**实验6：链表+文件1**

**姓名\_\_\_\_金宣成\_\_\_\_\_\_班级\_计科03\_\_学号\_202203151310\_**

* **请阅读此说明：实验6满分100分，做完实验后请按要求将代码和截图贴入该文档。然后将此文档、源代码文件（.hpp, .cpp）打包上传到学习通。**

**1、（总分60分）面向对象的链表类可以设计成：**

**typedef int DataType;**

**class node{**

**public:**

**node(DataType d,node\* ptr=nullptr) {data=d; next=ptr; }**

**DataType data;**

**node\* next;**

**};**

class linked\_list{

public:

linked\_list(); //无参构造，初始化一个空的链表（若定义为带哨兵的链表则默认包含一个结点）

linked\_list(const linked\_list& other); //拷贝构造

linked\_list& operator=(const linked\_list& right); //赋值重载

int list\_size() const; //求链表数据集中的数据规模

//集合并交叉也可以考虑设计为类外函数

linked\_list operator+(const linked\_list& right); //链表集合并

linked\_list operator-(const linked\_list& right);//链表集合差

linked\_list intersectionSet(const linked\_list& right); //链表集合交

node\* find(DataType value); //查找1 返回包含value值的结点地址，找不到返回空指针

bool find(DataType value,node\*& pre,node\*&p);

//查找2： 找到返回真：p为目标点，pre为前驱点; 找不到返回假：p和pre均为nullptr

void add\_front(DataType value); //添加到首

void add\_tail(DataType value);//添加到尾

void add\_pos\_after(DataType value, node \* pos); //添加到指定位置之后

void add\_pos\_before(DataType value,node \*pos);//添加到指定位置之前

void Delete(DataType value);//删除指定值

void delete\_pos\_after(node\* pos);//删除指定位置之后

void delete\_pos\_before(node\* pos);//删除指定位置之前

void delete\_all (DataType value);//删除所有包含value的结点

void delete\_repeat(); //将链表调整为无重复元素的集合

void reverse();//逆置链表

void sort();//升序排列当前链表

void display();//遍历链表，使用逗号间隔输出元素

~linked\_list(); //回收链表内的所有结点

private:

node\* head,\*tail; //表首、表尾

int size; //有效数据个数

}; //可以考虑设计 ，哨兵结点的信息可以代替size 存储表中结点的个数

* **实验要求：参考链表高级篇的讲义。补充完整我们描述的这个单向链表类。并通过给出的测试程序linked\_list\_demo。**

**在实现类的成员函数的过程中注意体会类成员函数和 课堂讨论过程化设计时的封装的独立函数 之间的区别。**

**可以为这个链表添加一个类外定义的普通函数，来完成有序表的合并操作：**

**linked\_lsit mergeSortedList(const linked\_list& L1, const linked\_list& L2 );**

**\*（选做）也可以使用模板类来描述这个单向链表类，则所有的定义系列变为模板：**

**template<class DataType>**

**class node{**

**public:**

**node(DataType d, node<DataType>\* ptr=nullptr) {data=d; next=ptr; }**

**DataType data;**

**node<DataType>\* next;**

**};**

**template<class DataType>**

class linked\_list{

public:

linked\_list(); //无参构造，初始化一个空的链表（若定义为带哨兵的链表则默认包含一个结点）

linked\_list(const **linked\_list<DataType>**& other); //拷贝构造

**linked\_list<DataType>**& operator=(const **linked\_list<DataType>**& right); //赋值重载

int list\_size() const; //求链表数据集中的数据规模

//集合并交叉也可以考虑设计为类外函数

**linked\_list<DataType>** operator+(const **linked\_list<DataType>**& right); //链表集合并

**linked\_list<DataType>** operator-(const **linked\_list<DataType>** & right);//链表集合差

**linked\_list<DataType>**  intersectionSet(const **linked\_list<DataType>**& right); //链表集合交

**node<DataType>**\* find(DataType value); //查找1 返回包含value值的结点地址，找不到返回空指针

bool find(DataType value, **node<DataType>**\*& pre,**node<DataType>**\*&p);

//查找2： 找到返回真：p为目标点，pre为前驱点; 找不到返回假：p和pre均为nullptr

void add\_front(DataType value); //添加到首

void add\_tail(DataType value);//添加到尾

void add\_pos\_after(DataType value, **node<DataType>** \* pos); //添加到指定位置之后

void add\_pos\_before(DataType value, **node<DataType>**\*pos);//添加到指定位置之前

void Delete(DataType value);//删除指定值

void delete\_pos\_after(**node<DataType>**\* pos);//删除指定位置之后

void delete\_pos\_before(**node<DataType>**\* pos);//删除指定位置之前

void delete\_all (DataType value);//删除所有包含value的结点

void delete\_repeat(); //将链表调整为无重复元素的集合

void reverse();//逆置链表

void sort();//升序排列当前链表

void display();//遍历链表，使用逗号间隔输出元素

~linked\_list(); //回收链表内的所有结点

private:

**node<DataType>**\* head,\*tail; //表首、表尾

int size; //有效数据个数

};

template<class DataType>

**linked\_list<DataType> mergeSortedList(const linked\_list<DataType>& L1, const linked\_list<DataType>& L2 );**

**或者：**

**template<class T>**

**T mergeSortedList(const T& L1, const T& L2 );**

* **测试程序：**

**//测试程序：**

**#include<iostream>**

**#include "linkedList.hpp"**

**using namespace std;**

**Int main()**

**{ linked\_list a1,a2, b,c; //若设计的类为模板，则采用linked\_list<int> a1,a2,b,c;**

**DataType data; //若设计的类为模板，则采用int data;**

**//正向和逆向建链测试**

**//输入2 ,6, 7, 3, 5, 9,12, 4 ,0**

**while(cin>>data){**

**if(data==0) break; //输入0结束**

**a1.add\_front(data);**

**a2.add\_tail(data);**

**}**

**a1.display(); //逆向链 4，12，9，5，3，7，6，2**

**a2.display(); //正向链2，6，7，3，5，9，12，4**

**//链表转置测试**

**//输入2 ,16, 3, 8, 15, 4, 9, 7 ,0**

**while(cin>>data) {**

**if(data==0) break; //输入0结束**

**b.add\_tail(data);**

**}**

**b.display(); //原始链表2,16,3,8,15,4,9,7**

**b.reverse();**

**b.display(); //转置结果7,9,4,15,8,3,16,2**

**c=a1+b; //测试集合并**

**c.display(); //4,12,9,5,3,7,6,2,15,8,16**

**c=a1-b; //测试集合差（属于a1且不属于b的数据）**

**c.display(); //12，5，6**

**c=a1.intersectionSet(b); //测试集合交**

**c.display(); //4,9,3,7,2**

**a1.sort(); //测试升序排序**

**a1.display(); //2,3,4,5,6,7,9,12**

**//思考需要降序排序如何做？**

**b.add\_tail(8); b.add\_tail(16); b.add\_tail(3); b.add\_front(3); b.add\_front(16);**

**b.display(); //16,3,7,9,4,15,8,3,16,2,8,16,3**

**b.delete\_all(16); //删除所有的16**

**b.display();//3,7,9,4,15,8,3,2,8,3**

**b.delete\_repeat(); //将b调整为无重复集合**

**b.display();//3,7,9,4,15,8,2**

**node\* pos=b.find(15); //若使用模板，则采用node<int>\* pos=b.find(15);**

**b. add\_pos\_after(18,pos);**

**b.display(); //3,7,9,4,15,18,8,2**

**b.add\_pos\_before(23,pos);**

**b.display();//3,7,9,4,23,15,18,8,2**

**b.delete\_pos\_after(pos);**

**b.display();//3,7,9,4,23,15,8,2**

**b.delete\_pos\_before(pos);**

**b.display();//3,7,9,4,15,8,2**

**b.Delete(7);**

**b.display();//3,9,4,15,8,2**

**b.Delete(8);**

**b.display();//3,9,4,15,2**

**b.sort();**

**b.display();//2,3,4,9,15**

**a2=mergeSortedList(a1,b);**

**a2.display(); //2,3,4,5,6,7,9,12,15**

**return 0;**

**}**

**实验解答：**

**❶黏贴链表类的实现代码：（标注你使用了基本单向链表 or带哨兵的单向链表？）**

**使用的链表类型：带哨兵的模板链**

**//file:linked\_list.cpp**

#include <iostream>

#include "linkedList.hpp"

using namespace std;

//无参构造函数

template <typename T>

linked\_list<T>::linked\_list()

{

    head = new node<T>;

    if (head == NULL)

    {

        cout << "内存分配失败！" << endl;

        exit(1);

    }

    tail = head;

    this->size = 0;

    tail->next = NULL;

    head->data = 0;

}

//拷贝构造函数

template <typename T>

linked\_list<T>::linked\_list(const linked\_list<T> &other){

    head = new node<T>;

    if (head == NULL)

    {

        cout << "内存分配失败！" << endl;

        exit(1);

    }

    this->size = other.size;

    node<T> \*p = other.head;

    node<T> \*q = this->head;

    p = p->next;

    while (p)

    {

        node<T> \*temp = new node<T>;

        temp->data = p->data;

        q->next = temp;

        q = q->next;

        p = p->next;

    }

    q->next = NULL;

    tail = q;

}

//赋值重载

template <typename T>

linked\_list<T>& linked\_list<T>::operator=(const linked\_list<T> &right){

    if(this == &right){

        return \*this;

    }//防止自赋值

    this->~linked\_list();//回收原有内存

    head = new node<T>;

    if (head == NULL)

    {

        cout << "内存分配失败！" << endl;

        exit(1);

    }

    this->size = right.size;

    node<T> \*p = right.head;

    node<T> \*q = this->head;

    p = p->next;

    while (p)

    {

        node<T> \*temp = new node<T>;

        temp->data = p->data;

        q->next = temp;

        q = q->next;

        p = p->next;

    }

    q->next = NULL;

    tail = q;

    return \*this;

}

//求链表数据集中的数据规模

template <typename T>

int linked\_list<T>::list\_size() const{

    return size;

}

//链表集合并

template <typename T>

linked\_list<T> linked\_list<T>::operator+(const linked\_list<T> &right){

    //遍历right，判断是否在this中存在，若不存在则插入

    linked\_list<T> temp(\*this);

    node<T> \*q = right.head->next;

    while(q){

        if(!temp.find(q->data)){

            temp.add\_tail(q->data);

            temp.size++;

        }

        q = q->next;

    }

    return temp;

}

//链表集合差

template <typename T>

linked\_list<T> linked\_list<T>::operator-(const linked\_list<T> &right){

    linked\_list<T> temp(\*this);

    node<T> \*p = right.head->next;

    node<T> \*q = NULL, \*pre = NULL;

    while(p){

        if(temp.find(p->data, pre, q)){

            pre->next = q->next;

            delete q;

            temp.size--;

            if(pre->next == NULL){

                temp.tail = pre;

            }

        }

        p = p->next;

    }

    return temp;

}

//链表集合交

template <typename T>

linked\_list<T> linked\_list<T>::intersectionSet(const linked\_list<T> &right){

    linked\_list<T> temp;

    node<T> \*p = this->head->next;

    while(p){

        if(right.find(p->data)){

            temp.add\_tail(p->data);

            temp.size++;

        }

        p = p->next;

    }

    return temp;

}

//查找1 返回包含value值的结点地址，找不到返回空指针

template <typename T>

node<T> \*linked\_list<T>::find(T value)const{

    node<T> \*p = head->next;

    while(p){

        if(p->data == value){

            return p;

        }

        p = p->next;

    }

    return NULL;

}

//查找2： 找到返回真：p为目标点，pre为前驱点; 找不到返回假：p和pre均为nullptr

template <typename T>

bool linked\_list<T>::find(T value, node<T> \*&pre, node<T> \*&p)const{

    pre = this->head;

    p = pre->next;

    while(p){

        if(p->data == value){

            return true;

        }else{

            pre = p;

            p = p->next;

        }

    }

    pre = p = NULL;

    return false;

}

//添加到首

template <typename T>

void linked\_list<T>::add\_front(T value){

    node<T> \*p = new node<T>;

    this->size++;

    p->next = head->next;

    head->next = p;

    p->data = value;

}

//添加到尾

template <typename T>

void linked\_list<T>::add\_tail(T value){

    node<T> \*p = new node<T>;

    this->size++;

    p->data = value;

    tail->next = p;

    tail = p;

}

//添加到指定位置之后

template <typename T>

void linked\_list<T>::add\_pos\_after(T value, node<T> \*pos){

    node<T> \*p = new node<T>;

    this->size++;

    p->data = value;

    p->next = pos->next;

    pos->next = p;

}

//添加到指定位置之前

template <typename T>

void linked\_list<T>::add\_pos\_before(T value, node<T> \*pos){

    //首先查找前驱点

    this->size++;

    node<T> \*p = this->head->next;

    node<T> \*pre = this->head;

    while(p){

        if(p == pos){

            break;

        }

        pre = p;

        p = p->next;

    }

    node<T> \*n = new  node<T>;

    n->data = value;

    n->next = pre->next;

    pre->next = n;

}

//删除指定值

template <typename T>

void linked\_list<T>::Delete(T value){

    this->size--;

    node<T> \*p = this->head->next;

    node<T> \*pre = this->head;

    this->find(value,pre,p);

    pre->next = p->next;

    delete p;

}

//删除指定位置之后

template <typename T>

void linked\_list<T>::delete\_pos\_after(node<T> \*pos){

    this->size--;

    node<T> \*p = pos->next;

    pos->next = p->next;

    delete p;

}

//删除指定位置之前

template <typename T>

void linked\_list<T>::delete\_pos\_before(node<T> \*pos){

    this->size--;

    node<T> \*p = this->head->next;

    node<T> \*pre = this->head;

    if(p == pos){

        cout << "无前驱点" << endl;

        return;

    }

    while(p){

        if(p->next == pos){

            break;

        }

        pre = p;

        p = p->next;

    }

    pre->next = p->next;

    delete p;

}

//删除所有包含value的结点

template <typename T>

void linked\_list<T>::delete\_all(T value){

    node<T> \*p = this->head->next;

    node<T> \*pre = this->head;

    while(p){

        if(p->data == value){

            pre->next = p->next;

            delete p;

            p = pre->next;

            this->size--;

        }else{

            pre = p;

            p = p->next;

        }

    }

}

//将链表调整为无重复元素的集合

template <typename T>

void linked\_list<T>::delete\_repeat(){

    node<T> \*p = this->head->next->next;

    if(!p) return;//如果只有一个数就不可能重复

    node<T> \*pre = this->head->next;

    node<T> \*temp = this->head->next;

    int flag;

    while(p){

        flag = 1;//是否要删除p

        while(temp != p){

            if(temp->data == p->data){

                flag = 0;

                break;

            }else{

                temp = temp->next;

            }

        }

        if(flag){

            pre = p;

            p = p->next;

            if(p == NULL){

                tail = pre;

            }

        }else{

            pre->next = p->next;

            if(p == tail){

                tail = pre;

            }

            delete p;

            this->size--;

            p = pre->next;

        }

        temp = this->head->next;//temp永远从第一个数开始，遍历数组，看是不是要把p删掉

    }

}

//逆置链表

template <typename T>

void linked\_list<T>::reverse(){

    node<T> \*p = this->head->next;

    node<T> \*pre = this->head;

    node<T> \*newhead = new node<T>;

    int flag = 0;

    while(p){

        pre = p;

        node<T> \*temp = new node<T>(p->data);

        temp->next = newhead->next;

        if(flag == 0){

            tail = temp;

            flag = 1;

        }

        newhead->next = temp;

        p = p->next;

        delete pre;

    }

    delete this->head;

    this->head = newhead;

}

//升序排列当前链表

template <typename T>

void linked\_list<T>::sort(){

    node<T> \*p = this->head->next;

    node<T> \*mini = this->head->next;

    node<T> \*q = this->head->next;

    while(p){

        while(q){

            if(q->data < mini->data){

                mini = q;

            }

            q = q->next;

        }

        if(mini != p){

            T temp = mini->data;

            mini->data = p->data;

            p->data = temp;

        }

        p = p->next;

        mini = p;

        q = p;

    }

}

//遍历链表，使用逗号间隔输出元素

template <typename T>

void linked\_list<T>::display(){

    node<T> \*p = head->next;

    while(p){

        cout << p->data;

        if(p->next){

            cout << ",";

        }

        p = p->next;

    }

    cout << endl;

}

//回收链表内的所有结点

template <typename T>

linked\_list<T>::~linked\_list(){

    node<T> \*p = this->head->next;

    node<T> \*pre = this->head;

    while(p){

        pre = p;

        p = p->next;

        delete pre;

    }

    delete this->head;

}

//合并并排序两个序列

template<class T>

linked\_list<T> mergeSortedList(const linked\_list<T>& L1, const linked\_list<T>& L2 ){

    node<T> \* p = L1.head->next;

    node<T> \* q = L2.head->next;

    linked\_list<T> temp ;

    while(p and q){

        if(p->data < q->data){

            temp.add\_tail(p->data);

            temp.size++;

            p = p->next;

        }else if (p->data > q->data)

        {

            temp.add\_tail(q->data);

            temp.size++;

            q = q->next;

        }else{

            temp.add\_tail(q->data);

            temp.size++;

            p = p->next;

            q = q->next;

        }

    }

    if(p){

        while(p){

            temp.add\_tail(p->data);

            temp.size++;

            p = p->next;

        }

    }else if(q){

        while(q){

            temp.add\_tail(q->data);

            temp.size++;

            q = q->next;

        }

    }

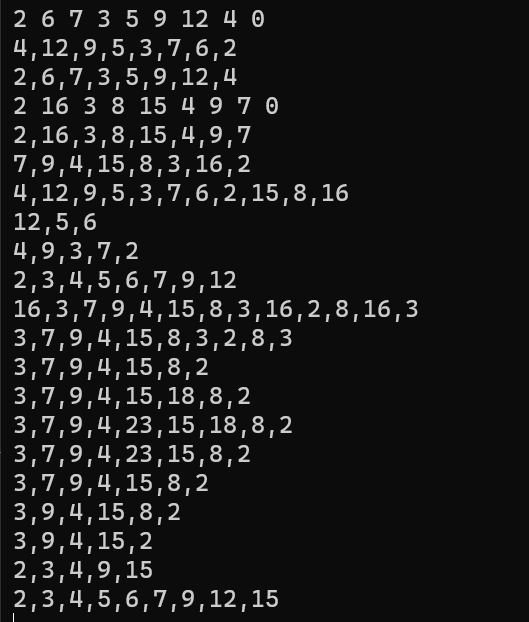
    return temp;

}

template class linked\_list<int>;

template linked\_list<int> mergeSortedList<int>(linked\_list<int> const&, linked\_list<int> const&);

**❷黏贴程序测试运行结果窗口（运行结果截屏）：**

****

**2、文件部分实验：学会ASCII文件的基本输入输出流读写（40分）**

**（1）写一个程序实现：随机生成两批数据，每批10个整数，范围自定。将第一组数据随机数写入 data1.txt；第二组随机书写入data2.txt （20分）**

**（2）写一个程序实现：读取data1.txt和data2.txt的共20个整数有序插入到数组a，再将数组a的结果输出到屏幕同时写入文件data3.txt存档。 （20分）**

**实验解答：**

**❶黏贴（1）的实现代码：**

#include <fstream>

#include <iostream>

using namespace std;

int main(){

    ofstream outfile1("../TXT/data1.txt");

    ofstream outfile2("../TXT/data2.txt");//这里用相对路径，相对于可执行文件的路径，对不同workspace的同学可能有不同的结果

    if (!outfile1 or !outfile2)

    {

        cerr << "open error!" << endl;

        abort(); //退出程序

    }

    //随机生成10个整数输入到两个文件中

    for(int i=0;i<10;i++){

        int temp=rand()%100;

        outfile1<<temp<<" ";

    }

    for(int i=0;i<10;i++){

        int temp=rand()%100;

        outfile2<<temp<<" ";

    }

    return 0;

}

**❷黏贴（2）的实现代码：**

#include <fstream>

#include <iostream>

#include <algorithm>

using namespace std;

int main()

{

    ifstream infile1("../TXT/data1.txt");

    ifstream infile2("../TXT/data2.txt");

    ofstream outfile("../TXT/data3.txt");

    int a[20];

    int b[10], c[10];

    for (int i = 0; i < 10; i++)

    {

        infile1 >> b[i];

        infile2 >> c[i];

    }

    sort(b, b + 10);

    sort(c, c + 10);

    int p = 0, q = 0;

    for (int i = 0; i < 20; i++)

    {

        if (b[p] < c[q] or q == 10)

        {

            a[i] = b[p];

            p++;

        }

        else

        {

            a[i] = c[q];

            q++;

        }

    }

    for (int i = 0; i < 20; i++)

    {

        outfile << a[i] << " ";

    }

}