

期中测试

问题1

问题描述：

对顺序表编写操作，将一整数序列中所有负数移到所有正数之前。

设计思想：

针对问题，我们发现仅需要将所有的负数移动到所有的正数之前，而无需保证其有序性，因此我们仅需遍历一遍单链表，将所有负数元素移动至链表开头既可以实现这个功能。

考虑优化，在从头遍历时，在遇到第一个正数之前，所有的负数都是符合要求的，因此我们可以先将操作节点遍历至第一个正数处，对于之后的负数元素移动至链表开头，如此可以避免将链表开头符合要求的负数元素再次移动至链表开头，减少操作次数。

时间复杂度：

基本语句： `pre->p = now->p, now->p = head->p, head->p = now, now = pre->p` (负数元素移动操作)

`now = now->p, pre = pre->p` (遍历操作)

执行次数： 最好情况下 $2N$ ，最坏情况下 $4N - 2$

时间复杂度： $O(N)$

代码：

```
template <class Element>
void LinkList<Element>::sortBySp1() {
    Node<Element>*pre = head, *now = head->p;
    while (now && now->data < 0) now = now->p, pre = pre->p;
    while (now)
        if (now->data < 0) pre->p = now->p, now->p = head->p, head->p = now, now = pre->p;
        else now = now->p, pre = pre->p;
}
```

运行截图：

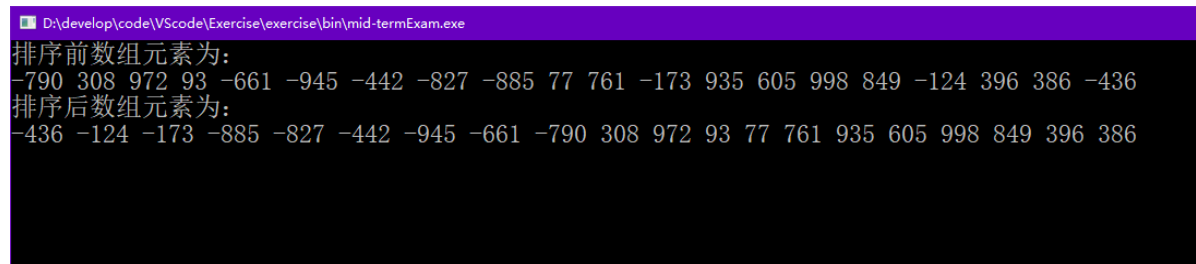
测试代码：

```

srand((unsigned)time(0));
int num[testSize];
for (int i = 0; i < testSize; i++) num[i] = rand() % 2001 - 1000;
LinkedList<int> ll(num, testSize);
std::cout << "排序前数组元素为: \n";
ll.printListByRow();
ll.sortBySp1();
std::cout << "排序后数组元素为: \n";
ll.printListByRow();

```

结果截图：



```

D:\develop\code\VScode\Exercise\exercise\bin\mid-termExam.exe
排序前数组元素为:
-790 308 972 93 -661 -945 -442 -827 -885 77 761 -173 935 605 998 849 -124 396 386 -436
排序后数组元素为:
-436 -124 -173 -885 -827 -442 -945 -661 -790 308 972 93 77 761 935 605 998 849 396 386

```

问题2

问题描述：

已知单链表 $L = (a_1, a_2, a_3, \dots, a_n)$ ，设计一个算法实现单链表的重新排列，即得到 $L' = (a_1, a_n, a_2, a_{n-1}, a_3, a_{n-2}, \dots)$ 。

设计思想：

针对单链表，先将后半段数据逆序，如果是奇数序列，取向下取整的一半数量逆序即可，在定义两个指针，一个从头开始，一个从中间节点开始，然后同时向后遍历，将后侧的节点依次插入前面的节点，最终实现目标排序。

时间复杂度：

基本语句： `mid = mid->p` （寻找中间节点）

`nex = now->p, now->p = pre, pre = now, now = nex;` （逆序操作）

`tmp = ba, ba = ba->p, tmp->p = fr->p, fr->p = tmp, fr = fr->p->p;` （插入操作）

执行次数： $N/2 + 4 * N/2 + 5 * N/2 = 5N$

时间复杂度： $O(N)$

代码：

```

template <class Element>
void LinkedList<Element>::sortBySp2() {
    int midLen = (this->getLength() >> 1) + (this->getLength() & 1);
    Node<Element>* mid = head;
    for (int i = 1; i <= midLen; i++) mid = mid->p;
    Node<Element>* now = mid->p, *nex = nullptr, *pre = nullptr;
    while (now) {
        nex = now->p, now->p = pre, pre = now, now = nex;
        if (!now) mid->p = pre;
    }
}

```

```

    }
    Node<Element>*fr = head->p, *ba = mid->p, *tmp = nullptr;
    mid->p = nullptr;
    while (ba) tmp = ba, ba = ba->p, tmp->p = fr->p, fr->p = tmp, fr = fr->p->p;
}

```

运行截图：

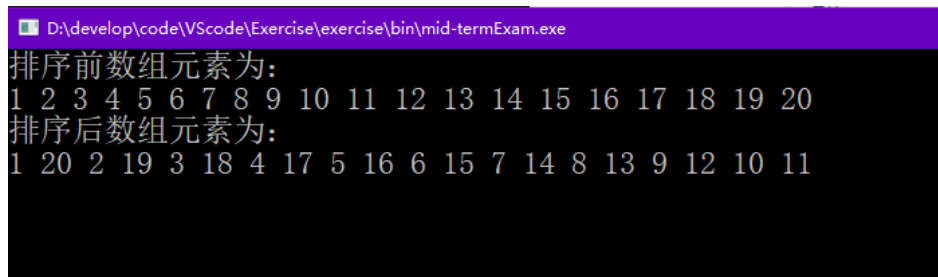
测试代码：

```

int num[testSize];
for (int i = 0; i < testSize; i++) num[i] = i + 1;
LinkList<int> ll(num, testSize);
std::cout << "排序前数组元素为：\n";
ll.printListByRow();
ll.sortBySp2();
std::cout << "排序后数组元素为：\n";
ll.printListByRow();

```

结果截图：



```

D:\develop\code\VScode\Exercise\exercise\bin\mid-termExam.exe
排序前数组元素为：
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
排序后数组元素为：
1 20 2 19 3 18 4 17 5 16 6 15 7 14 8 13 9 12 10 11

```

问题3

问题描述：

假设有两个按元素值递增次序排列的线性表，均以单链表形式存储。请编写算法将这两个单链表归并为一个按元素值递减次序排列的单链表，并要求利用原来两个单链表的结点存放归并后的单链表。

设计思想：

针对问题发现链表有序，且需要合并的两个单链表均为递增序列，这里我们取任意一个链表为主链表，称为 A ，将另一个链表合并到 A ，称另外一个链表为 B 。再定义两个指针，分别指向两个链表的首位元素，比较指针指向节点内元素大小，将元素较小的节点移动至 A 链表的头部，并将指针向后移动一位。考虑细节，若其中任何一个链表遍历完毕，则只需要再将另外一个链表的节点依次加入 A 链表头部即可，因为原两链表是递增的，一个链表遍历完毕证明另外一个链表的当前节点一定大于任意此链表所有节点元素，因此保证操作完毕后的 A 链表递减。

时间复杂度：

基本语句： `tmp = a->p, a->p = head->p, head->p = a, a = tmp;` (移动操作) (B 链表与 A 链表移动操作等价)

执行次数： $4 * N + 4 * M = 4(N + M)$ ，此处 N 为 A 链表长度， M 为 B 链表长度。

时间复杂度： $O(N + M)$ ，此处 N 为 A 链表长度， M 为 B 链表长度。

代码：

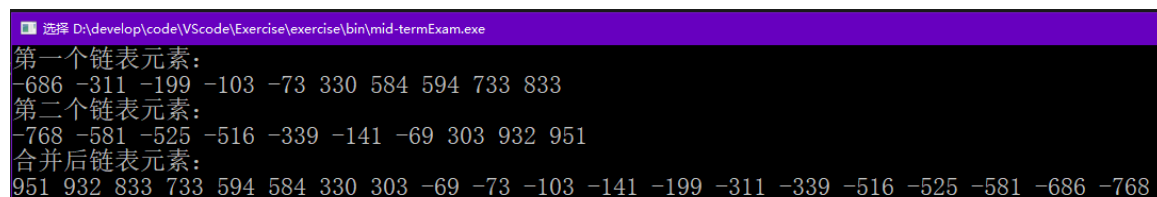
```
template <class Element>
void LinkedList<Element>::Merge(LinkedList<Element>& x) {
    Node<Element>*a = this->head->p, *b = x.head->p, *tmp = nullptr;
    this->head->p = nullptr, x.head->p = nullptr, this->size += x.size, x.size = 0;
    while (a && b)
        if (a->data < b->data) tmp = a->p, a->p = head->p, head->p = a, a = tmp;
        else tmp = b->p, b->p = head->p, head->p = b, b = tmp;
    while (!b && a) tmp = a->p, a->p = head->p, head->p = a, a = tmp;
    while (!a && b) tmp = b->p, b->p = head->p, head->p = b, b = tmp;
}
```

运行截图：

测试代码：

```
srand((unsigned)time(0));
int num[testSize];
for (int i = 0; i < testSize; i++) num[i] = rand() % 2001 - 1000;
std::sort(num, num + testSize);
LinkedList<int> l1(num, testSize);
std::cout << "第一个链表元素: \n";
l1.printListByRow();
for (int i = 0; i < testSize; i++) num[i] = rand() % 2001 - 1000;
std::sort(num, num + testSize);
LinkedList<int> l11(num, testSize);
std::cout << "第二个链表元素: \n";
l11.printListByRow();
l1.Merge(l11);
std::cout << "合并后链表元素: \n";
l1.printListByRow();
```

结果截图：



```
选择 D:\develop\code\VScode\Exercise\exercise\bin\mid-termExam.exe
第一个链表元素:
-686 -311 -199 -103 -73 330 584 594 733 833
第二个链表元素:
-768 -581 -525 -516 -339 -141 -69 303 932 951
合并后链表元素:
951 932 833 733 594 584 330 303 -69 -73 -103 -141 -199 -311 -339 -516 -525 -581 -686 -768
```

问题4

问题描述：

有非循环双链表 L ，需按照按值查找的频度排序，查找频率越高，越靠近头结点。 L 的结点中包含一个频度域，每次查找后，频度域+1，并调整结点在 L 中的位置，然后返回指向该结点的指针。试实现 $Locate(L, x)$ 函数。

设计思想：

根据题意，构造特殊节点 *DeNode*，结构体内添加频度变量 *fre*。对于初始化操作，我们将双链表中所有节点的频度变量初始化为0，对于头节点和尾节点赋值一个足够大的数字和一个足够小的数字，这里我们也可以进行特判操作，为实现方便，我们这里将其赋值为 $0x3f3f3f3f$ 和 $-0x3f3f3f3f$ 。对于 *Locate*(*L*, *x*)，我们传入后，先从头节点开始遍历元素，比对元素的值，寻找相同元素，注意这里从头开始遍历是因为，*Locate*(*L*, *x*)函数实现后会将频度大的元素置于链表前部，以优化查询时间。

对于查询操作，我们在查询到指定元素后，将其节点的频度+1，考虑到在链表中频度保证是递减，查询后频度增加只需向前比对频度，因此我们设指针向前遍历，遇到节点频度大于当前查询节点时，将当前查询节点插入该节点后，最终实现函数功能。

时间复杂度：

基本语句： `now = now->nex;` (查询操作)

`pos = pos->pre;` (比对频度操作)

`now->pre->nex = now->nex, now->nex->pre = now->pre, now->pre = pos, now->nex =`

`pos->nex, pos->nex->pre = now,`

`pos->nex = now;` (交换操作)

执行次数： 最好情况下 $1 + 1 + 6 = 8$ ，最坏情况下 $N + N + 8 = 2N + 8$

时间复杂度： $O(N)$

代码：

```
template <class Element>
DeNode<Element>* Locate(DeLinkedList<Element>& L, Element x) {
    DeNode<Element>* now = L.head->nex;
    while (now != L.tail) {
        if (now->data == x) break;
        now = now->nex;
    }
    if (now == L.tail) {
        std::cout << "Not Found!\n";
        return nullptr;
    }
    DeNode<Element>* pos = now->pre;
    now->fre++;
    while (now->fre > pos->fre) pos = pos->pre;
    now->pre->nex = now->nex, now->nex->pre = now->pre, now->pre = pos, now->nex
= pos->nex, pos->nex->pre = now, pos->nex = now;
    return now;
}
```

运行截图：

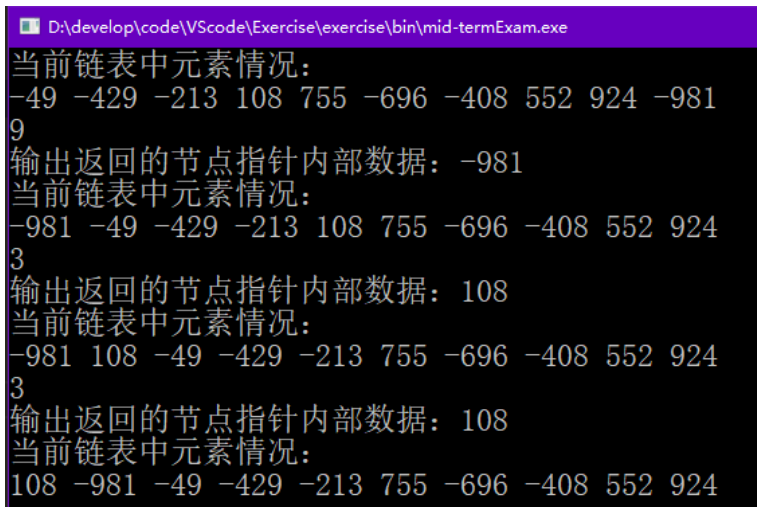
测试代码：

```

int num[testSize];
srand((unsigned)time(0));
for (int i = 0; i < testSize; i++) num[i] = rand() % 2001 - 1000;
DeLinkList<int> dll(num, testSize);
while (1) {
    std::cout << "当前链表中元素情况: \n";
    dll.printListByRow();
    int op; //输入-1退出测试, 输入[0, testSize - 1]中的数值查询链表元素。
    std::cin >> op;
    if (op == -1) break;
    else std::cout << "输出返回的节点指针内部数据: "<< Locate<int>(dll, num[op])->data
    << '\n';
}

```

结果截图:



```

D:\develop\code\VScode\Exercise\exercise\bin\mid-termExam.exe
当前链表中元素情况:
-49 -429 -213 108 755 -696 -408 552 924 -981
9
输出返回的节点指针内部数据: -981
当前链表中元素情况:
-981 -49 -429 -213 108 755 -696 -408 552 924
3
输出返回的节点指针内部数据: 108
当前链表中元素情况:
-981 108 -49 -429 -213 755 -696 -408 552 924
3
输出返回的节点指针内部数据: 108
当前链表中元素情况:
108 -981 -49 -429 -213 755 -696 -408 552 924

```