**《空间数据结构基础》**

**课程设计报告（地信21级）**

**姓名 马骁**

**班级 地信21-1班**

**学号 07212393**

**环境与测绘学院**

**目录**

[1. C++面向对象程序设计基础 1](#_Toc134312357)

[2. 链表的建立、合并与拆分 7](#_Toc134312358)

[3. 通讯簿管理（顺序表的应用） 13](#_Toc134312359)

[5. 将中缀表达式转换为后缀表达式 25](#_Toc134312360)

[6. 图元识别和替换（用队列实现） 29](#_Toc134312361)

[7. 队列的应用 35](#_Toc134312362)

[8. 二叉树的操作 44](#_Toc134312363)

[9. 字符串 52](#_Toc134312364)

# 1. C++面向对象程序设计基础

**【实验简介】**学会用算法语言C++描述抽象数据类型，使用模板建立数据结构。理解数据结构的组成分为两部分，第一部分是数据集（数据元素），第二部分是在此数据集上的操作。从面向对象的观点看，这两部分代表了对象的属性和方法。掌握用C++描述数据结构的基本方法，即通过建立类来描述抽象数据类型。类的数据成员提供对象属性，成员函数提供操作方法，方法是公共接口，用户通过调用方法实现对属性的访问。

**【实验内容】**

1. 定义三维空间的坐标点TPoint

2. 描述三维空间的球TBall，实现其主要操作（如计算体积和表面积，输出空间坐标等）。

**【主要代码】**

**定义三维空间的坐标点TPoint**

Tpoint.h

#ifndef **DATA\_STRUCTURE\_TPOINT\_H**#define **DATA\_STRUCTURE\_TPOINT\_H**#include<iostream>  
using namespace std;  
class TPoint{  
private:  
 double x;  
 double y;  
 double z;  
public:  
 TPoint(){  
 x = 0;  
 y = 0;  
 z = 0;  
 }  
  
 TPoint(double x1,double y1,double z1):x(x1),y(y1),z(z1){} *//带参构造函数* ~TPoint(){}  
 void move(double x1,double y1,double z1) *//移动位置* {  
 x = x1;  
 y = y1;  
 z = z1;  
 }  
 void show() *//输出坐标* {  
 cout<<"x = "<<x<<" "<<"y = "<<y<<" z = "<<z<<endl;  
 }  
 void setx(double x1)  
 {  
 x = x1;  
  
 }  
 void sety(double y1)  
 {  
 y = y1;  
 }  
 void setz(double z1)  
 {  
 z = z1;  
 }  
};  
  
#endif *//DATA\_STRUCTURE\_TPOINT\_H*

main.cpp

#include"TPoint.h"  
  
int main()  
{  
 TPoint a;  
 TPoint b(10,2.0,1.2);  
 cout<<"a点位置"<<endl;  
 a.show();  
 cout<<"b点位置"<<endl;  
 b.show();  
  
  
 double x1=0,y1=0,z1=0;  
 cout<<"设置a点坐标"<<endl;  
 cout<<"输入a点的x坐标"<<endl;  
 cin>>x1;  
 a.setx(x1);  
 cout<<"输入a点的y坐标"<<endl;  
 cin>>y1;  
 a.sety(y1);  
 cout<<"输入a点的z坐标"<<endl;  
 cin>>z1;  
 a.setz(z1);  
 cout<<"a点当前坐标"<<endl;  
 a.show();  
 cout<<"a点移动"<<endl;  
  
 double x2,y2,z2;  
 cout<<"输入移动后的a点x坐标： "<<endl;  
 cin>>x2;  
 cout<<"输入移动后的a点y坐标： "<<endl;  
 cin>>y2;  
 cout<<"输入移动后的a点z坐标： "<<endl;  
 cin>>z2;  
 a.move(x2,y2,z2);  
 cout<<"a点新坐标"<<endl;  
 a.show();  
  
 cout<<"a和b之间的距离为："<<endl;  
 a.distance(b);  
  
}

**描述三维空间的球TBall，实现其主要操作（如计算体积和表面积，输出空间坐标等）。**

Tball.h

#ifndef **DATA\_STRUCTURE\_TBALL\_H**#define **DATA\_STRUCTURE\_TBALL\_H**#define **PI** 3.14  
#include<iostream>  
#include<math.h>  
#include<math.h>  
using namespace std;  
  
class TBall{  
private:  
 double x,y,z,r; *//球心坐标和半径*public:  
 TBall();  
 TBall(double x1,double y1,double z1,double r1):x(x1),y(y1),z(z1),r(r1){}  
 void show()  
 {  
 cout<<"坐标位置： x: "<<x<<" y: "<<y<<" z: "<<z<<" 半径为： "<<r<<endl;  
 }  
 double volume()  
 {  
 return 4.0/3.0\***PI**\*pow(r,3);  
 }  
 double SurfaceArea()  
 {  
 return 4.0\***PI**\*pow(r,2);  
 }  
 void printPoint()  
 {  
 cout<<"球心坐标： ("<<x<<","<<y<<","<<z<<")"<<endl;  
 }  
 void distance(TBall b); *//计算两点的距离*};  
void TBall::distance( TBall b) {  
 double dis = sqrt(pow(x-b.x,2)+pow(y-b.y,2)+pow(z-b.z,2));  
 cout<<"两点之间的距离为： "<<dis<<endl;  
  
}

ball.cpp

#include "TBall.h"  
  
int main()  
{  
 TBall ballA(1,2,1,4);  
 cout<<"A球的信息："<<endl;  
 ballA.show();  
 cout<<"体积为： "<<ballA.volume()<<endl;  
 cout<<"表面积为： "<<ballA.SurfaceArea()<<endl;  
 ballA.printPoint();  
  
 cout<<"创建B球"<<endl;  
 double x1,y1,z1,r1;  
 cout<<"输入坐标x： "<<endl;  
 cin>>x1;  
 cout<<"输入坐标y： "<<endl;  
 cin>>y1;  
 cout<<"输入坐标z： "<<endl;  
 cin>>z1;  
 cout<<"输入半径r： "<<endl;  
 cin>>r1;  
 TBall ballB(x1,y1,z1,r1);  
 cout<<"B球的信息："<<endl;  
 ballB.show();  
 cout<<"B球体积为： "<<ballB.volume()<<endl;  
 cout<<"B球表面积为： "<<ballB.SurfaceArea()<<endl;  
  
 cout<<"A球和B球球心间的距离为： "<<endl;  
 ballA.distance(ballB);  
  
  
  
 return 0;  
}

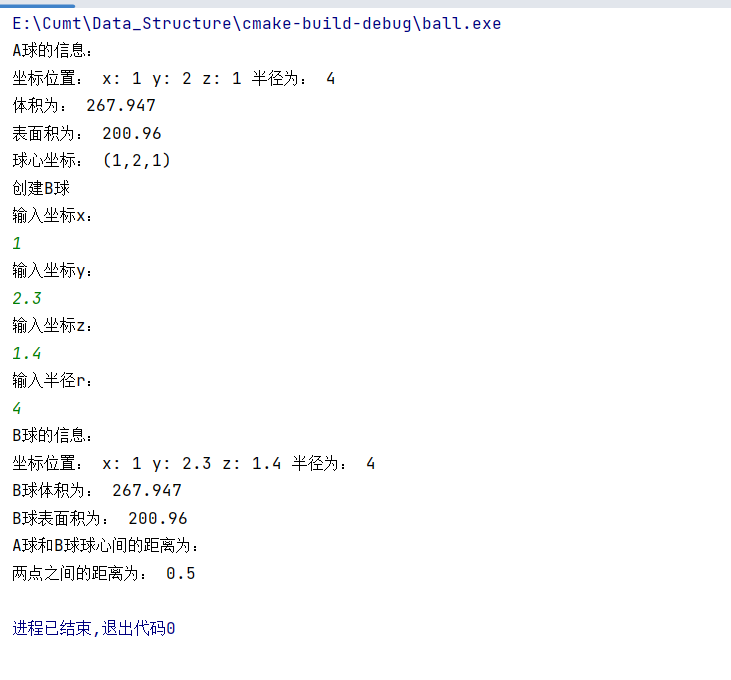
#endif *//DATA\_STRUCTURE\_TBALL\_H*

**【实验过程】**

首先定义一个点TPoint类用来表示空间坐标，其次定义一个球TBall类用来表示球以及定义球相关的操作。

输入TpointA的x,y,z坐标，并进行移动操作，最后自动计算点A和点B之间的距离。

在函数实现过程中，首先输入球的x、y、z坐标与球的半径，程序自动计算球的表面积与体积。

之后再以此输入球B的坐标，计算A球心和B球心之间的距离 

**【实验体会】**

本次实验，学会了用面向对象方法定义一个简单的抽象数据类型，抽象数据类型是由普通数据类型以及一组相关的操作构成，通过实验，切实体会理解了定义中相关的操作的含义。

通过这次实验，对类的构造函数有了理解，区分了有参数的构造函数与无参数的构造函数在类进行实例化时的不同与实例化时不同构造函数的调用方式。无参的构造函数在实例化时自动调用，有参的构造函数需要在实例化时给定需要的参数进行调用。

# 2. 链表的建立、合并与拆分

**【实验简介】**链表是用链接存储的方式来表达线性表，它用指针表示结点间的逻辑关系，链表适用于插入或删除频繁，存储空间需求不定的情形。

**【实验内容】**定义一个链表存储的线性表，除已给出的表元素插入、删除、查找等基本操作外，再提供表的合并、拆分和逆置等操作。在应用程序中建立两个整型的单链表对象A和B，应用线性表的基本操作对表的实例对象进行操作测试。

1. 设线性链表A=(a1,a2,…,am),，B=(b1,b2,…bm)，按下列规则合并A，B为线性表C的算法，即使得

C = (a1,b1,…,am,bm, b (m+1),…,bn) 当m<=n

或 C = (a1,b1,…,an,bn, a(n+1),…,am) 当m>n

C表利用A表和B表中的结点空间构成。

2. 将C表原地逆置。

3. 将C表的中偶数和奇数分别链接为两个循环链表D和E。

说明：每一次合并、拆分和逆置等操作的结果均要输出。

**【主要代码】**

头文件

#ifndef **DATA\_STRUCTURE\_LINKEDKIST\_H**#define **DATA\_STRUCTURE\_LINKEDKIST\_H**#include <iostream>  
using namespace std;  
  
struct ListNode {  
 int val;  
 ListNode\* next;  
 ListNode(int x) : val(x), next(**NULL**) {}  
};  
  
class LinkedList {  
public:  
 ListNode\* head;  
 LinkedList() {  
 head = new ListNode(0);  
 }  
 void insert(int x) {  
 ListNode\* node = new ListNode(x);  
 node->next = head->next;  
 head->next = node;  
 }  
 void print() {  
 ListNode\* p = head->next;  
 while (p != **NULL** && p!= head) {  
 cout << p->val << " ";  
 p = p->next;  
 }  
 cout << endl;  
 }  
  
 void append(int x) {  
 ListNode\* newNode = new ListNode(x);  
 ListNode\* p = head;  
 while (p->next != **NULL**) {  
 p = p->next;  
 }  
 p->next = newNode;  
 }  
 int Length()  
 {  
 int len = 0;  
 ListNode \*cur = head->next;  
 while(cur != nullptr)  
 {  
 cur = cur->next;  
 len++;  
 }  
 return len;  
 }  
  
  
};  
  
  
LinkedList mergeLinkedList(LinkedList A, LinkedList B) { *//合并链表* ListNode\* a = A.head->next;  
 ListNode\* b = B.head->next;  
 ListNode\* c = new ListNode(0);  
 ListNode\* p = c;  
 while (a != **NULL** && b != **NULL**) {  
 p->next = a;  
 a = a->next;  
 p = p->next;  
  
 p->next = b;  
 b = b->next;  
 p = p->next;  
 }  
 if (a != **NULL**) {  
 p->next = a;  
 }  
 else {  
 p->next = b;  
 }  
 LinkedList result;  
 result.head->next = c->next;  
 delete c;  
 return result;  
}  
  
LinkedList createList() {  
 LinkedList list;  
 int n;  
 cout << "请输入链表的长度："<<endl;  
 cin >> n;  
 cout << "请依次输入链表的每个元素："<<endl;  
 for (int i = 0; i < n; i++) {  
 int x;  
 cin >> x;  
 list.append(x);  
 }  
 return list;  
}  
  
void reverseLinkedList(LinkedList& C) { *//反转链表* ListNode\* prev = nullptr; *//前一个节点* ListNode\* curr = C.head->next; *//当前节点* ListNode\* next = nullptr; *//下一个节点* while (curr != nullptr) {  
 next = curr->next;  
 curr->next = prev; *//cur的下一个节点指向cur的前一个节点，实现反转* prev = curr;  
 curr = next;  
 }  
 C.head->next = prev;  
}  
  
void splitLinkedList(LinkedList C, LinkedList& D, LinkedList& E) { *//拆分为两个循环链表* ListNode\* p = C.head->next;  
 ListNode\* dTail = D.head;  
 ListNode\* eTail = E.head;  
 while (p != **NULL**) {  
 if (p->val % 2 == 0) { *//偶数进入D链表* dTail->next = p;  
 dTail = p;  
 } else {  
 eTail->next = p; *//奇数进入e链表* eTail = p;  
 }  
 p = p->next;  
 }  
 dTail->next = D.head; *//末尾指向头结点* eTail->next = E.head;  
}  
  
  
#endif *//DATA\_STRUCTURE\_LINKEDKIST\_H*

源文件

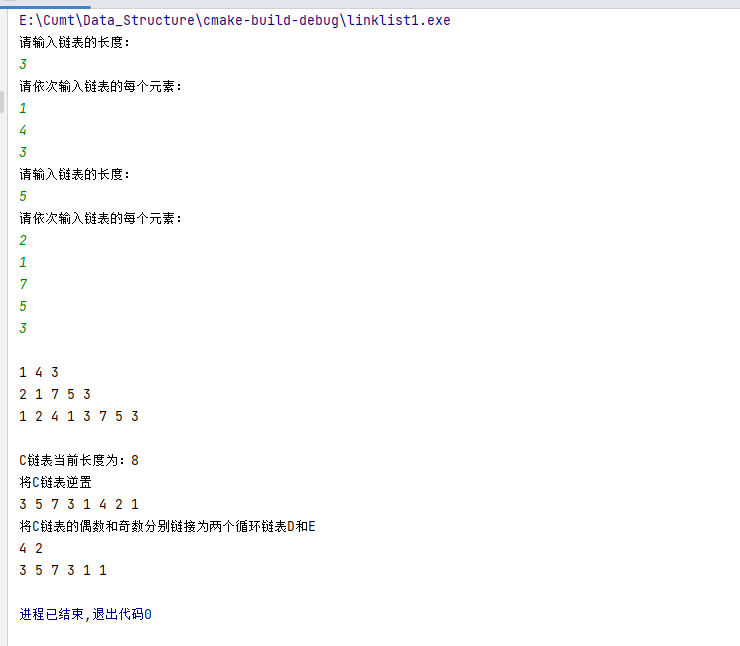
#include "linkedkist.h"  
  
  
int main() {  
 LinkedList A, B;  
 A = createList();  
 B = createList();  
 LinkedList D,E;  
 D.print();  
  
 A.print();  
 B.print();  
 LinkedList C = mergeLinkedList(A, B);  
 C.print();  
 cout << endl;  
 cout<<"C链表当前长度为："<<C.Length()<<endl;  
 cout<<"将C链表逆置"<<endl;  
 reverseLinkedList(C);  
 C.print();  
 cout<<"将C链表的偶数和奇数分别链接为两个循环链表D和E"<<endl;  
 splitLinkedList(C,D,E);  
 D.print();  
 E.print();  
  
 return 0;  
}

**【实验过程】**

首先创建一个结构体ListNode表示单链表的节点，然后建立单链表类LinkedList实现链表的插入，删除，输入，遍历等功能。

实现两个链表合并的功能，新建一个头节点，依次把a，b链表的元素连接到新建头节点上，最后在创建一个单链表，让单链表头结点指向的元素改为这个新建头结点指向的元素，最后将链表返回。

逆置链表通过三个节点指针，prev，curr，next实现逆置，拆分链表实现较为容易，只需判断链表元素奇偶性，进入相应的链表即可



**【实验体会】**

本次实验学会了使用结构体，定义的结构体可以被所有类共享。在对链表的建立中，使用后插法，每一个新加入的数据都在链表的后端。并且，了解了线性数据结构，掌握了链式存储的线性表的建立与使用。学习并实现了链表合并，逆置，拆分三个较难的操作，通过利用指针指向不同节点来实现，对指针的操作和理解要求较高。

# 3. 通讯簿管理（顺序表的应用）

**【问题描述】**通讯簿是一个线性表，可以存储一定数量的联系人记录，提供查找、插入、删除和修改等操作。通讯簿的特点是以查找为主要操作，要求快速查找到指定对象的位置，故宜采用具有随机访问功能的顺序表。

**【数据结构】**使用顺序表SeqList建立通讯簿。作为表项的联系人记录，至少应包括以下属性：

{ 序号，姓名，与本人关系，电话号码}

其中序号具有唯一性。序号和姓名可作为查找的主要关键字。与本人关系可枚举为“亲人”、“朋友”和“同事”，主要作用是为联系人分组，并作为次要关键字。

将上述联系人记录定义为一个结构（struct），在主程序中建立模板类顺序表SeqList的对象时用该结构实例化表结点的类型。

**【主要功能】**程序应为用户提供操作选择界面，必要的操作包括：查找某人电话号码，添加新记录，修改记录，删除记录，打印亲人清单、朋友清单或同事清单以及退出等。另外，为初始化方便，原始数据可存储在磁盘文件中。

**【主要代码】**

头文件

#ifndef **COURSE\_DESIGN\_TXB\_H**#define **COURSE\_DESIGN\_TXB\_H**#include<iostream>  
using namespace std;  
  
struct dataItem{  
 int No; *//序号* string name; *//姓名* string relation; *//与本人关系* int phonenum; *//电话号码*};  
  
  
class SeqList{  
protected:  
 dataItem \*data; *//创建一个动态存储的dataItem数组*public:  
 SeqList(){ data = new dataItem[100];}  
 dataItem \*getdata();  
 void Search(int tag,string nm);  
 void Change(int tag); *//tag为想要进行操作的数据的序号* void Remove(int tag);  
 void add();  
 void printFriend();  
 void printRelative();  
 void printWorkmate();  
 void printAll();  
  
};  
  
static int len =0;  
void SeqList::Search(int tag=0, string nm = "") {  
 if(tag != 0)  
 {  
 cout<<"按序号查找"<<endl;  
 cout<<"你想要查找序号为"<<tag<<"的数据"<<endl;  
 int c = 0;  
 for(int i=0;i<len;i++)  
 {  
  
 if(tag == data[i].No)  
 {  
 cout<<"序号："<<data[i].No<<" ";  
 cout<<"姓名："<<data[i].name<<" ";  
 cout<<"电话号码: "<<data[i].phonenum<<" ";  
 cout<<"关系："<<data[i].relation<<" ";  
 cout<<endl;  
 c = 1;  
 break;  
 }  
  
 }  
 if(!c)  
 {  
 cout<<"没有查找到序号为"<<tag<<"的数据"<<endl;  
 }  
 }  
 if(nm != "")  
 {  
 cout<<"按姓名查找"<<endl;  
 cout<<"你想要查找姓名为"<<nm<<"的数据"<<endl;  
 int c = 0;  
 for(int i=0;i<len;i++)  
 {  
  
 if(nm == data[i].name)  
 {  
 cout<<"序号："<<data[i].No<<" ";  
 cout<<"姓名："<<data[i].name<<" ";  
 cout<<"电话号码: "<<data[i].phonenum<<" ";  
 cout<<"关系："<<data[i].relation<<" ";  
 c = 1;  
 break;  
 }  
  
 }  
 if(!c)  
 {  
 cout<<"没有查找到姓名为"<<nm<<"的数据"<<endl;  
 }  
 }  
  
}  
  
void SeqList::add() {  
  
 cout << "请输入您想要添加的人的序号："<<endl;  
 cin >> data[len].No;  
 cout << "请输入您想要添加的人的姓名："<<endl;  
 cin >> data[len].name;  
 cout << "请输入您想要添加的人与本人的关系：(亲人、朋友、同事)"<<endl;  
 cin >> data[len].relation;  
 cout << "请输入您想要添加的人的电话号码："<<endl;  
 cin >> data[len].phonenum;  
 len++; *//通讯簿元素数量加一* cout<<"添加联系人成功 !"<<endl;  
}  
  
void SeqList::Change(int tag) {  
 int No1;  
 string name1;  
 string relation1;  
 int phonenum1;  
 int d = 0;  
 for(int i =0;i<len;i++)  
 {  
 if(tag = data[i].No)  
 {  
 d = 1;  
 cout<<"输入你想修改成的序号："<<endl;  
 cin>>No1;  
 data[i].No = No1;  
  
 cout<<"输入你想修改成的姓名："<<endl;  
 cin>>name1;  
 data[i].name = name1;  
  
 cout<<"输入你想修改成的关系："<<endl;  
 cin>>relation1;  
 data[i].relation = relation1;  
  
 cout<<"输入你想修改成的电话号："<<endl;  
 cin>>phonenum1;  
 data[i].phonenum = phonenum1;  
 }  
 }  
 if(d)  
 {  
 cout<<"修改成功 ！"<<endl;  
 }  
 else  
 {  
 cout<<"修改失败 ！"<<endl;  
 }  
}  
void SeqList::printAll() {  
 cout<<"-----------显示所有联系人---------"<<endl;  
 cout<<"-----------\*\*\*\*\*---------"<<endl;  
 cout<<"-----------\*\*\*\*\*---------"<<endl;  
 for(int i = 0;i<len;i++)  
 {  
 cout<<"序号："<<data[i].No<<" ";  
 cout<<"姓名："<<data[i].name<<" ";  
 cout<<"关系："<<data[i].relation<<" ";  
 cout<<"电话号码："<<data[i].phonenum<<" ";  
 cout<<endl;  
 }  
 cout<<"显示所有联系人成功 ！"<<endl;  
}  
  
void SeqList::Remove(int tag) {  
 int d = 0;  
 int cnt =-1;  
 for(int i =0;i<len;i++)  
 {  
 cnt++;  
 if(tag == data[i].No)  
 {  
 d = 1;  
 for(int i =cnt;i<len;i++)  
 {  
 data[i] = data[i+1];  
 }  
 len--;  
 }  
 }  
 if(d)  
 {  
 cout<<"删除成功 ！"<<endl;  
 }  
 else  
 {  
 cout<<"删除失败 ！"<<endl;  
 }  
}  
  
void SeqList::printFriend() {  
 for (int i = 0; i < len; i++) {  
 if (data[i].relation == "朋友") {  
 cout << "序号：" << data[i].No << " ";  
 cout << "姓名：" << data[i].name << " ";  
 cout << "关系：" << data[i].relation << " ";  
 cout << "电话号码：" << data[i].phonenum << " ";  
 cout << endl;  
 }  
  
 }  
 cout << "显示朋友列表成功 ！" << endl;  
}  
  
  
void SeqList::printRelative() {  
 for (int i = 0; i < len; i++)  
 {  
 if (data[i].relation == "亲人")  
 {  
 cout << "序号：" << data[i].No << " ";  
 cout << "姓名：" << data[i].name << " ";  
 cout << "关系：" << data[i].relation << " ";  
 cout << "电话号码：" << data[i].phonenum << " ";  
 cout << endl;  
 }  
  
 }  
 cout << "显示亲人列表成功 ！" << endl;  
}  
  
void SeqList::printWorkmate() {  
 for (int i = 0; i < len; i++)  
 {  
 if (data[i].relation == "同事")  
 {  
 cout << "序号：" << data[i].No << " ";  
 cout << "姓名：" << data[i].name << " ";  
 cout << "关系：" << data[i].relation << " ";  
 cout << "电话号码：" << data[i].phonenum << " ";  
 cout << endl;  
 }  
  
 }  
 cout << "显示同事列表成功 ！" << endl;  
}  
  
#endif *//COURSE\_DESIGN\_TXB\_H*

源文件

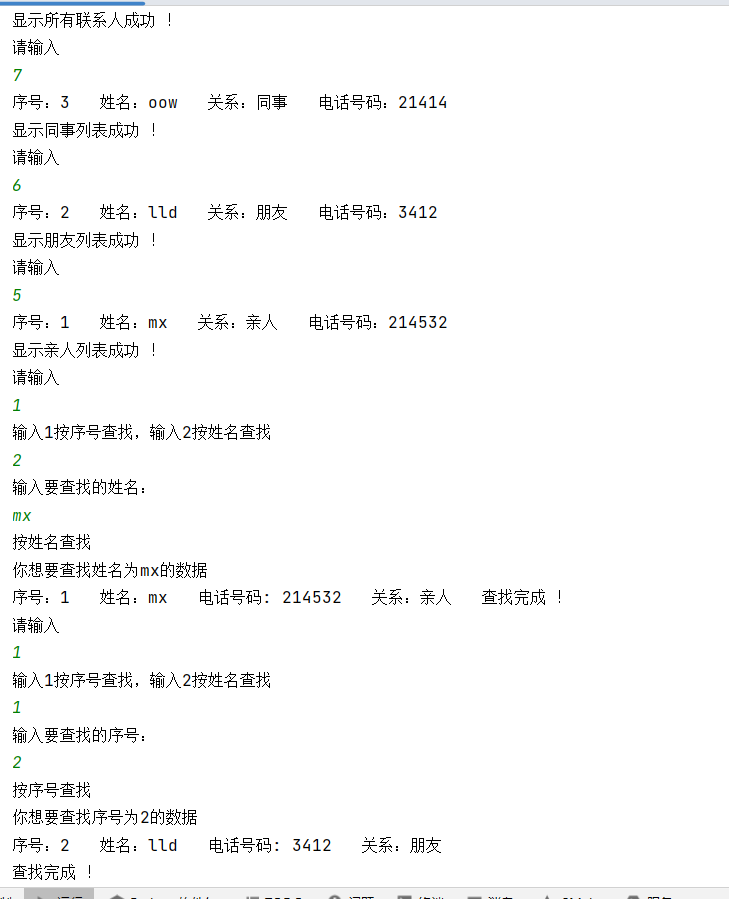
#include "txb.h"  
int main()  
{  
 cout << "┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓"<< endl;  
 cout << " | 通～讯～簿 |"<< endl;  
 cout << " | ☎ |"<< endl;  
 cout << " | 如果你选‘1’ - 查找电话号码 |"<< endl;  
 cout << " | 如果你选‘2’ - 可添加新记录 |"<< endl;  
 cout << " | 如果你选‘3’ - 可以修改记录 |"<< endl;  
 cout << " | 如果你选‘4’ - 可以删除记录 |"<< endl;  
 cout << " | 如果你选‘5’ - 打印亲人清单 |"<< endl;  
 cout << " | 如果你选‘6’ - 打印朋友清单 |"<< endl;  
 cout << " | 如果你选‘7’ - 打印同事清单 |"<< endl;  
 cout << " | 如果你选‘8’ - 打印所有清单 |"<< endl;  
 cout << " | 如果你选‘9’ - 我们下次再见 |"<< endl;  
 cout << " | |"<< endl;  
 cout << " | Copyright © 2023 MX.|" << endl;  
 cout << "┗━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┛"<< endl;  
  
 SeqList address;  
 while(1)  
 {  
 cout<<"请输入"<<endl;  
 int x;  
 cin>>x;  
  
 if(x == 1)  
 {  
 cout<<"输入1按序号查找，输入2按姓名查找"<<endl;  
 int sc;  
 cin>>sc;  
 if(sc == 1)  
 {  
 cout<<"输入要查找的序号："<<endl;  
 int tag;  
 cin>>tag;  
 address.Search(tag);  
 }  
 else if(sc == 2)  
 {  
 cout<<"输入要查找的姓名："<<endl;  
 string name;  
 cin>>name;  
 address.Search(0,name);  
 }  
 else  
 {  
 cout<<"输入有误请重新输入 ！"<<endl;  
 }  
 cout<<"查找完成 ！"<<endl;  
 }  
 if(x == 2)  
 {  
 address.add();  
 }  
 if(x == 3)  
 {  
 cout<<"请输入要修改的序号："<<endl;  
 int tag;  
 cin>>tag;  
 address.Change(tag);  
  
 }  
 if(x == 4)  
 {  
 cout<<"请输入要删除的序号："<<endl;  
 int tag;  
 cin>>tag;  
 address.Remove(tag);  
  
 }  
 if(x == 5)  
 {  
 address.printRelative();  
 }  
 if(x == 6)  
 {  
 address.printFriend();  
 }  
 if(x == 7)  
 {  
 address.printWorkmate();  
 }  
 if(x == 8)  
 {  
 address.printAll();  
 }  
 if(x == 9)  
 {  
 cout<<"----------再见---------"<<endl;  
 break;  
 }  
 }  
  
  
}

**【实验过程】**

设计结构体dataitem存储通讯簿的每一条记录，再编写SeqList类设计添加，删除，修改，查找记录等功能，查找记录时可以按序号或姓名查找。











**【实验体会】**

编写了通讯簿进行数据的记录和设计相关操作，利用数组存储结构体元素，设计过程功能较多，总体难度不大，提升对结构体，类的认识程度。

# 5. 将中缀表达式转换为后缀表达式

【问题描述】表达式转换。输入的中缀表达式为字符串，转换得到的后缀表达式存入字符数组中并输出。

例如： a\*(x+y)/(b-x) 转换后得： a x y + \* b x - /

【数据结构】

* 定义一个暂时存放运算符的转换工作栈opst。
* 中缀表达式字符串char \*infix;
* 后缀表达式字符串char \*postfix;

【算法提示】转换规则：把运算符移到它的两个操作数后面，删除掉所有的括号。

从头到尾扫描中缀表达式，对不同类型的字符按不同情况处理：

* 数字或小数点，直接写入字符串postfix，并在每个数值后面写入一个空格；
* 左括号，进栈，直到遇见相配的右括号，才出栈；
* 右括号，表明已扫描过括号内的中缀表达式，把从栈顶直到对应左括号之间的运算符依次退栈，并把结果推入栈内；
* 对于运算符，分两种情况处理：
  + - 该运算符的优先级大于栈顶符号的优先级，则入栈；
    - 若该运算符的优先级小于栈顶优先级，则先弹出栈顶运算符、写入postfix串；继续将该运算符与栈顶运算符比较，直到能把它推入栈内为止（即优先级大于栈顶运算符）。

说明：自行设计运算符优先级的表示。

【主要代码】

头文件

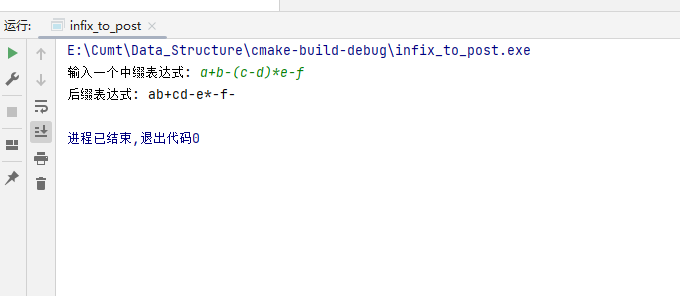
#ifndef **DATA\_STRUCTURE\_CHANGE\_H**#define **DATA\_STRUCTURE\_CHANGE\_H**#include <iostream>  
#include <stack>  
#include <string>  
using namespace std;  
  
*// 判断运算符的优先级*int precedence(char op)  
{  
 if (op == '+' || op == '-')  
 return 1;  
 else if (op == '\*' || op == '/')  
 return 2;  
 else if (op == '^')  
 return 3;  
 return 0;  
}  
  
*// 中缀转后缀函数*string infixToPostfix(const string& infix)  
{  
 stack<char> opst; *// 运算符栈* string postfix; *// 后缀表达式字符串* for (char ch : infix)  
 {  
 if (isalnum(ch)) *// 如果是字母或数字，直接添加到后缀表达式字符串* {  
 postfix += ch;  
 }  
 else if (ch == '(') *// 如果是左括号，推入栈中* {  
 opst.push(ch);  
 }  
 else if (ch == ')') *// 如果是右括号，弹出栈中运算符，直到遇到左括号* {  
 while (!opst.empty() && opst.top() != '(')  
 {  
 postfix += opst.top();  
 opst.pop();  
 }  
  
 if (!opst.empty() && opst.top() == '(')  
 opst.pop();  
 }  
 else *// 如果是运算符* {  
 *//!opst.empty()表示 栈opst不为空* while (!opst.empty() && precedence(ch) <= precedence(opst.top())) *// 弹出栈中优先级较高的运算符* {  
 postfix += opst.top();  
 opst.pop();  
 }  
  
 opst.push(ch); *// 将当前运算符推入栈中* }  
 }  
  
 while (!opst.empty()) *// 弹出栈中剩余的运算符* {  
 postfix += opst.top();  
 opst.pop();  
 }  
  
 return postfix; *// 返回后缀表达式字符串*}  
  
#endif *//DATA\_STRUCTURE\_CHANGE\_H*

源文件

#include"change.h"  
  
int main()  
{  
 string infix;  
 cout << "输入一个中缀表达式: ";  
 getline(cin, infix);  
  
 string postfix = infixToPostfix(infix);  
 cout << "后缀表达式: " << postfix << endl;  
  
 return 0;  
}

【实验过程】

利用了c++的STL标准模板库中的<stack>，提供了栈的相关操作进栈push()，退栈pop()等。首先设计运算符的优先级，我设计了一个简单的运算符优先级表示方法，避免判断栈内优先级和栈外优先级。在函数string infixToPostfix(const string &infix)中，我首先创建一个栈opst，一个后缀表达式字符串postfix，利用isalnum函数判断字符是字母还是数字，判断字符是左括号还是右括号，左括号直接进栈，右括号弹出栈中运算符，直到遇到左括号等等。



【实验体会】

利用STL模板库简化代码，进行栈的相关操作更加简单，设计转换函数时对栈，和字符串的知识了解要求较高，设计运算符优先级更体现如何快速实现功能的想法。

6. 图元识别和替换（用队列实现）

【问题描述】对于点阵式的图，在程序中可以用二维数组来表示。图中一片连续的具有相同颜色的点称为“图元”。现给出的问题是，对图中指定的图元用另一种颜色来替换。用队列实现。输入输出界面与实验指导书中算法实现\_程序3.1相同。

【主要代码】

头文件

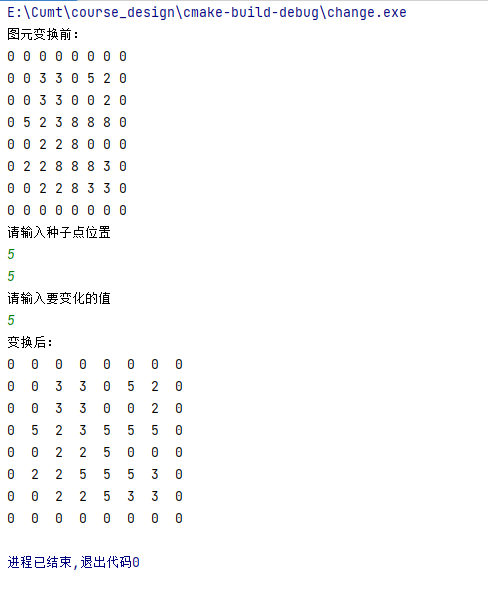
#ifndef **COURSE\_DESIGN\_TUYUAN\_H**#define **COURSE\_DESIGN\_TUYUAN\_H**#include<iostream>  
using namespace std;  
  
struct Pixel {  
 int row; *//行下标* int col; *//列下标*};  
  
Pixel offset[4]; *//每个像素具有上下左右四个相邻元素，此为方向移动数组，用于像素移位*class Queue {  
public:  
 Queue(int sz = 30);  
 ~Queue() { delete[] elements; }  
 bool push(Pixel& p);  
 bool pop(Pixel& p);  
 bool getFront(Pixel& p);  
 void makeEmpty() { front = rear = 0; }  
 bool isEmpty()  
 {  
 return (front == rear) ? true : false;  
  
 }  
 bool isFull()  
 {  
 return ((rear + 1) % maxSize == front) ? true : false;  
 }  
 int getSize()  
 {  
 return (rear - front + maxSize) % maxSize;  
 }  
  
private:  
 Pixel \*elements; *//存放Pixel的队列* int rear, front;  
 int maxSize;  
};  
  
Queue::Queue(int sz)  
{  
 rear = front = 0;  
 maxSize = sz;  
 elements = new Pixel[maxSize];  
}  
bool Queue::push(Pixel &p)  
{  
 if (isFull() == true)  
 {  
 return false;  
 }  
 elements[rear].row = p.row;  
 elements[rear].col = p.col;  
 rear = (rear + 1) % maxSize;  
 return true;  
}  
bool Queue::pop(Pixel& p)  
{  
 if (isEmpty() == true)  
 {  
 return false;  
 }  
 p.row = elements[front].row;  
 p.col = elements[front].col;  
 front = (front + 1) % maxSize;  
 return true;  
}  
  
  
bool Queue::getFront(Pixel& p)  
{  
 if (isEmpty() == true)  
 {  
 return false;  
 }  
 p.col = elements[front].col;  
 p.row = elements[front].row;  
 return true;  
}  
  
  
#endif *//COURSE\_DESIGN\_TUYUAN\_H*

**源文件**

#include"tuyuan.h"  
  
#define **M** 8  
#define **N** 8  
int main()  
{  
 int arr[**M**][**N**] = {  
 0,0,0,0,0,0,0,0,  
 0,0,3,3,0,5,2,0,  
 0,0,3,3,0,0,2,0,  
 0,5,2,3,8,8,8,0,  
 0,0,2,2,8,0,0,0,  
 0,2,2,8,8,8,3,0,  
 0,0,2,2,8,3,3,0,  
 0,0,0,0,0,0,0,0  
  
 };  
 cout << "图元变换前：" << endl;  
 for (int i = 0; i < **M**; i++)  
 {  
 for (int j = 0; j < **N**; j++)  
 {  
 cout << arr[i][j] << " ";  
 }  
 cout << endl;  
 }  
  
  
 Queue queue;  
 Pixel a;  
 Pixel offset[4];  
 offset[0].row = 0; offset[0].col = 1; *//右* offset[1].row = 1; offset[1].col = 0; *//下* offset[2].row = 0; offset[2].col = -1; *//左* offset[3].row = -1; offset[3].col = 0; *//上* cout << "请输入种子点位置" << endl;  
 int m1, n1,q;  
 cin >> m1 >> n1;  
 cout << "请输入要变化的值" << endl;  
 cin >> q;  
 int z = arr[m1][n1];  
  
 if (arr[m1][n1] != q) *//先判断是否是同像素,不是则赋予新的属性值* {  
 arr[m1][n1] = q;  
 a.row = m1;  
 a.col = n1;  
 queue.push(a); *//让该像素点入队* while (queue.isEmpty() == false) *//队不为空时执行* {  
 queue.pop(a); *//队头元素出队* int m = a.row;  
 int n = a.col;  
 for (int i = 0; i < 4; i++) *//判断出队元素右、下、左、上是否符合改变属性值的条件* {  
 a.row = m + offset[i].row;  
 a.col = n + offset[i].col;  
 if (arr[a.row][a.col] == z) *//符合则改变属性值后入队* {  
 arr[a.row][a.col] = q;  
 queue.push(a);  
 }  
 }  
 }  
 }  
 else  
 {  
 cout << "输入的像素属性值相同" << endl;  
 }  
  
  
 cout << "变换后：" << endl; *//输出变换后的点阵* for (int i = 0; i < **M**; i++)  
 {  
 for (int j = 0; j < **N**; j++)  
 {  
 cout << arr[i][j] << " ";  
 }  
 cout <<endl;  
 }  
  
  
 return 0;  
}

【实验过程】

设计了一个结构体Pixel用于存放每个元素的row，col两个属性，创建一个移位数组offset，记录每个元素的上下左右四个相邻属性。设计了一个Queue队列，实现了进队push，出队pop等功能，在主函数中进行判断，如果改变的值不与原来值相等，将该元素进队，并判断他的四个相邻元素是否等于输入值，相等也入队，最后打印改变后的二维数组。



【实验体会】

本次实验熟悉了队列的定义与应用，在队列的应用时，通过数据元素进队出队，对队列先进先出的特点有了理解。

并且，这次实验进行了图元识别与替换的问题解决，通过选定初始的种子点，判断是否需要进行替换，如果需要替换则将它替换后进队。之后进行出队，对出队元素的上下左右四个像素点进行判断，若需要替换则替换后进队，重复上述过程，直到出队后队为空，则完成图元的识别与替换。

该算法实现有一定的难度，需要仔细思考，设计入队和出队，并写好判断条件是关键，通过设计改算法，对队列的认识更加深刻。

# 7. 队列的应用

**【问题描述】**设计速冻食品厂的冷库管理系统。主要功能是入库操作和出库操作。速冻食品厂冷库管理的特点：先入库的产品必须先出库。当天入库的不得出库。管理对象：速冻箱号。设共有n个速冻箱，循环使用。

**【主要功能】**

要求设计算法实现以下操作功能：

1. 入库操作：指定产品数量（以盘为单位），检查空箱数量是否足够。
2. 出库操作：指定产品数量，检查入库时间以及产品数量是否足够。
3. 查询操作：返回库中的产品数量，入库时间最长的产品数量和存放天数。
4. 菜单选择上述操作及退出。

主程序执行时循环显示菜单，用户选择后，转入相应功能函数中。功能函数执行完毕返回主程序后仍保持菜单状态，直到选择退出。

**【主要代码】**

头文件

#ifndef **COURSE\_DESIGN\_GUANLI\_H**#define **COURSE\_DESIGN\_GUANLI\_H**#include<iostream>  
using namespace std;  
  
const int MaxSize = 100; *// 最大速冻箱数量*const int MqSize = 100; *// 循环队列大小*class ColdStorageSystem {  
public:  
 ColdStorageSystem(); *// 构造函数* void inStock(int quantity,int day); *// 入库操作* void outStock(int quantity); *// 出库操作* void query(); *// 查询操作* void date(); *//增加天数* void showDate();  
  
private:  
 int foodbox[MaxSize]; *// 速冻箱集合* int front; *// 循环队列头指针* int rear; *// 循环队列尾指针* int today;  
};  
  
  
ColdStorageSystem::ColdStorageSystem() {  
 front = 0;  
 rear = 0;  
 for (int i = 0; i < MaxSize; i++) {  
 foodbox[i] = -1; *// 初始值为-1，表示速冻箱为空* }  
 today = 1;  
}  
void ColdStorageSystem::inStock(int quantity,int day) {  
 if ((rear + 1) % MaxSize == front) {  
 cout << "冷库已满，无法入库！" << endl;  
 return;  
 }  
  
 if (quantity <= MaxSize - 1 - rear + front) {  
 for (int i = 0; i < quantity; i++) {  
  
 foodbox[rear] = day; *// 记录入库日期* rear = (rear + 1) % MaxSize;  
 }  
 cout << "入库成功！" << endl;  
 } else {  
 cout << "速冻箱数量不足，无法入库！" << endl;  
 }  
  
}  
  
void ColdStorageSystem::outStock(int quantity) {  
 int flag = 0;  
 if (front == rear) {  
 cout << "冷库为空，无法出库！" << endl;  
 return;  
 }  
  
 if (quantity <= rear - front + 1 && foodbox[front] <= today) { *// 判断出库数量是否超过了库存量以及出库产品是否为当天入库* for (int i = 0; i < quantity; i++) {  
 if(foodbox[front] == today)  
 {  
 cout<<"出库失败，当天入库的不得出库"<<endl;  
 flag = 1;  
  
 break;  
 }  
 else  
 {  
 front = (front + 1) % MaxSize;  
 }  
 }  
 if(flag == 0)  
 {  
 cout << "出库成功！" << endl;  
 }  
 }  
 else {  
 cout << "冷库库存不足，无法出库！" << endl;  
 }  
}  
void ColdStorageSystem::query() {  
 if (front == rear) {  
 cout << "冷库为空！" << endl;  
 return;  
 }  
 int totalCount = 0;  
  
 for(int i = front;i<rear;i++)  
 {  
 totalCount++;  
 }  
 int cnt = foodbox[0];  
 for(int i =front;i<=rear;i++)  
 {  
  
 if(foodbox[i]>cnt) cnt = foodbox[i];  
 }  
 int maxDayCount = 0;  
 for(int i =front;i<= rear;i++)  
 {  
 if(foodbox[i] == cnt ) maxDayCount++;  
 }  
  
 cout << "冷库中产品数量：" << totalCount<< endl;  
 cout << "入库时间最长的产品数量：" << maxDayCount << endl;  
 cout << "入库时间最长的产品存放天数：" << today-cnt+1<< endl;  
}  
void ColdStorageSystem ::date() {  
 today++;  
}  
void ColdStorageSystem::showDate() {  
 cout<<"当前是第"<<today<<"天"<<endl;  
}  
#endif *//COURSE\_DESIGN\_GUANLI\_H*

源文件

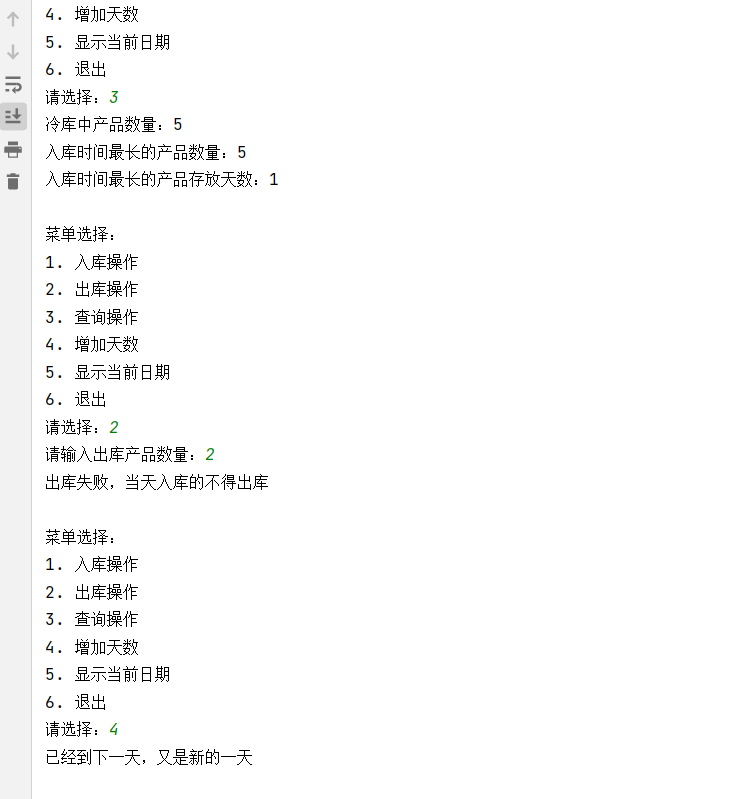
#include "guanli.h"  
#include <iostream>  
using namespace std;  
  
int main() {  
 ColdStorageSystem coldStorage;  
  
 int choice;  
 int quantity;  
 int day;  
  
 do {  
 cout << "菜单选择：" << endl;  
 cout << "1. 入库操作" << endl;  
 cout << "2. 出库操作" << endl;  
 cout << "3. 查询操作" << endl;  
 cout << "4. 增加天数"<<endl;  
 cout << "5. 显示当前日期"<<endl;  
 cout << "6. 退出" << endl;  
 cout << "请选择：";  
 cin >> choice;  
  
 switch (choice) {  
 case 1:  
 cout << "请输入入库产品数量："<<endl;  
 cin >> quantity;  
 cout<<"请输入入库时间: "<<endl;  
 cin>>day;  
 coldStorage.inStock(quantity,day);  
 break;  
 case 2:  
 cout << "请输入出库产品数量：";  
 cin >> quantity;  
 coldStorage.outStock(quantity);  
 break;  
 case 3:  
 coldStorage.query();  
 break;  
 case 4:  
 coldStorage.date();  
 cout<<"已经到下一天，又是新的一天"<<endl;  
 break;  
 case 5:  
 coldStorage.showDate();  
 break;  
 case 6:  
 cout << "已退出冷库管理系统。" << endl;  
 break;  
 default:  
 cout << "无效的选择，请重新输入。" << endl;  
 break;  
 }  
  
 cout << endl;  
  
 } while (choice != 6);  
  
 return 0;  
}

**【实验过程】**

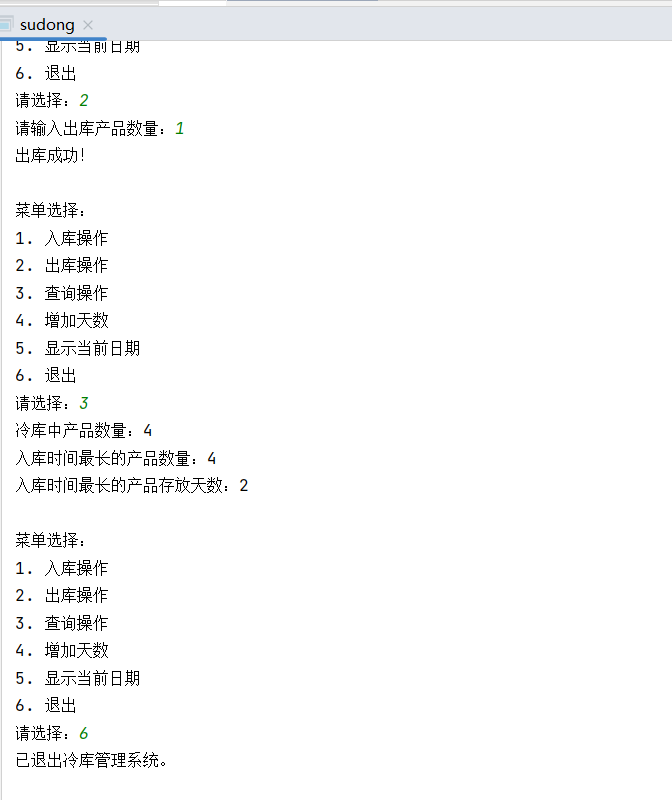
利用队列实现了简易版的速冻管理系统，利用队列先进先出的特性设计了入库，出库等操作，为了实现当天入库的货物不得出库，设计了一个foodBox数组，用于存放每个货物的进货日期，出库时如果货物的进货日期等于private中的today变量，不得出库。查询功能实现了对总货物，货物最长存放时间，最长存放时间对应的货物数量的查询。











**【实验体会】**

实现冷库管理系统主要利用了队列先进先出的功能，为了实现相应要求，花费了一些时间去构思，总体来说有一定难度，任务量稍大，不过受益良多。

# 8. 二叉树的操作

【实验简介】二叉树是树形结构的一种重要类型。通过本次实验，熟悉二叉树结点的结构，掌握二叉树的基本操作以及具体实现，学会利用递归方法编写对二叉树这种递归数据结构进行处理的算法。

【实验内容】

编写程序，实现对二叉树的以下操作：

1. 建立二叉树。
2. 按任一种遍历次序输出二叉树中的所有结点。
3. 求二叉树的深度。
4. 求二叉树中的所有节点数。
5. 求二叉树中的所有叶子节点数。
6. 清除二叉树，使之变成一只空树。

【主要代码】

BinaryTree.h

#ifndef **DATA\_STRUCTURE\_BINARYTREE\_H**#define **DATA\_STRUCTURE\_BINARYTREE\_H**#include<iostream>  
#include<iomanip>  
using namespace std;  
  
template<class T>  
struct Node{  
 T data;  
 Node \*left;  
 Node \*right;  
 Node():left(nullptr),right(nullptr){}  
 Node(T val):data(val),left(nullptr),right(nullptr){}  
};  
  
template<class T>  
class BinaryTree{  
private:  
 Node<T> \*root;  
 void destroy(Node<T> \*node) *//清除二叉树* {  
 if(node)  
 {  
 destroy(node->left);  
 destroy(node->right);  
 delete node;  
 }  
 }  
 void preOrder(Node<T> \*node) *//前序遍历* {  
 if(node)  
 {  
 cout<<node->data<<" ";  
 preOrder(node->left);  
 preOrder(node->right);  
 }  
 }  
 void inOrder(Node<T> \*node) *//中序遍历* {  
 if(node)  
 {  
 inOrder(node->left);  
 cout<<node->data<<" ";  
 inOrder(node->right);  
 }  
 }  
  
 void postOrder(Node<T> \*node) *//后序遍历* {  
 if(node)  
 {  
 postOrder(node->left);  
 postOrder(node->right);  
 cout<<node->data<<" ";  
 }  
 }  
 int getHeight(Node<T> \*node) *//求树的高度* {  
 if(!node)  
 {  
 return 0;  
 }  
 else{  
 int leftHeight = getHeight(node->left);  
 int rightHeight = getHeight(node->right);  
 return max(leftHeight,rightHeight)+1;  
 }  
 }  
  
 int getNodeCount(Node<T> \*node){ *//求节点数量* if(!node){  
 return 0;  
 }  
 else{  
 return getNodeCount(node->left) + getNodeCount(node->right) + 1;  
 }  
 }  
  
 void createBinaryTree(Node<T> \*node) {  
 cout<<"输入左子树"<<node->data<<"的左子树"<<" 输入-1结束"<<endl;  
 T data;  
 cin>>data;  
 if(data != -1)  
 {  
 node->left = new Node<T>(data);  
 createBinaryTree(node->left);  
 }  
  
 cout<<"输入右子树"<<node->data<<"的右子树"<<" 输入-1结束"<<endl;  
 cin>>data;  
 if(data != -1)  
 {  
 node->right = new Node<T>(data);  
 createBinaryTree(node->right);  
 }  
 }  
public:  
 BinaryTree():root(nullptr){}  
 ~BinaryTree(){  
 destroy(root);  
 }  
 bool isEmpty() {return (root == **NULL**)? true: false;}  
  
 void Insert(const T& val);  
 void preOrder()  
 {  
 preOrder(root);  
 cout<<endl;  
 }  
 void inOrder()  
 {  
 inOrder(root);  
 cout<<endl;  
 }  
 void postOrder()  
 {  
 postOrder(root);  
 cout<<endl;  
 }  
 int getHeight(){ return getHeight(root);}  
 int getNodeCount() { return getNodeCount(root);}  
 void createBinaryTree() {  
 std::cout << "Enter root value (or -1 for null): ";  
 T data;  
 std::cin >> data;  
 if (data == -1) {  
 root = nullptr;  
 } else {  
 root = new Node<T>(data);  
 createBinaryTree(root);  
 }  
 }  
  
 Node<T> \*getRoot() const{  
 return root;  
 }  
  
  
  
 int getLeafCount(Node<T> \*node, int count);  
};  
  
template<class T>  
void BinaryTree<T>::Insert(const T &val) {  
 if(!root)  
 {  
 root = new Node<T>(val);  
 }  
 else{  
 Node<T> \*cur = root;  
 while(1)  
 {  
 if(val<cur->data)  
 {  
 if(!cur->left)  
 {  
 cur->left = new Node<T>(val);  
 break;  
 }  
 cur = cur->left;  
 }  
 else{  
 if(!cur->right)  
 {  
 cur->right = new Node<T>(val);  
 break;  
 }  
 cur = cur->right;  
 }  
 }  
 }  
  
}  
  
template<class T>  
int BinaryTree<T>::getLeafCount(Node<T> \*node,int count) {  
 if(node != nullptr)  
 {  
 if(node->left == nullptr&& node->right == nullptr) count++;  
 count = getLeafCount(node->left,count);  
 count = getLeafCount(node->right,count);  
 }  
 return count;  
  
}  
  
  
#endif *//DATA\_STRUCTURE\_BINARYTREE\_H*

tree1.cpp  
  
#include "BinaryTree.h"  
  
  
int main() {  
 int count =0;  
 BinaryTree<int> tree;  
 tree.createBinaryTree();  
  
  
 cout << "前序遍历为: ";  
 tree.preOrder();  
  
 cout << "中序遍历为: ";  
 tree.inOrder();  
  
 cout << "后序遍历为: ";  
 tree.postOrder();  
  
 cout << "树的深度为: " << tree.getHeight() << endl;  
 cout << "树的节点数量为： " << tree.getNodeCount() << endl;  
 cout<<"树的叶子节点数量为： "<<tree.getLeafCount(tree.getRoot(),count)<<endl;  
  
 cout<<"插入元素4和3 "<<endl;  
 tree.Insert(4);  
 tree.Insert(3);  
  
  
 cout << "前序遍历为: ";  
 tree.preOrder();  
  
 cout << "中序遍历为: ";  
 tree.inOrder();  
  
 cout << "后序遍历为: ";  
 tree.postOrder();  
  
  
  
 return 0;  
}

【实验过程】

首先定义表示二叉树节点的结构体BinTreeNode，其次定义二叉树的类BinaryTree以及二叉树相关的操作。

函数实现时，先按前序遍历输入给定的二叉树，系统会给出此二叉树的前序遍历、中序遍历以及后序遍历。并给出二叉树的深度、节点个数以及叶节点个数。最后会删除二叉树。





【实验体会】

本次实验熟悉了二叉树的节点结构，掌握了二叉树的具体实现。区分了二叉树的前、中、后三种次序的遍历。并实现了前序遍历建立二叉树，计算二叉树的深度，节点数量，叶子节点数量，理解了三种遍历的特点。在二叉树的功能实现中，很多地方利用了递归的算法，对递归有了更好的认识.

# 9. 字符串

【实验简介】字符串是由零个或多个字符的顺序排列所组成的数据结构，字符串在计算机处理中使用非常广泛。通过本次实验理解字符串运算的原理，掌握主要算法的实现。

【实验内容】

建立字符串类，并实现求子串、字符串赋值、字符串连接等运算符重载函数，实现字符串的模式匹配功能。编写一个能够统计字符串中各个字符出现频度的函数。

【主要代码】

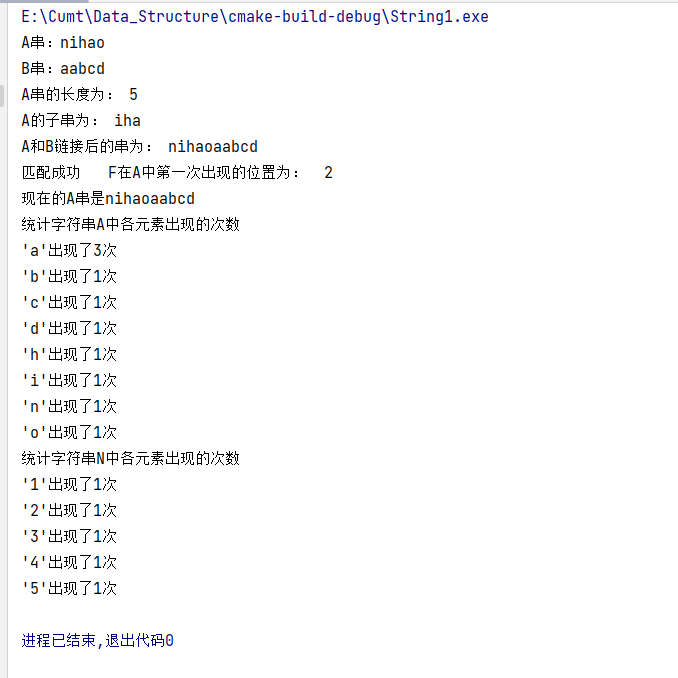
头文件#include<iostream>  
#include<string.h>  
#include<map>  
using namespace std;  
  
#ifndef **DATA\_STRUCTURE\_STRING\_H**#define **DATA\_STRUCTURE\_STRING\_H**#define **maxSize** 128  
  
class String{  
public:  
 String(int sz = **maxSize**); *//创建一个最大长度为maxSize，实际长度为0的串* String(const char \*init);  
 String(const String& ob); *//复制构造函数，用已有的ob创建一个新串* ~String()  
 {  
 delete []ch;  
 }  
 int Length()const  
 {  
 return curLength;  
 }  
 char& operator[](int i);  
 String& operator +=(String& ob); *//串链接* friend ostream &operator << (ostream &out,String &P); *//字符串输出* String& operator=(const String&ob);  
 String operator() (int pos,int len);  
 int Find(String& pat,int k);  
 void countChars(); *//统计字符串中每个字符出现的次数* char \*ch;  
 int curLength; *//字符串实际长度* int maxLength; *//存放数组的最大长度*};  
  
String::String(int sz) {  
 maxLength = sz;  
 ch = new char[maxLength +1];  
 if(ch == **NULL**){  
 cerr<<"分配错误"<<endl;  
 exit(1);  
 }  
 curLength = 0;  
 ch[0] = '\0';  
}  
  
String::String(const char \*init) {  
 int len = strlen(init);  
 maxLength = (len>**maxSize**)?len:**maxSize**;  
 ch = new char[maxLength+1];  
 if(ch == **NULL**){  
 cerr<<"分配错误"<<endl;  
 exit(1);  
 }  
 curLength = len;  
 strcpy(ch,init);  
}  
  
String::String(const String &ob) {  
 maxLength = ob.maxLength;  
 ch = new char[maxLength + 1];  
 if(ch == **NULL**)  
 {  
 cerr<<"存储分配失败"<<endl;  
 exit(1);  
 }  
 curLength = ob.curLength;  
 strcpy(ch,ob.ch);  
}  
String& String::operator=(const String &ob) {  
 if(&ob != this)  
 {  
 delete []ch;  
 ch = new char[ob.maxLength];  
 if(ch==**NULL**){  
 cerr<<"存储分配失败"<<endl;  
 exit(1);  
 }  
 curLength = ob.curLength;  
 strcpy(ch,ob.ch);  
 }  
 else cout<<"字符串自身赋值错误"<<endl;  
 return \*this;  
}  
  
  
char& String::operator[] (int i) {  
  
 if(i<0||i>=curLength)  
 {  
 cout<<"字符串下标超界限"<<endl;  
 exit(1);  
 }  
 return ch[i];  
}  
  
String& String::operator+=(String &ob) {  
 char \*tmp = ch;  
 int n = curLength+ob.curLength;  
 int m = (maxLength>=n)?maxLength:n;  
 ch = new char[m];  
 if(ch == **NULL**){  
 cerr<<"分配错误"<<endl;  
 exit(1);  
 }  
 maxLength = m; *//新字符串的最大长度 为m* curLength = n; *//新字符串的当前长度 n* strcpy(ch,tmp);  
 strcat(ch,ob.ch);  
 delete []tmp;  
 return \*this;  
}  
  
  
String String::operator()(int pos, int len) {  
 String tmp;  
 if(pos<0||pos+len>=maxLength||len<0)  
 {  
 tmp.curLength =0;  
 tmp.ch[0] = '\0';  
 }  
 else{  
 if(pos+len>=curLength) len = curLength-pos;  
 tmp.curLength = len;  
 for(int i =0,j = pos;i<len;i++,j++) tmp.ch[i] = ch[j];  
 tmp.ch[len] = '\0';  
 }  
 return tmp;  
}  
int String::Find(String &pat, int k) {  
 int i,j;  
 for(i = k;i<=curLength -pat.curLength;i++)  
 {  
 for(j = 0;j<pat.curLength;j++)  
 {  
 if(ch[i+j]!=pat.ch[j]) break;  
  
 }  
 if(j == pat.curLength) return i;  
  
 }  
 return -1;  
}  
  
ostream &operator << (ostream &out,String &P){ *//打印字符串* for(int i=0;i<P.curLength;i++)  
 cout<<P.ch[i];  
 return out;  
}  
  
void String::countChars() {  
 map<char,int>charCounts; *//利用STL容器map创建一个字符串每个元素，和元素出现的次数的一个键值对* for(int i =0 ;i<curLength;i++)  
 {  
 char c = ch[i];  
 charCounts[c]++;  
 }  
 for(map<char,int>::iterator it = charCounts.begin();it != charCounts.end();it++)  
 {  
 cout << "'" << it->first << "'出现了" << it->second << "次\n";  
 }  
}  
#endif *//DATA\_STRUCTURE\_STRING\_H*

源文件

#include "String.h"  
int main()  
{  
 char a[30] = "nihao"; *//数组存储的字符串* char b[6] = "aabcd";  
 char f[10] = "hao";  
 char c[10] = "abcLetabc";  
 char n[10] = "12345";  
 String A(a); *//定义为类对象的字符串* String B(b);  
 String C(c);  
 String F(f);  
 String N(n);  
 cout << "A串：" << A << endl;  
 cout << "B串：" << B << endl;  
 cout<<"A串的长度为： "<<A.Length()<<endl;  
 String D = A(1,3);  
 String E = A += B;  
 cout<<"A的子串为： "<<D<<endl;  
 cout<<"A和B链接后的串为： "<<E<<endl;  
 if(A.Find(F,0) != -1){  
 cout<<"匹配成功 "<<"F在A中第一次出现的位置为： "<<A.Find(F,0)<<endl; *//k表示从当前字符串第k个位置开始进行匹配* }  
 cout<<"现在的A串是"<<A<<endl;  
 cout<<"统计字符串A中各元素出现的次数"<<endl;  
 A.countChars();  
  
 cout<<"统计字符串N中各元素出现的次数"<<endl;  
 N.countChars();  
  
  
 return 0;  
}

【实验过程】

字符串类的建立是通过一个char\* 实现的，正确的写出了字符串的构造函数，编写了求字串，字符串赋值，运算，模式匹配等功能。



【实验体会】

在实现求解字符串中元素出现的次数功能时，我避免了用常规建立一个nums数组来统计字符串元素出现次数的数组，通过查阅资料，利用STL容器map实现字符和出现次数的一个键值对，通过遍历字符串，进行统计，代码简洁易懂，有学习了课外知识，受益良多。