МИНИСТЕРСТВО НАУКИ И ВЫСШЕГООБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

b

**Институт информационных технологий, математики и механики**

**ЛАБОРАТОРНАЯ РАБОТА**

на тему:

**«Аналитические преобразования полиномов**

**от нескольких переменных (списки)»**

**Выполнила:** студентка группы 3822Б1ФИ2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Фролова Е.А.

Подпись

**Проверил:** к.т.н, доцент каф. ВВиСП

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Кустикова В.Д./

Подпись

Нижний Новгород  
2024

**Содержание**

[Введение 3](#_Toc167474359)

[1 Постановка задачи 4](#_Toc167474360)

[2 Руководство пользователя 5](#_Toc167474361)

[2.1 Приложение для демонстрации работы кольцевого односвязного списка 5](#_Toc167474362)

[2.2 Приложение для демонстрации работы полиномов 10](#_Toc167474363)

[3 Руководство программиста 11](#_Toc167474364)

[3.1 Описание алгоритмов 11](#_Toc167474365)

[3.1.1 Линейный односвязный список 11](#_Toc167474366)

[3.1.2 Кольцевой список 14](#_Toc167474367)

[3.1.3 Моном 16](#_Toc167474368)

[3.1.4 Полином 18](#_Toc167474369)

[3.2 Описание программной реализации 22](#_Toc167474370)

[3.2.1 Описание структуры TNode 23](#_Toc167474371)

[3.2.2 Описание класса TList 25](#_Toc167474372)

[3.2.3 Описание класса TRingList 28](#_Toc167474373)

[3.2.4 Описание класса TMonom 29](#_Toc167474374)

[3.2.5 Описание класса TPolinom 35](#_Toc167474375)

[Заключение 40](#_Toc167474376)

[Литература 41](#_Toc167474377)

[Приложения 42](#_Toc167474378)

[Приложение А. Реализация класса TList 42](#_Toc167474379)

[Приложение Б. Реализация класса TRinglist 48](#_Toc167474380)

[Приложение В. Реализация класса TMonom 50](#_Toc167474381)

[Приложение Г. Реализация класса TPolinom 55](#_Toc167474382)

# Введение

Лабораторная работа направлена на изучение методов компьютерной обработки полиномов. С этой целью в лабораторной работе изучаются различные варианты структуры хранения и разрабатываются программы для обработки полиномов. Основной учебной целью работы является практическое освоение методов организации структур хранения данных с помощью списков. В ходе выполнения лабораторной работы разрабатывается общая форма представления линейных списков, разрабатываются программы работы со списками, которые могут быть использованы и в других областях приложений.

Актуальность полиномов:

Полиномы являются важным математическим инструментом, который находит применение в различных областях программирования и других наук. Некоторые из основных областей, где полиномы используются, включают:

1. Машинное обучение и искусственный интеллект. Полиномиальные функции часто используются для аппроксимации сложных зависимостей в данных, например, при построении регрессионных моделей.
2. Графический дизайн и компьютерная графика. Полиномы могут быть использованы для создания кривых и поверхностей, которые используются в создании графических изображений.
3. Криптография. Полиномы используются в различных алгоритмах шифрования и подписи сообщений, таких как RSA и Эллиптическая кривая.
4. Физика и инженерия. Полиномы используются для описания физических законов и моделирования различных процессов, например, в механике, электронике и аэродинамике.
5. Финансы. Полиномы могут использоваться для анализа финансовых данных и прогнозирования рыночных трендов.

Таким образом, полиномы играют важную роль во многих отраслях программирования и других наук, помогая решать различные задачи и проблемы.

# Постановка задачи

Цель – создание программных средств, поддерживающих эффективное представление полиномов и выполнение следующих операций над ними:

* ввод полинома
* организация хранения полинома
* удаление введенного ранее полинома;
* копирование полинома;
* сложение двух полиномов;
* вычисление значения полинома при заданных значениях переменных;
* вывод.

В качестве структуры хранения будут использоваться списки. В качестве дополнительной цели в лабораторной работе ставится также задача разработки некоторого общего представления списков и операций по их обработке. В числе операций над списками должны быть реализованы следующие действия:

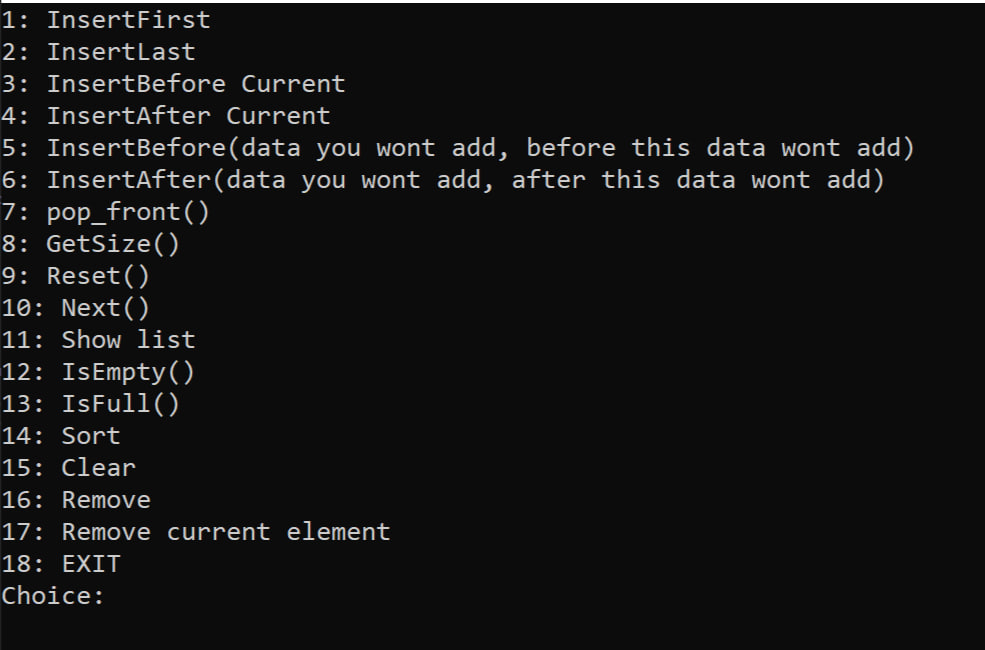
* поддержка понятия текущего звена;
* вставка звеньев в начало, после текущей позиции и в конец списков;
* удаление звеньев в начале и в текущей позиции списков;
* организация последовательного доступа к звеньям списка (итератор).

# Руководство пользователя

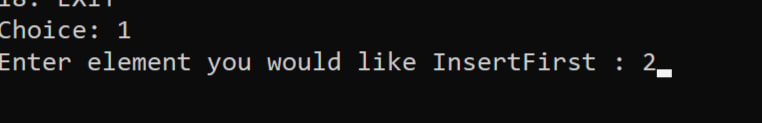
## Приложение для демонстрации работы кольцевого односвязного списка

1. Запустите приложение с названием sample\_tringlist.exe. В результате появится окно, показанное ниже (рис. 1). На выбор даются действия:

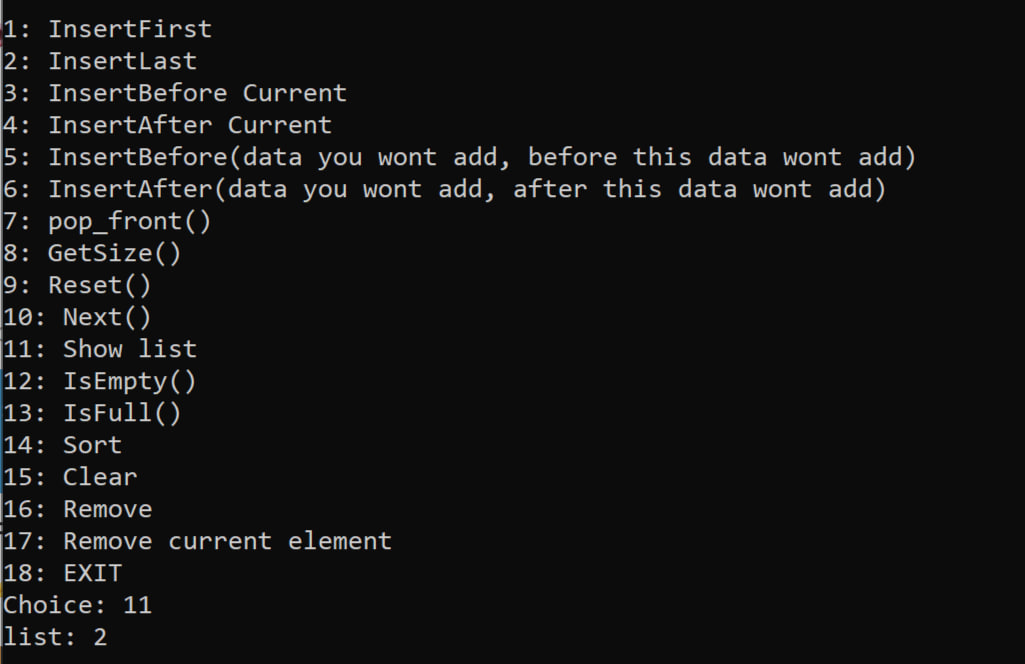
* 1: InsertFirst - вставляет элемент в начало списка.
* 2: InsertLast - вставляет элемент в конец списка.
* 3: InsertBefore Current - вставляет элемент перед текущим элементом.
* 4: InsertAfter Current - вставляет элемент после текущего элемента.
* 5: InsertBefore(data you wont add, before this data wont add) - вставка элемента перед существующим элементом.
* 6: InsertAfter(data you wont add, after this data wont add) - вставка элемента после существующего элемента.
* 7: pop\_front() - удаляет первый элемент списка.
* 8: GetSize() - возвращает количество элементов в списке.
* 9: Reset() - устанавливает указатель на первый элемент списка.
* 10: Next() - перемещает указатель на следующий элемент списка.
* 11: Show list - выводит все элементы списка.
* 12: IsEmpty() - проверяет, пуст ли список.
* 13: IsFull() - проверяет, полон ли список
* 14: Sort - сортирует список в убывающем порядке.
* 15: Clear - удаляет все элементы из списка.
* 16: Remove - удаляет элемент по указанному значению.
* 17: Remove current element - удаляет текущий элемент.
* 18: EXIT - завершает работу программы.



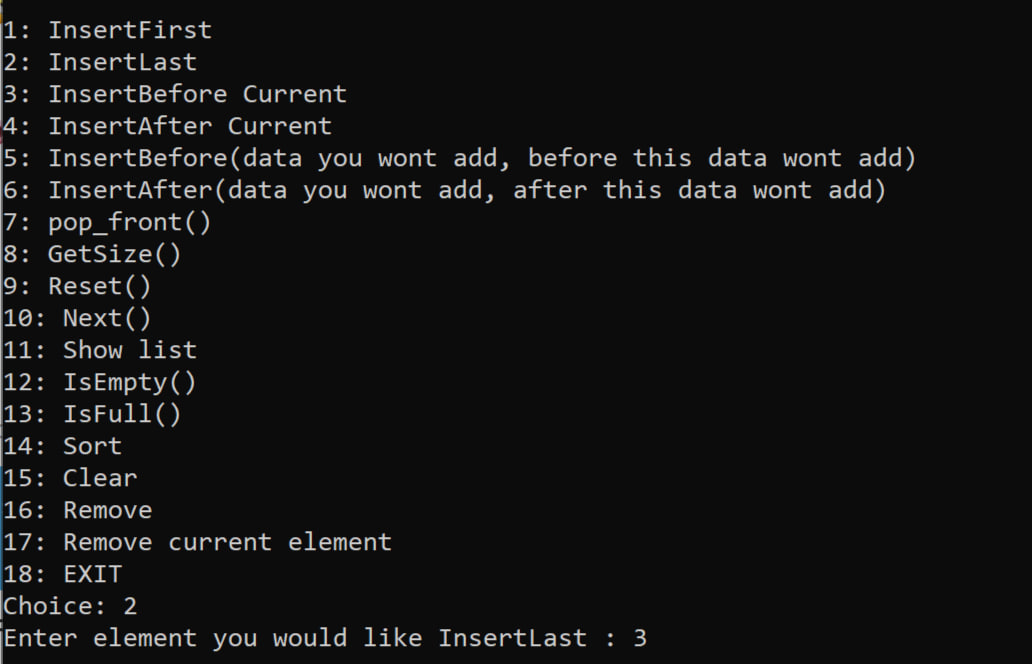
1. Основное окно программы.
2. Вставим элемент в начало. Для этого выбираем первый метод, в который передадим значение которое хотим поместить в список (рис. 2).



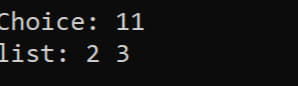
1. Вставка первого элемента.
2. Далее, выбрав метод Show list, можем увидеть элементы, которые хранятся в списке (рис. 3).



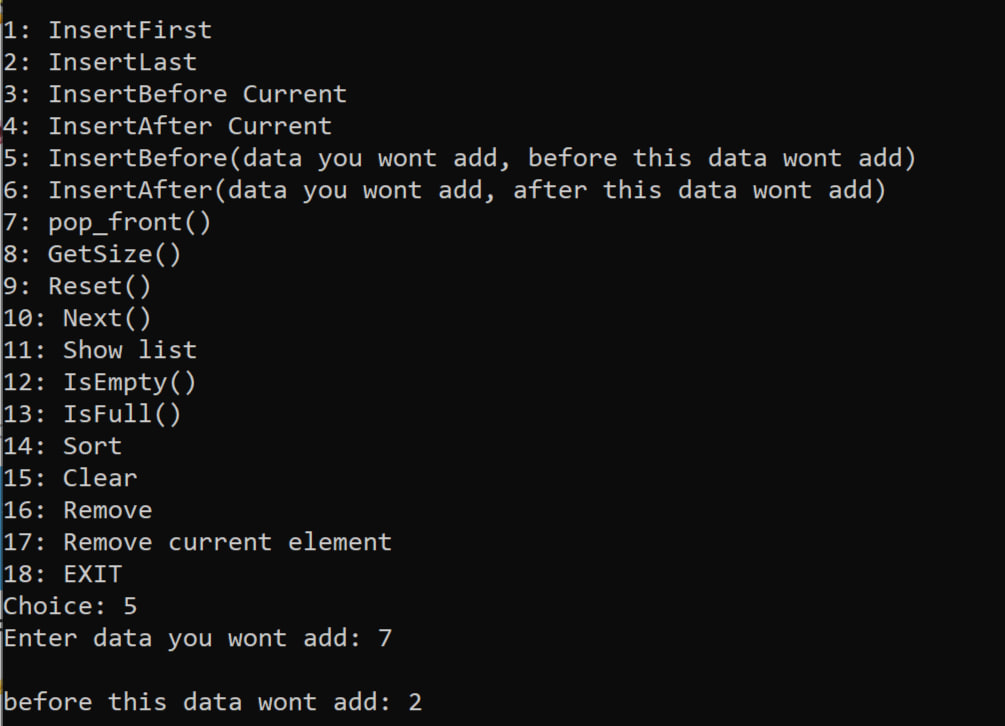
1. Вывод элементов списка.
2. Далее попробуем вставить элемент в конец списка, выбрав второй метод под названием InsertLast (рис. 4).



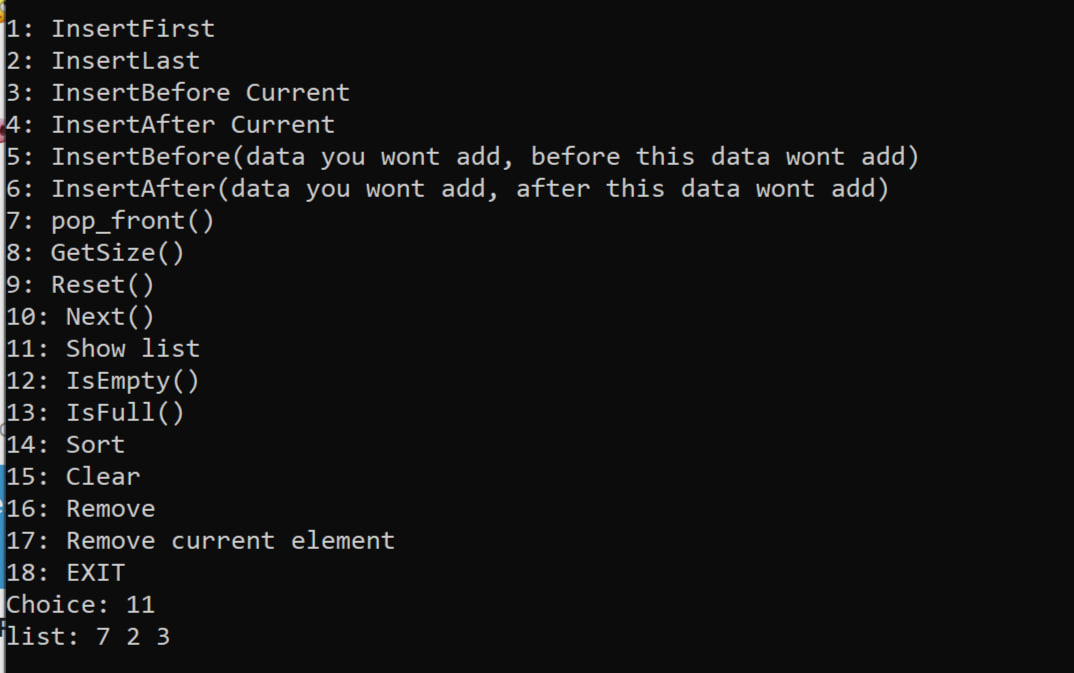
1. Вставка элемента 3 в конец списка.
2. Выведем значения из списка (рис. 5).



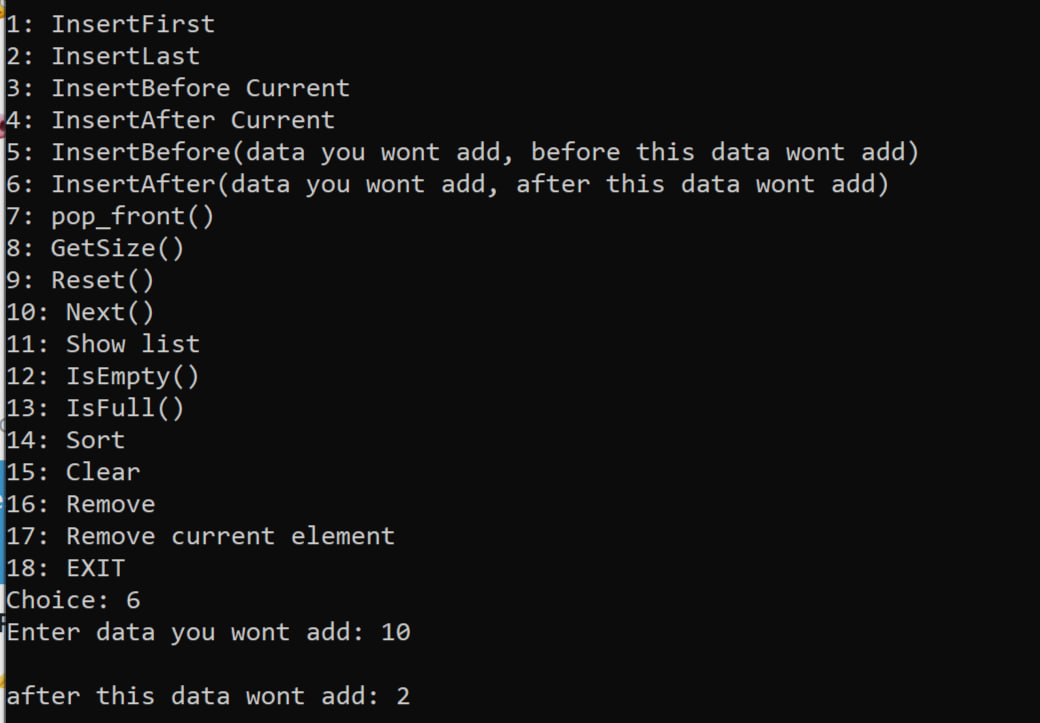
1. Два элемента в списке.
2. Теперь попробуем вставить элемент с помощью метода InsertBefore, в котором нам нужно будет указать новое значение, которое мы хотим добавить, и значение, которое уже имеется в списке, чтобы поставить новое значение перед ним(рис. 6).



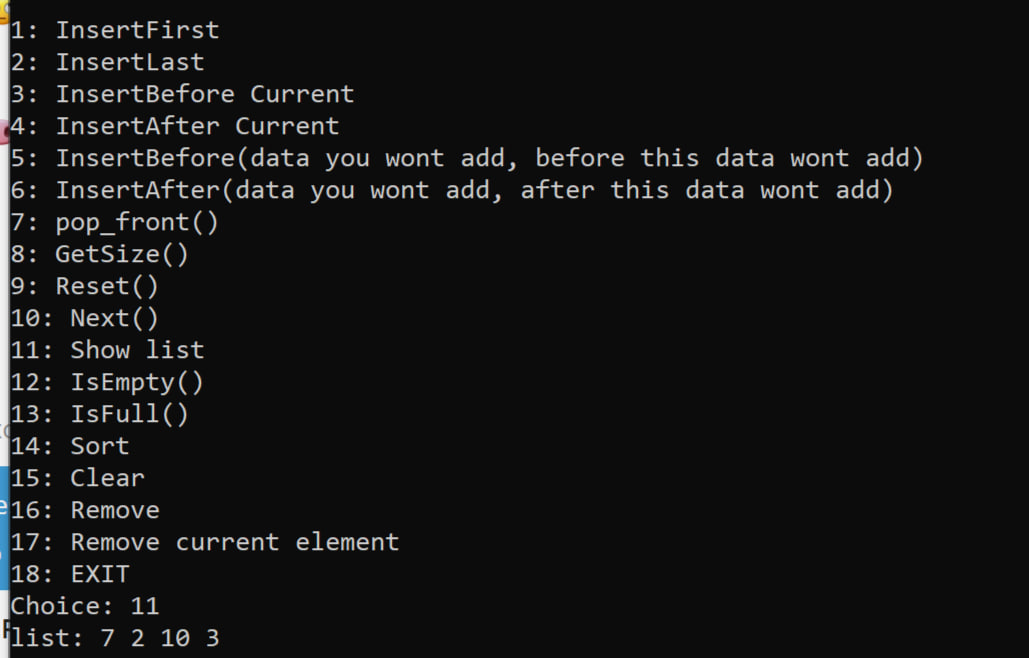
1. Основное окно программы.
2. Выведем значения, которые хранятся в списке (рис. 7).



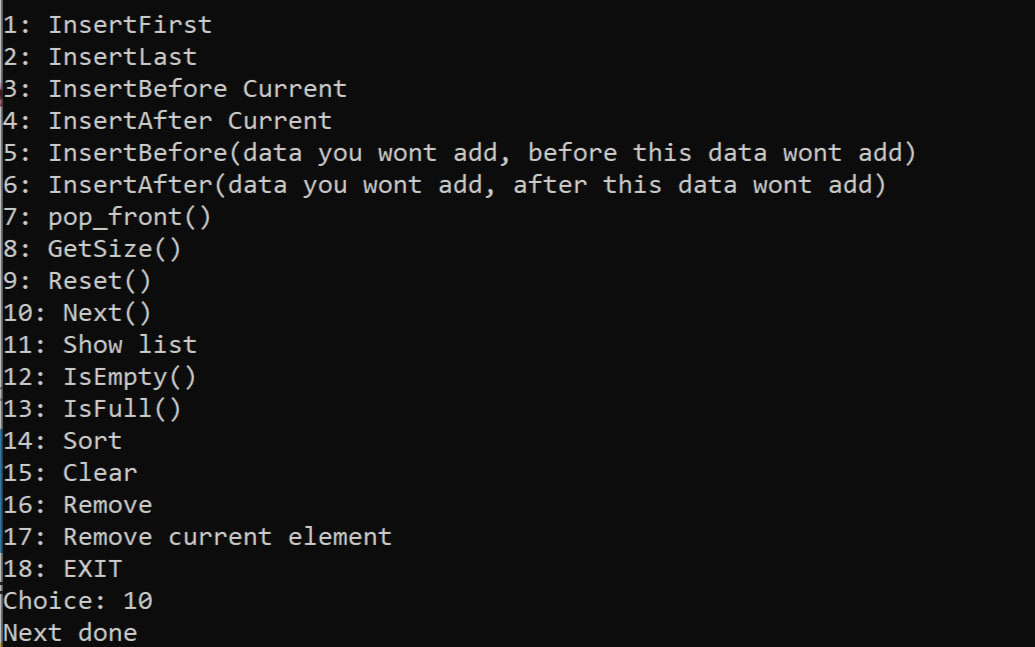
1. Значения списка.
2. Далее выберем метод InsrtAfter в который мы введём новое значение и значение после которого в списке будет храниться новое значение (рис. 8).



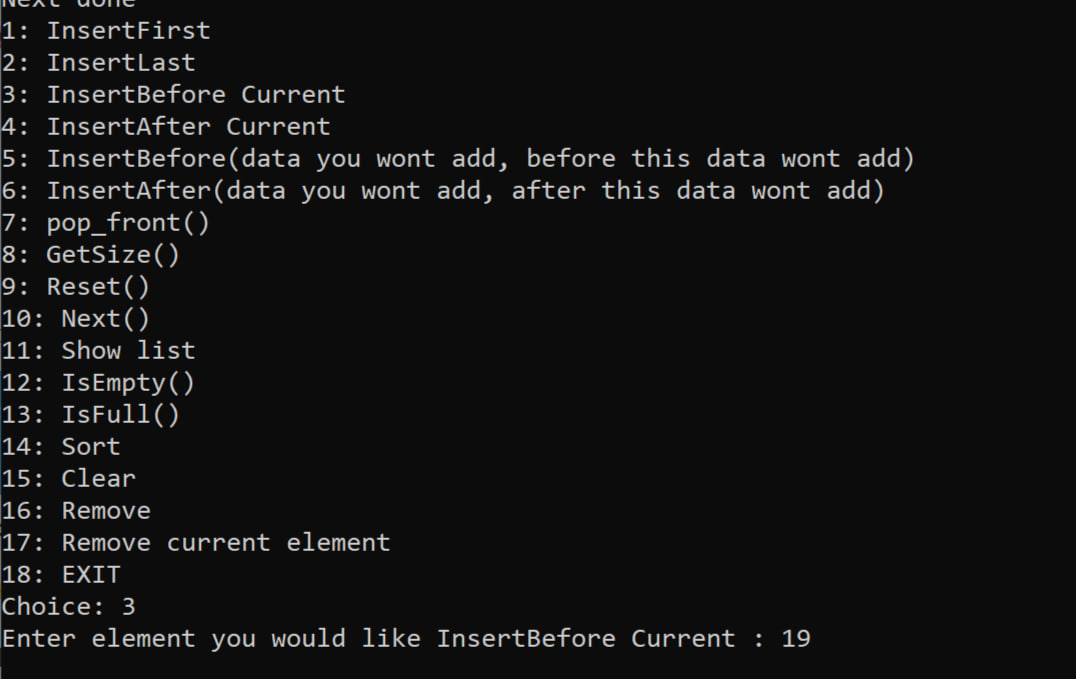
1. INsertAfter.
2. Выведем значения списка (рис. 9).



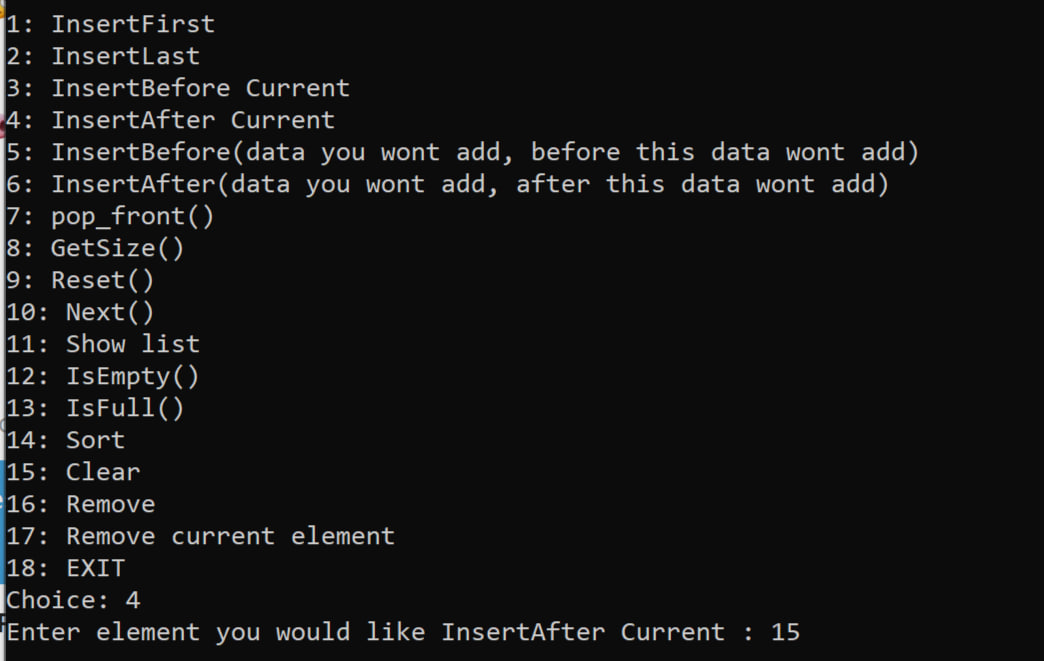
1. Основное окно программы.
2. Теперь сместим указатель текущего элемента (изначально он указывает на первый то есть это был 7. Теперь 2) (рис. 10).



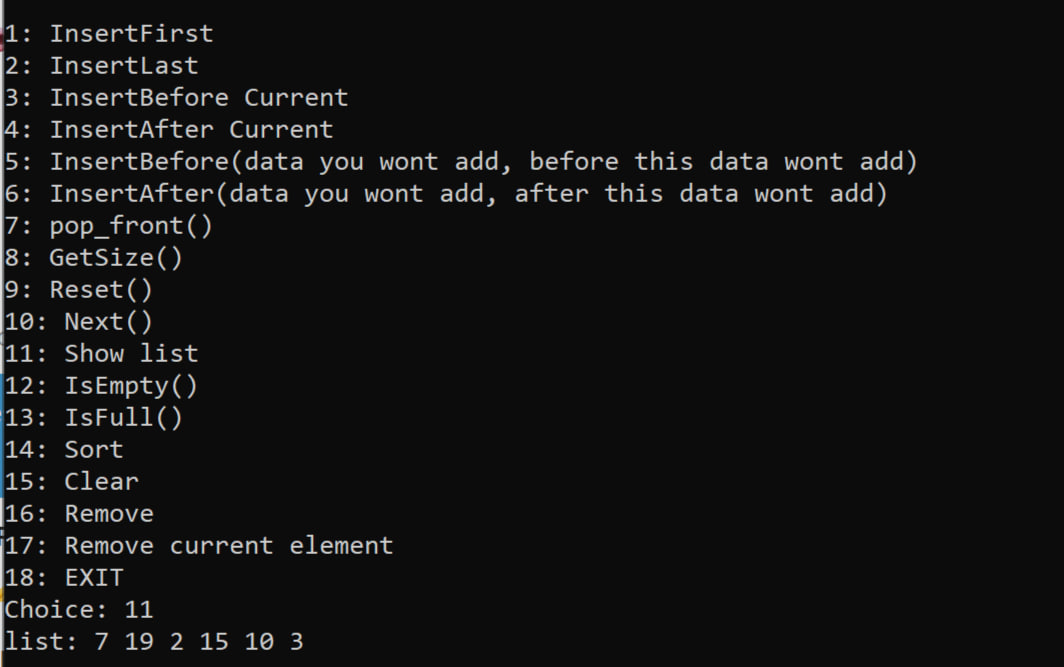
1. Next.
2. Теперь воспользуемся методами InsertBefore (рис. 11) current и InsertAfter (рис. 12) current таким образом рядом с нашим текущим элементом будут находиться новые данные (рис. 13).



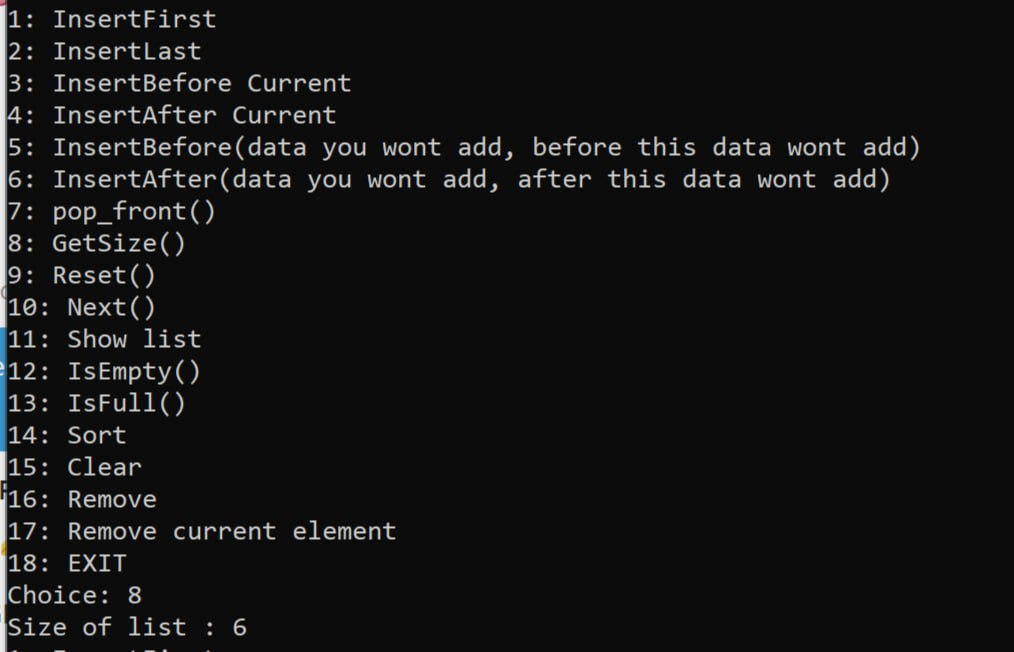
1. InsertBefore.



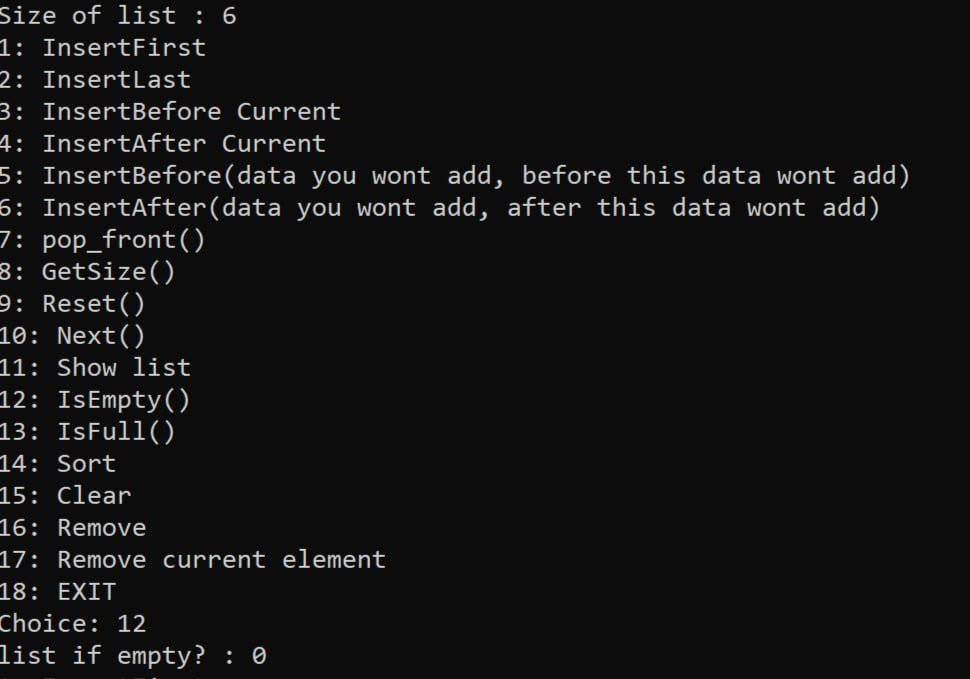
1. InsertAfter.



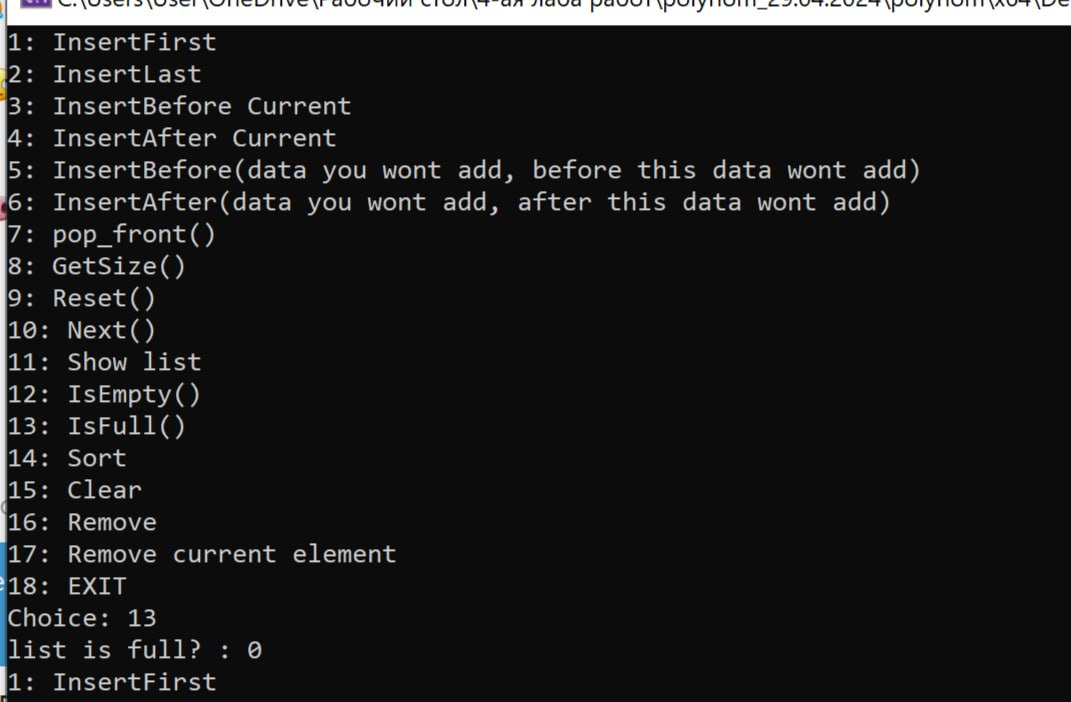
1. Основное окно программы.
2. Выведем размер списка (рис. 14).



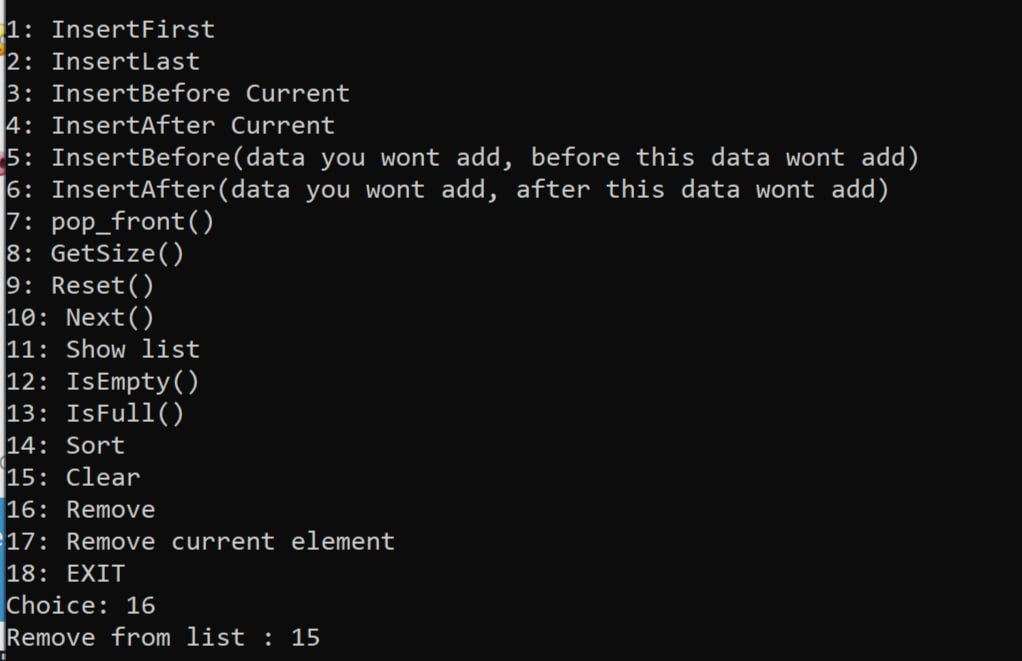
1. Размер списка.
2. Проверим список на пустоту (рис. 15) и полон ли он (рис. 16).



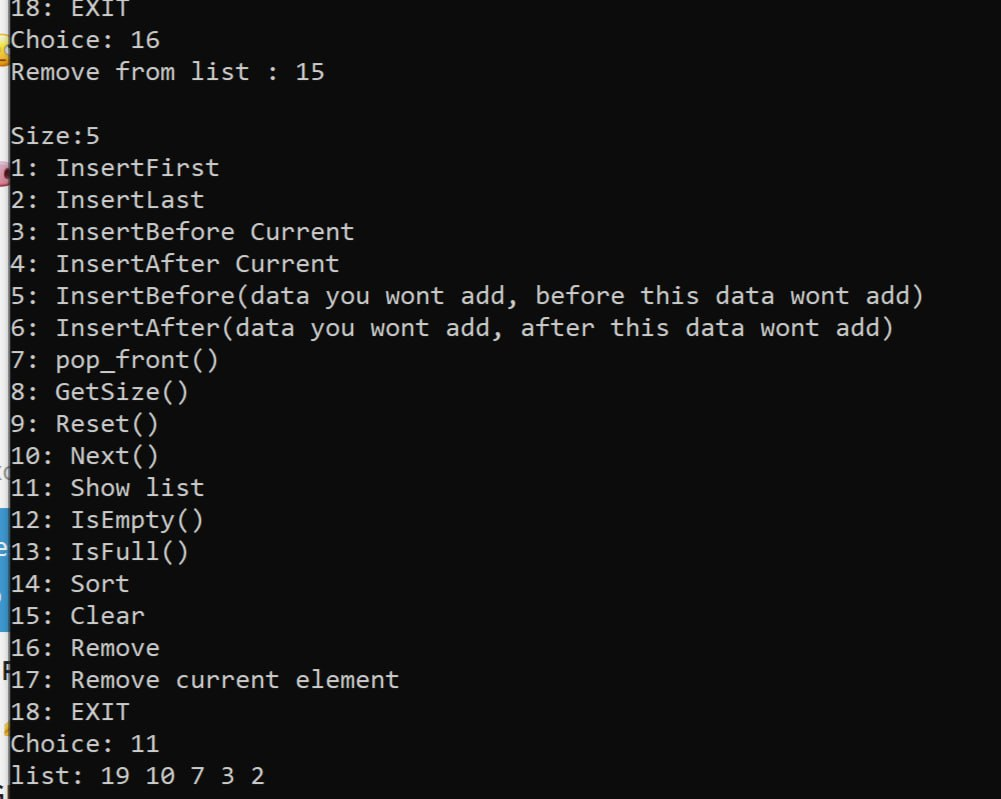
1. Список не пустой.



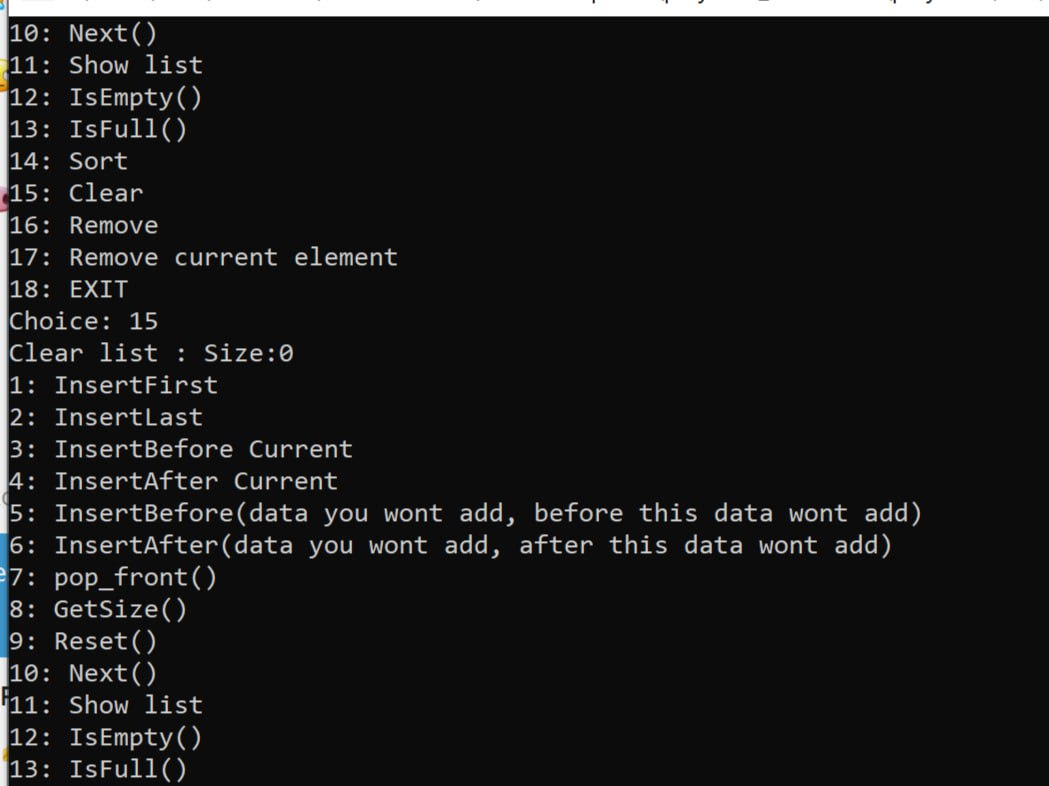
1. Список не полон.
2. Удалим из списка 15 (рис. 17). И выведем результат (Рис. 18).



1. Удаление элемента из списка.



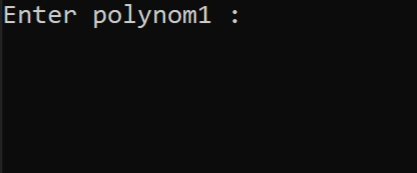
1. Результат.
2. Удалим все элементы из списка (рис. 19).



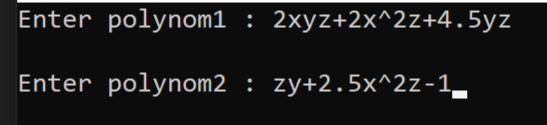
1. Удаление всех элементов.

## Приложение для демонстрации работы полиномов

1. Запустите приложение с названием sample\_tpolynom.exe. В результате появится окно, показанное ниже (рис. 20).



1. Требуется ввести полином.
2. Введём два полинома (рис. 21) и получим результат всех операций с полиномами (рис. 22) .



1. Вводим полиномы.



1. Результат.
2. В первых двух строках выводятся отсортированные полученные полиномы, далее дифееренциалы и результаты операций для двух полиномов: сумма, разность, произведение (рис. 22).
3. Потом программа запрашивает значения трех переменных x,y,z чтобы вывести значение полинома (рис. 23).



1. Значение полинома.

# Руководство программиста

## Описание алгоритмов

### Линейный односвязный список

Линейный односвязный список – это структура данных, которая представляет собой последовательность элементов, называемых узлами, где каждый узел содержит данные и указатель на следующий узел в списке. Односвязный список имеет три основных указателя:

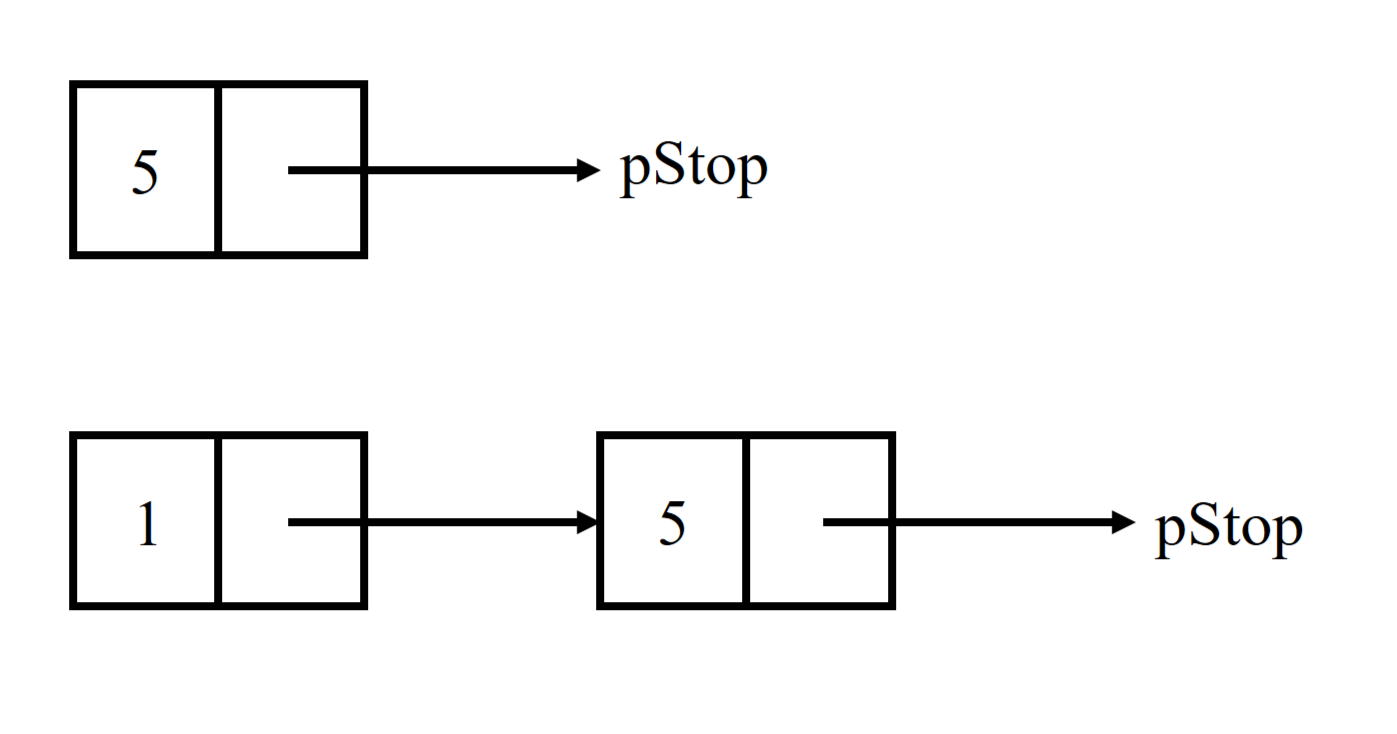
1. Указатель на первый узел (pFirst) – это указатель на начало списка, то есть на первый элемент. Он используется для доступа к первому элементу списка.
2. Указатель на текущий узел (pCurr)- это указатель, который используется для перемещения по списку. Он указывает на текущий элемент, с которым в данный момент происходит работа.
3. Указатель на последний узел (pLast)- это указатель на последний элемент списка. Он используется для быстрого доступа к последнему элементу без необходимости перебирать весь список.
4. Если узел является последним, то указатель на следующий элемент равен pStop (nullptr).

Структура узла в односвязном списке обычно содержит два поля: данные и указатель на следующий узел в списке.

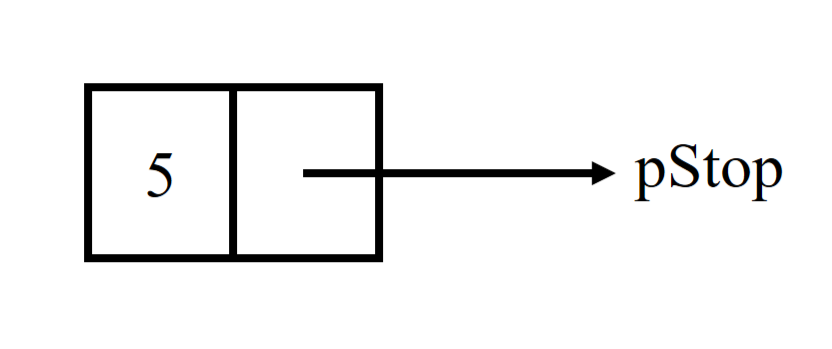
**Операция добавления в начало**

1. Когда вы добавляете элемент в начало кольцевого списка, создается новый узел. Этот новый узел должен знать, на какой узел он должен указывать далее.
2. Поэтому указатель на следующий элемент нового узла устанавливается на текущий первый узел списка.

Пример:



1. Добавление в начало список не пустой (добавили 1).



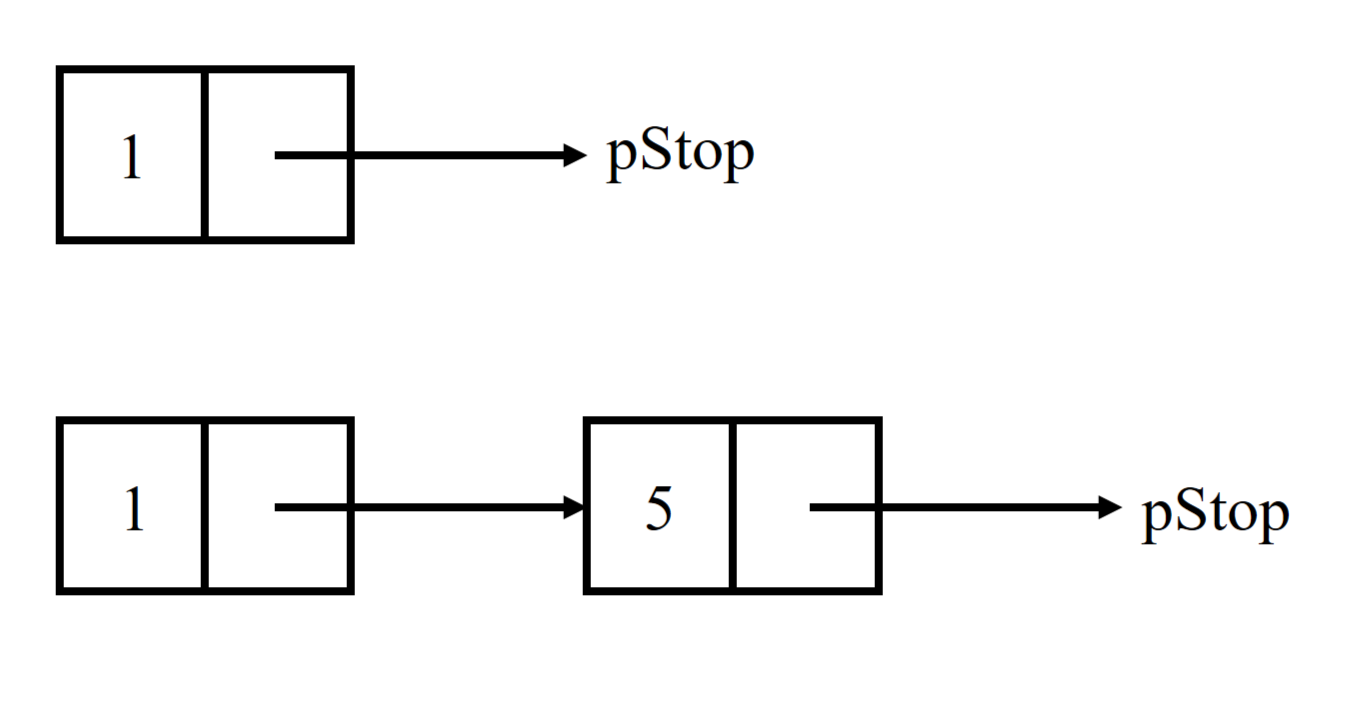
1. Добавление в начало список пустой (добавили 5).

**Операция добавления в конец**

Метод добавляет новый элемент в конец списка.

Если список пуст, новый элемент становится первым и последним элементом списка. Если список не пуст, новый элемент добавляется в конец списка, а указатели обновляются таким образом, чтобы список оставался связанным.

Пример:



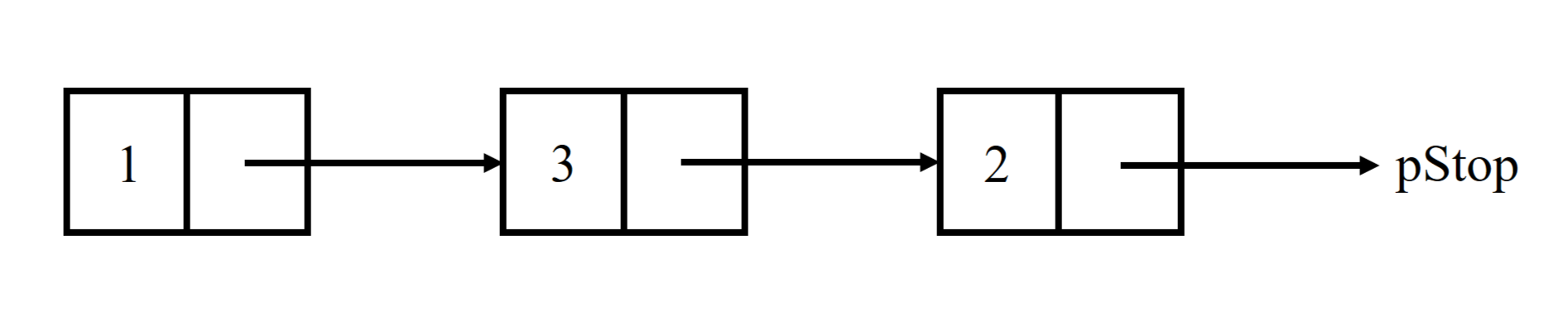
1. Добавление в конец список не пуст (добавили 5).

**Операция поиска**

Метод поиска ищет узел в списке, который содержит указанные данные. Он начинает с первого узла списка и последовательно переходит к следующему узлу, пока не найдет узел с совпадающими данными или не достигнет конца списка.

Если узел с совпадающими данными найден, метод возвращает указатель на этот узел. Если узел не найден, метод возвращает nullptr.

Пример:



1. Пример списка.

Найдём 5.

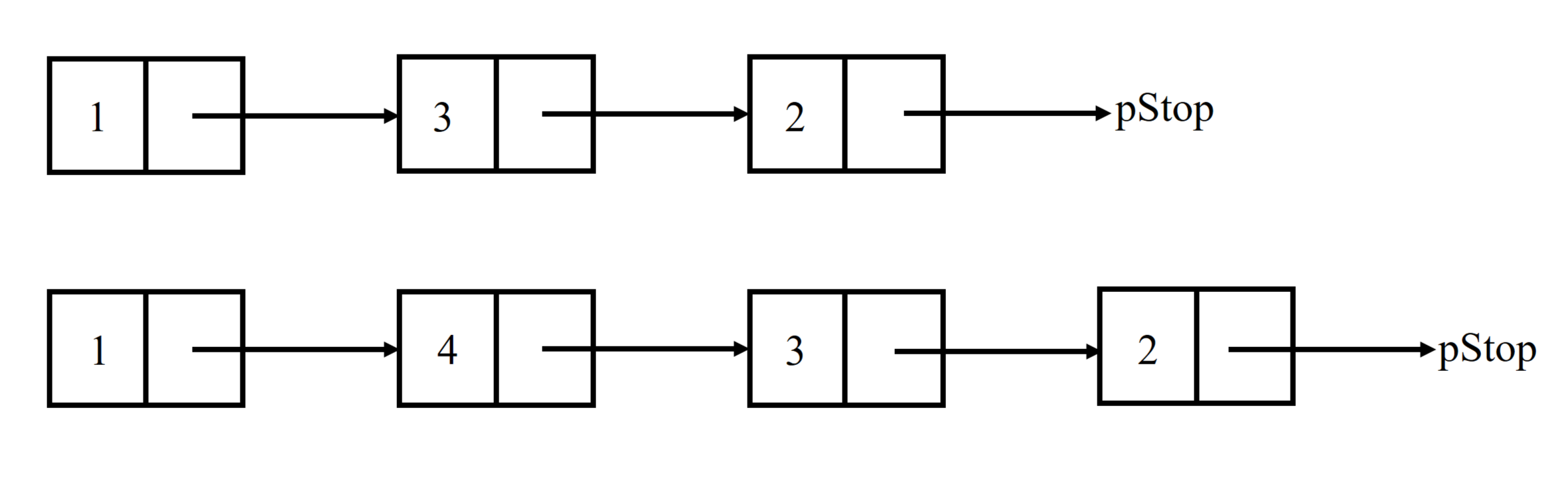
Вернется nullptr.

**Операция вставки перед заданным элементом**

Метод InsertBefore добавляет новый элемент в список перед текущим узлом. Если текущий узел является первым выполняется вставка в начало.

В противном случае новый элемент вставляется перед текущим узлом, и указатели обновляются таким образом, чтобы список оставался связанным.

Пример:



1. Вставка перед заданным элементом (вставим перед 3 4).

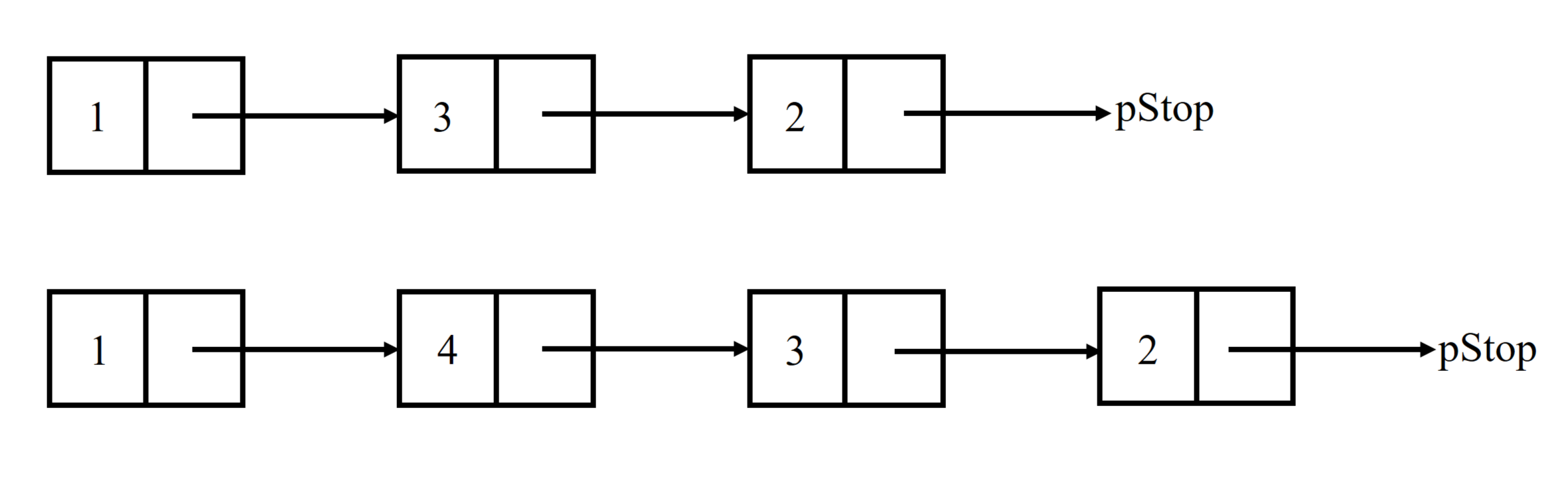
**Операция вставки после заданного элемента**

Метод InsertAfter добавляет новый элемент в список после текущего узла.

Если текущий узел является последним узлом списка, выполняется вставка в конец.

В противном случае новый элемент вставляется после текущего узла, и указатели обновляются таким образом, чтобы список оставался связанным.

Пример:



1. Вставка после заданного элемента (вставим 4 после 1).

**Операция удаления**

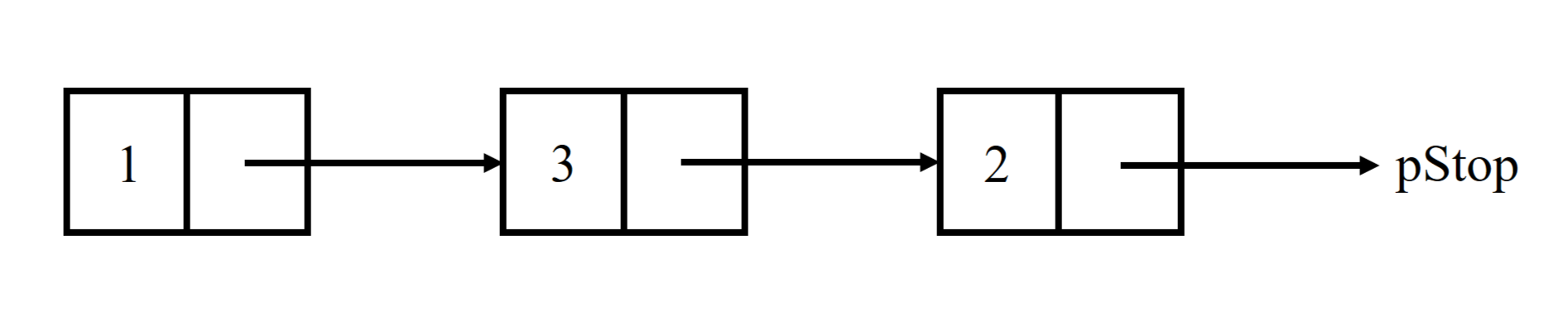
Метод удаляет из списка узел, содержащий указанные данные.

Он начинает с первого узла списка и последовательно переходит к следующему узлу, пока не найдет узел с совпадающими данными или не достигнет конца списка.

Если узел с совпадающими данными найден, он удаляется, а указатели обновляются таким образом, чтобы список оставался связанным.

Если узел с совпадающими данными не найден, вызывается исключение.

Пример:



1. Исходный список

удалим 2



1. Список после удаления

**Операция проверки на пустоту**

Метод проверяет, содержит ли список какие-либо элементы. Он делает это, сравнивая указатель на первый узел списка с указателем на последний узел списка. Если указатели равны, значит список пуст, и метод возвращает true. В противном случае метод возвращает false.

**Операция проверки на полноту**

Операция проверяет, может ли список вместить больше элементов. Он пытается создать новый узел и проверить, удалось ли это сделать. Если новый узел создан успешно, значит список не полон, и метод возвращает false. Если новый узел не удалось создать, значит список полон, и метод возвращает true.

### Кольцевой список

Структура данных TRinglist представляет собой кольцевой линейный односвязный список в виде шаблонного класса. При создании списка, создается фиктивный элемент pHead, указывающий на pFirst. Указатель на начало спиcка pFirst, на конец pStop, который равен pHead. Поддерживаются операции такие как: вставка в начало, вставка в конец, вставка до и после элемента, операция поиска, операция удаления из списка, проверка на пустоту и полноту.

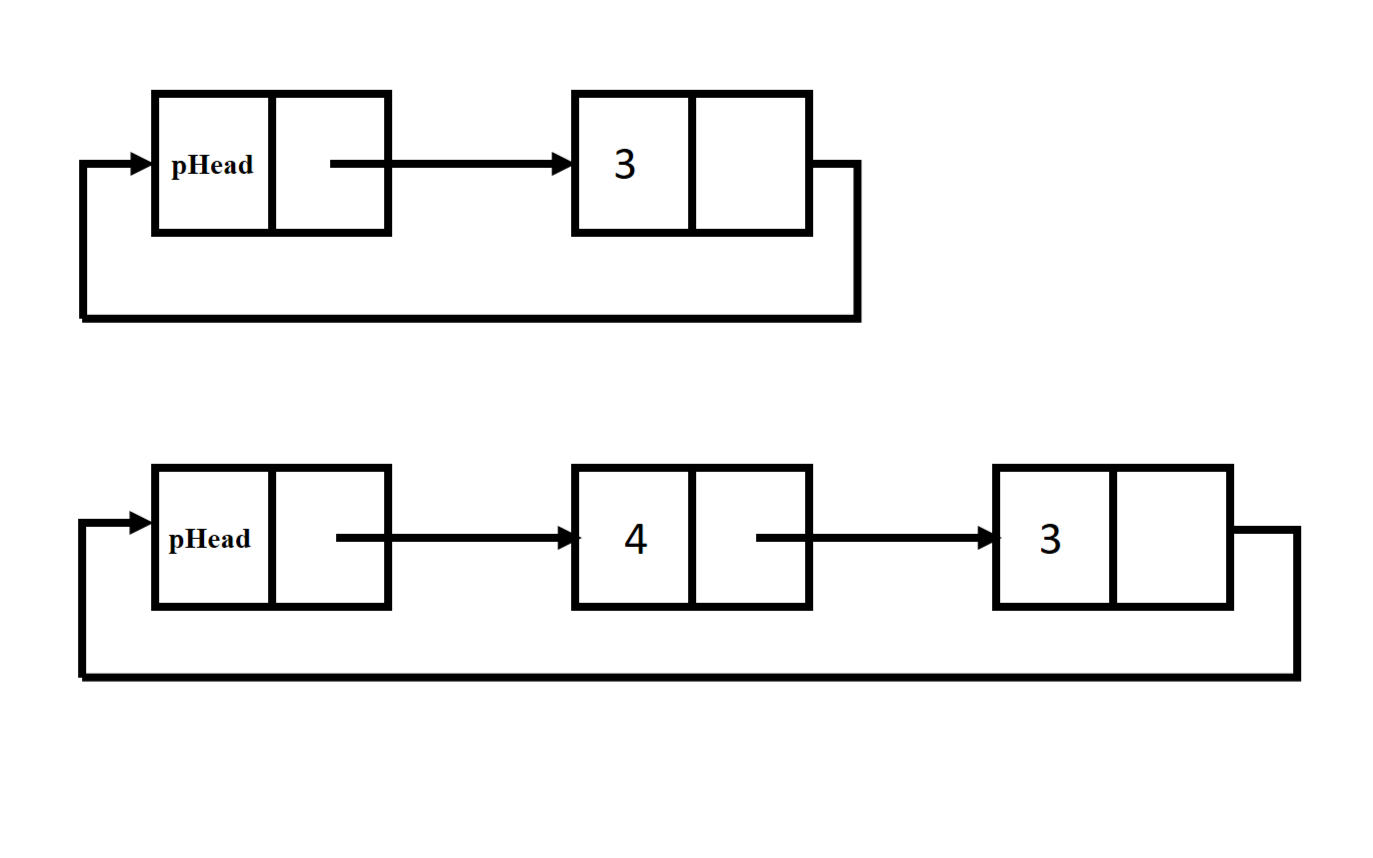
**Операция вставки в начало**

1. Создается новый узел и устанавливается в качестве первого узла в списке.
2. Если список был пуст (то есть не имел узлов), то новый узел становится и первым, и последним узлом.
3. Фиктивная голова списка (фиктивный узел, который всегда указывает на первый узел) теперь указывает на новый первый узел.
4. Последний узел списка снова указывает на фиктивную голову списка.

В результате:

* Новый узел становится первым узлом в списке.
* Все существующие узлы остаются на своих местах и связаны друг с другом.
* Список остается кольцевым, поскольку последний узел указывает на фиктивную голову списка.

Пример: добавим 4



1. Вставка в начало

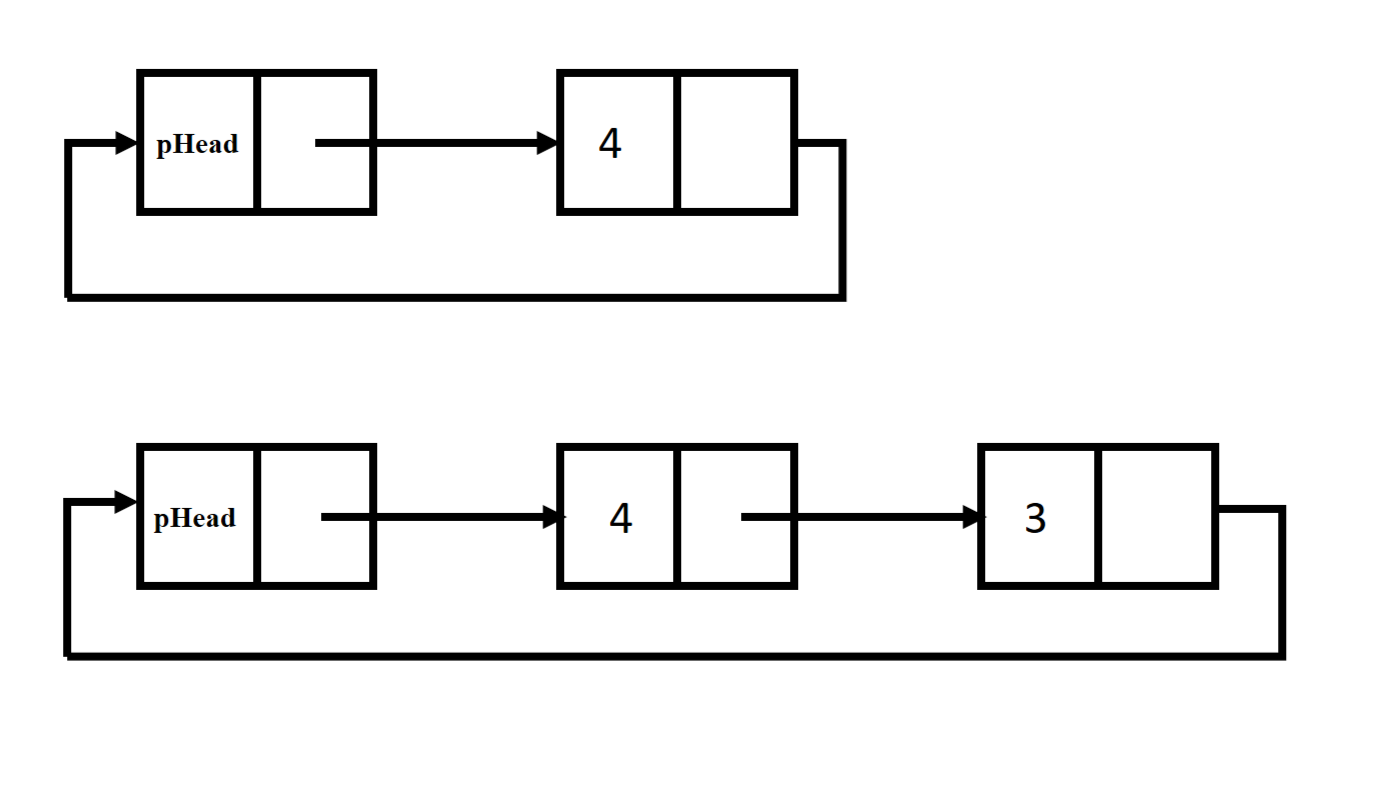
**Операция вставки в конец**

1. Сначала проверяется, пуст ли список. Если список пуст, то вызывает метод вставки в начало списка, который создает новый узел и устанавливает его в качестве первого и последнего узла в списке.
2. Если список не пуст, то создает новый узел и устанавливает его в качестве следующего узла после существующего последнего узла.
3. Затем обновляется указатель на последний узел, чтобы он указывал на новый узел.
4. Устанавливается указатель фиктивной головы списка на первый узел.
5. Устанавливается указатель нового последнего узла на фиктивную голову списка.

В результате:

* Новый узел добавляется в конец списка.
* Все существующие узлы остаются на своих местах и связаны друг с другом.
* Список становится кольцевым, поскольку последний узел теперь указывает на фиктивную голову списка.

Пример: добавим 3



1. Вставка в конец

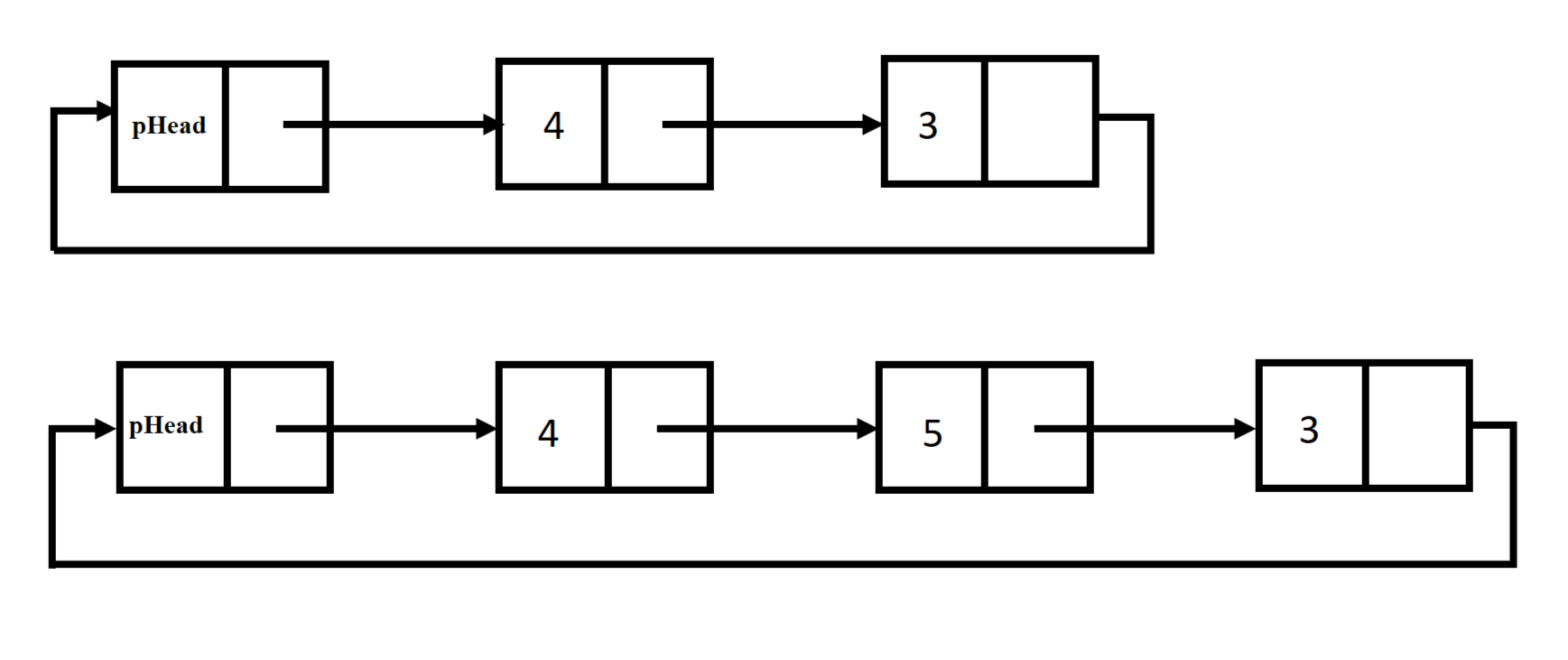
**Операция вставки до элемента**

1. Проверяет, не является ли указанный узел списка первым узлом. Если это так, то вызывается метод вставки в начало списка, который создает новый узел и устанавливает его в качестве первого узла в списке.
2. Если узел не указывает ни на первый, ни на последний узел, то создается новый узел и устанавливается перед заданным узлом.
3. Устанавливается указатель фиктивной головы списка на первый узел.
4. Устанавливает указатель последнего узла на фиктивную голову.

В результате:

* Новый узел вставляется перед узлом, на который указывает текущий узел.
* Все существующие узлы остаются на своих местах и связаны друг с другом.
* Список остается кольцевым, поскольку последний узел указывает на фиктивную голову списка.

Пример: вставим 5 до 3



1. Операция вставки до элемента

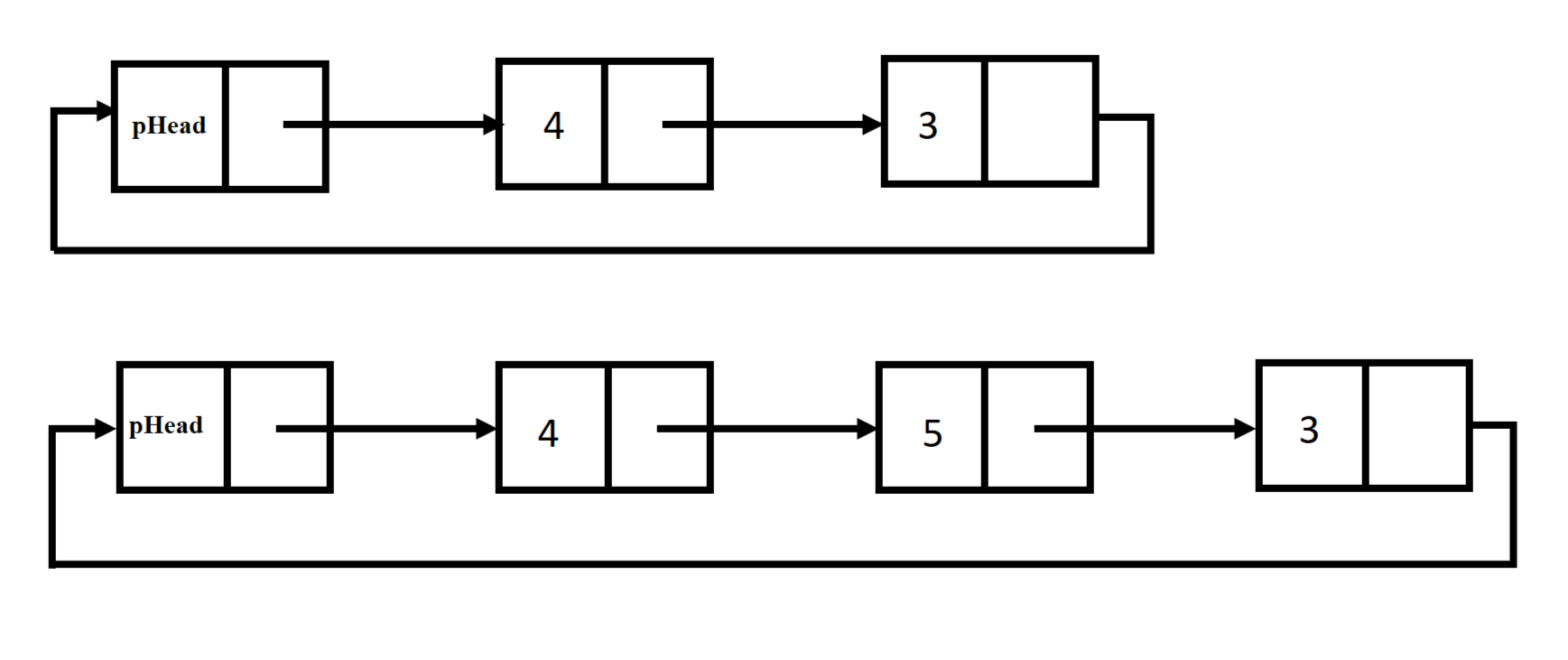
**Операция вставки после элемента**

1. Проверяется, не указывает ли заданный узел на последний узел в списке. Если это так, то вызывает метод вставки в конец списка, который создает новый узел и добавляет его в конец списка.
2. Если заданный узел не указывает на последний узел, то создается новый узел и устанавливается в качестве следующего узла после заданного узла.
3. Устанавливается указатель последнего узла на фиктивную голову списка.

В результате:

* Новый узел вставляется после узла, на который указывает текущий узел.
* Все существующие узлы остаются на своих местах и связаны друг с другом.
* Список остается кольцевым, поскольку последний узел указывает на фиктивную голову списка.

Пример: Вставим 5 после 4



1. Вставка после элемента

### Моном

Тип данных представляет собой структуру данных, которая используется для представления мономов. Моном имеет такие поля: коэффициент и степень монома (свёртка степеней x, y, z).

**Операция сравнения больше**

Оператор сравнения> сравнивает степени двух мономов. Сравниваются свертки степеней. Если свертка степеней первого монома больше чем второго, метод возвращает true, иначе возвращает false.

Пример:

**2xy^2z>3xy^2z** вернет false, так как степень первого монома не больше степени второго.

**2x^3y^2z>3xy^2z** вернет true так как степень первого монома больше степени второго.

**Операция сравнения меньше**

Оператор сравнения< сравнивает степени двух мономов. Сравниваются свертки степеней. Если свертка степеней первого монома меньше чем второго, метод возвращает true, иначе возвращает false.

Пример:

**4xy^2z<3xy^2z** вернет false, так как степень первого монома не меньше степени второго.

**4xyz <3xy^2z** вернет true, так как степень первого монома меньше степени второго.

**Операция проверки на равенство**

Оператор сравнения== сравнивает степени двух мономов. Сравниваются свертки степеней. Если свертка степеней первого монома равна второй, метод возвращает true, иначе возвращает false.

Пример:

**2xy^2z==2xy^2z** вернет true, так как степени обоих мономов одинаковы.

**Операция проверки на неравенство**

Оператор проверки на неравенство сравнивает степени двух мономов. Сравниваются свертки степеней. Если свертка степеней первого монома не равна второй, метод возвращает true, иначе возвращает false.

Пример:

**3xy^2z!=3xy^3z** вернет true, так как степени мономов различны.

**Операция сложения**

Складываются коэффициенты мономов, у которых равны свертки степеней.

Пример:

При попытке сложить **2x^5y^2z** и **3xy^2z** будет выброшено исключение, так как степени различны.

При попытке сложить **2xy^2z** и **3xy^2z** мы получим новый моном сложив коэффициенты **5xy^2z**.

**Операция вычитания**

Этот оператор вычитает коэффициенты мономов, если их свертки степеней равны.

Если свертки степеней не равны, выбрасывается исключение.

Предположим, у нас есть два монома: **3x^2y** и **2x^2y**. При вычитании их друг из друга получим:

**3x^2y - 2x^2y = x^2y**

**Операция перемножения мономов**

Этот оператор умножает два монома, коэффициенты мономов перемножаются, а свертки степеней складываются.

Если степень результата меньше 0 или больше 999, выбрасывается исключение.

Рассмотрим умножение двух мономов: **4xy^2 и 3x^2**. Результат умножения будет:

**4xy^2 \* 3x^2 = 12x^3y^2**

**Вычисление значения монома**

Этот метод вычисляет значение монома для заданных значений переменных x, y, z.

Сначала вычисляются значения переменных в степени, затем происходит умножение на коэффициент.

Пусть у нас есть моном **2x^3y** с переменными x=5, y=2. Значение этого монома будет:

2 \* 5^3 \* 2 = 500

**Производная по переменной**

Эти методы вычисляют частную производную монома по переменным x и y z соответственно.

Если степень по соответствующей переменной равна 0 (в свёртке степеней соответствующий разряд нулевой), возвращается нулевой моном (моном с коефициентом 0 и свёрткой степеней 0).

В противном случае производная вычисляется и возвращается как новый моном(Если x делим на 100 ,если y вычитаем 100 в степени x и делим на 10, если z вычитаем сумму степени икса умноженную на 100 и степень игрика умноженную на 10, таким образом получим степень переменной по которой берём производную домножаем коэффициент на это число и уменьшаем значени свертки ).

Рассмотрим моном **5x^3y^2**. Если мы возьмем частную производную по x, получим:

**dif\_x(5x^3y^2) = 15x^2y^2**

А если возьмем частную производную по y, получим:

**dif\_y(5x^3y^2) = 10x^3y**

### Полином

Структура данных TPolynom представляет собой класс, который содержит строку с полиномом и кольцевой линейный односвязный список с мономами. Структура данных поддерживает следующие операции: сложение, вычитание, умножение полиномов, взятие производных, вычисление значения полинома.

**Сложение полиномов**

Метод operator+ складывает два полинома, представленных списками мономов. Он создает новый список мономов для хранения результата сложения.

Алгоритм метода:

1. Итераторы двух списков мономов устанавливаются на начало соответствующих списков.
2. Цикл повторяется, пока не достигнут конец любого из списков мономов.
3. Если степени текущих мономов в списках совпадают, то их коэффициенты складываются и результат записывается в текущий моном в результирующем списке. Если итоговый коэффициент равен 0, то текущий моном удаляется из результирующего списка.
4. Если степени текущих мономов не совпадают, то итератор того списка, в котором степень меньше, переходит к следующему моному.
5. После обработки всех мономов из обоих списков в результирующем списке могут остаться необработанные моном из одного из списков. Они добавляются в результирующий список.
6. Если результирующий список мономов пуст, то создается моном с коэффициентом и степенью 0 и добавляется в список.
7. Результирующий список мономов сортируется по степени.
8. Новый список мономов, содержащий результат сложения, возвращается.

Пример:

Давайте разберем шаги сложения полиномов итеративно:

1. Исходные полиномы:

Полином A: 3x^2y + 2xy^2 + 5

Полином B: 4x^2y - 2xy^2 + 7

1. Создаем пустой результирующий полином res.
2. Сравниваем мономы:

Сначала сравниваем моном 3x^2y из полинома A с мономами в полиноме B.

Нет совпадений по степени, добавляем 3x^2y в res.

Теперь сравниваем моном 2xy^2 из полинома A с мономами в полиноме B.

Нет совпадений по степени, добавляем 2xy^2 в res.

Далее сравниваем константу 5 из полинома A с мономами в полиноме B.

Нет совпадений по степени, добавляем константу 5 в res.

1. Сравниваем мономы из полинома B с результирующим полиномом res:

Сначала сравниваем моном 4x^2y из полинома B.

Найдено совпадение по степени x^2y. Складываем коэффициенты (коэффициенты 3 и 4) и обновляем коэффициент в res: 3 + 4 = 7.

Затем сравниваем моном -2xy^2 из полинома B.

Найдено совпадение по степени xy^2. Складываем коэффициенты (коэффициенты 2 и -2) и обновляем коэффициент в res: 2 + (-2) = 0. Так как результат равен 0, этот моном удаляется из результата.

Наконец, рассматриваем константу 7 из полинома B.

Нет совпадений по степени, добавляем константу 7 в результирующий полином.

1. Результирующий полином:

Результат сложения: 7x^2y + 5

Таким образом, после итеративного процесса сложения мономов из полиномов A и B мы получаем результирующий полином 7x^2y + 5.

**Вычитание полиномов**

Алгоритм метода:

1. Создается новый список мономов, который будет содержать отрицательный второй список мономов.
2. Вызывается метод сложения полиномов.
3. Результат сложения возвращается как новый список мономов.

Пример:

Полином A: 3x^2y + 2xy^2 + 5

Полином B: 4x^2y - 2xy^2 + 7

1. Создаем новый список мономов, который будет содержать отрицательный второй список мономов:

Для полинома B меняем знак коэффициентов, чтобы получить отрицательные мономы:

Отрицательный полином B: -4x^2y + 2xy^2 - 7

1. Вызываем метод сложения полиномов:

Теперь мы вызываем метод сложения для полинома A и отрицательного полинома B.

1. Результат сложения возвращается как новый список мономов:

Мы применяем метод сложения, как описано в предыдущем ответе, и получаем результат.

Теперь рассмотрим вычитание полиномов итеративно:

* Исходные полиномы:

Полином A: 3x^2y + 2xy^2 + 5

Отрицательный полином B: -4x^2y + 2xy^2 - 7

* Вызываем метод сложения для полинома A и отрицательного полинома B:

Применяем шаги сложения, как описано в предыдущем ответе.

* Результат вычитания:

Результат сложения: 3x^2y + 2xy^2 + 5 - 4x^2y + 2xy^2 - 7

Упрощаем подобные термины: (-1)x^2y + 4xy^2 - 2

Таким образом, после итеративного процесса вычитания мономов из полиномов A и B мы получаем результирующий полином (-1)x^2y + 4xy^2 - 2.

**Умножение полиномов**

**Алгоритм метода:**

1. Итераторы двух списков мономов устанавливаются на начало соответствующих списков.
2. Внешний цикл перебирает моном из первого списка.
3. Внутренний цикл перебирает моном из второго списка и умножает их. Результат умножения добавляется в результирующий список мономов.
4. После обработки всех мономов из второго списка итератор первого списка переходит к следующему моному.
5. После обработки всех мономов из первого списка создается новый полином с результирующим списком мономов.
6. Новый полином возвращается как результат умножения.

Пример:

Теперь давайте выполним шаги умножения для этих двух полиномов:

Полином A: 2x + 3

Полином B: 4x - 1

1. Умножение первого монома из A на все мономы из B:

2x × 4x = 8x^2

2x × (-1) = -2x

1. Умножение второго монома из A на все мономы из B:

3 × 4x = 12x

3 × (-1) = -3

1. Сложение всех полученных мономов:

8x^2 - 2x + 12x - 3 = 8x^2 + 10x - 3

Таким образом, результат умножения полиномов 2x + 3 и 4x - 1 равен 8x^2 + 10x - 3.

**Взятие производных**

**Алгоритм метода:**

1. Создается копия списка мономов текущего полинома.

2. Итератор устанавливается на начало списка мономов.

3. Цикл перебирает моном из списка мономов (copy).

4. Если степень монома по переменной по которй берется производная больше 0, то вычисляется производная монома по этой переменной и добавляется в результирующий список мономов.

5. После обработки всех мономов из списка мономов создается новый полином с результирующим списком мономов.

6. Новый многочлен возвращается как результат вычисления производной по переменной.

Пример:

Давайте рассмотрим пример вычисления производной полинома

4x^3 + 2x^2 - 5x + 7 по переменной x с помощью описанного алгоритма.

Шаги вычисления производной итеративно:

Полином: 4x^3 + 2x^2 - 5x + 7

1. Создаем копию списка мономов текущего полинома.
2. Устанавливаем итератор на начало списка мономов.
3. Перебираем мономы из списка мономов (копия).
4. Вычисляем производную каждого монома по переменной x:

Производная монома 4x^3 по x равна 12x^2.

Производная монома 2x^2 по x равна 4x.

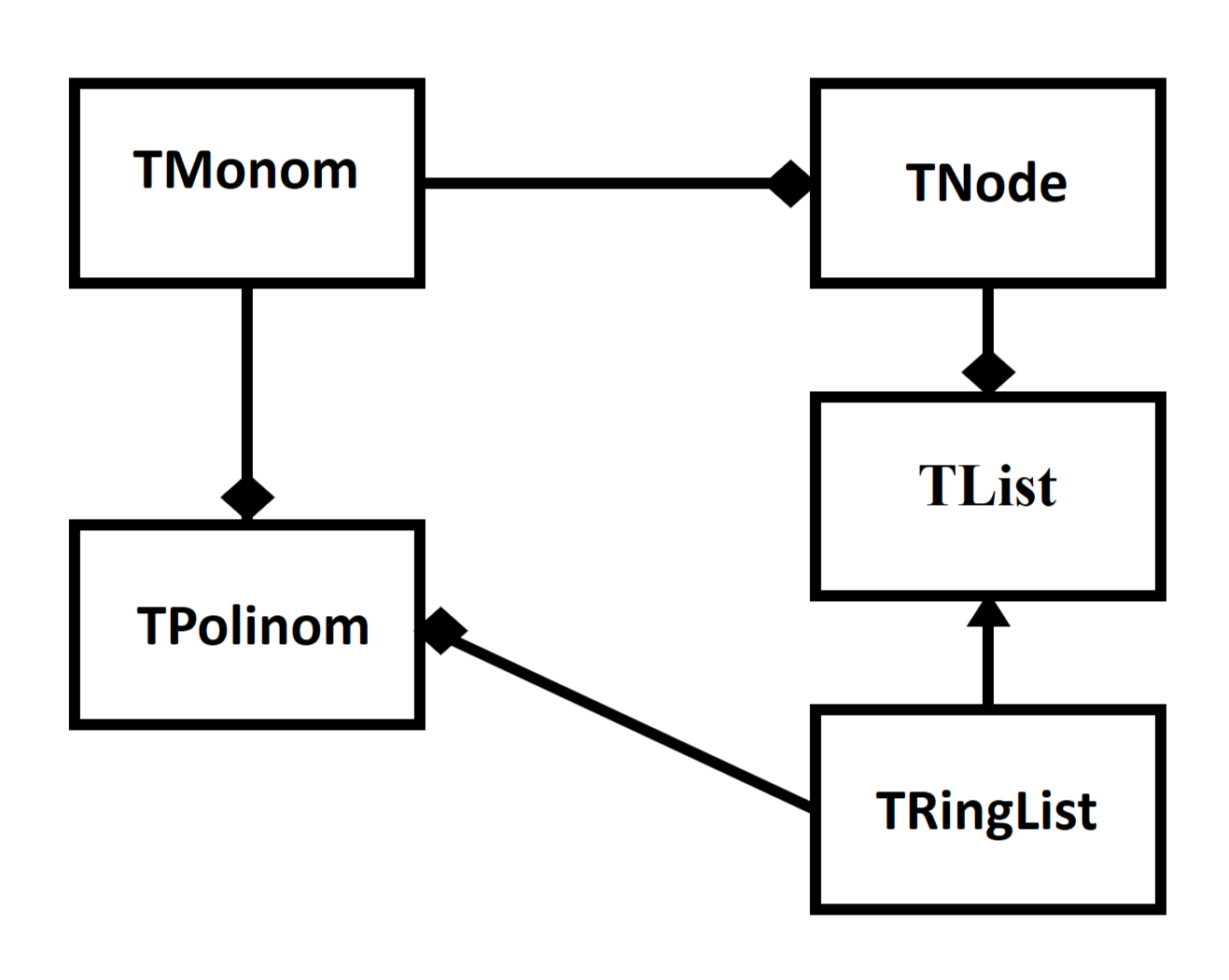
Производная монома -5x по x равна -5.

Производная константного члена не содержит x, поэтому равна 0.

1. Добавляем полученные производные в результирующий список мономов: 12x^2 + 4x - 5.
2. Создаем новый полином с результирующим списком мономов: 12x^2 + 4x - 5.
3. Возвращаем новый полином как результат вычисления производной по переменной x.

Таким образом, производная полинома 4x^3 + 2x^2 - 5x + 7 по переменной x равна 12x^2 + 4x - 5.

## Описание программной реализации



1. Схема наследования классов.

### Описание структуры TNode

struct TNode

{

T key;

TNode\* pNext;

TNode() : key(), pNext(nullptr) {};

TNode(const T& data) : key(data), pNext(nullptr) {};

TNode(TNode<T>\* \_pNext) : key(), pNext(\_pNext) {};

TNode(const T& data, TNode<T>\* \_pNext) : key(data), pNext(\_pNext) {};

friend ostream& operator<<(ostream& out, TNode<T>& list)

};

Назначение: представление узла.

Поля:

**key** – элемент типа **T**;

**pNext**- указатель на следующий элемент.

Методы:

**TNode()**

Назначение: создает пустой узел со значением **key** по умолчанию и ссылкой **pNext** на **nullptr**.

Входные параметры: нет

Выходные параметры: узел со значением **key** по умолчанию и ссылкой **pNext** на **nullptr**.

**TNode(const T& data)**

Назначение: создает узел с заданным значением **key** и ссылкой **pNext** на **nullptr**.

Входные параметры: **data** - значение для поля **key.**

Выходные параметры: узел со значением **key**, равным **data**, и указателем **pNext** на **nullptr.**

**TNode(TNode<T>\* \_pNext)**

Назначение: создает узел с значением **key** по умолчанию и указателем **pNext**, равным заданному указателю.

Входные параметры: **pNext**- ссылка на следующий узел.

Выходные параметры: узел со значением **key** по умолчанию и ссылкой **pNext**, равной **\_pNext**.

**TNode(const T& data, TNode<T>\* \_pNext)**

Назначение: создает узел с заданным значением **key** и указателем **pNext**.

Входные параметры: **data**- значение для поля **key**; **pNext**- указатель на следующий узел.

Выходные параметры: узел со значением **key**, равным **data**, и указателем **pNext**, равным **\_pNext**.

Перегруженный оператор `<<`

Назначение: перегружает оператор `<<` для вывода содержимого узла в поток **ostream**.

Входные параметры:

**Out**- поток **ostream**, в который будет выводиться содержимое узла;

**list**- ссылка на узел, содержимое которого будет выводиться.

Выходные параметры: поток **ostream** с выведенным содержимым узла.

### Описание класса TList

**template <typename T>**

**class TList**

**{**

**protected:**

**TNode<T>\* pFirst;**

**TNode<T>\* pLast;**

**TNode<T>\* pCurr;**

**TNode<T>\* pStop;**

**public:**

**TList();**

**TList(TNode<T>\* \_pFirst);**

**TList(const TList<T>& obj);**

**virtual ~TList();**

**bool IsEmpty() const;**

**bool IsFull() const;**

**bool Is\_End() const;**

**TNode<T>\* Search(const T& data);**

**TNode<T>\* GetCurrent() const;**

**void sort();**

**virtual void InsertFirst(const T& data);**

**virtual void InsertLast(const T& data);**

**const TList<T>& operator=(const TList<T>& p);**

**void InsertBefore(const T& data);**

**void InsertAfter(const T& data);**

**void InsertBefore(const T& data, const T& NextData);**

**void InsertAfter(const T& data, const T& PreviousData);**

**virtual void Remove(const T& data);**

**virtual void removeCurrentNode();**

**void pop\_front();**

**int GetSize() const;**

**void Clear();**

**void Reset();**

**void Next();**

**friend ostream& operator<<(ostream& out, TList<T>& list)**

**};**

Назначение: представление односвязного списка.

Поля:

**pFirst-**указатель на первый элемент односвязного списка.

**pLast-**указатель на последний элемент односвязного списка.

**pCurr-**указатель на текущий элемент односвязного списк.

**pStop-** указатель на **nullptr.**

Методы:

TList()

Назначение: конструктор по умолчанию.

Входные данные: нет.

**TList (TNode<T>\* \_pFirst)**

Назначение: создает новый список на основе существующего узла.

Входные параметры:**\_pFirst** - указатель на первый узел существующего списка.

**TList(const TList<T>& obj);**

Назначение: конструктор копирования

Входные данные: **obj** - экземпляр класса на основе которого создаём новый объект

IsEmpty()

Назначение: проверка, пуст ли односвязный список.

Входные параметры: Нет.

Выходные параметры: возвращает **true**, если список пуст, и **false** в противном случае.

IsFull()

Назначение: проверка, можно ли в список внести звено.

Входные параметры: нет.

Выходные параметры: возвращает **true**, если нельзя, и **false** в противном случае.

Is\_End()

Назначение: проверка, указывает ли текущий указатель на последний элемент списка.

Входные параметры: нет.

Выходные параметры: возвращает **true**, если указатель находится в конце списка, и **false** в противном случае.

Search(const T& data)

Назначение: поиск узла с заданным значением данных.

Входные параметры: значение данных, которое нужно найти.

Выходные параметры: указатель на найденный узел или **nullptr**, если узел не найден.

GetCurrent() const

Назначение: получение узла, на который в данный момент указывает текущий указатель.

Входные параметры: нет.

Выходные параметры: указатель на текущий узел или **nullptr**, если текущий узел не указывает ни на один узел.

sort()

Назначение: сортировка списка по заданному критерию.

Входные параметры: нет.

Выходные параметры: нет.

InsertFirst(const T& data)

Назначение: вставка нового узла со значением данных **data** в начало списка.

Входные параметры: значение данных нового узла.

Выходные параметры: нет (метод изменяет сам список).

InsertLast(const T& data)

Назначение: вставка нового узла со значением данных в конец списка.

Входные параметры: значение данных нового узла.

Выходные параметры: нет (метод изменяет сам список).

InsertBefore(const T& data)

Назначение: вставка нового узла со значением данных **data** перед текущим узлом.

Входные параметры: значение данных нового узла.

Выходные параметры: нет (метод изменяет сам список).

InsertAfter(const T& data)

Назначение: вставка нового узла со значением данных **data** после текущего узла.

Входные параметры: значение данных нового узла.

Выходные параметры: нет (метод изменяет сам список).

InsertBefore(const T& data, const T& NextData)

Назначение: вставка нового узла со значением данных **data** перед узлом, содержащим значение данных **NextData**.

Входные параметры: значение данных нового узла и значение данных узла, перед которым следует вставить.

Выходные параметры: нет (метод изменяет сам список).

InsertAfter(const T& data, const T& PreviousData)

Назначение: вставка нового узла со значением данных data после узла, содержащего значение данных **PreviousData**.

Входные параметры: значение данных нового узла и значение данных узла, после которого следует вставить.

Выходные параметры: нет (метод изменяет сам список).

Remove(const T& data)

Назначение: удаление узла со значением данных data из списка.

Входные параметры: значение данных узла, который нужно удалить.

Выходные параметры: нет (метод изменяет сам список).

removeCurrentNode()

Назначение: удаление текущего узла из списка.

Входные параметры: нет.

Выходные параметры: нет (метод изменяет сам список).

pop\_front()

Назначение: удаление узла из начала списка.

Входные параметры: нет.

Выходные параметры: нет (метод изменяет сам список).

GetSize() const

Назначение: получение количества узлов в списке.

Входные параметры: нет.

Выходные параметры: количество узлов в списке.

Clear()

Назначение: очистка списка, удаление всех узлов.

Входные параметры: нет.

Выходные параметры: нет (метод изменяет сам список).

Reset()

Назначение: сброс текущего указателя на первый узел списка.

Входные параметры: нет.

Выходные параметры: нет (метод изменяет сам список).

Next()

Назначение: перемещение текущего указателя на следующий узел списка.

Входные параметры: нет.

Выходные параметры: нет (метод изменяет сам список).

### Описание класса TRingList

class TRingList : public TList<T>

{

private:

TNode<T>\* pHead;

public:

TRingList();

TRingList(TNode<T>\* \_pFirst);

TRingList(const TRingList<T>& obj);

virtual ~TRingList();

void InsertFirst(const T& data);

void InsertLast(const T& data);

void Remove(const T& data);

void removeCurrentNode();

};

Назначение: реализация кольцевого связанного списка.

Класс наследуется (тип наследования pubic) от класса **TList**

Методы:

TRingList()

Назначение: конструктор по умолчанию.

Входные параметры: нет.

Выходные параметры: объект типа TRingList с пустым списком.

TRingList(TNode<T>\* \_pFirst)

Назначение: конструктор, инициализирующий список указанным первым узлом.

Входные параметры: указатель на первый узел списка.

Выходные параметры: объект типа TRingList с инициализированным списком.

TRingList(const TRingList<T>& obj)

Назначение: конструктор копирования.

Входные параметры: ссылка на другой объект типа TRingList.

Выходные параметры: объект типа TRingList, являющийся копией заданного объекта.

~TRingList()

Назначение: деструктор.

Входные параметры: нет.

Выходные параметры: освобождаются все узлы списка.

InsertFirst(const T& data)

Назначение: вставляет узел с заданными данными в начало списка.

Входные параметры: данные, которые должны быть добавлены в список.

Выходные параметры: список с добавленным узлом в начале.

InsertLast(const T& data)

Назначение: вставляет узел с заданными данными в конец списка.

Входные параметры: данные, которые должны быть добавлены в список.

Выходные параметры: список с добавленным узлом в конце.

Remove(const T& data)

Назначение: удаляет первый узел списка, содержащий заданные данные.

Входные параметры: данные, которые должны быть удалены из списка.

Выходные параметры: список с удаленным узлом (если он был найден).

removeCurrentNode()

Назначение: удаляет текущий узел списка.

Входные параметры: нет.

Выходные параметры: список с удаленным текущим узлом (если он не является первым узлом).

### Описание класса TMonom

**class TMonom**

**{**

**protected:**

**double coef;**

**int degree;**

**public:**

**TMonom();**

**TMonom(const TMonom& m);**

**TMonom(double \_coef, int \_degree);**

**TMonom(double \_coef, int DegreeX, int DegreeY, int DegreeZ);**

**bool operator>(const TMonom& m) const;**

**bool operator<(const TMonom& m) const;**

**bool operator==(const TMonom& m) const;**

**bool operator!=(const TMonom& m) const;**

**TMonom& operator=(const TMonom& m);**

**TMonom operator+(const TMonom& m)const;**

**TMonom operator-(const TMonom& m)const;**

**TMonom operator\*(const TMonom& m) const;**

**double GetValue(double x, double y, double z);**

**double GetCoef() { return coef; };**

**int GetDegree() { return degree; };**

**int GetDegreeX() const;**

**int GetDegreeY() const;**

**int GetDegreeZ() const;**

**TMonom dif\_x();**

**TMonom dif\_y();**

**TMonom dif\_z();**

**void SetCoef(double \_coef) { coef = \_coef; };**

**void SetDegree(int \_degree);**

**void SetDegreeXYZ(int DegreeX, int DegreeY, int DegreeZ);**

**string GetStr() const;**

**bool TheSameCoefANDDegree(const TMonom& m);**

**friend ostream& operator<<(ostream& out, const TMonom& monom)**

Назначение: представление монома.

Поля:

**Coef-**коэфициент монома**.**

**Degree-** свёртка степеней x y z.

Методы:

**TMonom()**

Назначение: инициализировать моном нулевым коэффициентом и нулевыми степенями.

Входные данные: отсутствуют.

**TMonom(const TMonom& m)**

Назначение: создание нового монома со значениями коэффициента и степеней, идентичными значению в другом мономе m.

Входные данные: m- ссылка на существующий моном, значения которого будут скопированы.

**TMonom(double \_coef, int \_degree)**

Назначение: создание монома с указанным коэффициентом **\_coef** и суммарной степенью**\_degree.**

Входные данные: **\_coef**- коэффициент нового монома, **\_degree**- свёртка степеней x,y,z.

**TMonom(double \_coef, int DegreeX, int DegreeY, int DegreeZ)**

Назначение: создание монома с указанным коэффициентом **\_coef** и индивидуальными степенями **DegreeX, DegreeY и DegreeZ.**

Входные данные**: \_coef**- коэффициент нового монома.

**DegreeX**- степень x нового монома.

**DegreeY**- степень y нового монома.

**DegreeZ**- степень z нового монома.

operator>(const TMonom& m) const

Назначение: сравнение текущего монома с другим мономом m и проверить, больше ли его степень.

Входные данные: ссылка на другой моном для сравнения.

Выходные данные: **true**, если степень текущего монома больше, чем у m; иначе **false**.

**operator<(const TMonom& m) const**

Назначение: сравнение текущего монома с другим мономом m и проверить, меньше ли его степень.

Входные данные: m-ссылка на другой моном для сравнения.

Выходные данные: **true**, если степень текущего монома меньше, чем у m; иначе false.

**operator==(const TMonom& m) const**

Назначение: сравнение текущего моном с другим мономом m и проверить, равны ли их степени.

Входные данные: **m**- ссылка на другой моном для сравнения.

Выходные данные: **true**, если степени текущего монома и m равны; иначе **false**.

**operator!=(const TMonom& m) const**

Назначение: сравнение текущего монома с другим мономом m и проверка, не равны ли их степени.

Входные данные:

**m**- ссылка на другой моном для сравнения.

Выходные данные: **true**, если степени текущего монома и m не равны; иначе **false**.

operator+:

Назначение: сложение двух мономов.

Входные параметры: моном m.

Выходные параметры: моном, являющийся суммой двух операндов.

operator-:

Назначение: вычитание одного монома из другого.

Входные параметры: моном **m**.

Выходные параметры: моном, являющийся разностью двух операндов.

operator\*:

Назначение: умножение двух мономов.

Входные параметры: моном **m**.

Выходные параметры: моном, являющийся произведением двух операндов.

GetDegree() const

Назначение: получение свертки степени текущего монома.

Входные данные: отсутствуют.

Выходные данные: свёртка степени текущего монома.

**SetDegree(int newDegree)**

Назначение: установка суммарной степени текущего монома в указанное значение **newDegree.**

Входные данные: **newDegree-** новая суммарная степень.

Выходные данные: отсутствуют.

**GetDegreeX() const**

Назначение: получение степени x текущего монома.

Входные данные: отсутствуют.

Выходные данные: степень x текущего монома.

**SetDegreeX(int newDegreeX)**

Назначение: установка степени x текущего монома в указанное значение **newDegreeX.**

Входные данные: **newDegreeX**- новая степень x.

Выходные данные: отсутствуют.

**GetDegreeY() const**

Назначение: получение степени y текущего монома.

Входные данные: отсутствуют.

Выходные данные: степень y текущего монома.

**SetDegreeY(int newDegreeY)**

Назначение: установка степени y текущего монома в указанное значение **newDegreeY**.

Входные данные: **newDegreeY**- новая степень y.

Выходные данные: отсутствуют.

**GetDegreeZ() const**

Назначение: получение степени z текущего монома.

Входные данные: отсутствуют.

Выходные данные: степень z текущего монома.

**SetDegreeZ(int newDegreeZ)**

Назначение: установка степени z текущего монома в указанное значение **newDegreeZ**.

Входные данные: **newDegreeZ**- новая степень z.

Выходные данные: отсутствуют.

**dif\_x()**

Назначение: вычисление частной производной монома по x.

Входные параметры: нет.

Выходные параметры: моном, являющийся частной производной исходного монома по x.

**dif\_y():**

Назначение: вычисление частной производной монома по y.

Входные параметры: нет.

Выходные параметры: моном, являющийся частной производной исходного монома по y.

**dif\_z():**

Назначение: вычисляетение частной производной монома по z.

Входные параметры: нет.

Выходные параметры: моном, являющийся частной производной исходного монома по z.

**SetCoef(double \_coef):**

Назначение: устанавка коэффициента монома.

Входные параметры: коэффициент **\_coef**.

Выходные параметры: нет (метод изменяет сам моном).

**SetDegree(int \_degree):**

Назначение: установка общей степени монома.

Входные параметры: степень **\_degree**.

Выходные параметры: нет (метод изменяет сам моном).

**SetDegreeXYZ(int DegreeX, int DegreeY, int DegreeZ)**:

Назначение: установка степени монома по переменным x, y и z.

Входные параметры: степень **DegreeX** по x, степень **DegreeY** по y и степень **DegreeZ** по z.

Выходные параметры: нет (метод изменяет сам моном).

**GetStr() const**:

Назначение: возвращает строковое представление монома.

Входные параметры: нет.

Выходные параметры: строковое представление монома.

**operator<<**

Входные параметры: **ostream& out**- Ссылка на выходной поток, в который будет выводиться моном.

**const TMonom& monom**-Ссылка на объект **TMonom**, который будет выводиться.

Выходные параметры: возвращает ссылку на выходной поток.

### Описание класса TPolinom

**class TPolinom**

**{**

**private:**

**TRingList<TMonom> monoms;**

**string name;**

**void NormalView();**

**void DeleteZero();**

**void IntoList(const string& n);**

**void Sort();**

**public:**

**TPolinom();**

**TPolinom(const string& n);**

**TPolinom(const TRingList<TMonom>& m);**

**TPolinom(const TPolinom& p);**

**~TPolinom();**

**TPolinom operator-() const;**

**TPolinom operator+(TPolinom& p);**

**TPolinom operator-(TPolinom& p);**

**TPolinom operator\*(TPolinom& p);**

**const TPolinom& operator=(const TPolinom& p);**

**double operator()(double \_x, double \_y, double \_z) const;**

**TPolinom dif\_x() const;**

**TPolinom dif\_y() const;**

**TPolinom dif\_z() const;**

**string get\_name() const { return name; }**

**string To\_string() const;**

**vector <string> Parse(string name);**

**bool IsConst(const string& lecsem);**

**bool isValidMonomial(const string& monomial);**

**bool isValidVariable(const string& monomial, int& i);**

**bool isValidCoefficient(const string& monomial, int& i);**

**bool checkPolynomial(const string& polyStr);**

**void ProcessVariable(const string& variable, int& degree, int& i, const vector<string>& elements);**

**void ProcessCoefficient(const string& element, double& coef, int& nextSign);**

Поля:

**Monoms -** список мономов;

**Name –** полином в виде строки;

Методы:

**void NormalView()**

Назначение: сортировка мономов в порядке убывания степеней и удаление нулевых мономов.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**void DeleteZero()**

Назначение: удаление всех нулевых мономов из списка.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**void IntoList(const string& n)**

Назначение: перевод строки полинома в список отдельных мономов.

Входные параметры: **n** – строка полинома.

Выходные параметры: отсутствуют.

**void Sort()**

Назначение: сортировка полинома по степеням мономов.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TPolinom()**

Назначение: конструктор по умолчанию. Создает пустой полином.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TPolinom(const string& n)**

Назначение: конструктор, создающий полином из заданной строки.

Входные параметры: строка, представляющая полином.

Выходные параметры: отсутствуют.

**TPolinom(const TRingList<TMonom>& m)**

Назначение: конструктор, создающий полином из заданного списка мономов.

Входные параметры: список мономов.

Выходные параметры: отсутствуют.

**TPolinom(const TPolinom& p)**

Назначение: конструктор копирования. Создает копию заданного полинома.

Входные параметры: ссылка на другой полином.

Выходные параметры: отсутствуют.

**~TPolinom()**

Назначение: деструктор. Удаляет полином.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TPolinom operator-() const**

Назначение: унарный оператор минус. Возвращает полином, противоположный по знаку исходному.

Входные параметры: отсутствуют.

Выходные параметры: новый полином.

**TPolinom operator+(TPolinom& p)**

Назначение: оператор сложения. Возвращает новый полином, являющийся суммой исходного полинома и заданного полинома.

Входные параметры: ссылка на другой полином.

Выходные параметры: новый полином.

**TPolinom operator-(TPolinom& p)**

Назначение: оператор вычитания. Возвращает новый полином, являющийся разностью исходного полинома и заданного полинома.

Входные параметры: ссылка на другой полином.

Выходные параметры: новый полином.

**TPolinom operator\*(TPolinom& p)**

Назначение: оператор умножения. Возвращает новый полином, являющийся произведением исходного полинома и заданного полинома.

Входные параметры: ссылка на другой полином.

Выходные параметры: новый полином.

**const TPolinom& operator=(const TPolinom& p)**

Назначение: оператор присваивания. Присваивает исходному полиному значения заданного полинома.

Входные данные: ссылка на другой полином.

Выходные данные: ссылка на исходный полином.

**double operator()(double \_x, double \_y, double \_z) const**

Назначение: вычисление значения полинома в точке с заданными координатами.

Входные параметры: координаты точки (x, y, z).

Выходные параметры: значение полинома в данной точке.

**TPolinom dif\_x() const**

Назначение: вычисление производной полинома по переменной x.

Входные параметры: отсутствуют.

Выходные параметры: новый полином, являющийся производной по x исходного полинома.

**TPolinom dif\_y() const**

Назначение: вычисление производной полинома по переменной y.

Входные параметры: отсутствуют.

• Выходные параметры: новый полином, являющийся производной по y исходного полинома.

**TPolinom dif\_z() const**

Назначение: вычисление производной полинома по переменной z.

Входные параметры: отсутствуют.

Выходные параметры: новый полином, являющийся производной по z исходного полинома.

**string get\_name() const**

Назначение: получение полинома в виде строки.

Входные параметры: отсутствуют.

Выходные параметры: строка, содержащая полином.

**string To\_string() const**

Назначение: преобразование полинома в строковое представление.

Входные параметры: отсутствуют.

Выходные параметры: строка, содержащая строковое представление полинома.

**vector <string> Parse(string name)**

Назначение: парсинг строки, содержащей полином.

Входные параметры: строка, содержащая полином.

Выходные параметры: вектор строк, содержащий элементы полинома.

**bool IsConst(const string& lecsem)**

Назначение: проверка, является ли лексема константой.

Входные параметры: лексема.

Выходные параметры: флаг, указывающий, является ли лексема константой (True) или нет (False).

**bool isValidMonomial(const string& monomial)**

Назначение: проверка, является ли заданная строка допустимым мономом.

Входные параметры: строка, содержащая моном.

Выходные параметры: флаг, указывающий, является ли строка допустимым мономом (True) или нет (False).

**bool isValidVariable(const string& monomial, int& i)**

Назначение: проверка, является ли заданная строка допустимой переменной.

Входные параметры: строка, содержащая переменную.

Выходные параметры: флаг, указывающий, является ли строка допустимой переменной (True) или нет (False), а также индекс переменной в строке.

**bool isValidCoefficient(const string& monomial, int& i)**

Назначение: проверка, является ли заданная строка допустимым коэффициентом.

Входные параметры: строка, содержащая коэффициент.

Выходные параметры: флаг, указывающий, является ли строка допустимым коэффициентом (True) или нет (False), а также индекс следующего знака в строке.

**bool checkPolynomial(const string& polyStr)**

Назначение: проверка, является ли заданная строка допустимым полиномом.

Входные параметры: строка, содержащая полином.

Выходные параметры: флаг, указывающий, является ли строка допустимым полиномом (True) или нет (False).

**void ProcessVariable(const string& variable, int& degree, int& i, const vector<string>& elements);**

Назначение: обработка переменной в мономе.

Входные параметры: переменная, степень переменной, индекс в строке, вектор элементов строки.

Выходные параметры: отсутствуют.

**void ProcessCoefficient(const string& element, double& coef, int& nextSign)**

Назначение: обработка коэффициента в мономе.

Входные параметры: элемент строки, коэффициент, следующий знак в строке.

Выходные параметры: отсутствуют.

# Заключение

В данной лабораторной работе были изучены различные варианты представления и обработки полиномов с использованием структур данных, в частности, списков. Были разработаны программы, позволяющие работать со списками, хранящими полиномы, включая ввод, вывод, сложение, вычитание, умножение и вычисление значений полиномов.

Проведенное исследование показало эффективность использования списков для хранения и обработки полиномов. Списковая структура данных обеспечивает удобный доступ к элементам полинома, что позволяет легко выполнять различные операции и манипуляции с ним.

# Литература

1. Лекция «Односвязные списки» Сысоев А.В [https://cloud.unn.ru/s/FkYBW5rJLDCgBmJ].

# Приложения

## Приложение А. Реализация класса TList

#ifndef TLIST\_H

#define TLIST\_H

#include <iostream>

#include "tnode.h"

using namespace std;

template <typename T>

class TList

{

protected:

TNode<T>\* pFirst;

TNode<T>\* pLast;

TNode<T>\* pCurr;

TNode<T>\* pStop;

public:

TList();

TList(TNode<T>\* \_pFirst);

TList(const TList<T>& obj);

virtual ~TList();

bool IsEmpty() const;

bool IsFull() const;

bool Is\_End() const;

TNode<T>\* Search(const T& data);

TNode<T>\* GetCurrent() const;

void sort();

virtual void InsertFirst(const T& data);

virtual void InsertLast(const T& data);

const TList<T>& operator=(const TList<T>& p);

void InsertBefore(const T& data);

void InsertAfter(const T& data);

void InsertBefore(const T& data, const T& NextData);

void InsertAfter(const T& data, const T& PreviousData);

virtual void Remove(const T& data);

virtual void removeCurrentNode();

void pop\_front();

int GetSize() const;

void Clear();

void Reset();

void Next();

friend ostream& operator<<(ostream& out, TList<T>& list)

{

TNode<T>\* tmp = list.pFirst;

if (list.IsEmpty())

{

out << "empty list\n";

return out;

}

while (tmp != list.pStop)

{

out << tmp->key << " ";

tmp = tmp->pNext;

}

out << endl;

return out;

}

};

template<typename T>

TList<T>::TList()

{

pStop = nullptr;

pFirst = pStop;

pLast = pStop;

pCurr = pStop;

}

template<typename T>

TList<T>::TList(TNode<T>\* \_pFirst)

{

pStop = nullptr;

pFirst = \_pFirst;

if (pFirst == pStop)

{

pLast = pStop;

pCurr = pStop;

return;

}

TNode<T>\* tmp = pFirst;

while (tmp->pNext != pStop)

{

tmp = tmp->pNext;

}

pLast = tmp;

}

template<typename T>

TList<T>::TList(const TList<T>& obj)

{

TNode<T>\* tmp = obj.pFirst;

if (obj.pFirst == obj.pStop)

{

pStop = nullptr;

pFirst = pStop;

pLast = pStop;

pCurr = pFirst;

return;

}

pFirst = new TNode<T>(obj.pFirst->key);

TNode<T>\* tmp2 = pFirst;

tmp = tmp->pNext;

while (tmp != obj.pStop)

{

tmp2->pNext = new TNode<T>(tmp->key);

tmp2 = tmp2->pNext;

tmp = tmp->pNext;

}

pLast = tmp2;

pStop = nullptr;

pCurr = pFirst;

}

template<typename T>

TList<T>::~TList()

{

Clear();

}

template<typename T>

bool TList<T>::IsEmpty() const

{

if (pFirst == pStop) return true;

else return false;

}

template<typename T>

bool TList<T>::IsFull() const

{

TNode<T>\* tmp = new TNode<T>;

if (tmp != nullptr)

{

delete tmp;

return false;

}

else return true;

}

template<typename T>

bool TList<T>::Is\_End() const

{

if (pCurr == pStop) return true;

else return false;

}

template<typename T>

TNode<T>\* TList<T>::Search(const T& data)

{

if (pFirst == nullptr) return nullptr;

TNode<T>\* tmp = pFirst;

while (tmp != pStop)

{

if (tmp->key != data)

{

break;

}

tmp = tmp->pNext;

}

if (tmp == pStop && tmp->key != data)

{

return nullptr;

}

return tmp;

}

template<typename T>

TNode<T>\* TList<T>::GetCurrent() const

{

return pCurr;

}

template<typename T>

void TList<T>::sort()

{

TNode<T>\* tmp1 = pFirst;

while (tmp1->pNext != pStop)

{

TNode<T>\* tmp2 = tmp1->pNext;

while (tmp2 != pStop)

{

if (tmp1->key < tmp2->key)

{

T elem = tmp1->key;

tmp1->key = tmp2->key;

tmp2->key = elem;

}

tmp2 = tmp2->pNext;

}

tmp1 = tmp1->pNext;

}

}

template<typename T>

void TList<T>::InsertFirst(const T& data)

{

pFirst = new TNode<T>(data, pFirst);

if (pFirst->pNext == pStop)

{

pLast = pFirst;

}

pCurr = pFirst;

}

template<typename T>

void TList<T>::InsertLast(const T& data)

{

if (pFirst == pStop)

{

InsertFirst(data);

}

else

{

TNode<T>\* tmp = new TNode<T>(data, pStop);

pLast->pNext = tmp;

pLast = tmp;

}

if (pCurr == pStop) Reset();

}

template<typename T>

const TList<T>& TList<T>::operator=(const TList<T>& p)

{

if (p.pFirst == p.pStop)

{

throw"Error";

}

{

return \*this;

}

Clear();

TNode<T>\* pNode = p.pFirst;

while (pNode != p.pStop)

{

InsertLast(pNode->key);

pNode = pNode->pNext;

}

return \*this;

}

template<typename T>

void TList<T>::InsertBefore(const T& data)

{

if (pCurr == pStop)

{

throw"change position";

}

if (pCurr == pFirst)

{

InsertFirst(data);

}

else

{

TNode<T>\* tmp = pFirst;

while (tmp->pNext != pCurr)

{

tmp = tmp->pNext;

}

tmp->pNext = new TNode<T>(data, pCurr);

}

}

template<typename T>

void TList<T>::InsertAfter(const T& data)

{

if (pCurr == pStop)

{

throw"change position";

}

if (Is\_End())

{

InsertLast(data);

}

else

{

pCurr->pNext = new TNode<T>(data, pCurr->pNext);

}

}

template<typename T>

void TList<T>::InsertBefore(const T& data, const T& NextData)

{

TNode<T>\* tmp = pFirst;

TNode<T>\* Before = nullptr;

while (tmp != pStop && tmp->key != NextData)

{

Before = tmp;

tmp = tmp->pNext;

}

if (tmp == pStop)

{

throw"Not found";

}

if (tmp == pFirst)

{

InsertFirst(data);

return;

}

Before->pNext = new TNode<T>(data, tmp);

if (pCurr == pStop) Reset();

}

template<typename T>

void TList<T>::InsertAfter(const T& data, const T& PreviousData)

{

TNode<T>\* tmp = pFirst;

while (tmp != pStop && tmp->key != PreviousData)

{

tmp = tmp->pNext;

}

if (tmp == pStop)

{

throw"Not found";

}

if (tmp == pLast)

{

InsertLast(data);

return;

}

tmp->pNext = new TNode<T>(data, tmp->pNext);

}

template<typename T>

void TList<T>::Remove(const T& data)

{

if (pFirst == pStop)

{

throw"List is empty";

}

TNode<T>\* tmp = pFirst;

TNode<T>\* previous = nullptr;

while (tmp!= pStop && tmp->key != data)

{

previous = tmp;

tmp = tmp->pNext;

}

if (tmp== pStop && tmp->key != data)

{

throw "Not found";

}

if (previous == nullptr)

{

pFirst = tmp->pNext;

delete tmp;

return;

}

previous->pNext = tmp->pNext;

delete tmp;

}

template<typename T>

void TList<T>::removeCurrentNode()

{

if (pCurr == nullptr) {

return;

}

TNode<T>\* pPrev = nullptr;

TNode<T>\* pNode = pFirst;

while (pNode != nullptr && pNode != pCurr) {

pPrev = pNode;

pNode = pNode->pNext;

}

if (pNode == nullptr) {

return; // Узел не найден, ничего не делаем

}

if (pPrev != nullptr) {

pPrev->pNext = pCurr->pNext;

}

else {

pFirst = pCurr->pNext;

}

if (pLast == pCurr) {

pLast = pPrev;

}

TNode<T>\* temp = pCurr;

pCurr = pCurr->pNext;

if (pCurr == pStop) Reset();

delete temp;

}

template<typename T>

void TList<T>::pop\_front()

{

if (pFirst == pStop)

{

throw"list is empty";

}

TNode<T>\* tmp = pFirst;

pFirst = pFirst->pNext;

delete tmp;

}

template<typename T>

int TList<T>::GetSize() const

{

int size = 0;

TNode<T>\* tmp = pFirst;

while (tmp != pStop)

{

tmp = tmp->pNext;

size++;

}

return size;

}

template<typename T>

void TList<T>::Clear()

{

while (GetSize())

{

pop\_front();

}

}

template<typename T>

void TList<T>::Reset()

{

pCurr = pFirst;

}

template<typename T>

void TList<T>::Next()

{

if (!Is\_End() && pCurr != pStop)

{

pCurr = pCurr->pNext;

}

else

{

Reset();

}

}

#endif

## Приложение Б. Реализация класса TRinglist

#ifndef TRINGLIST\_H

#define TRINGLIST\_H

#include <iostream>

#include "tlist.h"

using namespace std;

template<typename T>

class TRingList : public TList<T>

{

private:

TNode<T>\* pHead;

public:

TRingList();

TRingList(TNode<T>\* \_pFirst);

TRingList(const TRingList<T>& obj);

virtual ~TRingList();

void InsertFirst(const T& data);

void InsertLast(const T& data);

void Remove(const T& data);

void removeCurrentNode();

}

template<typename T>

TRingList<T>::TRingList() : TList<T>()

{

pHead = new TNode<T>(T());

this->pStop = pHead;

pHead->pNext = this->pFirst;

this->pFirst = pHead;

this->pLast = pHead;

this->pCurr = pHead;

}

template<typename T>

TRingList<T>::TRingList(TNode<T>\* \_pFirst)

{

pHead = new TNode<T>(T());

pHead->pNext = this->pFirst;

this->pStop = pHead;

if (this->pFirst == nullptr)

{

this->pFirst = pHead;

this->pLast = pHead;

this->pCurr = pHead;

}

else

{

TNode<T>\* tmp = this->pFirst;

while (tmp->pNext != nullptr)

{

tmp = tmp->pNext;

}

tmp->pNext = pHead;

this->pLast = tmp;

}

}

template<typename T>

TRingList<T>::TRingList(const TRingList<T>& obj)

{

pHead = new TNode<T>(T());

pHead->pNext = this->pFirst;

this->pStop = pHead;

if (obj.pFirst == obj.pStop)

{

this->pFirst = pHead;

this->pLast = pHead;

this->pCurr = pHead;

}

else

{

this->pFirst = new TNode<T>(obj.pFirst->key);

TNode<T>\* tmp = obj.pFirst;

TNode<T>\* tmp\_1 = this->pFirst;

while (tmp->pNext != obj.pStop)

{

tmp\_1->pNext = new TNode<T>(tmp->pNext->key);

tmp\_1 = tmp\_1->pNext;

tmp = tmp->pNext;

}

this->pLast = tmp\_1;

this->pLast->pNext = this->pStop;

this->pCurr = this->pFirst;

}

}

template<typename T>

TRingList<T>::~TRingList()

{

if (pHead)

delete pHead;

}

template<typename T>

void TRingList<T>::InsertFirst(const T& data)

{

TList<T>::InsertFirst(data);

this->pHead->pNext = this->pFirst;

this->pStop = this->pHead;

this->pLast->pNext = this->pStop;

}

template<typename T>

void TRingList<T>::InsertLast(const T& data)

{

TList<T>::InsertLast(data);

this->pHead->pNext = this->pFirst;

this->pStop = this->pHead;

this->pLast->pNext = this->pStop;

}

template<typename T>

void TRingList<T>::Remove(const T& data)

{

TList<T>::Remove(data);

if (this->pFirst != this->pStop && this->pFirst != nullptr)

{

this->pLast->pNext = this->pHead;

}

}

template<typename T>

void TRingList<T>::removeCurrentNode()

{

TList<T>::removeCurrentNode();

if (this->pFirst != this->pStop && this->pFirst != nullptr)

{

this->pLast->pNext = this->pHead;

}

}

#endif

## Приложение В. Реализация класса TMonom

#ifndef TMONOM\_H

#define TMONOM\_H

#include <iostream>

#include <string>

using namespace std;

class TMonom

{

protected:

double coef;

int degree;

public:

TMonom();

TMonom(const TMonom& m);

TMonom(double \_coef, int \_degree);

TMonom(double \_coef, int DegreeX, int DegreeY, int DegreeZ);

bool operator>(const TMonom& m) const;

bool operator<(const TMonom& m) const;

bool operator==(const TMonom& m) const;

bool operator!=(const TMonom& m) const;

TMonom& operator=(const TMonom& m);

TMonom operator+(const TMonom& m)const;

TMonom operator-(const TMonom& m)const;

TMonom operator\*(const TMonom& m) const;

double GetValue(double x, double y, double z);

double GetCoef() { return coef; };

int GetDegree() { return degree; };

int GetDegreeX() const;

int GetDegreeY() const;

int GetDegreeZ() const;

TMonom dif\_x();

TMonom dif\_y();

TMonom dif\_z();

void SetCoef(double \_coef) { coef = \_coef; };

void SetDegree(int \_degree);

void SetDegreeXYZ(int DegreeX, int DegreeY, int DegreeZ);

string GetStr() const;

bool TheSameCoefANDDegree(const TMonom& m);

friend ostream& operator<<(ostream& out, const TMonom& monom)

{

out << monom.coef;

if (monom.GetDegreeX() > 0)

{

out << "x";

if (monom.GetDegreeX() > 1)

{

out << "^" << monom.GetDegreeX();

}

}

if (monom.GetDegreeY() > 0)

{

out << "y";

if (monom.GetDegreeY() > 1)

{

out << "^" << monom.GetDegreeY();

}

}

if (monom.GetDegreeZ() > 0)

{

out << "z";

if (monom.GetDegreeZ() > 1)

{

out << "^" << monom.GetDegreeZ();

}

}

return out;

}

};

#include "tmonom.h"

#include <iostream>

#include <string>

using namespace std;

TMonom::TMonom()

{

coef = 0;

degree = 0;

}

TMonom::TMonom(const TMonom& m)

{

coef = m.coef;

degree = m.degree;

}

TMonom::TMonom(double \_coef, int \_degree)

{

if (\_degree > 999 || \_degree < 0)

{

throw "intervai for degree [0,999]";

}

coef = \_coef;

degree = \_degree;

}

TMonom::TMonom(double \_coef, int DegreeX, int DegreeY, int DegreeZ)

{

if (DegreeX < 0 || DegreeX>9 || DegreeY < 0 || DegreeY>9 || DegreeZ < 0 || DegreeZ>9)

{

throw"Interval for degrees [0,9]";

}

coef = \_coef;

degree = DegreeX \* 100 + DegreeY \* 10 + DegreeZ;

}

bool TMonom::operator>(const TMonom& m) const

{

if (degree > m.degree)

return true;

return false;

}

bool TMonom::operator<(const TMonom& m) const

{

if (degree < m.degree)

return true;

return false;

}

bool TMonom::operator==(const TMonom& m) const

{

if (degree == m.degree)

return true;

return false;

}

bool TMonom::operator!=(const TMonom& m) const

{

if (degree != m.degree)

return true;

return false;

}

TMonom& TMonom::operator=(const TMonom& m)

{

coef = m.coef;

degree = m.degree;

return \*this;

}

TMonom TMonom::operator+(const TMonom& m) const

{

if (degree != m.degree)

throw"Cant add different degrees";

TMonom res;

res.degree = degree;

res.coef = coef + m.coef;

return res;

}

TMonom TMonom::operator-(const TMonom& m) const

{

if (degree != m.degree)

throw"Cant add different degrees";

TMonom res;

res.degree = degree;

res.coef = coef - m.coef;

return res;

}

TMonom TMonom::operator\*(const TMonom& m) const

{

int ResultDegree = degree + m.degree;

if (ResultDegree < 0 || ResultDegree>999)

{

throw"Interwal for degree [0,999]";

}

double ResultCoef = coef \* m.coef;

TMonom res(ResultCoef, ResultDegree);

return res;

}

double TMonom::GetValue(double x, double y, double z)

{

double ResX = 1;

for (int i = 0; i < GetDegreeX(); i++)

{

ResX \*= x;

}

double ResY = 1;

for (int i = 0; i < GetDegreeY(); i++)

{

ResY \*= y;

}

double ResZ = 1;

for (int i = 0; i < GetDegreeZ(); i++)

{

ResZ \*= z;

}

return (coef \* ResX \* ResY \* ResZ);

}

int TMonom::GetDegreeX() const

{

if (degree < 100)

return 0;

return(degree / 100);

}

int TMonom::GetDegreeY() const

{

if (degree < 10)

return 0;

if (degree < 100)

return (degree / 10);

return((degree - GetDegreeX() \* 100) / 10);

}

int TMonom::GetDegreeZ() const

{

if (degree < 10)

return degree;

return(degree - GetDegreeX() \* 100 - GetDegreeY() \* 10);

}

TMonom TMonom::dif\_x()

{

if (GetDegreeX() == 0)

{

TMonom res(0, 0);

return res;

}

if (GetDegreeX() == 1)

{

int ResDegree = GetDegreeY() \* 10 + GetDegreeZ();

TMonom res(coef, ResDegree);

return res;

}

double ResCoef = coef \* GetDegreeX();

int ResDegree = (GetDegreeX() - 1) \* 100 + GetDegreeY() \* 10 + GetDegreeZ();

TMonom res(ResCoef, ResDegree);

return res;

}

TMonom TMonom::dif\_y()

{

if (GetDegreeY() == 0)

{

TMonom res(0, 0);

return res;

}

if (GetDegreeY() == 1)

{

int ResDegree = GetDegreeX() \* 100 + GetDegreeZ();

TMonom res(coef, ResDegree);

return res;

}

double ResCoef = coef \* GetDegreeY();

int ResDegree = GetDegreeX() \* 100 + (GetDegreeY() - 1) \* 10 + GetDegreeZ();

TMonom res(ResCoef, ResDegree);

return res;

}

TMonom TMonom::dif\_z()

{

if (GetDegreeZ() == 0)

{

TMonom res(0, 0);

return res;

}

if (GetDegreeZ() == 1)

{

int ResDegree = GetDegreeX() \* 100 + GetDegreeY() \* 10;

TMonom res(coef, ResDegree);

return res;

}

double ResCoef = coef \* GetDegreeZ();

int ResDegree = GetDegreeX() \* 100 + GetDegreeY() \* 10 + (GetDegreeZ() - 1);

TMonom res(ResCoef, ResDegree);

return res;

}

void TMonom::SetDegree(int \_degree)

{

if (\_degree < 0 || \_degree>999)

throw"digree must be in interval [0,999]";

degree = \_degree;

}

void TMonom::SetDegreeXYZ(int DegreeX, int DegreeY, int DegreeZ)

{

if (DegreeX < 0 || DegreeX>9 || DegreeY < 0 || DegreeY>9 || DegreeZ < 0 || DegreeZ>9)

throw"Interval for degrees [0,9]";

degree = DegreeX \* 100 + DegreeY \* 10 + DegreeZ;

}

string TMonom::GetStr() const

{

string res = " ";

res += to\_string(coef);

if (GetDegreeX() == 1)

res += "x";

if (GetDegreeX() > 1)

{

res += "x^" + to\_string(GetDegreeX());

}

if (GetDegreeY() == 1)

res += "y";

if (GetDegreeY() > 1)

{

res += "y^" + to\_string(GetDegreeY());

}

if (GetDegreeZ() == 1)

res += "z";

if (GetDegreeZ() > 1)

{

res += "z^" + to\_string(GetDegreeZ());

}

return res;

}

bool TMonom::TheSameCoefANDDegree(const TMonom& m)

{

if (coef == m.coef && degree == m.degree)

return true;

else

return false;

}

#endif

## Приложение Г. Реализация класса TPolinom

#ifndef POLYNOM\_H

#define POLYNOM\_H

#include <iostream>

#include <string>

#include <map>

#include <vector>

#include "tringlist.h"

#include "tmonom.h"

using namespace std;

class TPolinom

{

private:

TRingList<TMonom> monoms;

string name;

void NormalView();

void DeleteZero();

void IntoList(const string& n);

void Sort();

public:

TPolinom();

TPolinom(const string& n);

TPolinom(const TRingList<TMonom>& m);

TPolinom(const TPolinom& p);

~TPolinom();

TPolinom operator-() const;

TPolinom operator+(TPolinom& p);

TPolinom operator-(TPolinom& p);

TPolinom operator\*(TPolinom& p);

const TPolinom& operator=(const TPolinom& p);

double operator()(double \_x, double \_y, double \_z) const;

TPolinom dif\_x() const;

TPolinom dif\_y() const;

TPolinom dif\_z() const;

string get\_name() const { return name; }

string To\_string() const;

vector <string> Parse(string name);

bool IsConst(const string& lecsem);

bool isValidMonomial(const string& monomial);

bool isValidVariable(const string& monomial, int& i);

bool isValidCoefficient(const string& monomial, int& i);

bool checkPolynomial(const string& polyStr);

void ProcessVariable(const string& variable, int& degree, int& i, const vector<string>& elements);

void ProcessCoefficient(const string& element, double& coef, int& nextSign);

friend ostream& operator<<(ostream& out, TPolinom& p)

{

out << p.To\_string() << endl;

return out;

}

};

#endif

#include "tpolynom.h"

#include"tmonom.h"

#include <iostream>

#include <string>

using namespace std;

TPolinom::TPolinom()

{

name = " ";

IntoList(name);

}

TPolinom::TPolinom(const string& n)

{

if (checkPolynomial(n))

{

IntoList(n);

NormalView();

name = To\_string();

}

else

{

throw"Error";

}

}

void TPolinom::NormalView()

{

TRingList<TMonom> poly = monoms;

monoms.Clear();

poly.sort();

while (!poly.IsEmpty())

{

poly.Reset();

TMonom cur = poly.GetCurrent()->key;

double coef = poly.GetCurrent()->key.GetCoef();

while (!poly.Is\_End())

{

poly.Next();

if (poly.GetCurrent()->key == cur)

{

coef += poly.GetCurrent()->key.GetCoef();

}

if (poly.GetCurrent()->key != cur)

{

break;

}

}

if (coef != 0)

{

cur.SetCoef(coef);

monoms.InsertLast(cur);

}

poly.Reset();

while (!poly.Is\_End())

{

if (poly.GetCurrent()->key == cur)

{

poly.removeCurrentNode();

}

if (poly.GetCurrent()->key != cur)

{

break;

}

}

}

if (monoms.IsEmpty())

{

TMonom res(0, 0);

monoms.InsertLast(res);

}

}

void TPolinom::DeleteZero()

{

monoms.Reset();

while (!monoms.Is\_End())

{

if (monoms.GetCurrent()->key.GetCoef() == 0.0)

{

TMonom obj(monoms.GetCurrent()->key.GetCoef(), monoms.GetCurrent()->key.GetDegree());

monoms.Next();

monoms.Remove(obj);

}

else

{

monoms.Next();

}

}

if (monoms.IsEmpty())

{

TMonom monom(0, 0);

monoms.InsertLast(monom);

}

name = To\_string();

}

void TPolinom::ProcessVariable(const string& variable, int& degree, int& i, const vector<string>& elements)

{

if (i + 1 < elements.size())

{

if (elements[i + 1] == "^")

{

degree = stoi(elements[i + 2]);

if (i + 2 < elements.size())

i += 2;

}

else

{

degree = 1;

}

}

else

{

degree = 1;

}

}

void TPolinom::ProcessCoefficient(const string& element, double& coef, int& nextSign)

{

if (IsConst(element))

{

coef = stod(element);

}

else if (element == "+")

{

nextSign = 1;

}

else if (element == "-")

{

nextSign = -1;

}

}

void TPolinom::IntoList(const string& n)

{

vector<string> elements = Parse(n);

double coef = 1;

int degX = 0;

int degY = 0;

int degZ = 0;

int tmp\_next\_znak = 1;

for (int i = 0; i < elements.size(); i++)

{

if (i == 0 && (elements[0] == "-" || elements[0] == "+"))

{

tmp\_next\_znak = (elements[0] == "-") ? -1 : 1;

continue;

}

if (elements[i] == "x")

{

ProcessVariable("x", degX, i, elements);

continue;

}

if (elements[i] == "y")

{

ProcessVariable("y", degY, i, elements);

continue;

}

if (elements[i] == "z")

{

ProcessVariable("z", degZ, i, elements);

continue;

}

if (IsConst(elements[i]))

{

coef = stod(elements[i]);

continue;

}

if (elements[i] == "+" || elements[i] == "-")

{

TMonom monom(tmp\_next\_znak \* coef, degX, degY, degZ);

monoms.InsertLast(monom);

coef = 1;

degX = 0;

degY = 0;

degZ = 0;

tmp\_next\_znak = (elements[i] == "-") ? -1 : 1;

}

}

TMonom monom(tmp\_next\_znak \* coef, degX, degY, degZ);

monoms.InsertLast(monom);

}

TPolinom::TPolinom(const TRingList<TMonom>& m)

{

TRingList<TMonom> tmp = m;

tmp.Reset();

while (!tmp.Is\_End())

{

monoms.InsertLast(tmp.GetCurrent()->key);

tmp.Next();

}

NormalView();

name = To\_string();

}

TPolinom::TPolinom(const TPolinom& p)

{

name = p.name;

TRingList<TMonom> tmp = p.monoms;

tmp.Reset();

while (!tmp.Is\_End())

{

monoms.InsertLast(tmp.GetCurrent()->key);

tmp.Next();

}

}

TPolinom::~TPolinom()

{

name = "";

}

TPolinom TPolinom::operator-() const

{

TPolinom res(monoms);

res.monoms.Reset();

while (!res.monoms.Is\_End())

{

res.monoms.GetCurrent()->key.SetCoef(res.monoms.GetCurrent()->key.GetCoef() \* (-1));

res.monoms.Next();

}

return res;

}

TPolinom TPolinom::operator+(TPolinom& p)

{

TPolinom res(monoms);

TRingList<TMonom> copy = p.monoms;

res.monoms.Reset();

copy.Reset();

while (!res.monoms.Is\_End())

{

while (!copy.Is\_End())

{

if (res.monoms.GetCurrent()->key.GetDegree() == copy.GetCurrent()->key.GetDegree())

{

double coef;

coef = res.monoms.GetCurrent()->key.GetCoef() + copy.GetCurrent()->key.GetCoef();

if (coef == 0)

{

res.monoms.removeCurrentNode();

}

else res.monoms.GetCurrent()->key.SetCoef(coef);

copy.removeCurrentNode();

}

else copy.Next();

}

res.monoms.Next();

copy.Reset();

}

if (!copy.Is\_End())

{

copy.Reset();

while (!copy.Is\_End())

{

res.monoms.InsertLast(copy.GetCurrent()->key);

copy.Next();

}

}

if (res.monoms.IsEmpty())

{

TMonom monom(0, 0);

res.monoms.InsertLast(monom);

}

res.monoms.sort();

res.To\_string();

return res;

}

TPolinom TPolinom::operator-(TPolinom& p)

{

TPolinom poly = -(p);

return (\*this + poly);

}

TPolinom TPolinom::operator\*(TPolinom& p)

{

TRingList<TMonom> copy\_p = p.monoms;

TRingList<TMonom> copy = monoms;

TRingList<TMonom> res\_list;

copy\_p.Reset();

copy.Reset();

while (!copy\_p.Is\_End())

{

while (!copy.Is\_End())

{

res\_list.InsertLast(copy\_p.GetCurrent()->key \* copy.GetCurrent()->key);

copy.Next();

}

copy.Reset();

copy\_p.Next();

}

TPolinom res(res\_list);

return res;

}

const TPolinom& TPolinom::operator=(const TPolinom& p)

{

if (this == &p) return(\*this);

if (!monoms.IsEmpty())

{

monoms.Clear();

}

monoms = p.monoms;

return(\*this);

}

double TPolinom::operator()(double \_x, double \_y, double \_z) const

{

double res = 0;

TRingList<TMonom> copy = this->monoms;

copy.Reset();

while (!copy.Is\_End())

{

res += copy.GetCurrent()->key.GetValue(\_x, \_y, \_z);

copy.Next();

}

return res;

}

void TPolinom::Sort()

{

monoms.sort();

name = To\_string();

}

TPolinom TPolinom::dif\_x() const

{

TRingList<TMonom> copy = this->monoms;

TRingList<TMonom> res\_list;

copy.Reset();

while (!copy.Is\_End())

{

if (copy.GetCurrent()->key.GetDegreeX() > 0)

{

res\_list.InsertLast(copy.GetCurrent()->key.dif\_x());

}

copy.Next();

}

if (res\_list.IsEmpty())

{

TMonom monom(0, 0);

res\_list.InsertLast(monom);

}

TPolinom result(res\_list);

return result;

}

TPolinom TPolinom::dif\_y() const

{

TRingList<TMonom> copy = this->monoms;

TRingList<TMonom> res\_list;

copy.Reset();

while (!copy.Is\_End())

{

if (copy.GetCurrent()->key.GetDegreeY() > 0)

{

res\_list.InsertLast(copy.GetCurrent()->key.dif\_y());

}

copy.Next();

}

if (res\_list.IsEmpty())

{

TMonom monom(0, 0);

res\_list.InsertLast(monom);

}

TPolinom result(res\_list);

return result;

}

TPolinom TPolinom::dif\_z() const

{

TRingList<TMonom> copy = this->monoms;

TRingList<TMonom> res\_list;

copy.Reset();

while (!copy.Is\_End())

{

if (copy.GetCurrent()->key.GetDegreeZ() > 0)

{

res\_list.InsertLast(copy.GetCurrent()->key.dif\_z());

}

copy.Next();

}

if (res\_list.IsEmpty())

{

TMonom monom(0, 0);

res\_list.InsertLast(monom);

}

TPolinom result(res\_list);

return result;

}

string TPolinom::To\_string() const

{

string str = "";

TRingList<TMonom> tmp=monoms;

tmp.Reset();

while (!tmp.Is\_End())

{

if (tmp.GetCurrent()->key.GetCoef() < 0)

{

str += " ";

}

if (tmp.GetCurrent()->key.GetCoef() > 0)

{

str += "+";

}

str += tmp.GetCurrent()->key.GetStr();

tmp.Next();

}

return str;

}

vector<string> TPolinom::Parse(string name)

{

vector<string> tokens;

string current\_token;

for (char c : name) {

if (std::isdigit(c) || c == '.') {

current\_token += c;

}

else if (std::isalpha(c)) {

if (!current\_token.empty()) {

tokens.push\_back(current\_token);

current\_token.clear();

}

current\_token += c;

tokens.push\_back(current\_token);

current\_token.clear();

}

else {

if (!current\_token.empty()) {

tokens.push\_back(current\_token);

current\_token.clear();

}

current\_token += c;

tokens.push\_back(current\_token);

current\_token.clear();

}

}

if (!current\_token.empty()) {

tokens.push\_back(current\_token);

}

return tokens;

}

bool TPolinom::IsConst(const string& lecsem)

{

bool hasDot = false;

bool flag = true;

for (char c : lecsem) {

if (!std::isdigit(c)) {

if (c == '.' && !hasDot) {

hasDot = true;

}

else {

flag = false;

break;

}

}

}

return flag;

}

bool TPolinom::isValidMonomial(const string& monomial)

{

if (monomial.empty()) {

return false;

}

int i = 0;

if (isdigit(monomial[0]))

{

if (!isValidCoefficient(monomial, i)) {

return false;

}

}

for (; i < monomial.size(); i++)

{

if (isalpha(monomial[i]))

{

if (!isValidVariable(monomial, i)) {

return false;

}

}

else if (isdigit(monomial[i]) || monomial[i] == '\*')

{

return false;

}

}

return true;

}

bool TPolinom::isValidCoefficient(const string& monomial, int& i)

{

int findpoint = 0;

while (monomial[i] != 'x' && monomial[i] != 'y' && monomial[i] != 'z' && i < monomial.size() && monomial[i] != '\*' && (!isalpha(monomial[i])))

{

if (isdigit(monomial[i]))

{

i++;

continue;

}

if (monomial[i] == '.')

{

findpoint++;

i++;

continue;

}

if (findpoint > 1)

{

return false;

}

}

return true;

}

bool TPolinom::isValidVariable(const string& monomial, int& i)

{

if (monomial[i] == 'x' || monomial[i] == 'y' || monomial[i] == 'z')

{

char variable = monomial[i];

if (i + 1 < monomial.size() && monomial[i + 1] == '^')

{

if (i + 2 < monomial.size() && isdigit(monomial[i + 2]))

{

i += 2;

return true;

}

else

{

return false;

}

}

return true;

}

else

{

return false;

}

}

bool TPolinom::checkPolynomial(const string& polyStr)

{

size\_t start = 0;

size\_t end = polyStr.find\_first\_of("+-", start);

while (end != string::npos) {

string monomial;

if (start != end) {

monomial = polyStr.substr(start, end - start);

if (!isValidMonomial(monomial)) {

return false;

}

}

start = end + 1;

end = polyStr.find\_first\_of("+-", start);

}

string lastMonomial = polyStr.substr(start);

return isValidMonomial(lastMonomial);

}