**实验6：链表+文件1**

**姓名：陈王子**

**班级：大数据分析2101班**

**学号：202103150503**

**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

}; //可以考虑设计带哨兵的链表，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： *已经实现***

**实验解答：**

**❶链表类的实现代码：（我使用了带哨兵的单向链表）**

//测试程序：

/\*

#include<iostream>

#include "linkedList.hpp"

\*/

#include <iostream>

#include <unordered\_set>

using namespace std;

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)const; //查找1 返回包含value值的结点地址，找不到返回空指针

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

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

    //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

    node<DataType>\* get\_head() const {

        return head;

    }

private:

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

    int size; //有效数据个数

};

//无参构造，初始化一个空的链表

template<class DataType>

linked\_list<DataType>::linked\_list() {

    head = nullptr;

    tail = nullptr;

    size = 0;

}

//拷贝构造

template<class DataType>

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

    if (other.head == nullptr) {    // 如果被复制链表为空，则创建一个空链表对象

        head = tail = nullptr;

        size = 0;

        return;

    }

    head = new node<DataType>(other.head->data);

    node<DataType>\* p = head;    //指向当前结点

    node<DataType>\* q = other.head->next;    //指向被复制链表的下一个结点

    while (q != nullptr) {

        p->next = new node<DataType>(q->data);

        p = p->next;

        q = q->next;

    }

    tail = p;

    size = other.size;

}

//赋值重载

template<class DataType>

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

    if (&right == this) {

        return \*this;

    }

    node<DataType>\* p = head;

    while (p != nullptr) {

        node<DataType>\* tmp = p->next;

        delete p;

        p = tmp;

    }

    head = new node<DataType>(right.head->data);

    node<DataType>\* q = head; //指向当前结点

    node<DataType>\* r = right.head->next; //指向被复制链表的下一个结点

    while (r != nullptr) {

        q->next = new node<DataType>(r->data);

        q = q->next;

        r = r->next;

    }

    tail = q;

    size = right.size;

    return \*this;

}

//求链表数据集中的数据规模

template<class DataType>

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

    return size;

}

template<class DataType>

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

    linked\_list<DataType> result;

    std::unordered\_set<DataType> hash\_set;

    node<DataType>\* p = head;

    while (p != nullptr) {

        if (hash\_set.count(p->data) == 0) {    // 如果哈希表中没有该元素，则添加

            hash\_set.insert(p->data);

            result.add\_tail(p->data);

        }

        p = p->next;

    }

    p = right.head;

    while (p != nullptr) {

        if (hash\_set.count(p->data) == 0) {   // 如果哈希表中没有该元素，则添加

            hash\_set.insert(p->data);

            result.add\_tail(p->data);

        }

        p = p->next;

    }

    return result;

}

//链表集合差

template <typename DataType>

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

    linked\_list<DataType> result;

    node<DataType>\* p = head;

    unordered\_set<DataType> hash\_table;

    // 将 right 链表中的元素插入哈希表中

    for (node<DataType>\* q = right.head; q != nullptr; q = q->next) {

        hash\_table.insert(q->data);

    }

    while (p != nullptr) {

        // 如果当前节点不在 right 链表中，则将其添加到结果链表中

        if (hash\_table.find(p->data) == hash\_table.end()) {

            result.add\_tail(p->data);

        }

        p = p->next;

    }

    return result;

}

//链表集合交

template<class DataType>

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

    linked\_list<DataType> result;

    node<DataType>\* p = head;

    while (p != nullptr) {

        if (right.find(p->data)) {

            result.add\_tail(p->data);

        }

        p = p->next;

    }

    return result;

}

//查找1： 返回包含value值的结点地址，找不到返回空指针

template<class DataType>

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

    node<DataType>\* p = head;

    while (p != nullptr) {

        if (p->data == value) {

            return p;

        }

        p = p->next;

    }

    return nullptr;

}

//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<class DataType>

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

    pre = nullptr;

    p = head;

    while (p != nullptr) {

        if (p->data == value) {

            return true;

        }

        pre = p;

        p = p->next;

    }

    pre = nullptr;

    p = nullptr;

    return false;

}

//添加到首

template<class DataType>

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

    node<DataType>\* tmp = new node<DataType>(value);

    tmp->next = head;

    head = tmp;

    if (tail == nullptr) {

        tail = head;

    }

    size++;

}

//添加到尾

template<class DataType>

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

    if (head == nullptr) {

        add\_front(value);

        return;

    }

    node<DataType>\* tmp = new node<DataType>(value);

    tail->next = tmp;

    tail = tmp;

    size++;

}

//添加到指定位置之后

template<class DataType>

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

    if (pos == nullptr) {

        return;

    }

    node<DataType>\* tmp = new node<DataType>(value);

    tmp->next = pos->next;

    pos->next = tmp;

    if (tmp->next == nullptr) {

        tail = tmp;

    }

    size++;

}

//添加到指定位置之前

template<class DataType>

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

    if (pos == nullptr) {

        return;

    }

    if (pos == head) {

        add\_front(value);

        return;

    }

    node<DataType>\* pre = head;

    while (pre != nullptr && pre->next != pos) {

        pre = pre->next;

    }

    if (pre == nullptr) {

        return;

    }

    node<DataType>\* tmp = new node<DataType>(value);

    tmp->next = pos;

    pre->next = tmp;

    size++;

}

//删除指定值

template<class DataType>

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

    node<DataType>\* pre, \* p;

    if (find(value, pre, p)) {

        if (p == head) {

            head = p->next;

        }

        else {

            pre->next = p->next;

        }

        if (p == tail) {

            tail = pre;

        }

        delete p;

        size--;

    }

}

//删除指定位置之后

template<class DataType>

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

    if (pos == nullptr || pos == tail) {

        return;

    }

    node<DataType>\* tmp = pos->next;

    pos->next = tmp->next;

    if (tmp == tail) {

        tail = pos;

    }

    delete tmp;

    size--;

}

//删除指定位置之前

template<class DataType>

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

    if (pos == nullptr || pos == head) {

        return;

    }

    node<DataType>\* pre = head;

    while (pre->next != pos) {  // 在一次遍历中找到 pos 的前一个节点 pre

        pre = pre->next;

    }

    node<DataType>\* prepre = head;

    while (prepre->next != pre) {  // 在一次遍历中找到 pos 的前前一个节点 prepre

        prepre = prepre->next;

    }

    prepre->next = pre->next;  // 修改 pre 的 next 指针

    if (pre == tail) {  // 如果 pos 是尾节点，则需要更新 tail 指针

        tail = prepre;

    }

    delete pre;  // 删除 pos 节点

    size--;

}

//删除所有包含value的结点

template<class DataType>

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

    node<DataType>\* pre, \* p;

    while (find(value, pre, p)) {

        if (p == head) {

            head = p->next;

        }

        else {

            pre->next = p->next;

        }

        if (p == tail) {

            tail = pre;

        }

        delete p;

        size--;

    }

}

//将链表调整为无重复元素的集合

template<class DataType>

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

    node<DataType>\* curr = head;

    while (curr != nullptr && curr->next != nullptr) {  // 外层循环遍历链表

        node<DataType>\* p = curr;

        while (p->next != nullptr) {  // 内层循环遍历当前节点后面的所有节点

            if (p->next->data == curr->data) {  // 如果找到值相同的节点，则删除该节点

                node<DataType>\* temp = p->next;

                p->next = temp->next;

                delete temp;

            }

            else {

                p = p->next;

            }

        }

        curr = curr->next;

    }

}

//逆置链表

template<class DataType>

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

    if (head == nullptr || head == tail) {

        return;

    }

    node<DataType>\* pre = nullptr;

    node<DataType>\* p = head;

    node<DataType>\* nxt = p->next;

    while (p != nullptr) {

        p->next = pre;

        pre = p;

        p = nxt;

        if (nxt != nullptr) {

            nxt = nxt->next;

        }

        else {

            tail = head;

        }

    }

    head = pre;

}

//升序排列当前链表

template<class DataType>

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

    if (head == nullptr || head == tail) {

        return;

    }

    node<DataType>\* p = head;

    while (p != nullptr) {

        node<DataType>\* q = p->next;

        while (q != nullptr) {

            if (q->data < p->data) {

                std::swap(p->data, q->data);

            }

            q = q->next;

        }

        p = p->next;

    }

}

//遍历链表，使用逗号间隔输出元素

template<class DataType>

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

    node<DataType>\* p = head;

    while (p != nullptr) {

        std::cout << p->data;

        p = p->next;

        if (p != nullptr) {

            std::cout << ",";

        }

    }

    std::cout << std::endl;

}

//回收链表内的所有结点

template<class DataType>

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

    node<DataType>\* p = head;

    while (p != nullptr) {

        node<DataType>\* tmp = p->next;

        delete p;

        p = tmp;

    }

}

template<class DataType>

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

    linked\_list<DataType> result;

    node<DataType>\* p1 = L1.get\_head();

    node<DataType>\* p2 = L2.get\_head();

    while (p1 != nullptr && p2 != nullptr) {

        if (p1->data < p2->data) {  // 如果 p1 的值更小，则将 p1 插入到 result 中

            result.add\_tail(p1->data);

            p1 = p1->next;

        }

        else if (p2->data < p1->data) {  // 如果 p2 的值更小，则将 p2 插入到 result 中

            result.add\_tail(p2->data);

            p2 = p2->next;

        }

        else {  // 如果 p1 和 p2 的值相等，则将 p1 插入到 result 中，并让 p1 和 p2 都向前移动一位

            result.add\_tail(p1->data);

            p1 = p1->next;

            p2 = p2->next;

        }

    }

    // 下面两个循环处理未被遍历完的链表

    while (p1 != nullptr) {

        result.add\_tail(p1->data);

        p1 = p1->next;

    }

    while (p2 != nullptr) {

        result.add\_tail(p2->data);

        p2 = p2->next;

    }

    return result;

}

int main()

{

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

    int data; //若设计为模板类，这此处直接使用int data;

   //正向和逆向建链测试

   //输入2 ,6, 7, 3, 5, 9,12, 4 ,0

    //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

    //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<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

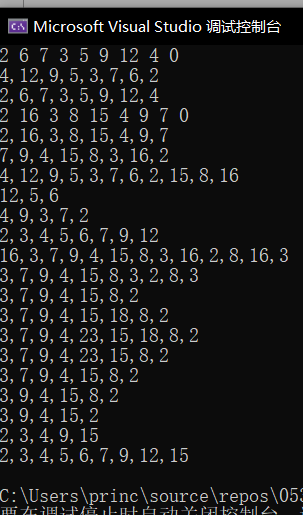
    a2 = mergeSortedList(a1, b);//?

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

    return 0;

}

**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 <iostream>

#include <fstream>

#include <random>

int main()

{

    std::default\_random\_engine rng(std::random\_device{}());

    std::uniform\_int\_distribution<int> dist(1, 100);  // 生成 1 到 100 的随机整数

    const int n = 10;  // 数组长度为 10

    int data1[n], data2[n];

    // 生成第一组数据

    for (int i = 0; i < n; ++i) {

        data1[i] = dist(rng);  // 生成随机数

    }

    // 将第一组数据写入文件 data1.txt

    std::ofstream fout1("data1.txt");

    if (fout1.is\_open()) {

        for (int i = 0; i < n; ++i) {

            fout1 << data1[i] << " ";  // 写入数字和空格

        }

        fout1.close();

    }

    else {

        std::cerr << "Failed to open file: data1.txt" << std::endl;

        return 1;

    }

    // 生成第二组数据

    for (int i = 0; i < n; ++i) {

        data2[i] = dist(rng);  // 生成随机数

    }

    // 将第二组数据写入文件 data2.txt

    std::ofstream fout2("data2.txt");

    if (fout2.is\_open()) {

        for (int i = 0; i < n; ++i) {

            fout2 << data2[i] << " ";  // 写入数字和空格

        }

        fout2.close();

    }

    else {

        std::cerr << "Failed to open file: data2.txt" << std::endl;

        return 1;

    }

    std::cout << "Data generated and saved successfully." << std::endl;

    return 0;

}

**❷黏贴（2）的实现代码：**

#include <iostream>

#include <fstream>

#include <algorithm>

int main()

{

    const int n = 20;  // 数组长度为 20

    int a[n] = {0};    // 初始化数组为 0

    // 读取第一组数据

    std::ifstream fin1("data1.txt");

    if (fin1.is\_open()) {

        for (int i = 0; i < 10; ++i) {

            fin1 >> a[i];  // 读取数字并赋值给数组元素

        }

        fin1.close();

    }

    else {

        std::cerr << "Failed to open file: data1.txt" << std::endl;

        return 1;

    }

    // 读取第二组数据

    std::ifstream fin2("data2.txt");

    if (fin2.is\_open()) {

        for (int i = 10; i < 20; ++i) {

            fin2 >> a[i];  // 读取数字并赋值给数组元素

        }

        fin2.close();

    }

    else {

        std::cerr << "Failed to open file: data2.txt" << std::endl;

        return 1;

    }

    // 对数组进行排序

    std::sort(a, a + n);

    // 输出数组到屏幕

    for (int i = 0; i < n; ++i) {

        std::cout << a[i] << " ";  // 输出数字和空格

    }

    std::cout << std::endl;

    // 将数组写入文件 data3.txt

    std::ofstream fout("data3.txt");

    if (fout.is\_open()) {

        for (int i = 0; i < n; ++i) {

            fout << a[i] << " ";  // 写入数字和空格

        }

        fout.close();

    }

    else {

        std::cerr << "Failed to open file: data3.txt" << std::endl;

        return 1;

    }

    std::cout << "Data processed and saved successfully." << std::endl;

    return 0;

}