《数据结构》上机报告

2019 年 9 月 27 日

姓名：李佳庚 学号：1852409 班级：计算机1班 得分：

<table>
<tr><td>实验题目</td><td colspan="2">链表</td></tr>
<tr><td>问题描述</td><td colspan="2">链表是顺序表的一种，是一种物理结构上不连续，但是逻辑结构中连续的存储结构。链表中的元素是通过指针依次连接而使其保持逻辑上连续的。
作业中要求使用链表解决插入、删除、按值查找、索引查找、逆序、合并等操作。</td></tr>
<tr><td rowspan="2">基本要求</td><td colspan="2">1. 掌握线性表的链式表示（单链表、循环链表、双向循环链表）；

2. 掌握链表实现线性表的基本操作，如建立、查找、插入、删除以及去重等；

3. 掌握有序线性表的插入、删除、合并操作；

4. 程序要添加适当的注释，程序的书写要采用 缩进格式 。

5. 程序要具在一定的 健壮性，即当输入数据非法时， 程序也能适当地做出反应，如 插入删除时指定的位置不对 等等。

6. 程序要做到界面友好，在程序运行时用户可以根据相应的提示信息进行操作。

7. 根据实验报告模板详细书写实验报告，在实验报告中给出主要算法的复杂度分析。

8. 给出逆置的算法和去重算法的流程图和复杂度分析。</td></tr>
<tr><td>已完成基本内容（序号）：</td><td>1,2,3,4,5,6,7,8</td></tr>
<tr><td rowspan="2">选做要求</td><td colspan="2">1. 使用少量现代 C++内容
2. 编写带头结点的双向链表
3. 降低 DLL 类耦合度
4. 提升 DLL 的健壮性</td></tr>
<tr><td>已完成选做内容（序号）</td><td>1，2，3，4</td></tr>
</table>

数据结构设计

```
template <typename T>
class DLL {

public:

    /*************************************
    嵌套类:     DLLNode
    作  用:     用来表示DLL的一个节点
    *************************************/
    template <typename T1>
    class DLLNode {
    public:
        T1          data;
        DLLNode*    next;
        DLLNode*    prev;

        DLLNode() = default;
    };



private:

    DLLNode<T>* head;
    size_t      length;
```

整个 DLL 链表拥有一个头节点，而每个节点的数据类型都为一个名字叫做 DLLNode 的 class。
单独的 DLLNode 这个 class 中还内置有数据域 data，前驱 prev，后驱 next。
当构造 DLL 链表时，会首先构造一个头节点。

这个头节点指向的是第一节点。含数据的节点从第一节点开始。

功能(函数)说明

（由于代码中注释写的十分详尽，所以不加更多的复杂描述）

```
    /*************************************
    函数名称:     DLL普通构造函数
    作    用:     形成一个空节点
    *************************************/
explicit DLL()
    : head(new DLLNode<T>), length(1) {
    head->next = head->prev = head;
}


    /*************************************
    函数名称:     DLL复制构造函数
    作    用:     完全复制一个双向链表
    *************************************/
```

```
explicit DLL(const DLL<T> &copied)
    : head(nullptr), length(0)
{

    DLLNode<T> *copyWork = &copied.getNode(0);
    DLLNode<T> *newWork = head = new DLLNode<T>;
    // 有关length，没生成一个节点，紧接着就++，保持一致
    length++;

    // 连续赋值，如普通构造函数
    head->next = head->prev = head;
    // 此次给第一个节点赋值
    // 于是后面的循环逻辑为：分配空间->赋值->分配空间->赋值……
    newWork->data = copied.getNode(0).data;

    // 当本链表length仍小于传入链表length时，说明还需要继续循环
    for (; length < copied.getLength();
        ++length,
        newWork = newWork->next, copyWork = copyWork->next) {

        newWork->next = new DLLNode<T>;
        newWork->next->prev = newWork;
        newWork->next->next = nullptr;
        newWork->next->data = copied.getNode(length).data;

    }
}
```

```
    /*************************************
    函数名称：    isEmpty
    输入参数：    void
    实现功能：    检验链表是否为空
    返回参数：    1/0
    *************************************/
constexpr bool isEmpty()                            const {
    return  (head->next == head && head->prev == head)
        ? true : false;
}
```

```
    /*************************************
    函数名称：    getLength
    输入参数：    void
    实现功能：    返回链表长度
    返回参数：    int
    *************************************/
constexpr size_t getLength()                        const {
    return length;
}
```

```
    /*************************************
    函数名称：    display
    输入参数：    void
    实现功能：    依次展示列表元素
    返回参数：    int
    *************************************/
```

```cpp
void display(const char *mode = "node", const char *beg = "sentinel")   const {

    int index = 0;
    DLLNode<T> *work;
    work = (0 == strcmp(beg, "sentinel")) ? head : head->next;
    if (0 == strcmp(mode, "node"))
        for (; work != nullptr; work = work->next)
            std::cout << "DLL[" << index++ << "]: " << work->data << '\n';

    if (0 == strcmp(mode, "num"))
        for (; work != nullptr; work = work->next)
            std::cout << work->data << " ";

    std::cout << '\n';

}
```

```
/************************************
函数名称：    push
输入参数：    T data
实现功能：    将data_封装成DLLNode后插入
返回参数：    void
************************************/
```

```cpp
void push(T data_)                                  noexcept {

    DLLNode<T> *work = head;
    for (size_t index = 0; index < length - 1; ++index)
        work = work->next;

    work->next = new DLLNode<T>;
    work->next->data = data_;
    work->next->next = nullptr;     work->next->prev = work;

    length++;

}
```

```
/************************************
函数名称：    insert
输入参数：    index, data_
实现功能：    在index对应位置插入data_的
              DLLNode对象
              插入的DLLNode的位置为index
返回参数：    void
************************************/
```

```cpp
void insert(const size_t index, T data_)            noexcept {

    // 不能在哨兵节点之前插入，但是可以在最后一个节点后插入
    // 本函数index是几，那么插入之后本次插入节点对应的就是几
    if (index > length || index <= 0) {
        std::cout << "insert index overflow" << '\n';
        return;
    }

    // 到达指定插入位置的前一个节点处
    // 如此，插入在work停留的节点后面
    // 新节点便在index对应的位置
    DLLNode<T> *work = head;
    DLLNode<T> *workNext;
    for (size_t i = 0; i < index - 1; i++)
        work = work->next;

    // 新建的指针可以指向work本来的下一个节点
    // work和worknext之间将插入一个节点
    // 但是，可能只有一个哨兵节点，所以要进行判定
    // 而只有哨兵节点时的特征就是 work->next == work
    // 一旦workNext被赋为nullptr，那么就代表要开始 insert 操作了
    workNext = (work->next == work) ? nullptr : work->next;

    work->next = new DLLNode<T>;
    work->next->data = data_;
    work->next->next = workNext;    work->next->prev = work;

    // 还要让新插入节点的下一个节点的指针指向新插入节点
    // 要注意的是，有可能 work->next->next 为 nullptr
    if (work->next->next)
        work->next->next->prev = work->next;

    length++;

}
```

```
/*************************************
函数名称：    insert
输入参数：    index, data_
实现功能：    在index对应位置插入data_的
              DLLNode对象
              插入的DLLNode的位置为index
返回参数：    void
*************************************/
```

```cpp
void insertSeq(const T data_, const char *mode = "ASC") {

    // 如果双向链表里面还没有节点
    DLLNode<T> *work = head;
    if (work->next == work) {
        push(data_);
        return;
    }

    work = work->next;
    size_t index = 1;
    if (0 == strcmp(mode, "ASC"))
        while (index < length && data_ <= work->data) {
            work = work->next;
            index++;
        }
    else if ( 0 == strcmp(mode, "DESC"))
        while (index < length && data_ >= work->data) {
            work = work->next;
            index++;
        }
    else;

    insert(index, data_);
    return;

}
```

```cpp
    /*************************************
    函数名称：    getNode
    输入参数：    index
    实现功能：    获得index对应位置的节点
    返回参数：    DLLNode<T> 对象
    *************************************/
constexpr DLLNode<T> getNode(const size_t index)    const {

    DLLNode<T> *work = head;
    for (size_t i = 0; i < index; i++)
        work = work->next;

    return *work;
}
```

```cpp
    /*************************************
    函数名称：    isExist
    输入参数：    value
    实现功能：    判断值为value的节点是否存在
    返回参数：    1 / 0
    *************************************/
constexpr bool isExist(const T value)              const {

    for (DLLNode<T> *work = head; work != nullptr; work = work->next)
        if (work->data == value)
            return true;

    return false;
}
```

```
/************************************
函数名称：    searchByValue
输入参数：    value
实现功能：    获得value对应位置的节点
返回参数：    size_t index
************************************/
size_t searchByValue(const T value)                const {

    DLLNode<T> *work = head;
    int index = 0;
    for (; index < length; index++, work = work->next)
        if (work->data == value)
            return index;

    return false;
}


/************************************
函数名称：    merge
输入参数：    DLL<T> first, second
实现功能：    合并某两个顺序DLL给本对象
返回参数：    void
************************************/
void merge(const DLL<T> &first, const DLL<T> &second) {

    // merge必须是一个空表
    if (!isEmpty())
        return;

    // 直接从第一节点开始
    DLLNode<T> *first_work = first.head->next;
    DLLNode<T> *second_work = second.head->next;
```

```
    // 如果 first 和 second 都为空表，那么直接 return，不多bb
    if (first.isEmpty() && second.isEmpty())
        return;
    // 如果 first 是空的，直接把 second 全 push 了
    else if (first.isEmpty())
        for (size_t index = 0; index < second.length - 1; index++, second_work = second_work->next)
            push(second_work->data);
    // 如果 second 是空的，直接把 first 全 push 了
    else if (second.isEmpty())
        for (size_t index = 0; index < first.length - 1; index++, first_work = first_work->next)
            push(first_work->data);
    // 但是如果全不是空的，那就开始排队
    else {
        size_t first_index = 0;
        size_t second_index = 0;
        while (first_index < first.length - 1 || second_index < second.length - 1) {

            if ((first_work && second_work && first_work->data >= second_work->data)
                || (first_work && !second_work)) {
                push(first_work->data);
                first_work = first_work->next;
                first_index++;
            }
            else if ((first_work && second_work && first_work->data < second_work->data)
                || (!first_work && second_work)) {
                push(second_work->data);
                second_work = second_work->next;
                second_index++;
            }
            else
                break;
        }
    }

}
```

```
/*************************************
函数名称：    reverse
输入参数：    DLLNode<T> *head_
实现功能：    逆序双向列表
返回参数：    void
*************************************/
```

```
void reverse(DLLNode<T> *head_)                    noexcept {

    if (nullptr == head_->next->next) {
        // 这里是递归的最内部的一层
        // head_指针应该指向尾节点的上一个节点

        // 尾节点           : head_->next
        // 尾节点上一节点    : head_

        // 将尾节点的next置为尾节点的上一节点，开始逆序
        head_->next->next = head_;
        // 现在head_->next就是第一节点了，第一节点指着哨兵
        head_->next->prev = head;
        // 哨兵的next应该指向第一节点
        head->next = head_->next;

        // 哨兵             : head
        // 第一节点         : head_->next
        // 第二节点         : head_

        // 第二节点的上一个应该为第一节点
        head_->prev = head_->next;
        // 因为递归最深处只有两个节点，所以第二节点的next应该为nullptr
        head_->next = nullptr;
    }
    else {

        // 通过reverse函数，head_指向的节点之后的所有节点逆序完成
        reverse(head_->next);
        // 对于最外层的递归，也就是递归的入口，这里的head_就是head
        // 但是由于最内层递归已经处理了head
        // 所以仅当head != head_
        // 才运行下列代码
        if (head != head_) {

            // 逆序完成的尾节点 : head_->next
            // 仍未逆序的节点    : head_

            // 尾节点指向仍未逆序的节点，把仍为逆序的，作为新的尾节点
            head_->next->next = head_;
            head_->prev = head_->next;
            // 新的尾节点置nullptr
            head_->next = nullptr;
        }

    }
}
```

```
/************************************
函数名称：    clear
输入参数：    void
实现功能：    delete所有节点
返回参数：    void
************************************/
```

```
inline void clear()                                    noexcept {

    // 将指针移动到最后一个节点处
    DLLNode<T> *work = &(this->operator[](length - 1));
    // 在循环开始的时候，将指针移动到尾节点的上一个
    // 这个循环的结束条件是：当work指向的是头节点
    // delete的节点是除了头节点和第一节点之外的所有节点

    // 即使是空链或只有第一节点的链也没有关系
    // 空链的prev指向自己，第一节点的prev指向空链
    // 只要在初始化条件之后 指向头节点，那么就会跳出循环
    for (work = work->prev; work->prev != work; work = work->prev)
        delete work->next;
    // delete头节点和第一节点
    delete work->next;
    delete work;

}
```

```
/************************************
函数名称：     deleted
输入参数：     index
实现功能：     删除index对应位置的节点
返回参数：     T data
************************************/
```

```
T deleted(const size_t index)                          noexcept {

    // 不能够删除第一个哨兵节点
    if ( index >= length || index <= 0 ) {
        std::cout << "delete index overflow" << '\n';
        return T();
    }


    DLLNode<T> *work = head;
    DLLNode<T> *workPrev;

    // 将work移动到被删除的那个节点的位置
    // 而workPrev在被删除的节点的上一个节点
    for (size_t i = 0; i < index; i++)
        work = work->next;
    workPrev = work->prev;

    // workPrev的next连向work->next
    // work->next的prev连向workPrev
    // 就跳过了中间应该被删除的节点了
    T tmp = work->data;
    workPrev->next = work->next;
    if ( work->next != nullptr )
        work->next->prev = workPrev;

    length--;

    return tmp;

}
```

开发环境	Win10 Microsoft Visual Studio Community 2017 15.9.3 Debug x86

调试分析

（运行结果截图）

HW3.1

```
第1行1个正整数n，表示创建的单链表的元素个数
第2行n个正整数ai，空格分割，表示n个元素值
4
10 20 40 30
```

```
第3行2个整数i x，空格分割，表示在第i个元素前插入新元素x。
1 25
```

```
第4行1个整数j，表示删除第j个元素
1
```

```
第5行1个整数y，表示查找整数y
40
```

```
第1行，输出创建后的元素。
30 40 20 10
```

```
第3行，执行删除第j个元素后的结果，若j位置不合法，输出-1，否则输出表中全部元素
25 30 40 20 10
```

```
第4行，执行查找y的结果，若查找成功，输出该位置值（y第一次发现的位置），若不存在，输出-1。
2
```

```
第5行，输出元素个数
4
```

HW2

```
第1行1个正整数n，表示创建的链表的元素个数
6
第2行n个正整数ai，空格分割，表示n个元素值
10 15 20 30 20 10
```

```
第1行，顺序输出用头插法创建的链表
10 20 30 20 15 10
第2行，顺序输出逆置后所有3的倍数
15 30
```

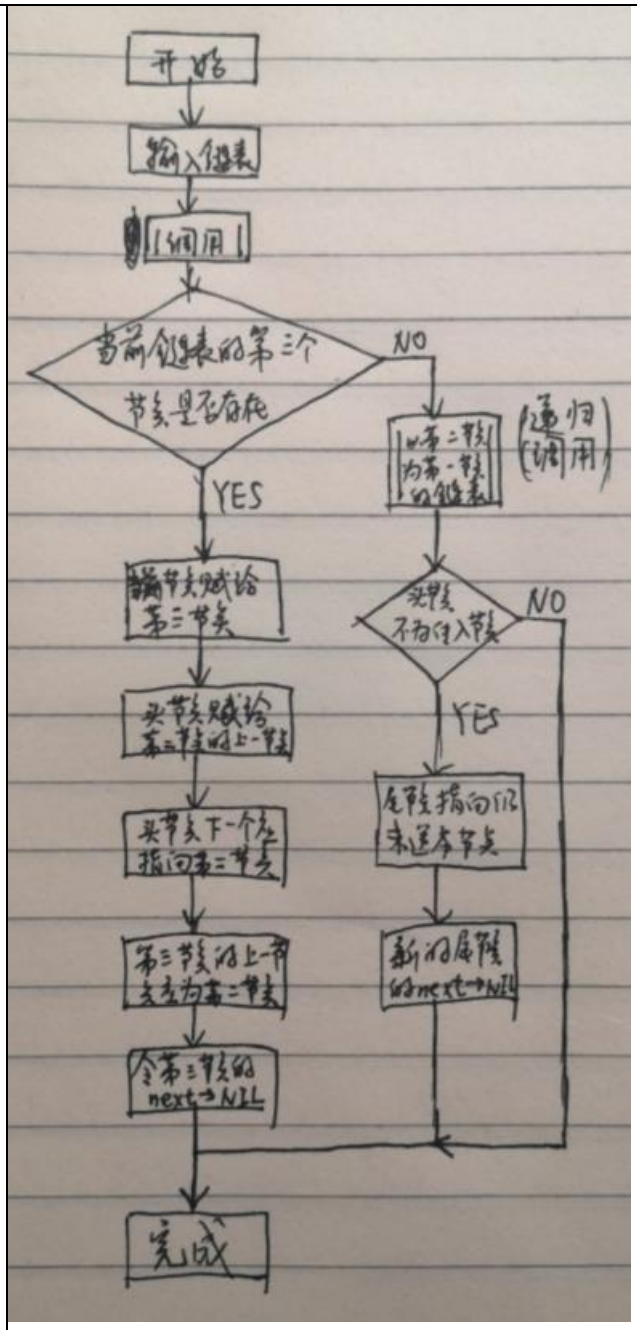
HW3

```
第1行1个正整数n，表示创建的链表的元素个数
7
第2行n个正整数ai，表示表中元素。
10 20 40 20 10 30 30
```

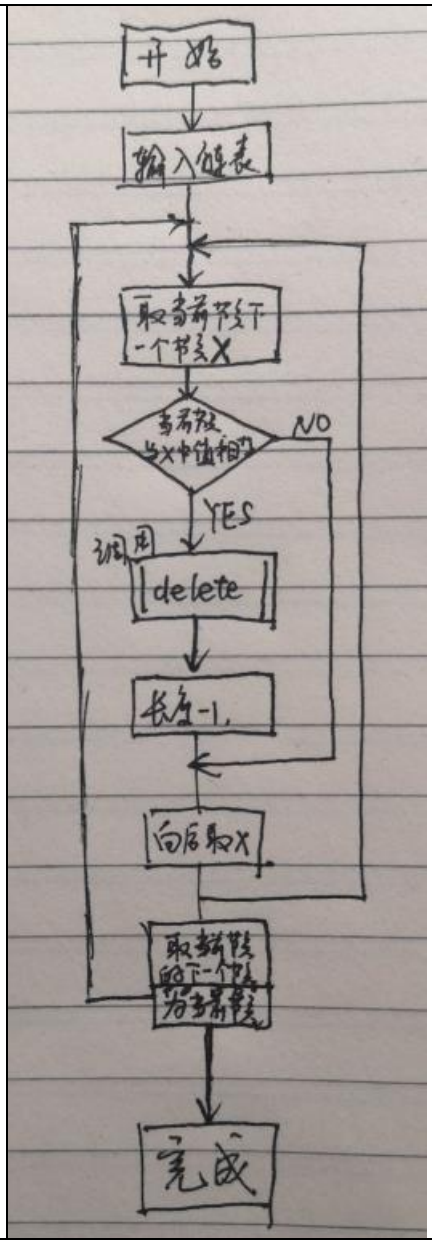
```
10 20 40 30
```

HW4

	第1行n个整数，当为0时结束，表示LA中的元素。 10 20 40 30 0 第2行m个整数，为0表示结束，表示LB中的元素。 5 10 15 0 1行，合并后的结果 5 10 10 15 20 30 40
心得体会	（对整个实验过程做出总结，对重要的算法做出性能分析。） 这次作业还是比较轻松愉快的。 因为我在这之前就写了一个双向链表的模板类。 对于 oj 里面的测试，可以说就是建个实例，然后用用函数就可以了。 特别是逆序和去重，我在还没有得知作业内容之前，就已经写好了去重和逆置算法，果然题目里面就出了这两道，所以没有调试几次就通过了。 对于线性表，比起连续空间，我还是更觉得链表用起来舒服。 链表更加灵活，虽然牺牲了查找的时间，但是在思路上，对其插入和删除都十分容易。 另外，链表可以在不进行物理层次上的移动而达到逻辑上的移动。仅仅通过链表指针的赋值，就可以使得链表进行逆序、去重、插入、删除、合并、排序等等一系列操作。而这是连续空间所不能替代的。 以下为链表逆序的程序框图 我采用的是递归思路。 假设某节点后的所有节点全部逆序完毕，然后调用该函数，将第 n 个节点和前面已经逆序完毕的 n-1 个节点反向连接，完成 n 个节点的逆序。 对于我的逆序方法，其时间复杂度为 O(n)



以下为链表去重算法的程序框图。
先确定一个节点，然后遍历此节点之后的所有节点。
如果节点值与现在确定的节点值相同，那么就调用删除函数删除指向的节点。
重复做此类操作，已达到去重的目的。
我的算法的时间复杂度为 $O(n^2)$


开始
输入链表
取当前节点下一个节点X
当前数据与X中值相等
NO
YES
调用
delete
长度-1
向后取X
取当前数据的下一个节点为当前节点
完成