2023—2024 学年第二学期

《数据结构与算法导论》实验报告

班级：2023211119

姓名：XXX

学号：2023210XXX

班内序号：07

报告日期：2024-04-15

数据结构实验报告

实验名称： 实验 1——线性表

学生姓名： XXX

班　　级： 2023211119

班内序号： 07

学　　号： 2023210XXX

日　　期： 2024 年 4 月 15 日

1．实验要求

[正文格式要求]

字体：汉字宋体、英文 Times New Roman

字号：五号

颜色：黑色

行距：单倍行距

[内容要求]

要求：实现线性表的基本功能

构造：使用头插法、尾插法两种方法；

插入：要求建立的链表按照关键字从小到大有序

删除；

查找；

获取链表长度；

销毁；

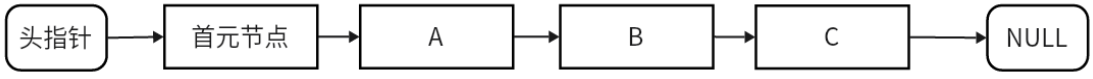
其他：可自行定义；

编写测试 main()函数测试线性表的正确性。

2. 程序分析

2.1 存储结构

不带头节点的单链表



2.2 关键算法分析

1.关键算法：

a.构造链表

a.1.根据传入的数组生成链表，通过尾插法将数据依次添加到链表尾部。

function LinkedList(const T* arr, int size) {

```
        // 初始化链表的头指针为 nullptr
        head = nullptr
        // 遍历数组，将每个元素插入到链表中
        for i from 0 to size-1 {
            insert(arr[i])
        }
    }
```

a.2.生成空链表。

```
    function LinkedList() {
        // 初始化链表的头指针为 nullptr，表示链表为空
        head = nullptr
        // 初始化链表的大小为 0
        size = 0
    }
```

b.添加元素

b.1.头插、尾插以及在指定位置插入。

```
    function insert(const T& value, int position = -1) {
        // 创建一个新节点
        newNode = createNode(value)
        // 如果链表为空或插入位置为 0（链表头部），将新节点插入头部
        if head is nullptr or position == 0 {
            newNode.next = head
            head = newNode
        } else {
            // 寻找插入位置的前一个节点
            current = head
            currentPos = 0
            while current.next is not nullptr and currentPos < position - 1 {
                current = current.next
                currentPos = currentPos + 1
            }
            // 插入新节点
            newNode.next = current.next
            current.next = newNode
        }
        // 增加链表大小
        size = size + 1
    }
```

b.2.根据大小自动插入。

```
    function autoInsert(const T& value) {
        // 创建一个新节点
        newNode = createNode(value)
        // 如果链表为空或头部节点的值大于新节点的值，将新节点插入头部
```

```
        if head is nullptr or head.data > value {
            newNode.next = head
            head = newNode
        } else {
            // 寻找插入位置的前一个节点
            current = head
            while current.next is not nullptr and current.next.data < value {
                current = current.next
            }
            // 插入新节点
            newNode.next = current.next
            current.next = newNode
        }
        // 增加链表大小
        size = size + 1
    }
```

c.删除

根据元素来删除链表节点，通过遍历删除最先出现的元素。

```
    function remove(const T& value) {
        // 如果链表为空，返回 false
        if head is nullptr {
            return false
        }
        // 如果头部节点的值等于要删除的值，删除头部节点
        if head.data == value {
            temp = head
            head = head.next
            delete temp
            size = size - 1
            return true
        }
        // 遍历链表寻找要删除的节点
        current = head
        while current.next is not nullptr {
            if current.next.data == value {
                temp = current.next
                current.next = current.next.next
                delete temp
                size = size - 1
                return true
            }
            current = current.next
        }
```

```
        // 如果没有找到要删除的节点，返回 false
        return false
    }
```

d.查找

d.1.根据元素索引位置。

```
    function findByValue(const T& value) {
        // 初始化当前节点为链表头
        current = head
        // 初始化索引为 0
        index = 0
        // 遍历链表
        while current is not nullptr {
            // 如果当前节点的值等于要查找的值，返回索引
            if current.data == value {
                return index
            }
            current = current.next
            index = index + 1
        }
        // 如果没有找到，返回-1
        return -1
    }
```

d.2.根据位置索引元素。

```
    function findByIndex(int index) {
        // 如果索引无效，返回无效值
        if index < 0 or index >= size {
            return invalid value
        }
        // 初始化当前节点为链表头
        current = head
        // 遍历链表到指定索引
        for i from 0 to index-1 {
            current = current.next
        }
        // 返回找到的元素
        return current.data
    }
```

e.长度

根据 private 中的信息显示当前链表元素个数。

```
    function getSize() {
        // 返回链表的大小
        return size
```

```
}
```

f.销毁

编写析构函数释放内存。

```
function ~LinkedList() {
    // 遍历链表，释放每个节点的内存
    current = head
    while current is not nullptr {
        temp = current
        current = current.next
        delete temp
    }
    // 清空链表大小
    size = 0
}
```

g.排序

对于能用“>”比较的数值，可以通过 sort 进行排序。

```
function sort() {
    // 如果链表为空或只有一个元素，直接返回
    if head is nullptr or head.next is nullptr {
        return
    }
    // 初始化一个布尔变量，用于标记是否发生了交换
    swapped = false
    // 遍历链表，进行冒泡排序
    do {
        swapped = false
        current = head
        // 遍历链表中非尾部的节点
        while current.next is not nullptr {
            // 如果当前节点的值大于下一个节点的值，交换它们
            if current.data > current.next.data {
                T temp = current.data
                current.data = current.next.data
                current.next.data = temp
                swapped = true
            }
            current = current.next
        }
    } while (swapped)
}
```

h.打印

将整个链表打印出来。

```
function print() {
    // 遍历链表，打印每个节点的值，直到链表尾部
    current = head
    while current is not nullptr {
        print(current.data)
        if current.next is not nullptr {
            print(" -> ")
        }
        current = current.next
    }
    print("\n")
}
```

2.代码详细分析

a.构造链表

a.1.带参数的构造函数

初始化链表的头指针为 nullptr，表示链表为空，并设置链表的元素计数为 0。

a.2.无参数的构造函数

接受一个数组和数组的长度作为参数。初始化链表的头指针为 nullptr，然后遍历数组，使用 insert 方法将数组中的每个元素按顺序插入链表的末尾，同时更新链表的大小。

b.添加元素

b.1.在指定 position 插入元素

创建一个新节点并赋值。如果链表为空（头指针为 nullptr）或插入位置为 0（表示链表头部），将新节点插入链表前端，更新头指针。否则，遍历链表找到指定位置的前一个节点，将新节点插入到该节点之后，并更新后续节点的指针。最后，链表大小加 1。

b.2.自动插入元素

创建一个新节点并赋值。如果链表为空或新节点的值小于等于头节点的值，将新节点插入链表前端。否则，遍历链表找到第一个值大于新节点值的节点，将新节点插入到该节点之前。最后，链表大小加 1。

c.删除

检查链表是否为空，如果为空则返回 false。检查头节点的值是否与要删除的值相等，如果相等则删除头节点并返回 true。否则，遍历链表找到要删除的节点，更新前一个节点的指针跳过当前节点，并删除当前节点。每次删除后，链表大小减 1。如果找不到要删除的值，返回 false。

d.查找

d.1.根据元素索引位置。

从头节点开始，遍历链表，比较每个节点的值与要查找的值。如果找到匹配的值，返回当前节点的索引。如果遍历完整个链表都没有找到，返回-1。

d.2.根据位置索引元素。

检查索引是否在链表的有效范围内。如果是，从头节点开始遍历链表，直到找到对应索引的节点，并返回该节点的值。如果索引无效，抛出异常。

e.长度

返回链表中元素的数量，即 size 成员变量的值。

f.销毁

释放链表中所有节点所占用的内存资源。从头节点开始，遍历链表，释放每个节点，直到链表为空。然后将头指针设置为 nullptr，并将链表大小重置为 0。

g.排序

对链表进行冒泡排序。遍历链表，比较相邻的节点值，如果前一个节点的值大于后一个节点的值，则交换这两个节点的值。重复这个过程，直到整个链表有序。

h.打印

从头节点开始，遍历链表的每个节点，打印节点的值。如果节点不是链表的最后一个节点，打印节点间的分隔符（例如" -> "）。遍历完成后，打印换行符。

3.时间和空间复杂度（n 是数组的大小）

a.构造链表

a.1.带参数的构造函数

时间复杂度：O(n)，空间复杂度：O(1)

a.2.无参数的构造函数

时间复杂度：O(1)，空间复杂度：O(1)

b.添加元素

b.1.在指定 position 插入元素

b.2.自动插入元素

时间复杂度: O(n)，空间复杂度: O(1)

c.删除

时间复杂度: O(n)，空间复杂度: O(1)

d.查找

d.1.根据元素索引位置。

时间复杂度: O(n)，空间复杂度: O(1)

d.2.根据位置索引元素。

时间复杂度: O(i)，空间复杂度: O(1)

e.长度

时间复杂度: O(1)，空间复杂度: O(1)

f.销毁

时间复杂度: O(n)， 空间复杂度: O(n)

g.排序

时间复杂度: O(n^2)，空间复杂度: O(1)

h.打印

时间复杂度: O(n)，空间复杂度: O(1)

3. 程序运行结果

测试主函数流程：

```
int main() {
    // 创建一个数组
    int arr[] = {10,20,30,40,50,60,70,80,90,100};
    // 使用数组初始化链表
    LinkedList<int> myList(arr, sizeof(arr) / sizeof(arr[0]));
    // 打印链表
    myList.print();
    // 在链表中自动插入值为 35 的节点
    myList.autoInsert(35);
    // 在链表的末尾插入值为 45 的节点
    myList.insert(45);
    // 打印链表
    myList.print();
    // 在链表的头部插入值为 55 的节点
    myList.insert(55, 0);
    // 打印链表
    myList.print();
    // 在链表的第 5 个位置插入值为 120 的节点
    myList.insert(120, 5);
    // 打印链表
    myList.print();
    // 从链表中删除值为 35 的节点，并打印操作结果
    cout << "Removed 35: " << (myList.remove(35) ? "Success" : "Failed") << endl;
    // 尝试从链表中删除值为 110 的节点，并打印操作结果
    cout << "Removed 110: " << (myList.remove(110) ? "Success" : "Failed") << endl;
    // 打印链表
    myList.print();
    // 对链表进行排序
    myList.sort();
    // 打印链表
    myList.print();
    // 查找值为 45 的节点，并打印结果
    cout << "Find value 45: " << myList.findByValue(45)<<endl;
    // 查找第 1 个节点，并打印结果
    cout << "Find index 1: " << myList.findByIndex(1)<<endl;
    // 打印链表的大小
    cout<<myList.getSize()<<endl;
    // 销毁链表
    myList.destroy();
}
```

测试结果：

10 -> 20 -> 30 -> 40 -> 50 -> 60 -> 70 -> 80 -> 90 -> 100
10 -> 20 -> 30 -> 35 -> 40 -> 50 -> 60 -> 70 -> 80 -> 90 -> 100 -> 45
55 -> 10 -> 20 -> 30 -> 35 -> 40 -> 50 -> 60 -> 70 -> 80 -> 90 -> 100 -> 45
55 -> 10 -> 20 -> 30 -> 35 -> 120 -> 40 -> 50 -> 60 -> 70 -> 80 -> 90 -> 100 -> 45
Removed 35: Success
Removed 110: Failed
55 -> 10 -> 20 -> 30 -> 120 -> 40 -> 50 -> 60 -> 70 -> 80 -> 90 -> 100 -> 45
10 -> 20 -> 30 -> 40 -> 45 -> 50 -> 55 -> 60 -> 70 -> 80 -> 90 -> 100 -> 120
Find value 45: 4
Find index 1: 20
13
测试条件：插入、删除元素的位置如何选择，尝试删除不存在的元素。
测试结论：功能实现完全。

4. 总结

1. 调试时出现的问题及解决方法：
内存泄漏
在早期版本的代码中，可能没有实现析构函数`~LinkedList()`，或者析构函数没有正确释放所有节点的内存。这会导致程序在创建和销毁多个链表对象时发生内存泄漏。
解决方法
确保每个链表对象在析构时通过`destroy()`函数正确释放所有节点的内存。在析构函数中，从头节点开始，遍历链表，释放每个节点所占用的内存。

2. 心得体会：
代码结构的重要性
通过编写和调试链表类，可以深刻体会到良好的代码结构对于程序的可读性和可维护性的重要性。清晰的类设计和函数分工有助于快速定位和解决问题。
内存管理的复杂性
内存管理是 C++编程中的一个重要方面。正确地分配和释放内存对于避免内存泄漏和其他资源管理问题至关重要。

3. 下一步的改进：
改进内存管理
可以进一步优化内存管理策略，例如使用智能指针来自动管理内存，减少手动`new`和`delete`操作，降低内存泄漏的风险。
优化算法
实现更高效的排序算法，并考虑对插入和删除操作进行优化，使其能够更快地适应大型数据集。
扩展功能
可以考虑为链表类增加更多的功能，如反转链表、合并链表、查找特定条件的节点等，以提高链表类的实用性。

数据结构实验报告

实验名称： 实验 4——一元多项式

学生姓名： XXX

班　　级： 2023211119

班内序号： 07

学　　号： 2023210XXX

日　　期： 2024 年 4 月 15 日

1．实验要求

[正文格式要求]

字体：汉字宋体、英文 Times New Roman

字号：五号

颜色：黑色

行距：单倍行距

[内容要求]

要求：

能够实现一元多项式的输入和输出；

能够进行一元多项式相加；

能够进行一元多项式相减；

能够计算一元多项式在 x 处的值；

能够计算一元多项式的导数；

能够进行一元多项式相乘；

编写测试 main()函数测试线性表的正确性获取链表长度；

销毁；

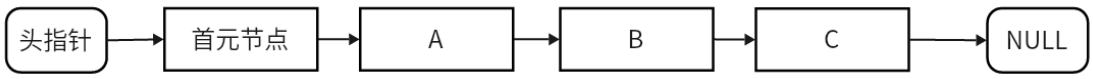
其他：可自行定义；

编写测试 main()函数测试线性表的正确性。

2. 程序分析

2.1 存储结构

不带头节点的单链表



2.2 关键算法分析

1.关键算法：

a.构造链表

a.1.根据传入的数组生成链表，通过比较指数实现从小到大排列存入。

```
Polynomial(const int *coefficients, const int *exponents, int size) {
    head = nullptr; // 初始化链表头部为 null
    size = size; // 初始化多项式的大小为传入的 size 值
    FOR i FROM 0 TO size-1 {
        insertTerm(coefficients[i], exponents[i]); // 插入每一项到多项式中
    }
}
```

a.2.根据传入的 size 生成空多项式。

```
Polynomial(int size) {
    head = nullptr; // 初始化链表头部为 null
    size = size; // 初始化多项式的大小为传入的 size 值
}
```

a.3.生成空链表。

```
function LinkedList() {
    // 初始化链表的头指针为 nullptr，表示链表为空
    head = nullptr
    // 初始化链表的大小为 0
    size = 0
}
```

b. 插入某一指定项

```
void insertTerm(int coefficient, int exponent) {
    IF coefficient == 0 THEN RETURN; // 如果系数为 0，则直接返回

    Term *current = head; // 定义 current 指针从链表头节点开始
    Term *prev = nullptr; // 定义 prev 指针为 null
    WHILE current != nullptr AND current->exponent < exponent {
        prev = current; // 如果 current 的指数小于要插入的指数，更新 prev
        current = current->next; // 移动 current 到下一个节点
    }
    IF current != nullptr AND current->exponent == exponent THEN {
        current->coefficient += coefficient; // 如果找到相同指数的项，累加系数
        IF current->coefficient == 0 THEN {
            IF prev != nullptr THEN {
                prev->next = current->next; // 移除 current 节点
            } ELSE {
                head = current->next; // 如果是头节点，则更新头节点
            }
            delete current; // 删除 current 节点
        }
    } ELSE {
        Term *newTerm = new Term(coefficient, exponent); // 创建新项
        IF prev != nullptr THEN {
            newTerm->next = prev->next; // 新项的 next 指向 prev 的下一个节点
```

```
                prev->next = newTerm; // 更新 prev 的下一个节点为新项
            } ELSE {
                newTerm->next = head; // 新项的 next 指向当前的头节点
                head = newTerm; // 更新头节点为新项
            }
        }
    }
```

c.加法

```
    Polynomial operator+(const Polynomial &other) const {
        Polynomial result; // 创建结果多项式
        Term *current1 = head; // 定义 current1 从第一个多项式的头节点开始
        Term *current2 = other.head; // 定义 current2 从第二个多项式的头节点开始

        WHILE current1 != nullptr AND current2 != nullptr {
            IF current1->exponent == current2->exponent THEN {
                result.insertTerm(current1->coefficient + current2->coefficient,
    current1->exponent); // 合并相同指数的项
                current1 = current1->next; // 移动 current1 到下一个节点
                current2 = current2->next; // 移动 current2 到下一个节点
            } ELSE IF current1->exponent < current2->exponent THEN {
                result.insertTerm(current1->coefficient, current1->exponent); // 插入
    index1 中较小指数的项
                current1 = current1->next; // 移动 current1 到下一个节点
            } ELSE {
                result.insertTerm(current2->coefficient, current2->exponent); // 插入
    index2 中较小指数的项
                current2 = current2->next; // 移动 current2 到下一个节点
            }
        }

        WHILE current1 != nullptr {
            result.insertTerm(current1->coefficient, current1->exponent); // 插入剩余的
    项
            current1 = current1->next;
        }

        WHILE current2 != nullptr {
            result.insertTerm(current2->coefficient, current2->exponent); // 插入剩余的
    项
            current2 = current2->next;
        }
        RETURN result; // 返回结果多项式
    }
```

d.减法

```
Polynomial operator-(const Polynomial &other) const {
    Polynomial result; // 创建结果多项式
    Term *current1 = head; // 定义 current1 从第一个多项式的头节点开始
    Term *current2 = other.head; // 定义 current2 从第二个多项式的头节点开始

    WHILE current1 != nullptr AND current2 != nullptr {
        IF current1->exponent == current2->exponent THEN {
            result.insertTerm(current1->coefficient - current2->coefficient,
current1->exponent); // 合并相同指数的项
            current1 = current1->next; // 移动 current1 到下一个节点
            current2 = current2->next; // 移动 current2 到下一个节点
        } ELSE IF current1->exponent < current2->exponent THEN {
            result.insertTerm(current1->coefficient, current1->exponent); // 插入
index1 中较小指数的项
            current1 = current1->next; // 移动 current1 到下一个节点
        } ELSE {
            result.insertTerm(-current2->coefficient, current2->exponent); // 插入负
的 index2 中较小指数的项
            current2 = current2->next; // 移动 current2 到下一个节点
        }
    }

    WHILE current1 != nullptr {
        result.insertTerm(current1->coefficient, current1->exponent); // 插入剩余的
项
        current1 = current1->next;
    }

    WHILE current2 != nullptr {
        result.insertTerm(-current2->coefficient, current2->exponent); // 插入负的剩
余的项
        current2 = current2->next;
    }

    RETURN result; // 返回结果多项式
}
```

e. 乘法

```
Polynomial operator*(const Polynomial &other) const {
    Polynomial result; // 创建一个新的多项式对象，用于存储乘法结果
    for (Term *term1 = this->head; term1 != nullptr; term1 = term1->next) {
        for (Term *term2 = other.head; term2 != nullptr; term2 = term2->next) {
            // 计算两个多项式的每个项的乘积
            int newCoefficient = term1->coefficient * term2->coefficient;
            int newExponent = term1->exponent + term2->exponent;
```

```
                    // 将乘积作为新项插入到结果多项式中
                    result.insertTerm(newCoefficient, newExponent);
                }
            }
            return result; // 返回乘法结果的多项式
        }
```

f.求值

```
        float evaluate(float x) const {
            float result = 0.0; // 初始化结果为 0

            FOR each term IN the polynomial {
                result += term->coefficient * pow(x, term->exponent); // 计算每一项的值并累加到结果中
            }

            RETURN result; // 返回计算结果
        }
```

g.求导

```
        Polynomial derivative() const {
            Polynomial result; // 创建一个新的多项式对象，用于存储求导结果
            Term *current = this->head; // 从当前多项式的头节点开始遍历
            while (current != nullptr) {
                // 如果项的指数不为 0，则进行求导操作
                if (current->exponent != 0) {
                    // 插入一项到结果多项式中，系数为原系数乘以指数，指数减 1
                    result.insertTerm(current->coefficient * current->exponent, current->exponent - 1);
                }
                current = current->next; // 移动到下一个项
            }
            return result; // 返回求导后的多项式
        }
```

h.析构

```
        void destroy() {
            Term *current = this->head; // 从当前多项式的头节点开始遍历
            while (current != nullptr) {
                Term *next = current->next; // 保存当前项的下一个项的地址
                delete current; // 释放当前项的内存
                current = next; // 移动到下一个项
            }
            this->head = nullptr; // 清空头节点指针
        }
```

2.代码详细分析

a.构造链表

a.1. Polynomial(): 默认构造函数，初始化一个空多项式，head 指针设置为 nullptr，size 设置为 0。

a.2. Polynomial(int size): 构造函数，接受一个整数参数 size，初始化一个具有指定大小的多项式。head 指针设置为 nullptr，size 设置为 size。

a.3. Polynomial(const int *coefficients, const int *exponents, int size): 构造函数，接受三个参数：系数数组、指数数组和项数。这个构造函数用于根据给定的系数和指数初始化多项式。它通过循环遍历系数和指数数组，使用 insertTerm 方法将每一项添加到多项式中。

b.插入某一指定项

void insertTerm(int coefficient, int exponent): 此方法用于向多项式中插入一个新的项。它首先检查系数是否为 0，如果是，则不执行任何操作。然后，它遍历链表以找到应该插入新项的位置。如果找到具有相同指数的项，则更新其系数。如果系数变为 0，则删除该项。如果新项应该插入到链表的开头，则将其设置为新的头节点。最后，如果新项具有唯一的指数，则将其添加到链表的末尾。

c.加法

Polynomial operator+(const Polynomial &other) const: 重载加法运算符以实现多项式的加法。它创建一个新的 Polynomial 对象来存储结果。对于两个多项式的每一项，如果指数相同，则合并它们的系数；如果一个多项式的项指数小于另一个，则将较大的项插入结果多项式。最后，它将剩余的项添加到结果中。

d.减法

Polynomial operator-(const Polynomial &other) const: 重载减法运算符以实现多项式的减法。与加法类似，它创建一个新的 Polynomial 对象来存储结果。不同之处在于，当遇到相同指数的项时，它从第一个多项式的项中减去第二个多项式的项。如果结果系数为负数，它将插入负号。

e. 乘法

Polynomial operator*(const Polynomial &other) const: 重载乘法运算符以实现多项式的乘法。它创建一个新的 Polynomial 对象来存储乘积的结果。对于第一个多项式的每一项和第二个多项式的每一项，它计算它们的乘积并将结果插入到结果多项式中。

f.求值

float evaluate(float x) const: 计算多项式在给定值 x 处的函数值。它通过遍历多项式的每一项，使用 pow 函数计算每一项的幂，并将其累加到结果中。

g.求导

Polynomial derivative() const: 计算多项式的导数。它创建一个新的 Polynomial 对象来存储导数的结果。对于多项式中的每一项，它将指数降低 1 并乘以原系数来得到导数的项。

h.析构

void destroy(): 释放多项式中所有项的内存。它遍历链表，删除每个节点，并将头指针设置为 nullptr。

3.时间和空间复杂度（n 是数组的大小）

a.构造链表

a.1.时间复杂度 O(1)，空间复杂度 O(1)。

a.2.时间复杂度 O(1)，空间复杂度 O(1)。

a.3.时间复杂度 O(size)，空间复杂度 O(size)。

b.插入某一指定项

时间复杂度 O(n)，空间复杂度 O(1)

c.加法

时间复杂度 O(n + m)，空间复杂度 O(n + m)。

d.减法

时间复杂度 O(n + m)，空间复杂度 O(n + m)。

e. 乘法

时间复杂度 O(n^2)，空间复杂度 O(n^2)。

f.求值

时间复杂度 O(n)，空间复杂度 O(1)。

g.求导

时间复杂度 O(n)，空间复杂度 O(n)。

h.析构

时间复杂度 O(n)，空间复杂度 O(1)。

2.3 其他

输出函数大篇幅采用 GPT 的辅助，在这里单独列出：

```
// 重载输出运算符，用于将多项式对象格式化输出到标准输出流
friend std::ostream &operator<<(std::ostream &os, const Polynomial &polynomial) {
    // 遍历多项式的每一项
    Polynomial::Term *current = polynomial.head; // 获取多项式的头节点
    bool firstTerm = true; // 标记是否为多项式的第一项
    // 遍历多项式的所有项
    while (current != nullptr) {
        // 如果当前项的系数为 0，则跳过，继续检查下一个项
        if (current->coefficient == 0) {
            current = current->next;
            continue;
        }
        // 如果是第一项且系数为负，则输出负号
        if (firstTerm && current->coefficient < 0) {
            os << "-";
        }
        // 如果不是第一项，根据当前项的系数是否为负来决定输出" - "或" + "
        if (!firstTerm) {
            if (current->coefficient < 0) {
                os << " - ";
            } else {
                os << " + ";
            }
        }
        // 如果当前项的指数为 0，或者系数不为-1（避免输出"-1"或"1"），则输出系数
        if (current->exponent == 0 || current->coefficient != -1) {
            if (current->coefficient < 0) {
                os << -(current->coefficient);
```

```
            } else {
                if (current->coefficient != 1 || current->exponent == 0) {
                    os << current->coefficient;
                }
            }
        }
        // 如果当前项的指数不为 0，则输出"x"和指数
        if (current->exponent != 0) {
            os << "x";
            if (current->exponent != 1) {
                os << "^" << current->exponent; // 输出指数
            }
        }
        // 移动到下一项，并标记当前项已输出
        current = current->next;
        firstTerm = false; // 设置标记为 false，因为已经输出了至少一项
    }
    // 返回引用，以便继续使用输出流
    return os;
}
```

时间复杂度 O(n)，空间复杂度 O(1)。

3. 程序运行结果

测试主函数流程：

```
int main()
{
    // 测试样例 1
    int coefficients1[] = {1, 0, 2, 0, 4, 5};
    int exponents1[] = {1, 2, 3, 4, 5, 6};
    Polynomial p1(coefficients1, exponents1, 6);
    cout << p1 << endl;
    cout << "The evaluation of p1 is " << p1.evaluate(1) << endl;
    // 测试样例 2
    int coefficients2[] = {-1, 2, -3, 4, -5};
    int exponents2[] = {0, 1, 2, 3, 4};
    Polynomial p2(coefficients2, exponents2, 5);
    cout << p2 << endl;
    cout << "The evaluation of p2 is " << p2.evaluate(-1) << endl;
    // 测试样例 3
    int coefficients3[] = {1, -1, 1, -1, 1};
    int exponents3[] = {0, 1, 2, 3, 4};
    Polynomial p3(coefficients3, exponents3, 5);
    cout << p3 << endl;
```

```
        cout << "The evaluation of p3 is " << p3.evaluate(2) << endl;
        Polynomial p4 = p1 + p2;
        cout << p4 << endl;
        Polynomial p5 = p1 - p2;
        cout << p5 << endl;
        Polynomial p6 = p1.derivative();
        cout << p6 << endl;
        Polynomial p7 = p1 * p2;
        cout << p7 << endl;
        p1.destroy();
        p2.destroy();
        p3.destroy();
        p4.destroy();
        p5.destroy();
        p6.destroy();
        p7.destroy();
        return 0;
    }
```

测试结果：

```
x + 2x^3 + 4x^5 + 5x^6
The evaluation of p1 is 12
-1 + 2x - 3x^2 + 4x^3 - 5x^4
The evaluation of p2 is -15
1 - x + x^2 - x^3 + x^4
The evaluation of p3 is 11
-1 + 3x - 3x^2 + 6x^3 - 5x^4 + 4x^5 + 5x^6
1 - x + 3x^2 - 2x^3 + 5x^4 + 4x^5 + 5x^6
1 + 6x^2 + 20x^4 + 30x^5
-x + 2x^2 - 5x^3 + 8x^4 - 15x^5 + 11x^6 - 12x^7 + x^8 - 25x^10
```

测试条件：插入、删除元素的位置如何选择，测试各个功能。

测试结论：功能实现完全。

4. 总结

1. 调试时出现的问题及解决方法：

把输出重载到“<<”时，总是不能正确显示系数为 1 或者-1 时的情况

对重载整体不是很熟悉，一开始 main 函数总是报错

解决方法

添加 if-else 语句，根据 firstTerm 和 x 的系数来判断是否显示

使用友元函数

2. 心得体会：

代码要尽可能的方便使用，同时要努力提高运行效率，节约空间。在写代码的时候，我将运算符重载，使得多项式的算术运算和输出变得自然和直观，提高了代码的可读性和易用性。

3. 下一步的改进：

在 operator+和 operator-中，可以优化合并和插入项的逻辑，以减少不必要的内存分配和释放。扩展功能

提供一个方法来从标准输入读取多项式，以便用户可以直接在命令行中输入多项式的系数和指数。

提供一个方法来输出多项式的字符串表示，而不是依赖于重载的<<运算符，这样可以提高代码的可测试性和灵活性。