苏 州 大 学 实 验 报 告

院、系	计算机学院	年级专业	19 计算机类	姓名	张昊	学号	1927405160
课程名称	数据结构课程实践					成绩	
指导教师	孔芳	同组实验者	无		实验日期	2020 年 9 月 25 日	

实 验 名 称 线性表

一、实验目的

通过本次实验要达到如下目的：

1. 理解线性表的抽象数据类型；
2. 掌握顺序表、单链表的存储思想；
3. 掌握顺序表、单链表、双链表、循环链表的基本算法和时间性能；
4. 学会使用线性表抽象实际问题中的数据结构。

二、实验内容

题目：长整数运算

【问题描述】

（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。

三、实验步骤和结果

1 线性表基本操作的实现

1.1 接口设计

线性表（linear list）是 $n\ (n\geq 0)$ 个数据元素的有限序列。一个非空表 $L=(a_1,a_2,\cdots,a_n)$ 中，除了表头元素a_1（无前驱）和表尾元素a_n（无后继）外，每一个数据元素a_i都有唯一的**前驱**a_{i-1}和**后继**a_{i+1}（$1<i<n$），而且对于每个数据元素都有且只有一个确定位置 i，称为**序号**（规定数据元素的序号从 1 开始）。这就形成了线性表的**数据模型**（Data Model）。线性表的数据模型的具体实现方法会根据实现线性表的存储结构的不同而存在差异，但相邻数据元素之间的序偶关系$<a_{i-1},a_i>\ (1<i\leq n)$是相同的。

线性表是一个灵活的数据结构，不仅可以对数据元素进行**查找**（定位）、**访问**、**插入**和**删除**，还可以对整个线性表进行**遍历**。这些操作称为线性表的基本操作。线性表的基本操作的具体算法会根据实现线性表的存储结构的不同而存在差异，但是对外的使用方法（调用接口）是不变的。

基于以上两点，可以定义线性表的抽象数据类型（ADT）。这里，基于调用接口的不变性，采用 C++中的**抽象类**来描述线性表的 ADT，到具体实现某一种存储结构的线性表（如顺序表、单链表等）时**继承**这个抽象类，并实现（覆盖）其中的方法，同时利用相应的存储结构来维护数据模型。

此外，线性表的数据元素的数据类型是抽象的（但同一表中的数据元素类型相同），在实际的问题中这种抽象型才会被具体化。这与 C++中的类模版的机制相似，所以考虑采用类的模板参数来作为数据元素的数据类型，在类模板实例化后，数据元素才有确定的数据类型。

综上所述，代表线性表抽象数据类型的抽象类 `ListType` 定义如下：

```
template<typename Entry>
class ListType {
public:
    virtual ~ListType() = default;
    virtual int length() const noexcept = 0;
    virtual Entry &get(int index) = 0;
    virtual int locate(const Entry &elem) const noexcept = 0;
    virtual void insert(int index, const Entry &elem) = 0;
    virtual Entry remove(int index) = 0;
    virtual bool empty() const noexcept = 0;
    virtual void traverse(void (*visit)(Entry &)) = 0;
    virtual void clear() noexcept = 0;
    static const int notFound;
};
```

其中，模板参数 `Entry` 为数据元素抽象数据类型；静态常量 `notFound` 初始化为 0，用于标识元素未找到，为所有派生类所共享。

具体说明如下：

1.1.1 构造函数（未在抽象类 `ListType` 中定义）

构造函数有三，分别用于初始化线性表（无输入，输出空表）；建立线性表（输入 n 个数据元素，输出具有 n 个数据元素的线性表）；拷贝线性表（输入同类线性表，输出拷贝的线性表）。

1.1.2 求线性表的长度 `length`

输入：无

输出：线性表中的元素个数

1.1.3 访问指定位置处的元素（按位查找） `get`

输入：元素的序号 `index`（从 1 开始）

输出：如果元素的序号合法，返回序号为 `index` 的元素的引用，否则抛出范围越界（`out_of_range`）异常

1.1.4 定位指定元素（按值查找） `locate`

输入：数据元素 `elem`

输出：如果查找成功，返回元素 `elem` 在线性表中的序号（从 `1` 开始），否则返回 `notFound`（`=0`）

1.1.5 在指定位置处插入元素 `insert`

输入：插入位置 `index`；待插入元素 `elem`

输出：如果插入不成功抛出包含失败信息的异常

1.1.6 删除指定位置的元素 `remove`

输入：删除位置 `index`

输出：如果删除成功，返回被删除的元素，否则抛出包含失败信息的异常

1.1.7 判段线性表是否为空表 `empty`

输入：无

输出：如果是空表返回 `true`，否则返回 `false`

1.1.8 遍历线性表 `traverse`

输入：用于访问数据元素的函数指针 `visit`

输出：无

1.1.9 清空线性表 `clear`

输入：无 输出：无

说明：使线性表恢复为空表（此时线性表仍可使用）

1.1.10 销毁线性表（析构函数）

输入：无 输出：无

说明：释放线性表占用的存储空间（此时线性表不可再次使用）

1.2 顺序表（线性表顺序存储结构）

1.2.1 定义

顺序表的基本思想是使用一段地址连续的存储单元来存储数据元素。由于存储地址连续，在基地址确定的情况下，表中每个元素的地址是其序号的一阶线性函数，故获取每个元素的地址所花费的时间均相等，是一种**随机存储**结构，同时也维护好了相邻数据元素之间的序偶关系$< a_{i-1}, a_i > (1 < i \leq n)$。对应 C++中使用一维数组来实现顺序表，并且给定静态常量 `maxSize` 来表示数组总长度，用字段 `size` 来表示线性表的长度。

综上所述，顺序表的 C++类定义如下：

```
template<typename Entry>
class SequenceList : public ListType<Entry> {
public:
    using ListType<Entry>::notFound;    // 声明引用基类的 notFound 常量
    SequenceList();
```

```
    SequenceList(Entry dataArray[], int len);
    SequenceList(const SequenceList<Entry> &list);
    ~SequenceList() override = default;
    // 实现抽象基类 ListType 中定义的抽象方法
protected:
    static const int maxSize;
    int size;
    Entry data[maxSize];
};
```

1.2.2 实现与分析

具体的代码实现详见附件“Src/util/container/sequence_list.hpp”

在实现过程中有以下几点需要注意：

- 由于 C++的数组下标从 0 开始，这里线性表中规定数据元素的序号从 1 开始，线性表中第 i 个元素要存储在数组的下标为 i-1 的位置。
- 序号（下标）的合法性问题：
  - 数组有最大长度 `maxSize` 和最小长度 0，故建立顺序表提供的元素数量 n 不能大于 `maxSize`，插入操作时顺序表的长度 `size` 必须小于 `maxSize`，删除操作时顺序表不能是空表。
  - 按位查找和删除操作提供的序号 `index` 不能超过顺序表的表长 `size`，且需要大于 0；插入操作提供的序号 `index` 需要大于 0 且至多为 `size+1`（是因为可以插在表的尾部）。
- 插入、删除、清空操作要维护好记录顺序表长度的字段。
- [不属于数据结构的问题] 由于基类 `ListType` 依赖于模板参数，当模版类作为基类时，派生类要用到基类中某个内容需要加入 `this->`或作用域限定，或者在类中使用 `using` 声明（如上代码片段），否则会出现符号的未定义问题。

1.2.2.1 常数阶（$O(1)$）操作实现简要分析

默认构造函数、析构函数和清空顺序表操作：由于顺序表是静态内存分配，构造和析构中整块存储空间的分配和回收由操作系统完成，只需要构造时将表长字段 `size` 初始化为 `0`，清空时表长字段 `size` 重新赋值为 `0`。

判空操作和求顺序表长度：字段 `size` 保存了表的长度，判空只需判断表长是否为 `0`，求表长只需返回字段 `size` 的值。

按位查找操作：由表中每个元素的地址是其序号的一阶线性函数 $location(a_i) = location(a_1) + (i-1) \times sizeof(\text{Entry})$，借助数组下标访问返回 `data[index-1]`，每次按位查找元素所耗时间固定且相同。

1.2.2.2 线性阶（$O(n)$）操作实现简要分析

创建顺序表和拷贝线性表等遍历操作均需遍历整个线性表一遍。

遍历算法可用 C++语言描述为：

```
template<typename Entry>
void SequenceList<Entry>::traverse(void (*visit)(Entry &)) {
    for (int i = 0; i < size; ++i) {
        visit(data[i]);
    }
}
```

定位算法（按值查找）需要逐个对数据元素进行比较，直到找到要找的元素为之，并返回序号（数组下标+1），否则返回未找到标志 notFound。算法的规模是表长 n，基本语句是比较，主要的影响就是比较动作发生的次数。假设要定位的元素在顺序表上的出现的位置是等概率的，则平均比较次数$P(n) = n/2 = O(n)$，为线性阶。

插入操作检查序号合法性后，主要需要将指定位置开始的所有元素向后移动一个位置（基本语句），表长自增，其算法可由如下 C++代码描述：

```
template<typename Entry>
void SequenceList<Entry>::insert(int index, const Entry &elem) {
    if (size >= maxSize) { throw std::overflow_error{"…"}; }
    if (index < 1 || index > size + 1) { throw std::out_of_range{"…"}; }
    for (int i = size; i > index; --i) {
        data[i] = data[i - 1];
    }
    data[index - 1] = elem;
    ++size;
}
```

同样假设插入点 index 的选定是等可能的，在第 i 个位置上插入需要将包括第 i 个的后面的$(n-i+1)$个元素后移，故平均移动次数$P(n) = \frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1) = \frac{n}{2} = O(n)$ ，为线性阶。

同理，**删除操作**检查序号合法性后，主要需要将指定位置之后的所有元素向前移动一个位置（基本语句），并注意提前取出元素，维护好表长 size。其算法描述（详见源代码文件）和时间复杂度分析与上面的类同，也为线性阶。

1.3 单链表（线性表链式存储结构）

1.3.1 定义

单链表是用一组任意的存储单元存放数据元素，并且每个存储单元同时

存储数据元素和后继元素的地址（这种存储映像称为**结点**），以按照逻辑顺序将数据元素链接在一起，从而不必关注每个数据元素在存储器中的实际位置，而是关心数据元素之间的逻辑关系。结点的结构可以定义如下：

```
template<typename Entry>
struct Node {
    Entry data;
    Node<Entry> *next;
};
```

为保证空链表和非空链表都有统一的处理方法，在单链表的开始结点之前添加一个类型相同的的结点，称为**头节点**（head node）。这样，也保证了对单链表的操作不会修改头节点。另外，对于单链表来说，其每个数据元素的存储空间是在其出现在表中才分配的，而且各个元素在物理上不连续，传统的方法对链表求长度都需要对链表进行一轮遍历，所以考虑在原始单链表中增添一个属性 `size` 来表示表长，这样类似顺序表，单链表也可以在按位查找、插入和删除等需要查找元素的方法中**提前做位置合法性检测**。

综上所述，单链表的 C++类定义如下：

```
template<typename Entry>
class LinkedList : public ListType<Entry> {
public:
    using ListType<Entry>::notFound;    // 声明引用基类的 notFound 常量
    LinkedList();
    LinkedList(Entry dataArray[], int len);
    LinkedList(const LinkedList<Entry> &list);
    ~LinkedList() override;
    // 实现抽象基类 ListType 中定义的抽象方法
protected:
    Node<Entry> *head;
    int size;
};
```

1.3.2 实现与分析

具体的代码实现详见附件“Src/util/container/linked_list.hpp”

在实现过程中有以下几点需要注意：

- 单链表遍历操作中，活动指针在链表数据元素中不断后移。使活动指针 p 后移为单链表各个方法的核心操作，为

  ```
  p = p->next;
  ```

- 在原单链表的基础上，增添了 `size` 字段来表示单链表的长度，需要在元素有变动时及时修改 `size` 的值。

下面就几个主要的单链表基本操作进行分析。

1.3.2.1 单链表的遍历、查找、插入与删除

对于普通的**遍历**操作，就是按照序号依次访问单链表的每个数据元素，也就是按照各结点的 next 域指示的逻辑关系访问每个结点。具体来说，设置一个活动指针从头节点依次指向下一个结点，直到遇到为标志（设置为 nullptr）。C++实现如下：

```
Node<Entry> *ptr = head->next;
while (ptr != nullptr) {
    // 具体操作
    ptr = ptr->next;
}
```

在单链表的**按值查找**操作中，可以添加累加器，在循环的“具体操作”中累加累加器，当查找成功直接返回累加器的值即为序号。

在**按位查找**的操作中，可以修改循环条件为

```
ptr != nullptr && --index
```

即每次循环将 index 减 1，并判断是否为 0，配合活动指针 ptr 的初始值 head->next，从而将 ptr 定位到位置 index 的元素结束循环。整个过程可用 C++描述如下：

```
template<typename Entry>
Entry &LinkedList<Entry>::get(int index) {
    if (index < 1 || index > size) {
        throw std::out_of_range{"The argument 'index' is out of range"};
    }
    Node<Entry> *ptr = head->next;
    while (ptr != nullptr && --index) {
        ptr = ptr->next;
    }
    return ptr->data;
}
```

更多地，在元素的**插入和删除**操作中，由于单链表的每个结点只能访问到它的后继结点，需要将 ptr 定位到要插入/删除的前一个元素的位置，这是只需在上述按位查找操作的基础上修改活动指针 ptr 的初始值为头节点（head）。如下 C++描述：

```
Node<Entry> *ptr = head;
// find the node in index-1
while (ptr != nullptr && --index) {
    ptr = ptr->next;
}
```

定位到 index 位置的前一个位置后，即可进行插入/删除。具体来说，**插入**需要申请新的结点（data 域为插入元素，next 域为原 ptr 的 next 域指向的地址），将它置于 ptr 的 next 域，并使 size 自增 1。如下 C++描述：

```
ptr->next = new Node<Entry>{elem, ptr->next};  ++size;
```

与此类似，**删除**需要保存数据元素和待删结点地址，将前一个结点的后继（next）指向删除节点的后继，并释放待删结点空间。

以上操作的问题规模都是单链表的长度 n。易知，遍历单链表的时间复杂度为 $O(n)$。可以看出，在等概率的前提下，查找（按值/按位）元素的时间复杂度是 $O(n)$。而无论是插入还是删除，时间都主要耗费在查找正确的位置上，因此时间复杂度也为 $O(n)$。

1.3.2.2 单链表的建立

a) 尾插法：新申请的结点置于尾结点的后面

```
Node<Entry> *thisPtr = head, *otherPtr = list.head->next;
while (otherPtr != nullptr) {
    thisPtr->next = new Node<Entry>{otherPtr->data, nullptr};
    thisPtr = thisPtr->next;
    otherPtr = otherPtr->next;
}
```

尾插法的一个优点为可以保证插入的顺序与元素在单链表中的序号是一致的，故用来进行单链表的拷贝。

b) 头插法：新申请的结点置于头结点之后

```
auto newNode = new Node<Entry>{dataArray[i], head->next};
head->next = newNode;
```

头插法的一个不足之处是插入顺序与序号逆序，用于初始化 n 个元素的单链表时可以**逆序遍历**这 n 个元素，以得到顺序的单链表。

c) 空链表的建立：生成头结点，next 域置空。

1.3.2.3 单链表的析构与清空

析构和清空的一个主要区别在于，析构之后整个链表的存储空间都会被收回，处于不可用的状态；而清空只是回收数据元素的空间，单链表的头结点还在，处于可用的状态。因此，在具体实现上区别体现在删除结点的开始位置不同：析构函数是从头结点开始回收，而清空方法时从头结点的后继节点开始回收。从 ptr 指向的位置开始回收的算法可用如下 C++代码描述：

```
while (ptr != nullptr) {
```

```
    auto delPtr = head;
    ptr = ptr->next;
    delete delPtr;
}
```

其中，析构函数 ptr 指向 head；清空方法 ptr 指向 head->next，且 size 置为 0。

1.4 双链表（单链表的扩展）

双链表与单链表类似，都是采用的链式的存储结构，都由头指针确定，而双链表可用很方便地访问到任一个结点的前驱结点。故其结点可定义为：

```
template<typename Entry>
struct DoubleNode {
    DoubleNode<Entry> *prior;
    Entry data;
    DoubleNode<Entry> *next;
};
```

在双链表中，查找（按值/按位）等遍历操作的实现与单链表基本相同，主要的区别在插入和删除的地方有差异。

具体的代码实现详见附件“Src/util/container/double_linked_list.hpp”

应该注意的是，以下步骤的顺序不可颠倒。

1.4.1 插入操作

为维护好每个结点的前驱后继的逻辑关系，在活动指针 ptr 指向的结点之后插入一个结点，在单链表的基础上要做以下调整：

- 生成新的结点 s，元素为插入元素，前驱为 ptr，后继为 ptr 的后继结点
- 如果 ptr 的后继不为空，则 ptr 后继的前驱修改为 s
- ptr 的后继修改为 s

C++代码描述如下：

```
auto newNode = new DoubleNode<Entry>{ptr, elem, ptr->next};
if (ptr->next != nullptr) {
    (ptr->next)->prior = newNode;
}
ptr->next = newNode;
```

注：尾插法由于总是在尾部插入，直接修改 ptr 的后继为新结点的地址即可。

1.4.2 删除操作

为维护好每个结点的前驱后继的逻辑关系，删除活动指针 ptr 指向的结点，在单链表的基础上要做以下调整：

- ptr 的前驱结点的后继结点修改为当前 ptr 的后继结点
- 如果修改后的 ptr 的后继结点不为空，修改其前驱结点为 ptr 的前驱结点

C++代码描述如下：

```
(ptr->prior)->next = ptr->next;
if (ptr->next != nullptr) {
    (ptr->next)->prior = ptr->prior;
}
```

2 长整数的设计实现

2.1 数据结构设计

比较线性表的两种存储结构，顺序表是预先分配存储空间的数据结构，其最大的元素数量是确定的，而且插入删除操作需要大量移动元素；而链表动态分配存储空间，在计算机存储空间充足的前提下，可以认为其可用存储的数据元素是无限大的，而且插入和删除操作的时间花费在查找元素上。基于以上两点，考虑到长整数的位数或许很大、或许很小，而且会不断的插入元素，为避免不必要的冗余和元素的移动，选用链表作为长整数的数据结构。

具体来说，私有继承上面实现好的 LinkedList 类，并指定模板参数为 int，以此来派生出 BigInteger 类。每个数据元素表示一位数字，构成链表的从前到后各位依次为长整数从小到大各位数字，以此作为长整数的**数据模型**。对内部的链表进行一次遍历，不断把得到的数字插入到字符串之前并去除前导 0 便可以得到该长整数（这里描述的是 to_string 算法）。**基本操作**为在这种数据模型上的加法、减法、乘法，并提供不改变操作数的方法和改变左操作数的原地操作方法（以 self_为前缀）。

注：为减少代码冗余和不一致性，不改变操作数的方法会调用实现好的原地操作方法。

此外，为了表示长整数的正负，设计枚举类 IntegerSign 来抽象正负：

```
enum class IntegerSign {
    positive, negative
};
```

并在 BigInteger 类中添加字段 sign 来表示正负。

声明详见附件“Src/util/integer.h”；代码实现细节详见附件“Src/util/integer.cpp”

使用 C++描述的长整数 `BigInteger` 类如下：

```
class BigInteger : LinkedList<int> {
public:
    BigInteger();
    BigInteger(const std::string &str);
    BigInteger(const BigInteger &integer);
    std::string to_string() const;
    BigInteger plus(const BigInteger &other) const;
    BigInteger &self_plus(const BigInteger &other);
    BigInteger multiply(const BigInteger &other) const;
    BigInteger &self_multiply(const BigInteger &other);
    BigInteger minus(const BigInteger &other) const;
    BigInteger &self_minus(const BigInteger &other);
private:
    IntegerSign sign;
    // 下面是为了简化基本操作设计的辅助函数
    void change_sign();
    bool abs_little_than(const BigInteger &other) const;
    static void do_minus(BigInteger &left, const BigInteger &right);
    static void do_multiply(const BigInteger &left,
                            int right,
                            BigInteger &result);
};
```

2.2 加法算法

【算法设计】

对于有符号整数的加法，可分为以下几类：（暂不考虑为 0 的情况）

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

对于(1)和(2)，即两数符号相同，可以直接使用同符号数的加法；对于(3)和(4)，即两数符号不同，可以视作左操作数减去右操作数的相反数，这样两操作数的符号相同，便于调用同符号数的减法。

注：取相反数由算法 `change_sign` 实现，该算法较为简单，仅改变整数的符号，具体详见附件“Src/util/integer.cpp”

总的来说，加法和减法都是**实现同符号数的运算**，符号不同则**转换为同符号数，再调用转换后的运算方法**，以**简化算法的复杂性**。如下描述：

```
BigInteger &BigInteger::self_plus(const BigInteger &other) {
    if (sign == other.sign) {  // 符号相同相加
```

```
        /* …… */
    } else {  // 符号不同则调用减法
        BigInteger integer(other);
        integer.change_sign();
        self_minus(integer);
    }
    return *this;
}
```

由于不改变操作数的方法 `plus` 会调用实现好的原地操作方法 `self_plus`，故下面讨论原地操作方法 `self_plus` 的算法。

算法的总体思想是：使用两个活动指针 `p`，`q` 分别同时向后遍历两数并做加法（加到左操作数上），若大于 10 则将十位数加到左操作数的下一位上：

```
p->data += q->data;
if (p->data >= 10) {
    (p->next)->data += p->data / 10;
    p->data %= 10;
}
```

直到两数之一先达到末尾，进入尾部处理。尾部处理是，若左操作数没有遍历到末尾 `(p != nullptr)`，则继续遍历，若当前位置数据域大于 10 则将十位数加到左操作数的下一位上。

应该注意的是，**加法操作过程中**和**尾部处理当前位置数据与大于 10 时**若发现左操作数的下一位为空时就新增一个结点，数据域置 0，再执行后续操作。

```
if (p->next == nullptr) {
    p->next = new Node<int>{0, nullptr};  ++size;
}
```

【时间复杂度】

算法的数据规模为两长整数的位数（包括前导 0）m 和 n，基本操作为按位加法，其执行次数取决于更长的整数的位数，记$N = \max\{m,n\}$，故时间复杂度为$O(N)$，为线性阶。

2.3 减法算法

【算法设计】

与加法算法处理思路基本相同：**实现同符号数的运算**，符号不同则**转换为同符号数，再调用转换后的运算方法**，以**简化算法的复杂性**。具体来说，减法运算中符号不同的两操作数，改变右操作数的符号，并调用加法；对于符号相同的两操作数，直接使用减法算法。

由于不改变操作数的方法 `minus` 会调用实现好的原地操作方法

`self_minus`，故下面讨论原地操作方法 `self_minus` 的算法。

考虑到减法中同符号的两操作数绝对值小的数字减去绝对值大的数字会发生符号的变化，而不处理好这种变化会出现溢出现象，所以为保证操作的统一性，设计**保证左操作数的绝对值一定比右操作数的绝对值大**，在此基础上实现减法。

具体来说，减法算法需要三个子算法来组成，分别是主要算法 `do_minus` 专门负责减法运算（左操作数的绝对值大于右操作数的绝对值）、辅助算法 `abs_little_than` 比较长整数绝对值大小。

由于长整数的数据模型不保证最高位的后继不存在前导零，因此辅助算法 `abs_little_than` 的主要思路是去比较两长整数遍历形成不含前导零的字符串的大小（利用 `to_string` 算法，详见代码）。长度相同则按照字典序比较，否则按长度比较。

主要算法 `do_minus` 的主要思路是使用两个活动指针 `p`，`q` 分别同时向后遍历两数并做减法（在左操作数上做减法），若小于 0 则加 10，且左操作数的下一位减 1：

```
p->data -= q->data;
if (p->data < 0) {
    (p->next)->data -= 1;
    p->data += 10;
}
```

直到两数之一先达到末尾，进入尾部处理。尾部处理是，若左操作数没有遍历到末尾 `(p != nullptr)`，则继续遍历，若当前位置数据域小于 0 则加 10，且左操作数的下一位减 1。

由于保证了左操作数的绝对值大于右操作数的绝对值，每次判断左操作数的下一位都会保证不为空，故不需特殊处理。

另外交换两长整数的算法是通过基类的清空算法和加法算法实现的。

综上所述，减法算法可由如下 C++代码描述：

```
BigInteger &BigInteger::self_minus(const BigInteger &other) {
    if (sign == other.sign) {    // 符号相同相减
        // 确保左操作数的绝对值大于右操作数的绝对值
        if (this->abs_little_than(other)) {
            // 如果左操作数的绝对值 < 右操作数的绝对值
            // 交换两操作数
            const BigInteger right(*this);  // 右操作数
            clear(); insert(1, 0);  // 左操作数 = 0
            self_plus(other);
            // 交换完成，相减
            do_minus(*this, right);
```

```
            change_sign();  // 取相反数
        } else {
            // 左操作数的绝对值大于右操作数的绝对值，直接相减
            do_minus(*this, other);
        }
    } else {    // 符号不同调用加法
        BigInteger integer(other);
        integer.change_sign();
        self_plus(integer);
    }
    return *this;
}
```

【时间复杂度】

算法的数据规模为两长整数的位数（包括前导 0）m 和 n，记$N = \max\{m,n\}$，易知各子算法的时间复杂度为$O(N)$，且各子算法在减法算法中的执行次数为常量，故减法的时间复杂度也为$O(N)$，为线性阶。

2.4 乘法算法

【算法设计】

a) 数字处理

乘法可以看作是右操作数的每位数字逐一乘以左操作数，并将结果按右操作数数字的位置扩大到相应倍数后累加的过程。因此乘法可以分解成普通整数乘以长整数算法 `do_multiply`、扩大倍数算法和加法三个子算法。其中扩大倍数算法可以利用基类中已经实现好的插入算法，不断在位置 1 插入元素 0 即可实现按倍数扩大；加法算法直接使用上面实现的 `self_plus` 方法。

同样的，不改变操作数的方法 `multiply` 会调用实现好的原地操作方法 `self_multiply`。下面着重讨论普通整数乘以长整数算法 `do_multiply`。

`do_multiply` 算法的主要思路是：用变量保存要进到本位的数字（初始化为 `0`），使用活动指针 `p` 向后遍历长整数的每一位，乘以普通整数并加上进位数字保存到结果数字中。如果本位大于 `10` 则将十位数保存到进位数字中；否则进位数字置为 `0`。结果数字下一位为空时就新增一个结点，数据域置为 0（注意维护好结果数字的 `size` 属性）：

注：左操作数为长整数，右操作数为普通整数，结果整数为长整数 0。

```
auto p = left.head->next, q = result.head->next;
int next_bit = 0;
while (p != nullptr) {
    q->data = p->data * right + next_bit;
    if (q->data >= 10) {
```

```
            next_bit = q->data / 10;
            q->data %= 10;
        } else {
            next_bit = 0;
        }
        q->next = new Node<int>{0, nullptr};  result.size++;
        p = p->next;  q = q->next;
    }
```

循环结束进入尾部处理：如果进位数字不为 0，则将进位数字置于结果数字的末尾。

b) 符号处理

有符号数的乘法的符号遵循“同号为正，异号为负”的规则。算法实现上依照此规则很容易实现。

3 总结与反思

针对本实验，程序提供了一个简易的长整数计算器用于测试（编译后生成的 IntegerCalculator 二进制文件，源代码见附件“Src/calculator.cpp”），同时也是一个批处理大量长整数加减乘运算的命令行工具。 (下面为该工具编译运行的截图)

```
holger@HONOR-LAPTOP:/mnt/d/WorkSpace/Codes/School/DS-Class/List$ cd Src
holger@HONOR-LAPTOP:/mnt/d/WorkSpace/Codes/School/DS-Class/List/Src$ mkdir build
holger@HONOR-LAPTOP:/mnt/d/WorkSpace/Codes/School/DS-Class/List/Src$ cd build
holger@HONOR-LAPTOP:/mnt/d/WorkSpace/Codes/School/DS-Class/List/Src/build$ cmake ..
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/WorkSpace/Codes/School/DS-Class/List/Src/build
holger@HONOR-LAPTOP:/mnt/d/WorkSpace/Codes/School/DS-Class/List/Src/build$ make
Scanning dependencies of target IntegerCalculator
[ 33%] Building CXX object CMakeFiles/IntegerCalculator.dir/util/integer.cpp.o
[ 66%] Building CXX object CMakeFiles/IntegerCalculator.dir/calculator.cpp.o
[100%] Linking CXX executable IntegerCalculator
[100%] Built target IntegerCalculator
holger@HONOR-LAPTOP:/mnt/d/WorkSpace/Codes/School/DS-Class/List/Src/build$ ./IntegerCalculator
Big Integer Calculator (Sept. 30 2020) By Holger Zhang
Type "exit" to exit the shell.

[[NOTICE]] The operands and the operator must be divided by at least one blank.
<Sample Input>
>>> 6 * 7     # LeftOperand Operator RightOperand
42

>>> 12345678987654321 * 19207
237123456315876543447
>>> exit
holger@HONOR-LAPTOP:/mnt/d/WorkSpace/Codes/School/DS-Class/List/Src/build$
```

长整数不仅仅可以通过单链表来实现，也可以使用顺序表来模拟，而且存储空间的冗余问题在两者身上都有所体现：单链表为维护元素之间的逻辑关系，存储了大量的地址信息；顺序表的存储空间是一次性分配的，在不太大的数字上也存在冗余。基于这一问题，我认为可以在本实验的基础上改进，每个数据元素不在保存一位数，而是保存多位数（例如在 C/C++语言中元素类型为 int 时，保存范围在 0 ~ 999999999 的数字），这样可以解决大量指针占用的存储空间这一问题。