《数据结构》课程实践报告

院、系	计算机学院	年级专业	21 计算机科学与技术	姓名	赵鹏	学号	2127405037
实验布置日期	9.5		提交日期	2022.10.19		成绩	

课程实践实验 3：长整型数运算

一、问题描述及要求

题目：长整数运算

（1）实现线性表顺序存储结构、链式存储结构的基本操作，包括顺序表、单链表、双链表、单循环链表、双循环链表等;

（2）设计并实现两个长整数的加、减、乘运算。

二、概要设计

本次实验由两个部分组成，故在此将两部分分开阐述。

1. 线性表的基本操作

此部分为实现线性表顺序存储结构以及链式存储结构的相关操作，依据课堂所学习的知识编写代码，实现线性表的构造、插入、删除、按值查找、按位查找、销毁（析构函数）等即可。

1.1 函数的接口设计

线性表的基本操作包括查找、访问、插入、删除等。尽管线性表的具体实现由于存储结构的不同（顺序、链式）而不同，但对外的函数接口是一致的。在线性表的实现中，我们定义了以下接口用于使用线性表的各种功能。

函数名	功能
Get(int index)	返回线性表第 index 个元素
Find(DataType elem)	返回线性表中第一个值为 elem 的元素下标
Insert(int index, DataType elem)	在 index 位置插入 elem
Delete(int index)	删除第 index 个元素

Length()	返回元素个数
Clear()	清空线性表
Empty()	判断表是否为空
Print()	输出所有元素

1.2 顺序表

1.2.1 顺序表的定义

顺序表是线性表的顺序存储结构,其基本思想是使用一段地址连续的存储单元依次存储线性表的数据元素，设每个元素占用 c 个存储单元，则第 i 个元素的存储地址为 Loc(ai)=Loc(a1)+(i-1)*c,容易看出，确定了起始地址后计算任意元素的存储地址的时间是相等的，因此顺序表是一种随机存取结构，在 C++中可以使用数组来实现，定义 MAXSIZE 来约束静态数组的最大长度。

1.2.2 算法的实现与分析

按位查询 Get(index) O(1)

由与顺序表采用连续的存储单元，因此 Loc(ai)=Loc(a1)+(i-1)*c，直接返回 data[index-1]即可，每次操作均为 O(1)。

求长度 Length () O(1)与判空 Empty() O(1)

在析构、插入、删除的时候同时维护线性表类的成员变量 size，求长度直接返回 size，判空即判断 size 是否等于 0。

按值查找 Find (elem) O (n)

按下标 index 对表从前向后遍历，不断判断 if(data[index]==elem)找到后返回是表的第 index 个元素，由于 data 数组的下标从 0 开始，故返回 index+1 即可，若没有找到返回 0 即可。平均比较次数 P(n)=(0+n)/2=O(n)。

插入元素 Insert(index,elem) O(n)

首先需要检查插入位置的合法性，即当 index<1 或 index>size+1 判为不合法。对于顺序表的插入操作，只需要从插入位置开始将后面 size-index+1 个元素向后移动一个位置即可，同时需要注意维护表长 size，在插入位置随机的情况下，平均移动次数为 n/2,故该操作的时间

复杂度为 O(n)。

删除元素 Delete(index)

删除元素的操作与插入元素大体类似，首先检查位置的合法性。对于删除操作，只需要将删除位置后的 size-index+1 个元素向前移动一个位置即可，用删除位置后继元素覆盖删除元素，同时维护 size。在删除位置随机的情况下，平均移动次数为 n/2,故该操作的时间复杂度为 O(n)。

打印顺序表 Print()

打印顺序表的操作较为简单，只需要下标 index 从 0 开始到 size-1 依次遍历，输出每一个元素即可。因为需要遍历整个表，故时间复杂度为 O(n)。

销毁顺序表 Clear ()

销毁顺序表的操作较为简答，只需要将 size 修改为 0，即可在逻辑上销毁整个顺序表。

1.3 单链表

1.3.1 单链表的定义

单链表使用一组任意的存储单元存放线性表的元素,存储单元可以连续也可以不连续,每个存储单元(结点)存储数据以及后续结点的地址,用逻辑顺序把元素链接在一起。结点的定义如下:

```
1. template<typename Entry>
2. struct Node
3. {
4. Entry data;
5. Node<Entry>* next;
6. };
```

由于链表的每个结点的存储地址存放在前一结点的 next 域中，而第一个元素无前驱，因此需要设定一个头结点指向第一个元素所在的结点地址，单链表的存取也需要从头结点开始。所以单链表的定义如下:

```
1. template<typename DataType>
```

```
class   linkedList
{
public:
linkedList();
linkedList(DataType a[], int len);
~linkedList();
linkedList(const linkedList<DataType>&);
//some basic linear list operations …
protected:
Node<DataType>* head;
int size;
};
```

1.3.2 单链表的实现与分析

单链表的建立

单链表的建立主要有两种方法：头插法和尾插法，由于使用插法得到的绪论为所给定序列的逆序列，所以在使用数组初始化链表时使用可以生成顺序列的尾插法。尾插法的基本思路为生成列表时存储当前链表的最后一个元素的地址，每次添加新元素时生成一个新结点，原来的尾结点接到新生成的结点上，再更新最后一个元素的地址。

代码实现如下：

```
template<typename DataType>
linkedList<DataType>::linkedList(DataType a[], int n)
{
head = new Node<DataType>;
Node<DataType>* r = head, * s = NULL;
for (int i = 0; i < n; i++)
{
s = new Node<DataType>; s->data = a[i];
r->next = s; r = s;
}
r->next = NULL;
}
```

单链表的清空与析构

单链表的清空与析构即删除链表里除了头结点以外的所有元素，因此只需不断删除最开头的元素，直至 head->next==NULL 即可。由于实现较为简单，故在此不再放置代码。

单链表的插入元素

单链表插入元素时需要定义指针 p 从头结点依次遍历到需要插入位置的前一个位置。随后生成新插入的结点并定义指针 s 指向新结点并进行指针的操作实现插入，具体代码实现如下：

```
1. s = new Node<DataType>;
2. s->data = x;
3. s->next = p->next;
4. p->next = s;
```

单链表的删除元素

单链表的删除元素与插入类似，首先将指针 p 遍历到需要删除元素的上一个元素，随后申请新指针 q 指向待删除元素，再操作 p 指针指向元素的后继指向待删除元素的后继，最后释放待删除元素即可，具体代码实现如下：

```
1. q = p->next; x = q->data;
2. p->next = q->next;
3. delete q;
```

单链表的求输出与求长度

单链表求输出与求长度 c 操作类似只需定义指针 p 从头结点开始遍历直至 p->next==NULL，过程中输出 p->data 与计数即可。由于实现较为简单，在此不在展示代码。

单链表的按值查找与按位查找

定义指针 p 从头结点开始向后遍历，按值查找只需不断往后遍历并计数直至 p->data==elem 即可，若最后 p->next==NULL 说明无法找到，返回 0 即可，否则返回计数器的值。单链表的按位查找与也同样是简单的遍历，定义计数器，每次向后遍历一个元素便把计数器+1，直至 count==index 时返回 p->data，注意需要对 index 进行合法性判断。

单链表的判空

单链表为空即头结点为最后一个结点，因此判断 head->next==NULL 即可。

可以看出，上述的插入、删除、查找等都需要遍历链表，因此在等概率的情况下，这些操作的时间复杂度均为 O（n）。

由于链式存储结构的种类很多，但大部分函数的实现思路与代码基本一致，故以下部分阐述单循环链表、双链表、双循环链表时对于大部分实现基本一致的部分不再重复阐述，只阐述各个数据结构的特殊部分。

1.4 单循环链表

1.4.1 单循环链表的定义

在单链表中如果将原终点结点的指针由空指针 NULL 改为头结点，就可以使整个单链表在逻辑上形成一个环，这种头尾相接的单链表即为单循环链表。

1.4.2 单循环链表的实现

正如单循环链表的定义一般，单循环链表的实现与单链表的实现基本完全一致，只是在从头结点向后遍历时判断链表结束时把 p->next==NULL 改为 p->next==head.其余部分基本完全一致，故在此不再重复阐述实现。

1.5 双链表

1.5.1 双链表

为了快速确定单链表中任一结点的前驱结点,可以在单链表中每个结点中再设置一个指向其前驱结点的指针 prior,故双链表的结点定义如下:

```
1. template<typename Entry>
2. struct Node
3. {
4. Entry data;
5. Node<Entry>* prior;
6. Node<Entry>* next;
7. };
```

1.5.2 双链表的实现与分析

双链表的遍历、查找、求长度、判空等操作均与单链表一致，主要区别在于插入和删除操作的实现，故下述重点阐述双链表的插入与删除操作。

由于每个结点有两个指针的存在,故双链表操作的重点在于维护好结点间的逻辑关

系。指针操作的顺序不可颠倒，否则极易出现错误。

双链表的插入操作

为了在操作时维护好逻辑关系，双链表的插入操作需要按如下顺序操作:

1. 定义指针 p 移动到待插入位置的前一个位置。
2. 生成新结点 s，将其前驱设为 p，后继设为 p->next。
3. 若 p 的后继不为空，将其前驱修改为 s。
4. 将 p 的后继修改为 s

代码实现如下:

```
1. s = new Node<DataType>;
2. s->data = x;
3. if (p->next != NULL)
4. p->next->prior = s;
5. s->next = p->next;
6. s->prior = p;
7. p->next = s;
```

双链表的删除操作

为了在操作时维护好逻辑关系，双链表的删除操作需要按如下顺序操作:

1. 定义指针 p 移动到待删除的位置。
2. P 的前驱的后继修改为 p 的后继
3. 若 p 的后继不为空，将 p 的后继的前驱修改为 p 的前驱。

代码实现如下:

```
1. q = p->next; x = q->data;
2. p->next = q->next;
3. p->next->prior = p;
4. delete q;
```

1.6 双循环链表

1.6.1 双循环链表的定义

在双链表中如果将原终点结点的指针由空指针 NULL 改为头结点，把头结点的前驱改为尾结点，就可以使整个双链表在逻辑上形成一个环，这种头尾相接的单链表即为单循环链表。

1.6.2 双循环链表的实现

正如双循环链表的定义一般，双循环链表的实现与双链表的实现基本完全一致，只是在从头结点向后遍历时判断链表结束时把 p->next==NULL 改为 p->next==head.其余部分基本完全一致，故在此不再重复阐述实现。

2 文件结构设计

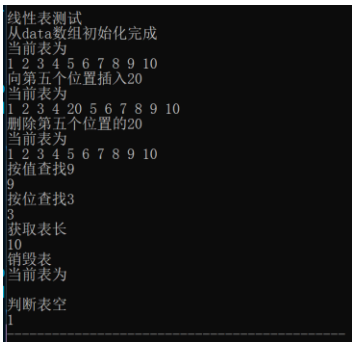
本项目在实现过程中设计了以下文件

linkedListNode.h	定义并实现单指针结点
doubleLinkedListNode.h	定义并实现双指针结点
sequenceList.hpp	定义并实现顺序表的相关操作
linkedList.hpp	定义并实现链表的相关操作
circularLinkedList.hpp	定义并实现循环链表的相关操作
doubleLinkedList.hpp	定义并实现双链表的相关操作
doubleCircularLinkedList.hpp	定义并实现双循环的链表的相关操作

其中单链表和单循环链表包含 linkedListNode.h，双链表和双循环链表包含 doubleLinkedListNode.

1.7 程序测试

1.7.1 顺序表的测试



通过测试

1.7.2 单链表的测试

```
单链表表测试
从data数组初始化完成
当前表为
1 2 3 4 5 6 7 8 9 10
向第五个位置插入20
当前表为
1 2 3 4 20 5 6 7 8 9 10
删除第五个位置的20
当前表为
1 2 3 4 5 6 7 8 9 10
按值查找9
9
按位查找3
3
获取表长
10
销毁表
当前表为

判断表空
1
```

通过测试

1.7.3 单循环链表的测试

```
单循环表测试
从data数组初始化完成
当前表为
1 2 3 4 5 6 7 8 9 10
向第五个位置插入20
当前表为
1 2 3 4 20 5 6 7 8 9 10
删除第五个位置的20
当前表为
1 2 3 4 5 6 7 8 9 10
按值查找9
9
按位查找3
3
获取表长
10
销毁表
当前表为

判断表空
1
```

通过测试

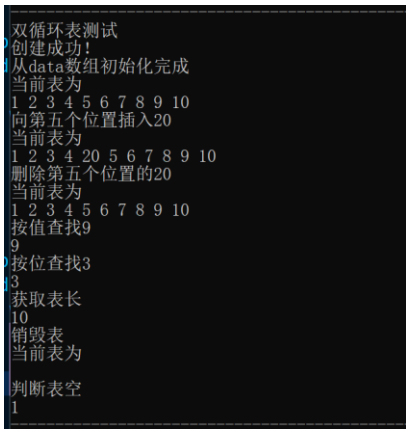
1.7.4 双链表的测试

```
双链表测试
从data数组初始化完成
当前表为
1 2 3 4 5 6 7 8 9 10
向第五个位置插入20
当前表为
1 2 3 4 20 5 6 7 8 9 10
删除第五个位置的20
当前表为
1 2 3 4 5 6 7 8 9 10
按值查找9
9
按位查找3
3
获取表长
10
销毁表
当前表为

判断表空
1
```

通过测试

1.7.5 双循环链表的测试



通过测试

2. 长整数运算

长整数运算的基本思路是将整数的每一位分开存储在单个存储单元中，对于加、减、乘的操作，手动模拟进位、借位等。

对于长整数的存储可以采取线性存储或链式存储结构，但比较两种存储结构，使用顺序存储结构需要预先分配大量的空间，对于数字长度较短时会出现空间浪费，对于数字较长时会出现空间不足，为了避免以上问题，存储长整数更适合采取动态分配空间的结构，即链式存储结构。我们可以使用链表或直接使用一些 C++自带的动态分配空间的容器进行存储。在这里，我们直接使用 C++标准模板库中的 vector 容器进行存储长整数的每一位。

对于长整数的计算，除了数字的问题，另一个值得考虑的地方是数字的符号，在定义长整数时可以使用 vector 存储数字绝对值的每一位并添加 sign 表示这个数字的正负。

在设计长整数类时，首先定义了三个私有函数 add,sub,multi，实现对两个长整数绝对值的加、减、乘运算。对于支持正负的运算通过重载+-*运算符来实现。除此以外，为长整数类定义了三种构造函数，分别是从字符串构造、从 vector 和符号位构造、拷贝构造函数。

综上，长整数类的定义如下：

```
class BigNum
{
public:
std::vector<int>num;//存储每一位数字
BigNum();
BigNum(const BigNum& Num);
BigNum(const std::vector<int> Num, int Sign);
BigNum(const std::string& str);
BigNum operator +( BigNum& Num);
void print();
BigNum operator -( BigNum& num);
BigNum operator *( BigNum& num);
private:
```

```
int sign;//正负
bool cmp(std::vector<int>& A, std::vector<int>& B);
std::vector<int> add(const std::vector<int>& A, const  std::vecto
r<int>& B);
std::vector<int> sub(const std::vector<int>& A, const std::vector
<int>& B);
std::vector<int> multi (std::vector<int>& A, std::vector<int>& B);
};
```

由于直接考虑实现支持正负的运算较为困难，故我们首先实现非负数长整数的运算。为了便于运算时的进位、借位等，我们在存储数字时采取倒序存储，即把 123456 存储为以下形式：num[0]=6,num[1]=5……

2.1 非负数加法

对于非负数加法只需从个位开始不断向前遍历，同时维护一个余数 t，对于每一位的结果设定为(A[i]+B[i]+t)%10，对于进位只需要将 t/=10 即可。需要注意的是最后一位也可能有进位，需要在遍历结束后进行特判。

代码实现如下：

```
std::vector<int> C;
int t = 0;
for (int i = 0; i < A.size(); i++)
{
t += A[i];
if (i < B.size()) t += B[i];
C.push_back(t % 10);
t /= 10;
}
if (t) C.push_back(t);
```

2.2 非负数减法

与非负数加法类似，减法的实现也是从前往后遍历，同时维护一个借位数字 t，每一位的数字首先减去原来的借位即 t=A[i]-t，随后在减去减数的这一位 B[i]，考虑到不够减需要借位的情况，这一位的结果即为（t+10）%10。若 t<0 则将 t 置为 1 表示向下一位借了一位。由于最后一位结果可能是 0，因此同样需要在最后进行特判。

代码实现如下：

```
std::vector<int> C;
for (int i = 0, t = 0; i < A.size(); i++)
{
t = A[i] - t;
if (i < B.size()) t -= B[i];
C.push_back((t + 10) % 10);
if (t < 0) t = 1;
else t = 0;
```

```
}
while (C.size() > 1 && C.back() == 0) C.pop_back();
```

2.3 非负数乘法

非负数乘法也是模拟竖式运算乘法的过程，即枚举两个数的每一位相乘，对于 A 的第 i 位和 B 的第 j 位，把相乘的结果存储在 C[i+j]中即可，最后从前向后处理进位即可。

代码实现如下：

```
std::vector<int> C(A.size() + B.size(), 0);
for (int i = 0; i < A.size(); i++) {
for (int j = 0; j < B.size(); j++) {
C[i + j] += A[i] * B[j];
}
}
int t = 0;
for (int i = 0; i < C.size(); i++) {
t = t + C[i];
C[i] = t % 10;
t /= 10;
}
while (C.size() > 1 && C.back() == 0) C.pop_back();
```

3.1 加法

有了上述的非负数运算，我们只需要对两个运算数的符号进行讨论即可得到支持正负的运算。

对于两个数的加运算有以下四种情况：

(1) 正数+正数 (2) 负数+负数 (3) 正数+负数 (4) 负数+正数

对于（1）、（2）直接采取绝对值运算然后使用构造函数分别返回结果为正、负的运算结果即可。

对于（3）、（4）可以将其看出绝对值的减法运算（3）为 A-B，(4)为 B-A，为了获得结果的正负，需要在类里定义比较两个非负长整数的比较函数 cmp。根据比较结果返回对应符号的运算结果即可。

设数字长度为 n，由于需要一重循环遍历两个数字，因此执行两个长整数加法的时间复杂度为 O(n)。

3.2 减法

对于两个数的减法运算有以下四种情况：

(1) 正数-正数 (2) 负数-负数 (3) 正数-负数 (4) 负数-正数

对于（1）直接调用 cmp 函数判断正负和 sub 函数进行相减即可。

对于（2）,可以转化为正数+负数，直接调用重载的加法即可。

对于（3），可以转化为两正数相加，直接调用 add 函数并返回结果和符号即可。

对于（4），可以转化为两整数相加，把符号取反，直接调用 add 函数并把符号设为负号返回即可。

设数字长度为 n，由于需要一重循环遍历两个数字，因此执行两个长整数减法的时间复杂度为 O(n)。

3.3 乘法

对于两个数的减法运算有以下两种情况：

(1) 两数同号（2）两数异号

由于在定义符号时 0 表示 positive,1 表示 negative。所以可以通过两个数符号的异或运算进行判断。

对于（1），即 A.sign^B.sign==1，调用 mutil 函数获得两数相乘的绝对值并返回符号位负号的 BigNum 类即可，

对于（2），即 A.sign^B.sign==0，调用 mutil 函数获得两数相乘的绝对值并返回符号位正号的 BigNum 类即可，

设数字长度为 n，由于需要两重循环遍历两个数字，因此执行两个长整数减法的时间复杂度为 O(n^2)。

3.4 长整数运算测试

```
输入两个长整数,将自动输出两个数的四种运算结果
10 20
10+20=30
10-20=-10
20-10=10
10*20=200
----------------------------------------
输入两个长整数,将自动输出两个数的四种运算结果
99999 88888
99999+88888=188887
99999-88888=11111
88888-99999=-11111
99999*88888=8888711112
----------------------------------------
输入两个长整数,将自动输出两个数的四种运算结果
100000000 100000000000000000
100000000+100000000000000000=100000000100000000
100000000-100000000000000000=-99999999900000000
100000000000000000-100000000=99999999900000000
100000000*100000000000000000=10000000000000000000000000
----------------------------------------
输入两个长整数,将自动输出两个数的四种运算结果
0 10
0+10=10
0-10=-10
10-0=10
0*10=0
----------------------------------------
输入两个长整数,将自动输出两个数的四种运算结果
```

测试通过

三、小结

通过本次实验，我对线性表的顺序存储以及链式存储结构都有了更加深刻的理解，也通过长整数运算部分更加深刻理解的动态分配空间的链式存储结构的实际运用。

但在对于代码部分仍有很多不足，例如长整数运算部分仅仅做了简单的非法字符判断，程序健壮性部分仍有待提高。以及 vector 的每一位仅存储数字的一位，对于空间有着较多的浪费。对于线性表部分，可以使用抽象类统一描述线性表的 ADT 等。