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：

0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>
```

```cpp
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++;
}
```

```cpp
//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;
    }
}
```

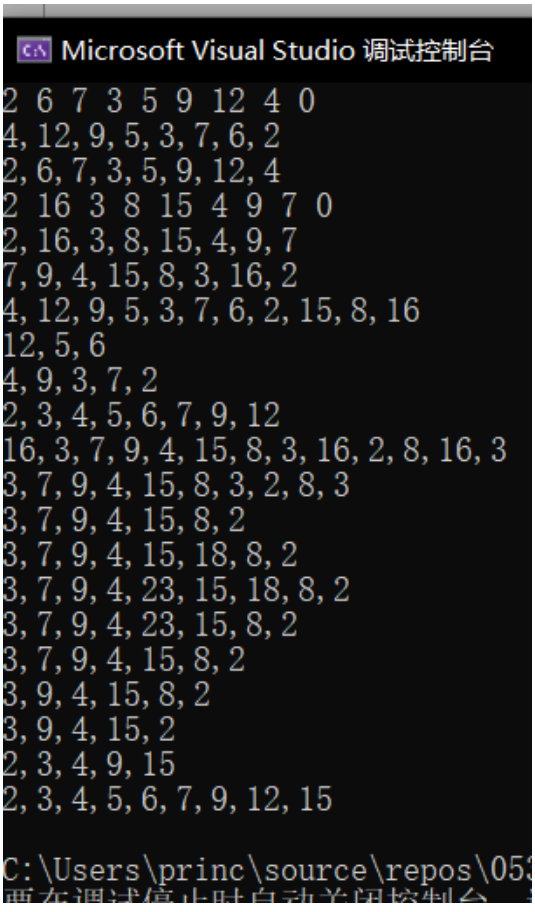
```cpp
//逆置链表
template<class DataType>
void linked_list<DataType>::reverse() {
    if (head == nullptr || head == tail) {
        return;
    }
    node<DataType>* pre = nullptr;
    node<DataType>* p = head;
    node<DataType>* nxt = p->next;
    while (p != nullptr) {
        p->next = pre;
        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;
}
```

0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：

0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;
}
```

0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;
}
```