

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
Федеральное государственное автономное образовательное учреждение  
высшего образования  
«Национальный исследовательский  
Нижегородский государственный университет им. Н.И. Лобачевского»  
(ННГУ)

Институт информационных технологий, математики и механики

## ЛАБОРАТОРНАЯ РАБОТА

на тему:  
«Аналитические преобразования полиномов от нескольких  
переменных (списки)»

**Выполнил(а):** студент группы  
3822Б1ФИ2

\_\_\_\_\_ / Хохлов А.Д./  
Подпись

**Проверил:** к.т.н, доцент каф. ВВиСП  
\_\_\_\_\_ / Кустикова В.Д./

Подпись

Нижний Новгород  
2024

# Содержание

Введение.....	3
1 Постановка задачи .....	4
2 Руководство пользователя .....	5
2.1 Приложение для демонстрации работы списка.....	5
2.2 Приложение для демонстрации работы аналитических преобразований полиномов от нескольких переменных .....	5
3 Руководство программиста.....	7
3.1 Описание алгоритмов.....	7
3.1.1 Линейный односвязный список .....	7
3.1.2 Кольцевой односвязный список.....	12
3.1.3 Моном.....	13
3.1.4 Полином .....	15
3.2 Описание программной реализации .....	17
3.2.1 Описание структуры TNode .....	17
3.2.2 Описание класса TList.....	18
3.2.3 Описание класса TRingList.....	23
3.2.4 Описание класса TMonom .....	24
3.2.5 Описание класса TPolynom .....	29
Заключение .....	33
Литература .....	34
Приложения .....	35
Приложение А. Реализация класса TNode.....	35
Приложение В. Реализация класса TList .....	35
Приложение С. Реализация класса TRingList.....	42
Приложение D. Реализация класса TMonom.....	43
Приложение Е. Реализация класса TPolynom .....	47

## **Введение**

В современном мире информационных технологий большую роль играют арифметические операции. Одной из важных операций является эффективная работа с полиномами.

Знание и понимание структуры и принципов хранения помогают оптимизировать использование памяти и увеличивать эффективность вычислений.

Таким образом, данная лабораторная работа является актуальной и полезной для студентов и специалистов в области информационных технологий, которые имеют необходимость эффективно работать с битами и битовыми множествами.

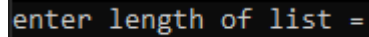
# 1 Постановка задачи

Целью данной лабораторной работы является создание структуры хранения и перевода полинома в структуру данных на языке программирования C++. В рамках работы необходимо разработать классы TNode, TList, TRingList, TMonom и TPolynom, которые будут предоставлять функциональность для работы с полиномами. Основной задачей является реализация основных операций с полиномами.

## 2 Руководство пользователя

### 2.1 Приложение для демонстрации работы списка

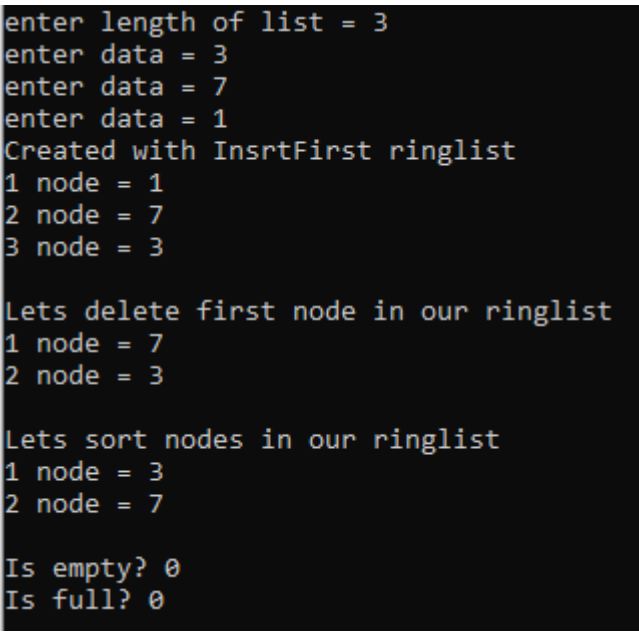
1. Запустите приложение с названием sample\_tringlist.exe. В результате появится окно, показанное ниже (рис. 1).



```
enter length of list =
```

Рис. 1. Основное окно программы

2. Введите длину кольцевого списка. Далее введите значения для каждого звена (рис. 2).



```
enter length of list = 3
enter data = 3
enter data = 7
enter data = 1
Created with InsrtFirst ringlist
1 node = 1
2 node = 7
3 node = 3

Lets delete first node in our ringlist
1 node = 7
2 node = 3

Lets sort nodes in our ringlist
1 node = 3
2 node = 7

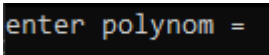
Is empty? 0
Is full? 0
```

Рис. 2. Основное окно программы

3. В результате выведутся основные операции для работы с кольцевым списком (вставка, удаление, сортировка, проверка на пустоту и полноту)

### 2.2 Приложение для демонстрации работы аналитических преобразований полиномов от нескольких переменных

1. Запустите приложение с названием sample\_tpolynom.exe. В результате появится окно, показанное ниже (рис. 3).



```
enter polynom =
```

Рис. 3. Основное окно программы

2. Ведите пример. Это окно покажет основные функции класса TPolynom (дифференциал, разность, присваивание, подсчет) (рис. 4).

```

enter polynom = x^3*y^2-27*z^4

new polynom = -27.000000*z^4+1.000000*x^3*y^2

b: -2.000000*x^2*y^4*z^7+34.000000*x^3*y^1
c: -27.000000*z^4+1.000000*x^3*y^2
b.dif(): -112.000000*x^1*y^3*z^6
b-c: -2.000000*x^2*y^4*z^7+34.000000*x^3*y^1-1.000000*x^3*y^2
c=b: -2.000000*x^2*y^4*z^7+34.000000*x^3*y^1-1.000000*x^3*y^2
x = 3
y = 2
z = 1
c(x, y, z) = 1440

```

Рис. 4. Основное окно программы

## 3 Руководство программиста

### 3.1 Описание алгоритмов

#### 3.1.1 Линейный односвязный список

Список – это динамическая структура данных, состоящая из звеньев, содержащие данные и указатель на следующее звено. Данная структура поддерживает такие операции как: вставка (в начало, конец, до, после), поиск, удаление, проверка на пустоту и полноту.

Для реализации некоторых операций, текущий элемент должен изменяться, поэтому он является отдельным полем в структуре.

##### Операция добавления в начало:

Операция добавления элемента реализуется при помощи указателя на первый элемент.

1. Если структура хранения пуста, то создаем новый элемент (рис. 5).
2. Иначе создаём новый элемент. Указатель на первый элемент присваиваем значение нового. Указатель на следующий элемент нового звена присваивает значение старого первого звена.

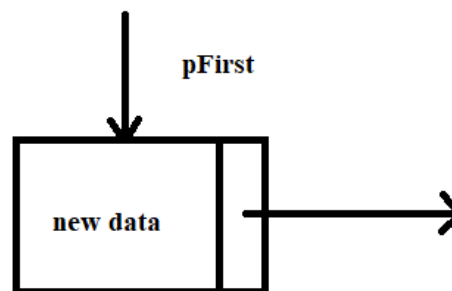


Рис. 5. Добавление в начало, если список пустой

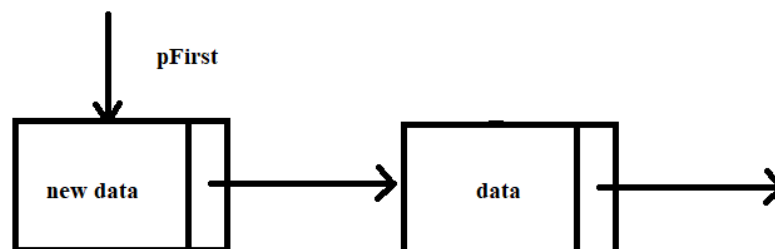


Рис. 6. Добавление в начало, если список не пустой

##### Операция добавления в конец:

Операция добавления элемента реализуется при помощи указателя на последний элемент.

1. Если структура хранения пуста, то создаем новый элемент с помощью вставки в начало (рис. 5).
2. Если список не пуст, то создаем новый элемент. Меняем указатель последнего элемента на следующий равный новому.
3. Меняем указатель на последний элемент равный новому (рис. 7).

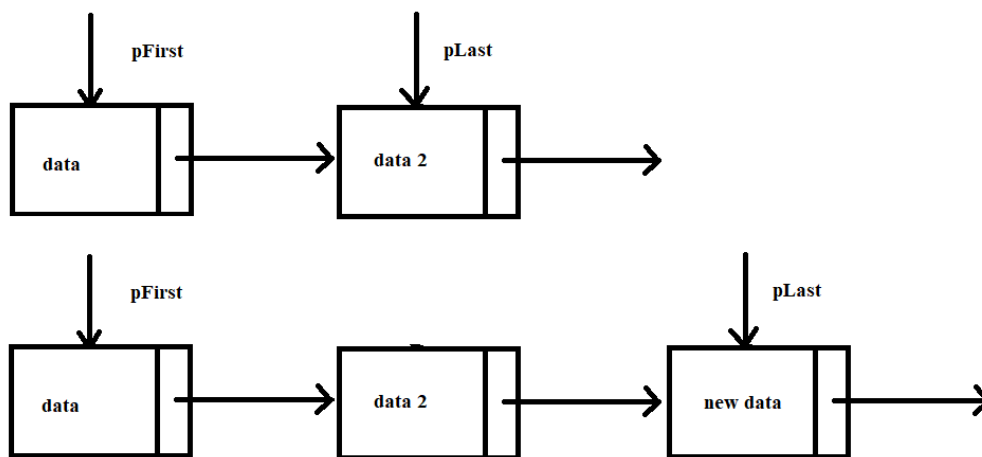


Рис. 7. Операция добавления элемента в конец

### Операция поиска:

Операция поиска ищет элемент в списке по значению.

1. Проходимся от начала списка, сравнивая значения звеньев с указанными пользователем.
2. Если значения не равны, идем к следующему звену, меняя указатель на текущий элемент.
3. Найдя нужное звено, возвращаем указатель на найденный элемент (рис. 8).

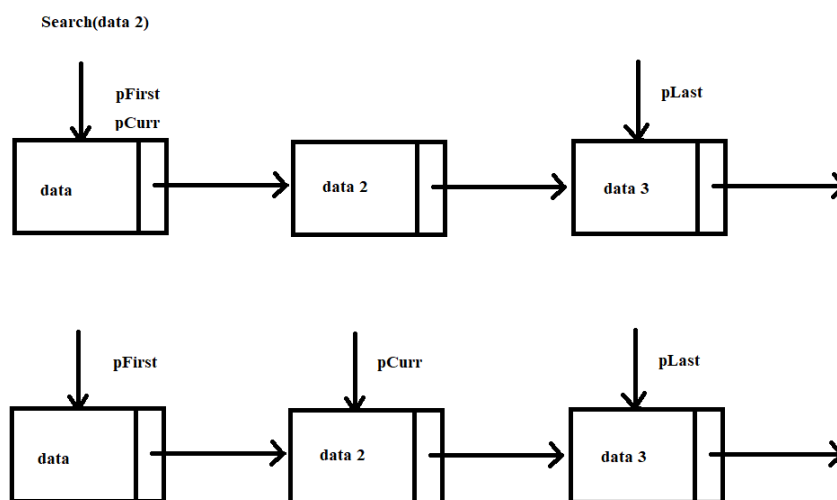


Рис. 8. Операция поиска элемента (3) в списке

### Операция добавления после:



Операция добавления элемента реализуется при помощи указателя на текущий элемент.

Находим элемент, после которого хотим вставить, создаем новое звено и сдвигаем указатели (рис. 9):

1. Находим элемент, после которого хотим выполнить операцию вставки с помощью операции поиска (рис. 8).
2. Указатель нового звена на следующее равен указателю текущего.
3. Указатель текущего звена на следующее равен новому.

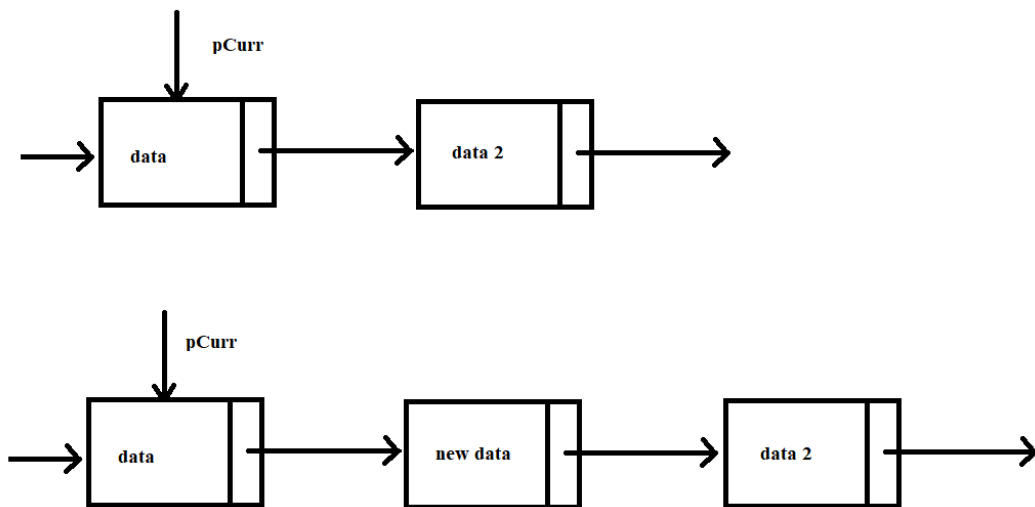


Рис. 9. Операция добавления элемента после текущего.

#### **Операция добавления перед:**

Операция добавления элемента реализуется при помощи указателя на текущий элемент.

Находим элемент, перед которым хотим вставить, создаем новое звено и сдвигаем указатели (рис. 10):

1. Находим элемент, перед которым хотим выполнить операцию вставки с помощью операции поиска (рис. 8).
2. Указатель на следующее звено предыдущего элемента равен новому элементу.
3. Указатель нового элемента на следующее звено равен текущему.

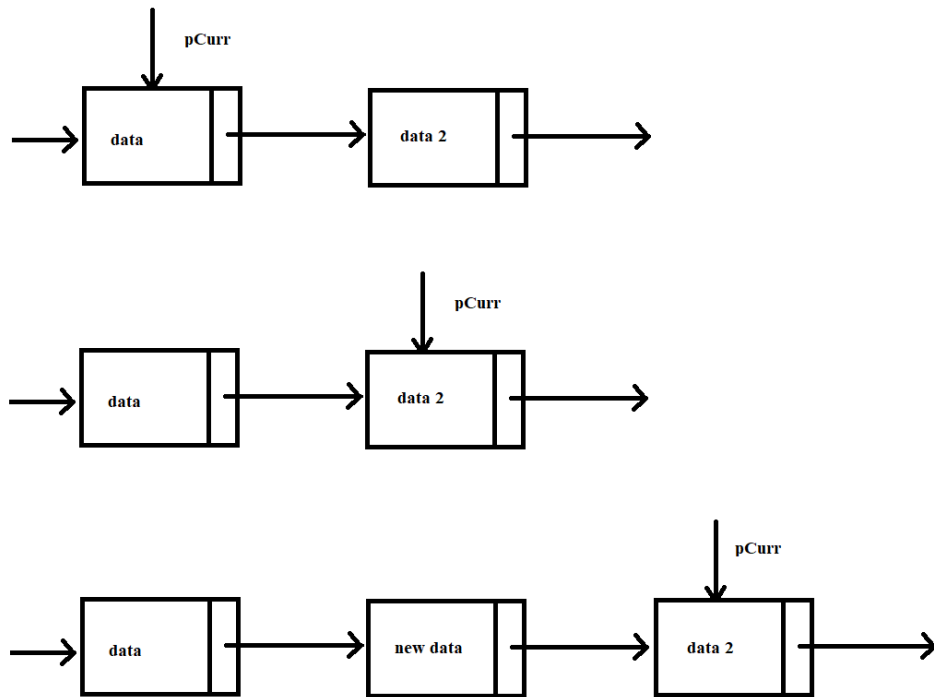


Рис. 10. Операция вставки нового звена перед текущим.

#### Операция удаления первого элемента:

Операция удаления элемента реализуется при помощи указателя на первый элемент.

1. Удаляем первый элемент.
2. Указатель на первое звено переопределяется на следующее звено (рис. 11).

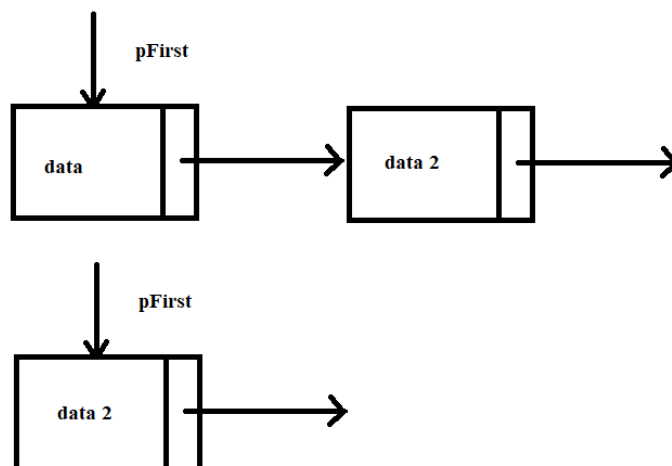


Рис. 11. Операция удаления первого элемента

#### Операция удаления текущего элемента:

Операция удаления элемента реализуется при помощи указателя на текущий элемент (рис. 12):

1. Находим элемент, который хотим удалить с помощью операции поиска (рис. 8).
2. Указателю предыдущего звена на следующее присваивается значение указателя на следующее звено текущего.
3. Освобождается выделенная память под текущее звено.

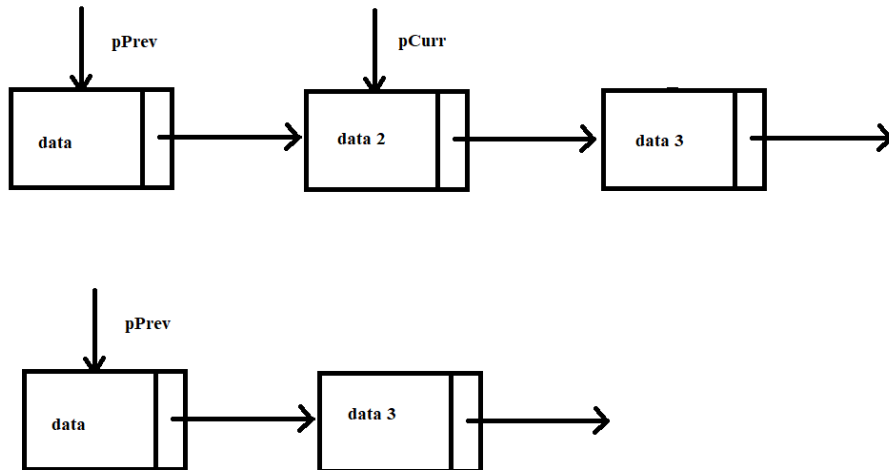


Рис. 12. Операция удаления текущего элемента

#### Операция удаления элемента по его данным:

Операция удаления схожа с удалением текущего элемента:

1. Выполняется поиск элемента по его значению (рис. 8).
2. Если элемент найден, выполняется операция удаления текущего элемента (рис. 12).
3. Иначе ничего не происходит.

#### Операция получения текущего элемента:

Операция получения текущего элемента реализуется при помощи указателя на текущий элемент. Возвращает указатель на текущий.

#### Операция проверки на пустоту:

Операция проверки на полноту проверяет, есть ли хотя бы один элемент в списке.

1. Если первый элемент существует, возвращает true.
2. Иначе false.

#### Операция сортировки:

Операция сортировки позволяет сортировать список, используется оптимизированная сортировка пузырьком (рис. 13):

1. Начиная с первого элемента, текущий сравнивается с последующими элементами, до последнего.
2. Если найден элемент, значение которого меньше текущего, они меняются местами.
3. Алгоритм повторяется до тех пор, пока текущий элемент не будет равен последнему.

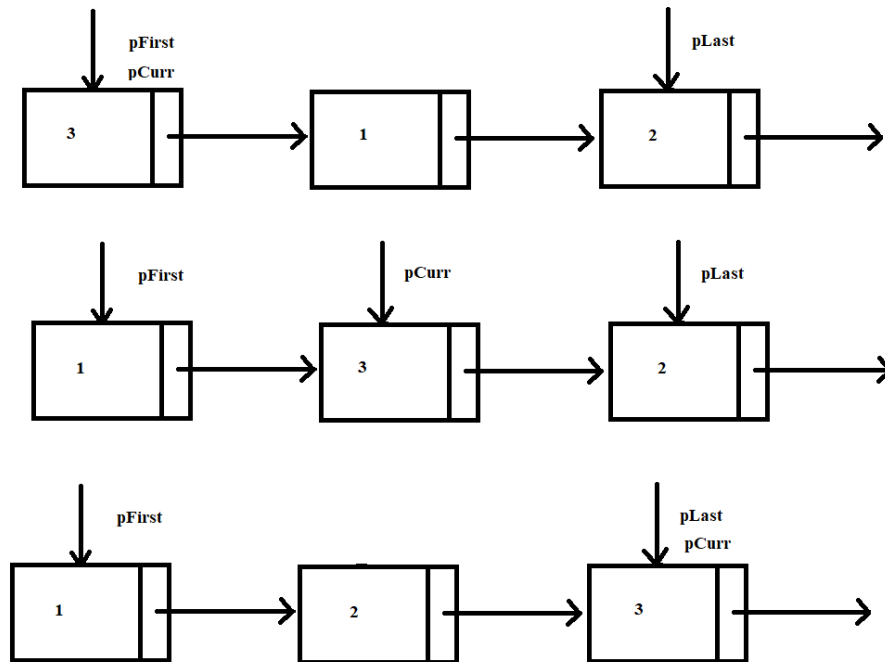


Рис. 13. Сортировка

### 3.1.2 Кольцевой односвязный список

Кольцевой список – линейный односвязный список, последний элемент которого всегда указывает на фиктивную голову, которая в свою очередь указывает на первое звено.

Все операции для линейного односвязного списка применимы к кольцевому, за исключением вставки в начало и удаления первого элемента, так как они меняют указатель на следующий элемент для фиктивной головы.

#### Операция добавления в начало:

Добавляет новое звено в начало списка.

1. Указатель на следующее звено нового элемента присваивает значение первого.
2. Указатель на следующее звено фиктивной головы присваивает значение нового элемента.
3. Переопределяется указатель на первый элемент.

4. Если текущий список не содержит элементов, указатель на следующее звено фиктивной головы присваивает значение нового элемента. Указатель на следующее звено нового элемента присваивает значение головы (рис. 14).

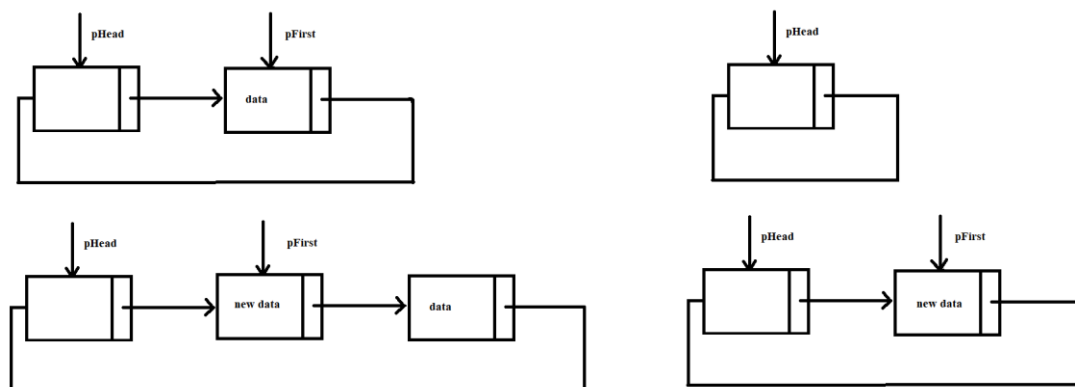


Рис. 14. Операция добавления элемента в начало

### Операция удаления первого элемента:

Операция удаляет первый элемент в списке.

1. Первому звену присваивается значение указателя на следующий элемент.
2. Указатель на следующий элемент фиктивной головы присваивает значение нового первого элемента.
3. Освобождается выделенная память под старое звено.
4. Если список содержал один элемент, голова замыкается (рис. 15).
5. Если список пустой, ничего не происходит.

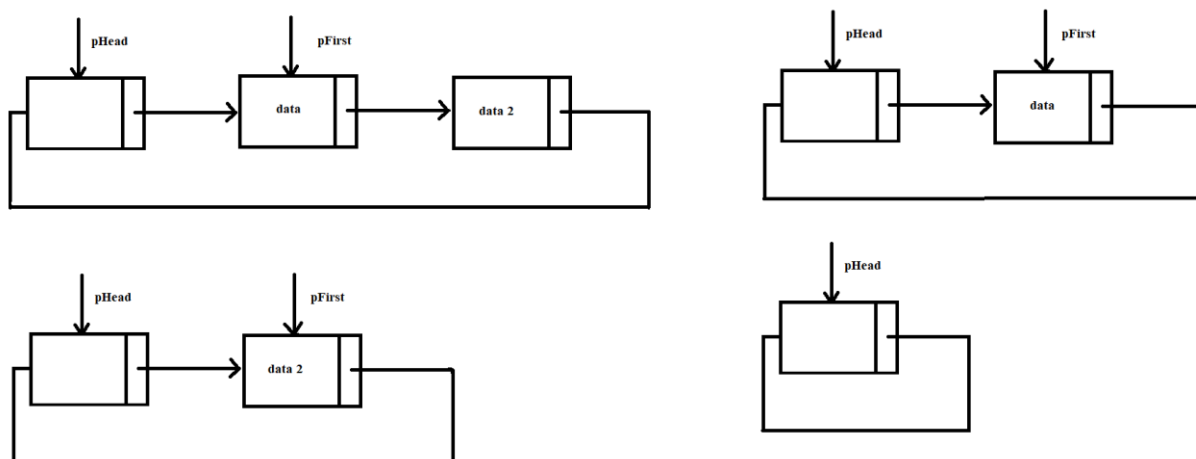


Рис. 15. Удаление первого элемента.

### 3.1.3 Моном

Моном – структура данных, содержащая два поля: коэффициент и степень. Степень представляет собой число от 0 до 999, каждый разряд которого является степенью для X,

Y и Z. Структура данных поддерживает операции сравнения, сложения, вычитания, умножения, дифференцирования и вычисления.

#### **Операция сравнения на равенство:**

Операция сравнивает два монома на равенство.

1. Функция сравнивает степень.
2. Если степени не совпадают, возвращает false, иначе сравнивает коэффициент.

Пример:  $2*x*z^3 = 2*x*z^2$

Функция вернет false, так как степень слева равна 103, а справа 102.

#### **Операция сравнения на неравенство:**

Возвращает обратное значение от сравнения на равенство.

Пример:  $2*x*z^3 = 2*x*z^2$

Функция вернет true, так как степень слева равна 103, а справа 102.

#### **Операция сравнения больше:**

Операция возвращает результат сравнения.

1. Функция сравнивает степени мономов. Если они не совпадают, возвращает результат сравнения.
2. Если степени совпадают сравнивает коэффициенты.

Пример:  $3*x*z^3 > 2*x*z^3$

Функция вернет true, так как степени совпадают, но коэффициент монома слева больше.

#### **Операция сравнение меньше:**

Операция возвращает результат сравнения.

1. Функция сравнивает степени мономов. Если они не совпадают, возвращает результат сравнения.
2. Если степени совпадают сравнивает коэффициенты.

Пример:  $3*x*z^3 > 2*x*z^3$

Функция вернет false, так как степени совпадают, но коэффициент монома слева больше.

#### **Операция сложения:**

Функция складывает два монома.

1. Сравнивает коэффициенты.

2. Если они равны складывает коэффициенты.

Пример:  $3*x*z^3 + 2*x*z^3$

Функция вернет  $5*x*z^3$ , так как степени мономов совпадают.

#### **Операция вычитания:**

Функция считает разность двух мономов.

1. Сравнивает коэффициенты.
2. Если они равны вычитает коэффициенты.

Пример:  $3*x*z^3 - 2*x*z^2$

Функция ничего не вернет, так как степени мономов не совпадают.

#### **Операция умножения:**

Функция считает произведение мономов.

1. Коэффициенты перемножаются.
2. Степени складываются.
3. Если степень любого разряда становится больше 9, возвращает 0.

Пример:  $3*x*z^3 * 2*x*z^2$

Функция вернет  $6*x^2*z^5$ .

#### **Операция дифференцирования:**

Функция вычисляет полный дифференциал от монома.

1. Коэффициент умножается на каждый разряд степени.
2. Каждый разряд степени уменьшается на единицу.

Пример:  $3*x^5*y*z^3$

Функция вернет  $45*x^4*z^2$ .

#### **Операция вычисления:**

Функция вычисляет значение монома от заданных пользователем параметров.

Пример:  $3*x^5*y*z^3$ ,  $x = 2$ ,  $y = 1$ ,  $z = 3$

Функция вернет 2592.

### **3.1.4 Полином**

Полином – структура данных, поля которой: кольцевой односвязный список мономов, имя. Имя – строка полинома. Данная структура данных поддерживает такие

операции как: сложение, вычитание, произведение, умножение на константу, полный дифференциал, вычисление.

#### **Операция сложения:**

Данная функция складывает два полинома.

1. Проходясь по первому полиному, выполняет вставку во второй полином.
2. Если во втором полиноме есть моном с такой же степенью, складывает их коэффициенты.

Пример:  $(3*x^5*y*z^3+2*x*z^2) + (6*x^5*y*z^3 - x*y*z)$

Функция вернет  $9*x^5*y*z^3+2*x*z^2-x*y*z$

#### **Операция умножения на константу:**

Данная функция умножает коэффициенты всех мономов на заданную константу.

Пример:  $(3*x^5*y*z^3+2*x*z^2) * (2)$

Функция вернет  $6*x^5*y*z^3+4*x*z^2$ .

#### **Операция вычитания:**

Данная функция вызывает операцию сложения, умножая операнд на константу равную -1.

Пример:  $(3*x^5*y*z^3+2*x*z^2) - (6*x^5*y*z^3 - x*y*z)$

Функция вернет  $-3*x^5*y*z^3+2*x*z^2+x*y*z$

#### **Операция произведения:**

Данная функция перемножает все мономы обоих полиномов.

Пример:  $(3*x^5*y*z^3+2*x*z^2) * (6*x^4*y*z^3 - x*y*z)$

Функция вернет  $18*x^9*y^2*z^6 - 3*x^6*y^2*z^4 + 12*x^5*y*z^5 - 2*x^2*y*z^3$

#### **Операция дифференцирования:**

Данная функция вычисляет полный дифференциал всех мономов.

1. Коэффициент умножается на каждый разряд.
2. Каждый разряд степени уменьшается на единицу.

Пример:  $3*x^5*y*z^3+2*x*z^2$

Функция вернет  $45*x^4*z^2$

#### **Операция вычисления:**



Данная функция вычисляет все мономы по заданным значениям параметров.

Пример:  $3*x^5*y*z^3+2*x*z^2$ ,  $x = 2$ ,  $y = 1$ ,  $z = 3$

Функция вернет 2628.

## 3.2 Описание программной реализации

### 3.2.1 Описание структуры TNode

```
struct TNode
{
    T data;
    TNode<T>* pNext;
    TNode()
    TNode(const T& dat);

    bool operator<(const TNode<T>& nd) const;
    bool operator>(const TNode<T>& nd) const;
    bool operator==(const TNode<T>& nd) const;
    bool operator!=(const TNode<T>& nd) const;
};
```

Назначение: представление звена.

Поля:

**data** — данные ячейки.

**pNext** — указатель на следующее звено.

Методы:

**TNode()** ;

Назначение: конструктор по умолчанию.

**TNode(const T& dat)** ;

Назначение: конструктор с параметром.

Входные параметры:

**dat** — данные.

**bool operator<(const TNode<T>& nd) const;**

Назначение: перегрузка оператора меньше.

Входные параметры:

**nd** - константный указатель на звено.

Выходные параметры:

Результат сравнения.

```
bool operator>(const TNode<T>& nd) const;
```

Назначение: перегрузка оператора больше.

Входные параметры:

`nd` - константный указатель на звено.

Выходные параметры:

Результат сравнения.

```
bool operator==(const TNode<T>& nd) const;
```

Назначение: перегрузка оператора сравнения на равенство.

Входные параметры:

`nd` - константный указатель на звено.

Выходные параметры:

Результат сравнения.

```
bool operator!=(const TNode<T>& nd) const;
```

Назначение: перегрузка оператора сравнения на неравенство.

Входные параметры:

`nd` - константный указатель на звено.

Выходные параметры:

Результат сравнения.

### 3.2.2 Описание класса TList

```
class TList
{
protected:
    TNode<T>* pFirst;
    TNode<T>* pLast;
    TNode<T>* pCurr;
    TNode<T>* pPrev;
    TNode<T>* pStop;
public:
    TList();
    TList(const TNode<T>* pFirst);
    TList(const TList<T>& TList);

    TNode<T>* search(const T& data);
    TNode<T>* search(const TNode<T>* node);

    virtual ~TList();
    virtual void clear();

    void Next();
    void Reset();
};
```

```

bool IsEmpty() const;
bool IsFull() const;
bool IsEnd() const;

void Sort();

TNode<T>* getpF() const;
TNode<T>* getpS() const;
TNode<T>* getpC() const;

T get_pFirst() const;
T get_pLast() const;
T get_pCurr() const;
T get_pPrev() const;

virtual void InsertFirst(const T& data);
virtual void InsertLast(const T& data);
virtual void InsertBefore(const T& data, const TNode<T>* before_node);
virtual void InsertAfter(const T& data, const TNode<T>* after_node);
virtual void InsertBeforeCurrent(const T& data);
virtual void InsertAfterCurrent(const T& data);

virtual void DeleteFirst();
virtual void DeleteLast();
virtual void DeleteBefore(const TNode<T>* before_node);
virtual void DeleteAfter(const TNode<T>* after_node);
virtual void DeleteData(const T& data);
virtual void DeleteCurrent();

virtual const TList<T>& operator=(const TList<T>& pList);

friend std::ostream& operator<<(std::ostream& out, TList<T>& list);
friend std::istream& operator>>(std::istream& in, TList<T>& list);
};

```

Назначение: предоставление структуры данных линейный односвязный список.

Поля:

**pFirst** – указатель на первое звено.

**pLast** – указатель на последнее звено.

**pCurr** – указатель на текущее звено.

**pPrev** – указатель на предыдущее звено.

**pStop** – указатель на границу списка.

Методы:

**TList();**

Назначение: конструктор по умолчанию.

**TList(const TNode<T>\* pFirst);**

Назначение: конструктор с параметром.

Входные параметры:

**pFirst** – указатель на звено.

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

Назначение: конструктор копирования.

Входные параметры:

**TList** – ссылка на список.

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

Назначение: поиск звена.

Входные параметры:

**data** – ссылка на данные.

Выходные параметры:

Указатель на звено.

**TNode<T>\* search(const TNode<T>\* node);**

Назначение: поиск звена.

Входные параметры:

**node** – указатель на звено.

Выходные параметры:

Указатель на звено.

**virtual ~TList();**

Назначение: деструктор.

**virtual void clear();**

Назначение: удаление всех элементов списка.

**void Next();**

Назначение: переход к следующему элементу.

**void Reset();**

Назначение: присвоение текущему звену указатель на первое.

**bool IsEmpty() const;**

Назначение: проверка на пустоту.

Выходные параметры:

Результат проверки.

**bool IsFull() const;**

Назначение: проверка на полноту.

Выходные параметры:

Результат проверки.

**bool IsEnd() const;**

Назначение: проверка находится ли текущее звено конец списка.

Выходные параметры:

Результат проверки.

**void Sort() ;**

Назначение: сортировка списка.

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

Назначение: получение указателя на первое звено

Выходные параметры:

Указатель на первое звено.

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

Назначение: получение указателя на границу списка.

Выходные параметры:

Указатель на границу списка.

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

Назначение: получение указателя на текущее звено.

Выходные параметры:

Указатель на текущее звено.

**T get\_pFirst() const;**

Назначение: получение данных первого звена.

Выходные параметры:

Данные первого звена.

**T get\_pLast() const;**

Назначение: получение данных последнего звена.

Выходные параметры:

Получение данных последнего звена.

**T get\_pCurr() const;**

Назначение: получение данных текущего звена.

Выходные параметры:

Данные текущего звена.

**T get\_pPrev() const;**

Назначение: получение данных предыдущего звена.

Выходные параметры:

Данные предыдущего звена.

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

Назначение: вставка в начало.

Входные параметры:

**data** — данные для нового звена.

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

Назначение: вставка в конец

**data** – данные для нового звена.

**void InsertBefore(const T& data, const TNode<T>\* before\_node);**

Назначение: вставка до.

**data** – данные для нового звена.

**before\_node** – указатель на звено.

**void InsertAfter(const T& data, const TNode<T>\* after\_node);**

Назначение: вставка после

Входные параметры:

**data** – данные для нового звена.

**after\_node** – указатель на звено.

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

Назначение: вставка до текущего

**data** – данные для нового звена.

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

Назначение: вставка после текущего.

**data** – данные для нового звена.

**virtual void DeleteFirst();**

Назначение: удаление первого звена

**void DeleteLast();**

Назначение: удаление последнего звена.

**void DeleteBefore(const TNode<T>\* before\_node);**

Назначение: удаление звена до.

Входные параметры:

**before\_node** – указатель на звено.

**void DeleteAfter(const TNode<T>\* after\_node);**

Назначение: удаление звена после.

Входные параметры:

**after\_node** – указатель на звено.

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

Назначение: удаление по данным.

Входные параметры:

**data** – данные удаляемого звена.

```
void DeleteCurrent();
```

Назначение: удаление текущего звена.

```
virtual const TList<T>& operator=(const TList<T>& pList);
```

Назначение: перегрузка оператора присваивания.

Входные параметры:

**pList** – указатель на список.

Выходные параметры:

Ссылка на список.

```
friend std::ostream& operator<<(std::ostream& out, TList<T>& list);
```

Назначение: перегрузка оператора потокового вывода.

Входные параметры:

**out** – поток вывода.

**list** – ссылка на список.

Выходные параметры:

Ссылка на поток.

### 3.2.3 Описание класса TRingList

```
class TRingList : public TList<T>
{
private:
    TNode<T>* pHead;
public:
    TRingList();
    TRingList(const T& data);
    TRingList(const TNode<T>* pF);
    TRingList(const TRingList<T>& ringList);

    ~TRingList();
    void clear();

    void InsertFirst(const T& data);
    void DeleteFirst();
    const TRingList<T>& operator=(const TRingList<T>& pRingList);
};
```

Назначение: предоставление структуры данных кольцевой односвязный список.

Поля:

**pHead** – указатель на фиктивную голову.

Методы:

```
TRingList();
```

Назначение: конструктор по умолчанию.

**TRingList(const T& data);**

Назначение: конструктор с параметром.

Входные параметры:

**data** — данные для нового звена.

**TRingList(const TNode<T>\* pF);**

Назначение: конструктор с параметром

Входные параметры:

**pF** — указатель на звено.

**TRingList(const TRingList<T>& ringList);**

Назначение: конструктор копирования.

Входные параметры:

**ringList** — ссылка на список.

**~TRingList();**

Назначение: деструктор.

**void clear();**

Назначение: удаление всех элементов.

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

Назначение: вставка в начало.

Входные параметры:

**data** — данные для нового звена.

**void DeleteFirst();**

Назначение: удаление первого элемента.

**const TRingList<T>& operator=(const TRingList<T>& pRingList);**

Назначение: перегрузка оператора присваивания.

Входные параметры:

**pRingList** — ссылка на список.

Выходные параметры:

Ссылка на список.

### 3.2.4 Описание класса TMonom

```
class TMonom {  
private:  
    double coeff;  
    int degree;  
public:
```



```

TMonom();
TMonom(const TMonom& monom);
TMonom(double coeff, int degree);
TMonom(const string& monom);

void SetCoeff(const double cf);
void SetDegree(const int dgr);
double GetCoeff()const { return coeff; }
int GetDegree()const { return degree; }

bool operator<(const TMonom& data) const;
bool operator>(const TMonom& data) const;
bool operator==(const TMonom& data) const;
bool operator!=(const TMonom& data) const;

TMonom operator+(const TMonom& data);
TMonom operator-(const TMonom& data);
TMonom operator*(const TMonom& data);

const TMonom& operator=(const TMonom& monom);

TMonom def_X()const;
TMonom def_Y()const;
TMonom def_Z()const;

double operator()(double x, double y, double z)const;

friend std::ostream& operator<<(std::ostream& out, TMonom& monom);

friend std::istream& operator>>(std::istream in, TMonom& monom);
};

```

Назначение: предоставление структуры данных моном.

Поля:

**coeff** – коэффициент.

**degree** – степень переменных.

Методы:

**TMonom()** ;

Назначение: конструктор по умолчанию.

**TMonom(const TMonom& monom)** ;

Назначение: конструктор копирования.

Входные параметры:

**monom** – ссылка на моном.

**TMonom(double coeff, int degree)** ;

Назначение: конструктор с параметром.

Входные параметры:

**coeff** – коэффициент.

**degree** – степень переменных.

**TMonom(const string& monom);**

Назначение: конструктор с параметром.

Входные параметры:

**Monom** – строка монома.

**void SetCoeff(const double cf);**

Назначение: изменение коэффициента

Входные параметры:

**cf** – коэффициент.

**void SetDegree(const int dgr);**

Назначение: изменение степени.

Входные параметры:

**degree** – степень переменных.

**double GetCoeff() const**

Назначение: получение коэффициента

Выходные параметры:

Коэффициент.

**int GetDegree() const**

Назначение: получение степени.

Выходные параметры:

Степень.

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

Назначение: перегрузка оператора меньше.

Входные параметры:

**data** – моном.

Выходные параметры:

Результат сравнения.

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

Назначение: перегрузка оператора больше.

Входные параметры:

**data** – моном.

Выходные параметры:

Результат сравнения.

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

Назначение: перегрузка оператора сравнения на равенство.

Входные параметры:

**data** – моном.

Выходные параметры:

Результат сравнения

```
bool operator!=(const TMonom& data) const;
```

Назначение: перегрузка оператора сравнения на неравенство.

Входные параметры:

**data** – моном.

Выходные параметры:

Результат сравнения

```
TMonom operator+(const TMonom& data);
```

Назначение: перегрузка оператора сложения

Входные параметры:

**data** – моном.

Выходные параметры:

Моном.

```
TMonom operator-(const TMonom& data);
```

Назначение: перегрузка оператора вычитания.

Входные параметры:

**data** – моном.

Выходные параметры:

Результат вычитания.

```
TMonom operator*(const TMonom& data);
```

Назначение: перегрузка оператора умножения.

Входные параметры:

**data** – моном.

Выходные параметры:

Моном.

```
const TMonom& operator=(const TMonom& monom);
```

Назначение: перегрузка оператора присваивания.

Входные параметры:

**data** – моном.

Выходные параметры:

Моном.

**TMonom def\_X() const;**

Назначение: дифференцирование по икс.

Выходные параметры:

Моном.

**TMonom def\_Y() const;**

Назначение: дифференцирование по игрек.

Выходные параметры:

Моном.

**TMonom def\_Z() const;**

Назначение: дифференцирование по зет.

Выходные параметры:

Моном.

**double operator()(double x, double y, double z) const;**

Назначение: вычисление монома.

Входные параметры:

**x** – значение переменной x.

**y** – значение переменной y.

**z** – значение переменной z.

Выходные параметры:

Результат вычисления.

**friend std::ostream& operator<<(std::ostream& out, TMonom& monom);**

Назначение: перегрузка оператора потокового вывода.

Входные параметры:

**out** – поток вывода.

**monom** – ссылка на моном.

Выходные параметры:

Ссылка на поток.

**friend std::istream& operator>>(std::istream in, TMonom& monom);**

Назначение: перегрузка оператора потокового ввода.

Входные параметры:

**in** – поток ввода.

**monom** – ссылка на моном.

Выходные параметры:

Ссылка на поток.

### 3.2.5 Описание класса TPolynom

```
class TPolynom
{
private:
    TRingList<TMonom> monoms;
    string name;
    void update();
    void updatename();
    void updatenull();
public:
    TPolynom();
    TPolynom(const string& name);
    TPolynom(const TRingList<TMonom> monoms);
    TPolynom(const TPolynom& p);
    TPolynom operator+(const TPolynom& polynom);
    TPolynom operator-(const TPolynom& polynom);
    TPolynom operator*(const TPolynom& polynom);
    TPolynom operator*(const double c);
    const TPolynom& operator=(const TPolynom& polynom);
    TPolynom dif() const;
    TPolynom dif_x() const;
    TPolynom dif_y() const;
    TPolynom dif_z() const;
    string get_name() const;
    void InsertSort(const TMonom& m);
    double operator()(double x, double y, double z) const;
    friend ostream& operator<< (ostream& out, const TPolynom& polynom);
    friend istream& operator>> (istream& in, TPolynom& polynom);
};
```

Назначение: предоставление структуры данных полином.

Поля:

**monoms** – кольцевой список мономов.

**name** – строка полинома.

Методы:

**TPolynom();**

Назначение: конструктор по умолчанию.

**TPolynom(const string& name);**

Назначение: конструктор с параметром.

Входные параметры:

**name** – строка полинома.

**TPolynom(const TRingList<TMonom> monoms);**

Назначение: конструктор с параметром.

Входные параметры:

**monoms** – кольцевой список мономов.

**TPolynom(const TPolynom& p);**

Назначение: конструктор копирования.

Входные параметры:

**p** – полином.

**TPolynom operator+(const TPolynom& polynom);**

Назначение: перегрузка оператора сложения.

Входные параметры:

**polynom** – полином.

Выходные параметры:

Полином.

**TPolynom operator-(const TPolynom& polynom);**

Назначение: перегрузка оператора вычитания.

Входные параметры:

**polynom** – полином.

Выходные параметры:

Полином.

**TPolynom operator\*(const TPolynom& polynom);**

Назначение: перегрузка оператора произведения полиномов.

Входные параметры:

**polynom** – полином.

Выходные параметры:

Полином.

**TPolynom operator\*(const double c);**

Назначение: перегрузка оператора умножения на константу.

Входные параметры:

**c** – константа.

Выходные параметры:

Полином.

**const TPolynom& operator=(const TPolynom& polynom);**

Назначение: перегрузка оператора присваивания.

Входные параметры:

**polynom** – полином.

Выходные параметры:

Полином.

**TPolynom dif() const;**

Назначение: полный дифференциал.

Выходные параметры:

Полином.

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

Назначение: дифференцирование по *икс*.

Выходные параметры:

Полином.

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

Назначение: дифференцирование по *игрек*.

Выходные параметры:

Полином.

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

Назначение: дифференцирование по *зет*.

Выходные параметры:

Полином.

**string get\_name() const;**

Назначение: получение строки полинома.

Выходные параметры:

Строка полинома.

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

Назначение: вставка монома в отсортированный полином.

Входные параметры:

*m* – моном.

**double operator()(double x, double y, double z) const;**

Назначение: вычисление полинома.

Входные параметры:

*x* – значение переменной *x*.

*y* – значение переменной *y*.

*z* – значение переменной *z*.

Выходные параметры:

Результат вычисления.

**friend ostream& operator<< (ostream& out, const TPolynom& polynom);**

Назначение: перегрузка оператора потокового вывода.

Входные параметры:

`out` – поток вывода.

`polynom` – ссылка на полином.

Выходные параметры:

Ссылка на поток.

```
friend istream& operator>> (istream& in, TPolynom& polynom);
```

Назначение: перегрузка потокового ввода.

Входные параметры:

`in` – поток ввода.

`polynom` – ссылка на полином.

Выходные параметры:

Ссылка на поток.



## **Заключение**

В ходе выполнения работы "Полином" были изучены и практически применены концепции кольцевого односвязного списка и предоставление структуры данных полином.

Были достигнуты следующие результаты:

1. Были изучены теоретические основы линейного односвязного списка и алгоритмы с полиномом.
2. Была разработана программа, реализующая необходимые операции. В ходе экспериментов была оценена эффективность работы этих операций и сравнена с другими подходами.
3. Были проанализированы полученные результаты и сделаны выводы о преимуществах и ограничениях использования кольцевого односвязного списка. Оказалось, что эта структура данных особенно полезна при работе с большими объемами данных, где компактность представления и эффективность операций являются ключевыми факторами.

## **Литература**

1. Сысоев А.В., Алгоритмы и структуры данных, лекция 07, 17 октября.

# Приложения

## Приложение А. Реализация класса TNode

```
template <typename T>
bool TNode<T>::operator<(const TNode<T>& nd) const
{
    return this->data < nd.data;
}
template <typename T>
bool TNode<T>::operator>(const TNode<T>& nd) const
{
    return this->data > nd.data;
}
template <typename T>
bool TNode<T>::operator==(const TNode<T>& nd) const
{
    return this->data == nd.data;
}
template <typename T>
bool TNode<T>::operator!=(const TNode<T>& nd) const{
    return this->data != nd.data;
}
```

## Приложение В. Реализация класса TList

```
template <typename T>
TList<T>::TList()
{
    pFirst = nullptr;
    pLast = nullptr;
    pCurr = nullptr;
    pPrev = nullptr;
    pStop = nullptr;
}
template <typename T>
TList<T>::TList(const TNode<T>* pFirst)
{
    if (pFirst == nullptr)
    {
        pFirst = nullptr;
        pLast = nullptr;
        pCurr = nullptr;
        pPrev = nullptr;
        pStop = nullptr;
        return;
    }
    TNode<T>* tmp = pFirst->pNext;
    this->pFirst = new TNode<T>(pFirst->data);
    this->pLast = this->pFirst;
    pCurr = this->pFirst;
    while (tmp != nullptr) {
        pLast->pNext = new TNode<T>(tmp->data);
        pLast = pLast->pNext;
        tmp = tmp->pNext;
    }
    pStop = nullptr;
    pPrev = pStop;
}

template <typename T>
```

```

TList<T>::TList(const TList<T>& TList)
{
    if (TList.pFirst == nullptr)
    {
        pFirst = nullptr;
        pLast = nullptr;
        pCurr = nullptr;
        pPrev = nullptr;
        pStop = nullptr;
        return;
    }
    pFirst = new TNode<T>(TList.pFirst->data);
    TNode<T>* tmp1 = TList.pFirst;
    TNode<T>* tmp2 = pFirst;
    pCurr = tmp2;
    while (tmp1 ->pNext != TList.pStop) {
        tmp1 = tmp1->pNext;
        tmp2->pNext = new TNode<T>(tmp1->data);
        pPrev = tmp2;
        tmp2 = tmp2->pNext;
    }
    pStop = nullptr;
    pLast = tmp2;
    pLast->pNext = pStop;
}

template <typename T>
TList<T>::~~TList()
{
    clear();
}

template <typename T>
void TList<T>::clear()
{
    while (!IsEmpty()) {
        TNode<T>* tmp = pFirst;
        pFirst = pFirst->pNext;
        delete tmp;
    }
    pCurr = nullptr;
    pPrev = nullptr;
    pLast = nullptr;
}

template <typename T>
void TList<T>::Next()
{
    if (pCurr == pStop)
        return;
    pPrev = pCurr;
    pCurr = pCurr->pNext;
}

template <typename T>
bool TList<T>::IsEmpty() const
{
    return pFirst == pStop;
}

template <typename T>
bool TList<T>::IsFull() const
{

```

```

        TNode<T>* tmp = new (std::nothrow) TNode<T>();
        if (tmp)
            return false;
        else
            return true;
    }

template <typename T>
bool TList<T>::IsEnd() const
{
    if (pCurr == pStop)
        return true;
    return false;
}

template <typename T>
TNode<T>* TList<T>::search(const T& data)
{
    Reset();
    while (pCurr->pNext != pStop && pCurr->data != data)
    {
        Next();
    }
    if (pCurr->pNext == pStop && pCurr->data != data)
        throw std::exception("data not found(Search)");
    return pCurr;
}

template <typename T>
TNode<T>* TList<T>::search(const TNode<T>* node)
{
    Reset();
    while (pCurr->pNext != pStop && pCurr->data != node->data)
    {
        Next();
    }
    if (pCurr->pNext == pStop && pCurr->data != node->data)
        throw std::exception("data not found(Search)");
    return pCurr;
}

template <typename T>
void TList<T>::InsertFirst(const T& data)
{
    if (IsFull())
        throw std::exception("out of memory(InsF)");
    if (IsEmpty())
    {
        TNode<T>* node = new TNode<T>(data);
        pFirst = node;
        pLast = pFirst;
        pCurr = pFirst;
        pLast->pNext = pStop;
    }
    else
    {
        TNode<T>* node = new TNode<T>(data);
        TNode<T>* tmp = pFirst;
        pFirst = node;
        pFirst->pNext = tmp;
        Reset();
    }
}

```

```

    }
}

template <typename T>
void TList<T>::InsertLast(const T& data)
{
    if (IsFull())
        throw std::exception("out of memory(InsL)");
    if (IsEmpty())
    {
        InsertFirst(data);
        return;
    }
    pLast->pNext = new TNode<T>(data);
    pLast = pLast->pNext;
    pLast->pNext = pStop;
}

template <typename T>
void TList<T>::InsertBefore(const T& data, const TNode<T>* before_node)
{
    if (IsFull())
        throw std::exception("out of memor(InsB)");
    TNode<T>* new_node = new TNode<T>(data);
    pCurr = pFirst;
    search(before_node);
    if ((pCurr->pNext == pStop) && (pCurr != before_node))
        throw std::exception("node not found(InsB)");
    InsertBeforeCurrent(data);
    Reset();
}

template <typename T>
void TList<T>::InsertAfter(const T& data, const TNode<T>* after_node)
{
    if (IsFull())
        throw std::exception("out of memory(InsA)");
    Reset();
    search(after_node);
    InsertAfterCurrent(data);
    Reset();
}

template <typename T>
void TList<T>::InsertBeforeCurrent(const T& data)
{
    if (IsEmpty()) {
        InsertFirst(data);
        return;
    }
    if (IsEnd()) {
        InsertLast(data);
        return;
    }
    if (IsFull()) {
        throw std::exception("out of memory(InsBC)");
    }
    TNode<T>* new_node = new TNode<T>(data);
    pPrev->pNext = new_node;
    pPrev = pPrev->pNext;
    pPrev->pNext = pCurr;
    pCurr = pFirst;
    pPrev = pStop;
}

```

```

}

template <typename T>
void TList<T>::InsertAfterCurrent(const T& data)
{
    if (IsEmpty()) {
        InsertFirst(data);
        return;
    }
    if (IsEnd()) {
        InsertLast(data);
        return;
    }
    TNode<T>* new_node = new TNode<T>(data);
    if (new_node == NULL) {
        throw std::exception("out of memory(InsAC)");
    }
    Next();
    pPrev->pNext = new_node;
    pPrev = pPrev->pNext;
    pPrev->pNext = pCurr;
    pCurr = pFirst;
    pPrev = pStop;
}

template <typename T>
void TList<T>::DeleteFirst()
{
    if (IsEmpty())
        throw std::exception("empty list(DelF)");
    if (pFirst == pLast)
    {
        pFirst = pStop;
        return;
    }
    TNode<T>* tmp = pFirst->pNext;
    delete pFirst;
    pFirst = tmp;
    pCurr = pFirst;
}

template <typename T>
void TList<T>::DeleteLast()
{
    if(IsEmpty())
        throw std::exception("empty list(DelL)");
    if (pLast == pFirst)
    {
        delete pFirst;
        return;
    }
    pCurr = pFirst;
    while (pCurr->pNext != pStop)
        Next();
    delete pCurr;
    pPrev->pNext = pStop;
    pLast = pPrev;
    pCurr = pFirst;
    pPrev = pStop;
}

template <typename T>

```

```

void TList<T>::DeleteCurrent()
{
    if (IsEmpty())
        throw std::exception("empty list(DelC)");
    if (!IsEnd())
    {
        if (pCurr == pFirst) {
            DeleteFirst();
        }
        else {
            TNode<T>* tmp = pCurr;
            pPrev->pNext = pCurr->pNext;
            pCurr = pCurr->pNext;
            if (pPrev->pNext == pStop)
                pLast = pPrev;
            delete tmp;
        }
    }
    else {
        DeleteLast();
    }
}

template <typename T>
void TList<T>::DeleteBefore(const TNode<T>* before_node)
{
    if (IsEmpty())
        throw std::exception("empty list(DelB)");
    Reset();
    search(before_node);
    if (pCurr->pNext == pStop && pCurr != before_node)
        throw std::exception("node not found(DelB)");
    if (pCurr == pFirst)
    {
        DeleteFirst();
        pCurr = pFirst;
        return;
    }
    else
    {
        pPrev->pNext = pCurr->pNext;
        delete pCurr;
        pCurr = pFirst;
        pPrev = pStop;
    }
    Reset();
}

template <typename T>
void TList<T>::DeleteAfter(const TNode<T>* after_node)
{
    if (IsEmpty())
        throw std::exception("empty list(DelA)");
    if (pLast == after_node)
        return;
    Reset();
    search(after_node);
    if (pCurr->pNext == pStop && pCurr != after_node)
        throw std::exception("node not found(DelB)");
    Next();
    pPrev->pNext = pCurr->pNext;
    delete pCurr;
    pPrev = pStop;
}

```



```

        pCurr = pFirst;
        Reset();
    }

template <typename T>
void TList<T>::DeleteData(const T& data)
{
    if (IsEmpty())
        throw std::exception("empty list(DelD)");
    Reset();
    search(data);
    if (pCurr->pNext != pStop && pCurr->data != data)
        throw std::exception("data not found(DelD)");
    if (pPrev == pStop)
        DeleteFirst();
    else
    {
        pPrev->pNext = pCurr->pNext;
        delete pCurr;
        pCurr = pFirst;
        pPrev = pStop;
    }
    Reset();
}

template <typename T>
void TList<T>::Sort()
{
    TNode<T>* elem1 = pFirst;
    while (elem1->pNext != pStop)
    {
        TNode<T>* elem2 = elem1->pNext;
        while (elem2 != pStop)
        {
            if (elem1->data > elem2->data)
            {
                T tmp = elem1->data;
                elem1->data = elem2->data;
                elem2->data = tmp;
            }
            elem2 = elem2->pNext;
        }
        elem1 = elem1->pNext;
    }
}

template <typename T>
const TList<T>& TList<T>::operator=(const TList<T>& pList)
{
    if (pList.pFirst == pList.pStop)
        throw std::exception("Invalid pList (=)");
    if (this == &pList)
        return *this;
    clear();
    TNode<T>* pNode = pList.pFirst;
    while (pNode != pList.pStop)
    {
        InsertLast(pNode->data);
        pNode = pNode->pNext;
    }
    return *this;
}

```

```

template <typename T>
void TList<T>::Reset()
{
    if (IsEmpty()) {
        pCurr = pStop;
    }
    else {
        pCurr = pFirst;
        pPrev = pStop;
    }
    return;
}

```

## Приложение С. Реализация класса TRingList

```

template <typename T>
TRingList<T>::TRingList() {
    pHead = new TNode<T>(T());
    this->pStop = pHead;
    pHead->pNext = this->pFirst;
    this->pFirst = pHead;
    this->pLast = pHead;
    this->pCurr = pHead;
    this->pPrev = pHead;
}

template <typename T>
TRingList<T>::TRingList(const TNode<T>* pF) : TList<T>::TList(pF)
{
    pHead = new TNode<T>(T());
    pStop = pHead;
    pHead->pNext = pFirst;
    pPrev = pHead;
    pLast->pNext = pStop;
}

template <typename T>
TRingList<T>::TRingList(const TRingList<T>& obj) : TList<T>::TList(obj)
{
    pHead = new TNode<T>(T());
    pStop = pHead;
    if (obj.pFirst == obj.pStop)
    {
        this->pFirst = pHead;
        this->pLast = pHead;
        this->pCurr = pHead;
        this->pPrev = pHead;
    }
    pPrev = pStop;
    pHead->pNext = pFirst;
    pLast->pNext = pStop;
}

template <typename T>
void TRingList<T>::clear()
{
    TList<T>::clear();
    pHead->pNext = pHead;
}

template <typename T>
TRingList<T>::~~TRingList() {

```

```

        delete pHead;
    }

    template<typename T>
    void TRingList<T>::InsertFirst(const T& data)
    {
        TList<T>::InsertFirst(data);
        pHead->pNext = pFirst;
    }

    template <typename T>
    void TRingList<T>::DeleteFirst()
    {
        TList<T>::DeleteFirst();
        pHead->pNext = pFirst;
    }

    template <typename T>
    const TRingList<T>& TRingList<T>::operator=(const TRingList<T>& pRingList) {
        TList<T>::operator=(pRingList);
        pHead->pNext = this->pFirst;

        return *this;
    }

```

## Приложение D. Реализация класса TMonom

```

TMonom::TMonom()
{
    degree = -1;
    coeff = 0;
}

TMonom::TMonom(double coeff, int degree)
{
    this->coeff = coeff;
    this->degree = degree;
}

TMonom::TMonom(const TMonom& monom)
{
    coeff = monom.coeff;
    degree = monom.degree;
}

TMonom::TMonom(const string& monom)
{
    string str = monom;
    string strcoeff = "";
    degree = 0;
    coeff = 1;
    int tmpdegree = 0;
    int i = 0;
    while (isdigit(str[i]))
    {
        strcoeff += str[i];
        i++;
    }
    while (i < str.length())
    {
        if (str[i] == 'x')
        {
            if (i == str.length() - 1)

```

```

        {
            tmpdegree += 100;
            break;
        }
        if (str[i + 1] == '*')
        {
            tmpdegree += 100;
            i++;
            continue;
        }
        if (str[i + 1] == '^')
        {
            i++;
            continue;
        }
        i++;
        continue;
    }
    if (str[i] == 'y')
    {
        if (i == str.length() - 1)
        {
            tmpdegree += 10;
            break;
        }
        if (str[i + 1] == '*')
        {
            tmpdegree += 10;
            i++;
            continue;
        }
        if (str[i + 1] == '^')
        {
            i++;
            continue;
        }
        i++;
        continue;
    }
    if (str[i] == 'z')
    {
        if (i == str.length() - 1)
        {
            tmpdegree += 1;
            break;
        }
        if (str[i + 1] == '*')
        {
            tmpdegree += 1;
            i++;
            continue;
        }
        if (str[i + 1] == '^')
        {
            i++;
            continue;
        }
        i++;
        continue;
    }
    else if (str[i] == '^')
    {
        i++;
    }

```

```

        continue;
    }
    else if (str[i] == '*')
    {
        i++;
        continue;
    }
    else if (isdigit(str[i]))
    {
        if (str[i - 2] == 'x' && str[i - 1] == '^')
            tmpdegree += (str[i] - 48) * 100;
        if (str[i - 2] == 'y' && str[i - 1] == '^')
            tmpdegree += (str[i] - 48) * 10;
        if (str[i - 2] == 'z' && str[i - 1] == '^')
            tmpdegree += (str[i] - 48);
        i++;
    }
    else
        throw std::exception("invalid string");
}
if (strcoeff.length() != 0)
    coeff = stod(strcoeff);
else
    coeff = 1;
if (tmpdegree > 999)
    throw std::exception("invalid string");
degree = tmpdegree;
}

bool TMonom::operator==(const TMonom& data) const
{
    if (degree == data.degree && coeff == data.coeff)
        return true;
    return false;
}

bool TMonom::operator!=(const TMonom& data) const
{
    return !(*this == data);
}

bool TMonom::operator<(const TMonom& data) const
{
    if (degree < data.degree)
        return true;
    if (degree == data.degree)
        if (coeff < data.coeff)
            return true;
    return false;
}

bool TMonom::operator>(const TMonom& data) const
{
    if (degree > data.degree)
        return true;
    if (degree == data.degree)
        if (coeff > data.coeff)
            return true;
    return false;
}

```

```

TMonom TMonom::operator+(const TMonom& data)
{
    if (degree != data.degree)
        throw std::exception("Invalid monoms(op+)");
    return TMonom(coeff + data.coeff, degree);
}

TMonom TMonom::operator-(const TMonom& data)
{
    if (degree != data.degree)
        throw std::exception("Invalid monoms(op-)");
    return TMonom(coeff - data.coeff, degree);
}

TMonom TMonom::operator*(const TMonom& data)
{
    int newdegree = degree + data.degree;
    if ((degree / 100 + data.degree / 100) > 9
        || ((degree % 100) / 10 + (data.degree % 100) / 10) > 9
        || (degree % 10 + data.degree % 10) > 9)
        return TMonom();
    return TMonom(coeff * data.coeff, newdegree);
}

TMonom TMonom::def_X() const
{
    if ((int)(degree/100) == 0)
        return TMonom(0, 0);
    return TMonom(coeff * (degree / 100), degree - 100);
}

TMonom TMonom::def_Y() const
{
    if ((int)(degree % 100) / 10 == 0)
        return TMonom(0, 0);
    return TMonom(coeff * ((degree % 100) / 10), degree - 10);
}

TMonom TMonom::def_Z() const
{
    if((int)(degree%10) == 0)
        return TMonom(0, 0);
    return TMonom(coeff * (degree % 10), degree - 1);
}

double TMonom::operator()(double x, double y, double z) const
{
    double res = coeff;
    if (degree / 100 != 0)
        for (int i = 0; i < degree / 100; i++)
            res *= x;
    if ((degree % 100) / 10 != 0)
        for (int i = 0; i < (degree % 100) / 10; i++)
            res *= y;
    if (degree % 10 != 0)
        for (int i = 0; i < degree % 10; i++)
            res *= z;
    return res;
}

const TMonom& TMonom::operator=(const TMonom& monom)
{

```

```

        if (*this == monom)
            return *this;
        coeff = monom.coeff;
        degree = monom.degree;
        return *this;
    }

std::istream& operator>>(std::istream in, TMonom& monom)
{
    int x, y, z;
    std::cout << "coeff = ";
    std::cin >> monom.coeff;
    std::cout << std::endl;
    if (monom.coeff == 0)
        return in;
    do {
        std::cout << "x = ";
        std::cin >> x;
        std::cout << std::endl;
    } while (x > 9 || x < 0);
    do {
        std::cout << "y = ";
        std::cin >> y;
        std::cout << std::endl;
    } while (y > 9 || y < 0);
    do {
        std::cout << "z = ";
        std::cin >> z;
        std::cout << std::endl;
    } while (z > 9 || z < 0);
    monom.degree = x * 100 + y * 10 + z;
    return in;
}

void TMonom::SetCoeff(const double cf)
{
    coeff = cf;
}

void TMonom::SetDegree(const int dgr)
{
    degree = dgr;
}

```

## Приложение Е. Реализация класса TPolynom

```

TPolynom::TPolynom(const string& name)
{
    if (name[0] == '\\0')
        throw std::exception("invalid string");
    string tmpstr = name;
    std::string::iterator end_pos =
        std::remove(tmpstr.begin(), tmpstr.end(), ' ');
    tmpstr.erase(end_pos, tmpstr.end());
    string strmonom = "";
    TMonom tmpmonom;

```

```

int flag = 0;
double tmpcoeff = 1;
int tmpdegree = 0;
for (int i = 0; i < tmpstr.length(); i++)
{
    if (i == 0 && tmpstr[i] == '+')
        continue;
    if (strmonom == "" && tmpstr[i] == '-')
    {
        flag++;
        continue;
    }
    if(i != 0)
        if (tmpstr[i-1] == '-')
            flag++;
    if (tmpstr[i] != '+' && tmpstr[i] != '-' )
    {
        strmonom += tmpstr[i];
        if(i != tmpstr.length()-1)
            continue;
    }
    tmpmonom = TMonom(strmonom);
    if (flag > 0)
    {
        tmpmonom.SetCoeff((-1) * tmpmonom.GetCoeff());
    }
    strmonom = "";
    flag = 0;
    monoms.InsertLast(tmpmonom);
}
update();
}

TPolynomial::TPolynomial(const TRingList<TMonom>& monoms)
{
    this->monoms = monoms;
    update();
}

TPolynomial::TPolynomial(const TPolynom& p)
{
    *this = p;
}

```



```

}

void TPolynom::update()
{
    monoms.Sort();
    monoms.Reset();
    while (!monoms.IsEmpty() && !monoms.IsEnd())
    {
        TNode<TMonom>* i = monoms.getpC();
        TNode<TMonom>* j = i->pNext;
        if (j != monoms.getpS() &&
            i->data.GetDegree() == j->data.GetDegree())
        {
            i->data = i->data + j->data;
            i->pNext = j->pNext;
            delete j;
        }
        else
        {
            monoms.Next();
        }
    }
    updatenull();
    updatename();
}

void TPolynom::updatenull()
{
    monoms.Reset();
    int flag = 0;
    while (!monoms.IsEmpty() && !monoms.IsEnd())
    {
        flag = 0;
        if (monoms.get_pCurr().GetCoeff() == 0.0)
        {
            monoms.DeleteCurrent();
            flag++;
        }
        if (flag == 0)
            monoms.Next();
    }
}

```

```

    monoms.Reset();
}

void TPolynom::updatename()
{
    string tmpname = "";
    TMonom tmpmonom;
    int flag = 0;
    while (!monoms.IsEnd())
    {
        tmpmonom = monoms.get_pCurr();
        if (tmpmonom.GetDegree() == 0)
        {
            tmpname += to_string(tmpmonom.GetCoeff());
            monoms.Next();
            flag++;
            continue;
        }
        if (tmpmonom.GetCoeff() > 0 && flag != 0)
            tmpname += "+";
        tmpname += to_string(tmpmonom.GetCoeff());
        if (tmpmonom.GetDegree() / 100 > 0)
        {
            tmpname += "x^" + to_string(tmpmonom.GetDegree() / 100);
        }
        if ((tmpmonom.GetDegree() % 100) / 10 > 0)
        {
            tmpname += "y^" + to_string((tmpmonom.GetDegree() % 100)
                                         / 10);
        }
        if (tmpmonom.GetDegree() % 10 > 0)
        {
            tmpname += "z^" + to_string(tmpmonom.GetDegree() % 10);
        }
        flag++;
        monoms.Next();
    }
    name = tmpname;
    monoms.Reset();
}

const TPolynom& TPolynom::operator=(const TPolynom& polynom)

```

```

{
    this->monoms = polynom.monoms;
    this->name = polynom.name;
    return *this;
}

void TPolynom::InsertSort(const TMonom& m)
{
    monoms.Reset();
    while (monoms.get_pCurr().GetDegree() != m.GetDegree()
           && !monoms.IsEnd() && monoms.get_pCurr() < m)
        monoms.Next();
    if (monoms.get_pCurr().GetDegree() == m.GetDegree())
    {
        monoms.getpC()->data.SetCoeff(monoms.get_pCurr().GetCoeff() +
                                       m.GetCoeff());
        return;
    }
    monoms.InsertBeforeCurrent(m);
    monoms.Reset();
}

TPolynom TPolynom::operator+(const TPolynom& polynom)
{
    TPolynom tmp(polynom);
    while (!tmp.monoms.IsEnd() && !tmp.monoms.IsEmpty())
    {
        if (tmp.monoms.get_pCurr().GetCoeff() != 0)
            InsertSort(tmp.monoms.get_pCurr());
        tmp.monoms.Next();
    }
    monoms.Reset();
    updatenull();
    updatename();
    return *this;
}

TPolynom TPolynom::operator-(const TPolynom& polynom)
{
    *this = *this + polynom * (-1);
    return *this;
}

```

```

TPolynom TPolynom::operator*(const double c) const
{
    TPolynom p(*this);
    if (c == 0)
    {
        return TPolynom();
    }
    p.monoms.Reset();
    while (!p.monoms.IsEnd())
    {
        p.monoms.getpC()->data.SetCoeff(p.monoms.getpC()->data.GetCoeff()
                                         * c);

        p.monoms.Next();
    }
    p.monoms.Reset();
    p.updatename();
    return p;
}

TPolynom TPolynom::operator*(const TPolynom& polynom) const
{
    TPolynom p(*this);
    if (polynom.monoms.IsEmpty())
        return polynom;
    TPolynom tmp(polynom);
    TPolynom newp;
    p.monoms.Reset();
    while (!p.monoms.IsEnd())
    {
        tmp.monoms.Reset();
        while (!tmp.monoms.IsEnd())
        {
            newp.monoms.InsertLast(p.monoms.getpC()->data *
                                   tmp.monoms.getpC()->data);
            tmp.monoms.Next();
        }
        p.monoms.Next();
    }
    newp.update();
}

```

```

    return newp;
}

TPolynom TPolynom::dif() const
{
    TPolynom tmp(*this);
    tmp = tmp.dif_x();
    tmp = tmp.dif_y();
    tmp = tmp.dif_z();
    tmp.updatenull();
    tmp.updatename();
    return tmp;
}

TPolynom TPolynom::dif_x() const
{
    TPolynom tmp(monoms);
    TPolynom newp;
    TMonom tp;
    while (!tmp.monoms.IsEnd())
    {
        tp = tmp.monoms.getpC()->data.def_X();
        newp.monoms.InsertLast(tp);
        tmp.monoms.Next();
    }
    return newp;
}

TPolynom TPolynom::dif_y() const
{
    TPolynom tmp(monoms);
    TPolynom newp;
    TMonom tp;
    while (!tmp.monoms.IsEnd())
    {
        tp = tmp.monoms.getpC()->data.def_Y();
        newp.monoms.InsertLast(tp);
        tmp.monoms.Next();
    }
    return newp;
}

TPolynom TPolynom::dif_z() const

```

```

{
    TPolynom tmp(monoms);
    TPolynom newp;
    TMonom tp;
    while (!tmp.monoms.IsEnd())
    {
        tp = tmp.monoms.getpC()->data.def_Z();
        newp.monoms.InsertLast(tp);
        tmp.monoms.Next();
    }
    return newp;
}

double TPolynom::operator()(double x, double y, double z) const
{
    TPolynom tmp(*this);
    double res = 0;
    while (!tmp.monoms.IsEnd())
    {
        res += tmp.monoms.getpC()->data(x, y, z);
        tmp.monoms.Next();
    }
    return res;
}

```