p=L->next;

//位置计数器

**int** count=1;

**while**(p!=**NULL**&&p->data!=e)

{

p=p->next;

count++;

}

**if**(p==**NULL**)

{

cout<<"查找失败 无对应元素"<<endl;

**return** 0;

}

**else**

{

**return** count;

}

}

}

//查找前驱节点数据函数 cur\_e表示查找值 pre\_e用来接受前驱值

**void** PriorElem(LinkNode\* &L,TYPE cur\_e,TYPE &pre\_e)

{

**if**(LinkListEmpty(L))

{

cout<<"链表为空 查找失败"<<endl;

**return**;

}

**else**

{

//q用来指向指定节点的前一节点

LinkNode \*q=L;

LinkNode \*p=L->next;

**int** count=1;

**while**(p!=**NULL**&&p->data!=cur\_e)

{

p=p->next;

q=q->next;

count++;

}

**if**(p==**NULL**)

{

cout<<"未找到对应数据"<<endl;

**return**;

}

**else** **if**(count==1)

{

cout<<"第一个匹配值为首元素 首元素无前驱值"<<endl;

**return**;

}

**else**

{

pre\_e=q->data;

}

}

}

//查找后继节点数据函数

**void** NextElem(LinkNode\* &L,TYPE cur\_e,TYPE &next\_e)

{

**if**(LinkListEmpty(L))

{

cout<<"链表为空 查找失败"<<endl;

**return**;

}

**else**

{

LinkNode \*p=L->next;

LinkNode \*q=p->next;

**int** count=1;

**while**(p!=**NULL**&&p->data!=cur\_e)

{

p=p->next;

q=p->next;

count++;

}

**if**(p==**NULL**)

{

cout<<"未找到对应数据"<<endl;

**return**;

}

**else** **if**(q==**NULL**)

{

cout<<"表尾节点无后继元素"<<endl;

}

**else**

{

next\_e=q->data;

}

}

}

//遍历链表函数

**void** LinkListTraverse(LinkNode\* &L)

{

//申请辅助指针指向首元节点

LinkNode \*p=L->next;

**while**(p!=**NULL**)

{

cout<<p->data<<" ";

p=p->next;

}

cout<<endl;

}

//按位置插入值函数 num表示位置 e表示要插入的值

//算法：找到需插入位置的前一个节点 然后对其和目标节点进行操作

**void** InsertElem(LinkNode\* &L,**int** num,TYPE e)

{

//初始化

//由于可能出现头插 所以辅助指针的初始位置应该是头节点

LinkNode \*p=L;

//计数器 用以判断位置 注意从头节点开始应初始化为0

**int** count=0;

//找到num-1的位置 注意这个循环包含了在链表尾端进行插入的情况

**while**(p!=**NULL**&&count<num-1)

{

p=p->next;

count++;

}

//左到右分别对应两种错误情况 即num过大或num过小

//若找不到插入位的前一位节点 则一定不可插入

**if**(p==**NULL**||count>num-1)

{

cout<<"未找到指定插入位置 插入失败"<<endl;

}

//若流程正常 此时p指向的是指定插入位前一位的节点

**else**

{

//申请插入节点的空间

LinkNode \*s=**new** LinkNode;

s->data=e;

//先令插入节点指针域指向指定位置原来的节点

s->next=p->next;

//再令前一位的节点指向插入节点

p->next=s;

//注意以上两个步骤不可以交换 否则会丢失指定位原节点

}

}

//按位置删除节点函数 num表示指定位置 e用来接收被删除节点的数据域

**void** DeleteElem(LinkNode\* &L,**int** num,TYPE &e)

{

//申请两个辅助指针 p用以指向被删除节点前面的节点 q用以在释放前保存被删除节点以返回原数据

LinkNode \*p,\*q;

//下为寻找前驱节点的操作

p=L;

**int** count=0;

**while**(p!=**NULL**&&count<num-1)

{

p=p->next;

count++;

}

**if**(p==**NULL**||count>num-1)

{

cout<<"输入有误 无法删除节点"<<endl;

**return**;

}

//此时p指向被删除节点的前驱

**else**

{

q=p->next;

p->next=p->next->next;//也可以写作q->next

e=q->data;

**delete** q;

}

}

//尾插法创建链表函数 n表示所创建链表的节点数 注意先初始化再创建链表

**void** CreateLinkList(LinkNode\* &L,**int** n)

{

//申请两个辅助指针 r指向当前尾节点 p用以反复指向即将插入的节点

LinkNode \*r,\*p;

//初始化

r=L;

**for**(**int** i=0;i<n;i++)

{

//生成新的尾节点

p=**new** LinkNode;

cout<<"请输入第"<<i+1<<"个节点的数据"<<endl;

cin>>p->data;

p->next=**NULL**;

//先插入表尾

r->next=p;

//r指向新表尾节点

r=p;

cout<<"第"<<i+1<<"个节点尾插成功"<<endl;

}

}

**int** main()

{

//申请头指针

LinkNode \*L;

**return** 0;

}

2、

题目：线性表倒序

算法思想：对于顺序表，交换头尾并依次向中间迈进；对于链表，使用头插法

T(O):O(n)

S(O):O(1)

附源程序。

#include<iostream>

#include<list>

**using** **namespace** std;

//顺序结构类型声明 直接初始化

**class** SqList

{

**public**:

**int** \* v=**new** **int**[1000];

**int** length=0;

};

//链式结构节点类型声明

**class** LinkNode

{

**public**:

**int** data;

LinkNode \* next;

};

//顺序结构创建函数

**void** CreateSqList(SqList &L,**int** n)

{

**for**(**int** i=0;i<n;i++)

{

cout<<"请输入第"<<i+1<<"个值"<<endl;

cin>>L.v[i];

L.length++;

}

**if**(L.length==n)

{

cout<<"顺序表初始化成功"<<endl;

}

}

//链式结构创建函数

**void** CreateLinkList(LinkNode\* &L,**int** n)

{

LinkNode \*r,\*p;

r=L;

**for**(**int** i=0;i<n;i++)

{

p=**new** LinkNode;

cout<<"请输入第"<<i+1<<"个值"<<endl;

cin>>p->data;

r->next=p;

r=p;

}

cout<<"链式表初始化成功"<<endl;

}

//遍历顺序表函数

**void** SqListTraverse(SqList &L)

{

**for**(**int** i=0;i<L.length;i++)

{

cout<<L.v[i]<<" ";

}

cout<<endl;

}

//遍历链表函数

**void** LinkListTraverse(LinkNode\* &L)

{

LinkNode \*p=L->next;

**while**(p!=**NULL**)

{

cout<<p->data<<" ";

p=p->next;

}

cout<<endl;

}

//顺序表倒序函数

**void** SqListReverse(SqList &L)

{

**for**(**int** i=0;i<L.length/2;i++)

{

**int** temp=L.v[i];

L.v[i]=L.v[L.length-1-i];

L.v[L.length-1-i]=temp;

}

}

//链表倒序函数 头插法

**void** FirstWay\_LinkListReverse(LinkNode\* &L)

{

//申请辅助节点 p用以指向前一节点 q用以指向当前被移动节点 r用以指向下一节点

LinkNode \*p,\*q,\*r;

p=L->next;

q=p->next;

r=q->next;

**while**(q!=**NULL**)

{

//先删除移动结点

p->next=r;

//结点头插

q->next=L->next;

L->next=q;

**if**(r!=**NULL**)

{

r=r->next;

}

q=p->next;

}

}

//链表倒序函数 逐个倒转指针 指针对数据节点进行操作

**void** SecondWay\_LinkListReverse(LinkNode\* &L)

{

//先假设没有头节点 申请两个辅助指针

LinkNode \*pre,\*cur;

//初始化

pre=**NULL**;

cur=L->next;//cur从第一个数据节点开始 其前面假设没有头节点

//开始倒转指针操作

**while**(cur!=**NULL**)

{

//为防止出现使用NULL->next的情况 每次开始循环时重新寻找当前节点的下一节点

LinkNode \*nex=cur->next;

cur->next=pre;//倒转指针

//移动操作

pre=cur;

cur=nex;

}

//循环完成后 pre指向新的首元节点 此时令头节点指针域指向它

L->next=pre;

}

**int** main()

{

//申请顺序表

SqList sql;

//申请链表头指针并初始化

LinkNode \* ll=**new** LinkNode;

ll->next=**NULL**;

// //线性表测试

// CreateSqList(sql, 5);

// SqListTraverse(sql);

// SqListReverse(sql);

// SqListTraverse(sql);

//链表测试

CreateLinkList(ll, 5);

LinkListTraverse(ll);

SecondWay\_LinkListReverse(ll);

LinkListTraverse(ll);

}

3、

题目：原空间删除重复元素

算法：对两种线性表结构，均采用双重遍历的方法，指定元素后依次遍历其后元素，遇重复删除，然后从该元素后移到下一元素重复该过程。

T(O):O(n^2)

S(O):O(1)

附源代码。

#include<iostream>

**using** **namespace** std;

**class** SqList

{

**public**:

**int** \*v=**new** **int**[1000];

**int** length=0;

};

**class** LinkNode

{

**public**:

**int** data;

LinkNode \*next;

};

//创建线性表

**void** CreateSqList(SqList &L,**int** n)

{

**for**(**int** i=0;i<n;i++)

{

cout<<"请输入第"<<i+1<<"个元素的值"<<endl;

cin>>L.v[i];

L.length++;

}

cout<<"线性表初始化成功"<<endl;

}

//线性表输出

**void** PrintSqList(SqList &L)

{

**for**(**int** i=0;i<L.length;i++)

{

cout<<L.v[i]<<" ";

}

cout<<endl;

}

//线性表去重函数

**void** SqListDeleteDuplicated(SqList &L)

{

**for**(**int** i=0;i<L.length;i++)

{

**for**(**int** j=i+1;j<=L.length-1;j++)

{

**if**(L.v[i]==L.v[j])

{

**for**(**int** k=j+1;k<=L.length-1;k++)

{

L.v[k-1]=L.v[k];

}

L.length--;

j--;

}

}

}

}

//从零创建链表函数

**void** CreateLinkList(LinkNode\* &L,**int** n)

{

LinkNode \*r=L;//用以指向最后一个节点 初始化为头节点

**for**(**int** i=0;i<n;i++)

{

LinkNode \*p=**new** LinkNode;

cout<<"请输入第"<<i+1<<"个节点的数据"<<endl;

cin>>p->data;

//尾插

r->next=p;

r=p;

}

}

//输出链表函数

**void** PrintLinkList(LinkNode\* &L)

{

LinkNode \*p=L->next;

**while**(p!=**NULL**)

{

cout<<p->data<<" ";

p=p->next;

}

cout<<endl;

}

//无序链表去重函数

**void** LinkListDeleteDuplicated(LinkNode\* &L)

{

//p指向当前比对节点 r用于遍历 q指向r前一节点（删除方便）

LinkNode \*p,\*q,\*r;

q=L;

p=L->next;

**while**(p!=**NULL**)

{

q=p;

r=p->next;

**while**(r!=**NULL**)

{

**if**(r->data==p->data)

{

q->next=r->next;

**delete** r;

r=q->next;

**continue**;

}

r=r->next;

q=q->next;

}

p=p->next;

}

}

**int** main()

{

SqList sql;

LinkNode \*ll;

ll=**new** LinkNode;

ll->next=**NULL**;

**return** 0;

}

4、

题目：西西艾弗岛核酸证明

算法：对每一个指定的做核酸时间 检查每个出行计划的时间是否在核算报告的有效时间内

附源代码。

#include<iostream>

#include<vector>

**using** **namespace** std;

//声明计划类

**class** Plan

{

**public**:

**int** in\_time;

**int** require\_hour;

};

//查询函数

**void** HowMany(vector<Plan> &plan,vector<**int**> &q,**int** &k)

{

**for**(vector<**int**>::iterator qit=q.begin();qit!=q.end();qit++)

{

**int** count=0;

**for**(vector<Plan>::iterator pit=plan.begin();pit!=plan.end();pit++)

{

**if**((\*pit).in\_time>=(\*qit)+k&&(\*pit).in\_time<=(\*qit)+k+(\*pit).require\_hour-1)

{

count++;

}

}

cout<<count<<endl;

}

}

**int** main()

{

//n：计划数目，m查询个数，k核酸等待时间

**int** n,m,k;

vector<Plan> plan;

vector<**int**> q;

cin>>n>>m>>k;

plan.reserve(n);

q.reserve(m);

**for**(**int** i=0;i<n;i++)

{

Plan p;

cin>>p.in\_time>>p.require\_hour;

plan.push\_back(p);

}

**for**(**int** i=0;i<m;i++)

{

**int** x;

cin>>x;

q.push\_back(x);

}

HowMany(plan, q, k);

**return** 0;

}

5、

题目：小球碰撞

算法：每一秒后维护一次小球位置的函数 根据传入时间t调用t次该函数

附源代码。

#include<iostream>

#include<vector>

**using** **namespace** std;

**class** Ball

{

**public**:

**int** InitPosition;

**int** NowPosition;

**int** v=1;

};

//1秒后维护小球位置函数

**void** AfterOneSecond(vector<Ball> &ball,**int** &n,**int** &L)

{

**for**(**int** i=0;i<n;i++)

{

ball[i].NowPosition=ball[i].NowPosition+ball[i].v;

}

**for**(**int** i=0;i<n;i++)

{

**if**(ball[i].NowPosition==0||ball[i].NowPosition==L)

{

ball[i].v=ball[i].v\*(-1);

**continue**;

}

**for**(**int** j=i+1;j<n;j++)

{

**if**(ball[i].NowPosition==ball[j].NowPosition)

{

ball[i].v=ball[i].v\*(-1);

ball[j].v=ball[j].v\*(-1);

}

}

}

}

**int** main()

{

**int** n,L,t;

cin>>n>>L>>t;

vector<Ball> ball;

ball.reserve(n);

**for**(**int** i=0;i<n;i++)

{

Ball b;

cin>>b.InitPosition;

b.NowPosition=b.InitPosition;

ball.push\_back(b);

}

**for**(**int** i=0;i<t;i++)

{

AfterOneSecond(ball, n, L);

}

**for**(**int** i=0;i<n;i++)

{

cout<<ball[i].NowPosition<<" ";

}

cout<<endl;

**return** 0;

}

......

二、代码行数及小结

行数：903

小结：

1.本次实验csp题目较为基础，没有过多的算法考察，基本是简单的数组操作实现

2.前三题分别用顺序表和链表实现的过程中感受到了两种线性表数据结构的不同之处。顺序表结构简单，便于理解，遍历方便，使用排序算法时思路简单，但计算机实现的重复过程较多：链表结构相对顺序表而言较为复杂，遍历也比较慢，但是插入删除等操作，计算机的实现会比顺序表效率更高。在实际应用中，应根据实际需求合理选择合适的数据结构。