Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Алгебра полиномов»**

**Выполнили**:

студенты группы 3821Б1ФИ3

Косарев Е.А.

Лупша Е.А.

Мамаева О.А.

Мортина А.О.

Шушпанов Н.М.

**Проверил**:

Заведующий лабораторией суперкомпьютерных технологий и высокопроизводительных вычислений

Лебедев И.Г.

Нижний Новгород

2023

Содержание

[Введение 3](#_Toc100783489)

[1. Постановка задачи 4](#_Toc100783490)

[2. Руководство пользователя 5](#_Toc100783491)

[3. Руководство программиста 8](#_Toc100783492)

[3.1. Описание структуры программы 11](#_Toc100783493)

[3.2. Описание алгоритмов 12](#_Toc100783494)

[4. Результаты тестов 16](#_Toc100783495)

[Заключение 18](#_Toc100783496)

[Литература 19](#_Toc100783497)

[Приложение 20](#_Toc100783498)

# Введение

Таблицей называется динамическая структура данных, где базисное множество – это семейство линейных структур из записей, а базисное отношение включения определяется операциями вставки и удаления записей.

Важность понятия таблицы определяется: организацией доступа по имени для управления информацией в привычной для человека форме, абстрагирование от проблем распределения памяти при размещении данных, а также реализацией концепции ассоциативной памяти*.*

Характерной логической особенностью таблиц, которая определяет их отдельное рассмотрение, является то, что доступ к элементам таблицы производится не по номеру (индексу), а по ключу – по значению одного из свойств объекта, описываемого элементом таблицы. Ключ – это свойство, идентифицирующее данную запись во множестве однотипных записей. Как правило, к ключу предъявляется требование уникальности в данной таблице. Ключ может включаться в состав записи и быть одним из ее полей, но может и не включаться в запись, а вычисляться по положению записи.

В данной лабораторной работе мы будем проводить арифметические операции над полиномами: сложение, вычитание, умножение и подсчет в точке, а также дифференцирование, интегрирование и составление новых выражений из имен введённых многочленов. Для хранения полиномов будет реализовано 6 видов таблиц.

В отчете приводится постановка задачи, описание алгоритмов, а также дается описание программы и правил ее использования, прилагается текст программы и результаты выполнения тестов.

# Постановка задачи

В рамках лабораторной работы ставится задача создания программных средств, поддерживающих работу с шестью видами таблиц, предназначенных для эффективного хранения полинома, его имени и ссылки на объект класса, реализующего работу с полиномами.

Таблицы для реализации хранения полиномов:

* неупорядоченные таблицы:
  + линейная таблица на массиве;
  + линейная таблица на списке;
* упорядоченные таблицы:
  + таблица на поисковом дереве (АВЛ или красно-черном);
  + таблица на массиве;
* хеш-таблицы:
  + хеш-таблица с открытым перемешиванием;
  + хеш-таблица со списками (метод цепочек);

Для работы с полиномами необходимо реализовать:

* операции над полиномами:
  + ввод полинома и его имени;
  + удаление введенного ранее полинома;
  + вывод полиномов и результата операций;
* организация хранения полинома;
* арифметические методы:
  + сложение;
  + вычитание;
  + умножение;
  + деление;
  + нахождение производной при заданной переменной дифференцирования;
  + нахождение неопределенного интеграла;
  + возможность получения выражения из полиномов;
  + вычисление значения полинома при заданных значениях переменных.

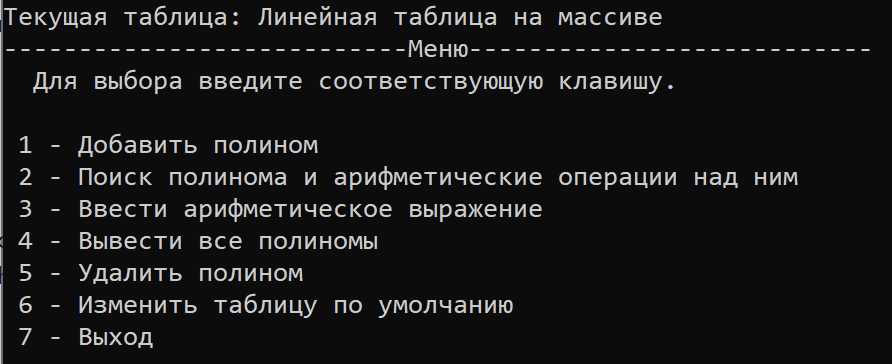
В каждой таблице должна быть возможность удаления, добавления и поиска полинома по имени. Удаление и добавление полинома должно происходить сразу во всех таблицах, поиск же должен осуществляться только в активной таблице.

В процессе выполнения лабораторной работы требуется использовать систему контроля версий Git и фрэймворк для разработки автоматических тестов Google Test.

# Руководство пользователя

Программа имеет вид консольного приложения.

На первом этапе пользователю будет представлено главное меню программы (рис.1), где также при запуске указана таблица по умолчанию (линейная таблица на массиве).



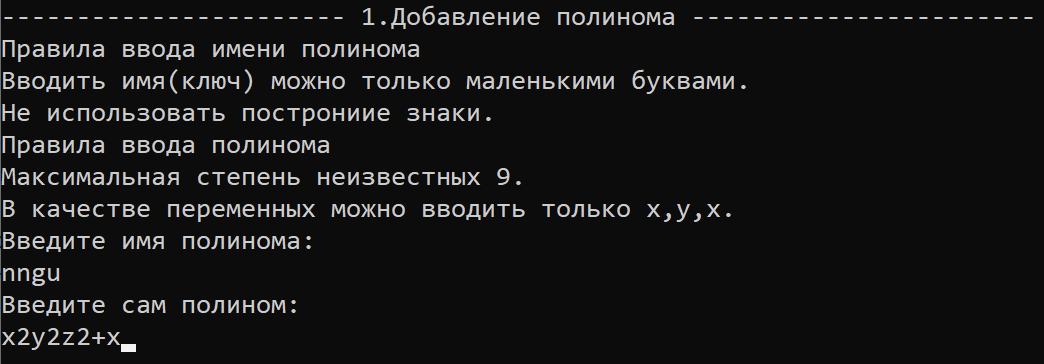
**Рис. 1 Главное меню**

Далее пользователь должен добавить новый полином в таблицу. Сделать он это может, нажав на соответствующий пункт в меню (рис. 2).



**Рис. 2 Добавить полином**

На следующем этапе появятся правила ввода полинома и его имени (рис. 3). Пользователю необходимо ввести все поля, согласно правилам!



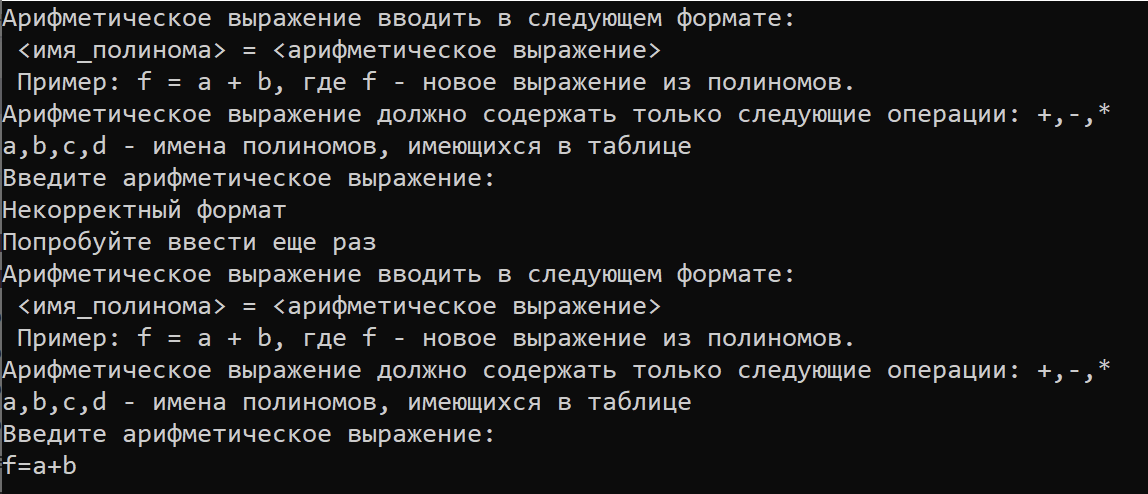
**Рис. 3 Правила ввода полинома**

Если пользователь добавил несколько полиномов, то у него есть возможность получить новый многочлен, составив арифметическое выражение из имен. Для этого нужно выбрать нужный пункт меню (рис. 4).



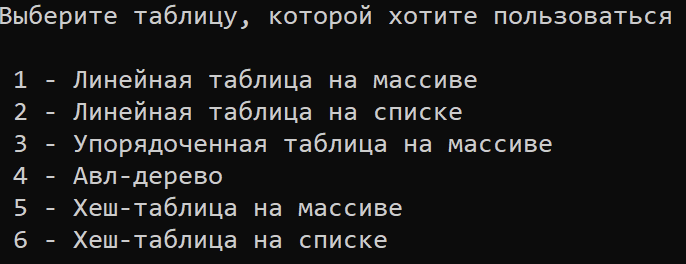
**Рис. 4 Ввод арифметического выражения**

Далее пользователь должен ввести новое выражение, следуя инструкции на экране (рис. 5).



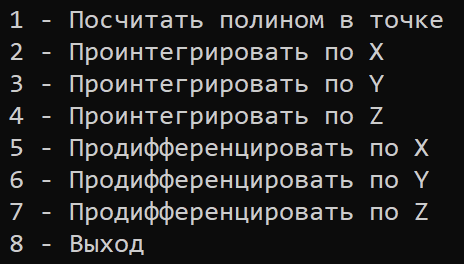
**Рис. 5 Составление арифметического выражения**

Если же пользователь перейдет в пункт меню № 2, то у него появится возможность произвести различные арифметические операции над полиномом. Для этого ему нужно будет выбрать таблицу, по которой будут работать алгоритмы (рис. 6).



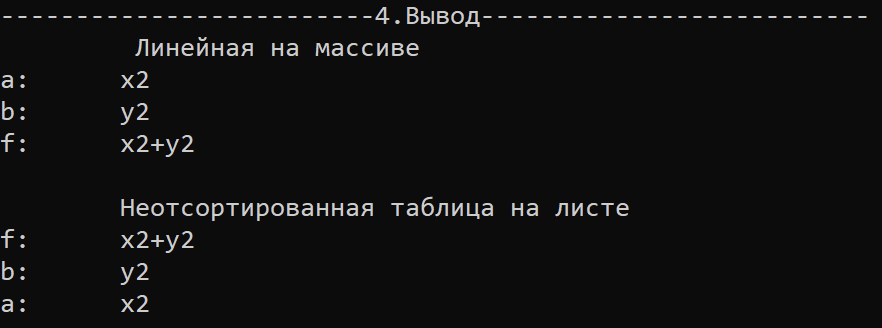
**Рис. 6 Выбор таблицы**

Далее нужно будет ввести имя для поиска нужного полинома (если такого нет, то об этом будет сообщено пользователю) и будет предложено выбрать арифметическую операцию (рис. 7).



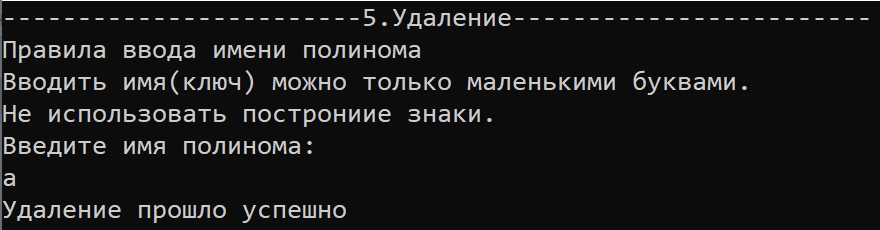
**Рис. 7 Выбор операции над полиномом**

Удовлетворенный результатом пользователь может вернуться в главное меню и вывести все полиномы, которые хранятся в таблице, выбрав соответствующий пункт меню (рис. 8).



**Рис. 8 Фрагмент вывода хранящихся полиномов**

Вернувшись в главное меню, пользователь может удалить какой-то из введенных ранее многочленов, выбрав соответствующий пункт меню. Для этого необходимо ввести имя этого полинома (рис. 9).



**Рис. 9 Удаление полинома**

Также при необходимости можно поменять таблицу по умолчанию (рис. 10).



**Рис. 10 Изменение таблицы по умолчанию**

# Руководство программиста

Программа написана на основе суперкласса BaseTable при помощи таких классов, как List, Polinim, Postfix, Stack и Inteface. От суперкласса наследовано 6 классов для реализации различных видов таблиц: LineTable, UnorderedTableOnList, OrderedTableOnArray, SearchTreeTable, HashTableSomething и HashTableOnList.

**Класс BaseTable содержит три поля типа private:**

1. string key – ключ, имя полинома;
2. string polynomString – исходная строка, введенная пользователем;
3. Polynom \_polynom – полином.

**Класс BaseTable содержит методы virtual:**

1. virtual Data\* find(const std::string key) – метод класса, реализующий поиск полинома по ключу, принимает в качестве аргумента имя полинома;
2. virtual void insert(const Data& data) – метод класса, реализующий вставку полинома в таблицу, принимает в качестве аргумента ссылку объект структуры Data, содержащий информацию о новом полиноме;
3. virtual void deleteByKey(const std::string& key) – метод класса, отвечающий за удаление полинома из таблицы по ключу;
4. virtual void print() – метод класса, реализующий печать всех полиномов, содержащихся в таблице;
5. virtual ~BaseTable() – деструктор класса.

**Класс Polynom содержит два поля в модификаторе доступа private:**

1. string prefix – исходная строка, введенная пользователем;
2. List <Monom> polinom – список мономов с ненулевыми коэффициентами.

**Класс Polynom содержит следующие методы:**

1. bool IsSign(char op) – метод класса, проверяющий является ли символ знаком (является private методом);
2. bool IsVariable(char op) – метод класса, проверяющий является ли символ переменной (является private методом);
3. int GetDegree(char p) – метод класса, возвращающий преобразованную из char в int степень(является private методом);
4. Polynom(string \_str = "") – конструктор класса, принимающий в качестве параметра строку;
5. Polynom(const Polynom& pl) – конструктор копирования;
6. void Convert() – метод, разделяющий исходную строку на мономы;
7. int CalculateInPoint(int \_x, int \_y, int \_z) – метод, осуществляющий подсчет значения в точке, принимает в качестве параметров значения неизвестных;
8. Polinom operator+(TPolinom& pl) – перегрузка оператора сложения, принимающая в качестве параметра правое слагаемое;
9. TPolynom operator-(Polynom& pl) – перегрузка оператора вычитания, принимающая в качестве параметра вычитаемое;
10. Polynom operator\*(Polynom& pl) – перегрузка оператора произведения, принимающая в качестве параметра правый множитель;
11. Polynom& operator=(const Polynom& pl) – перегрузка оператора приравнивания, принимающая в качестве параметра полином, поля которого будут присвоены;
12. friend ostream& operator<<(ostream& os, Polynom& pl) перегрузка оператора вывода;
13. friend istream& operator>>(istream& is, Polynom& pl) – перегрузка оператора ввода;
14. bool operator == (const Polynom& pl) const – перегрузка оператора сравнения;
15. Polynom Sort() – метод, сортирующий в лексикографическом порядке мономы полинома;
16. string GetPrefix() – метод, возвращающий исходную строку, введенную пользователем.

**Класс List содержит пять полей с доступом private:**

1. struct Node – звено списка;
2. int numberOfElements – переменная, хранящая число элементов в списке;
3. Node <type>\* pFirst – указатель на начало списка;
4. Node <type>\* pCurrent – указатель на текущее звено списка;
5. class iterator – класс, содержащий одно поле типа private и методы, позволяющие работать с итераторами.

**Класс List содержит следующие методы в модификаторе доступа public:**

1. List() – конструктор класса по умолчанию;
2. List(const List<type>& lst) – конструктор копирования;
3. List<type>& operator = (const List<type>& lst) – – перегрузка оператора приравнивания, принимающая в качестве параметра лист, поля которого будут присвоены;
4. bool operator==(const List& lst)const – перегрузка оператора сравнения на равно;
5. bool operator!=(const List& lst)const – перегрузка оператора сравнения на неравно;
6. void PushBack(type tmp) – метод, заполняющий следующее звено списка, если список не полон;
7. type PopFront() – метод, который возвращает информацию первого звена из списка и удаляет его;
8. void Insert(iterator it, type val) – метод, вставляющий звено в список;
9. void Erase(iterator it) – метод, стирающий звено из списка;
10. void Clear() – метод, освобождающий список;
11. iterator Begin() – метод, возвращающий указатель на начало списка;
12. iterator End() – метод, возвращающий указатель на конец списка;
13. int GetSize() – метод, возвращающий размер списка;
14. bool IsEmpty() – метод, проверяющий список на пустоту;
15. ~List() – деструктор класса.

**Класс LineTable содержит три поля private и один метод:**

1. Data\* dat – указатель на структуру Data;
2. int size – переменная, хранящая размер таблицы;
3. int capacity – ёмкость массива;
4. void repacking() – метод, увеличивающий размер таблицы.

**Класс LineTable содержит следующие методы в модификаторе доступа public:**

1. LineTable() : BaseTable() – конструктор класса по умолчанию;
2. void deleteByKey(const string& key) – метод, удаляющий полином по ключу;
3. void insert(const Data& d) –метод, отвечающий за вставку полинома в таблицу;
4. Data\* find(const string key) – метод, находящий полином по ключу, возвращает указатель на объект структуры Data;
5. void print() – метод печати таблицы;
6. ~LineTable() – деструктор класса.

**Класс UnorderedTableOnList содержит два приватных поля:**

1. Link\* Head – указатель на структуру Link;
2. int Amount – переменная, хранящая размер таблицы.

**Класс UnorderedTableOnList содержит следующие методы в модификаторе доступа public:**

1. UnorderedTableOnList () – конструктор класса по умолчанию;
2. ~UnorderedTableOnList () – деструктор класса;
3. void insert(const Data& d) –метод, отвечающий за вставку полинома в таблицу;
4. Data\* find(const string key) – метод, находящий полином по ключу, возвращает указатель на объект структуры Data;
5. void print() – метод печати таблицы.

**Класс OrderedTableOnArray содержит два приватных поля:**

1. Link\* Head – указатель на структуру Link;
2. int Amount – переменная, хранящая размер таблицы.

**Класс OrderedTableOnArray содержит следующие методы в модификаторе доступа public:**

1. OrderedTableOnArray () – конструктор класса по умолчанию;
2. ~ OrderedTableOnArray() – деструктор класса;
3. void insert(const Data& d) –метод, отвечающий за вставку полинома в таблицу;
4. Data\* find(const string key) – метод, находящий полином по ключу, возвращает указатель на объект структуры Data;
5. void print() – метод печати таблицы.

**Класс SearchTreeTable содержит приватное поле и метод:**

1. TreeNode\* pRoot – указатель на структуру TreeNode;
2. void TempPrint(TreeNode\* p) – приватный метод класса.

**Класс SearchTreeTable содержит следующие методы в модификаторе доступа public:**

1. SearchTreeTable () – конструктор класса по умолчанию;
2. void balance(int& a, int& b, int p) – метод, отвечающий за балансировку дерева, принимающий три аргумента;
3. void rotateLeft(TreeNode\* a) – метод, реализующий «поворот влево», принимает в качестве аргумента объект структуры TreeNode;
4. void rotateRight(TreeNode\* a) – метод, реализующий «поворот вправо», принимает в качестве аргумента ссылку на объект структуры TreeNode;
5. void rotateBigLeft(TreeNode\* a) – метод, реализующий «большой поворот влево», принимает в качестве аргумента ссылку на объект структуры TreeNode;
6. void rotateBigRight(TreeNode\* a) – метод, реализующий «большой поворот вправо», принимает в качестве аргумента ссылку на объект структуры TreeNode;
7. ~ SearchTreeTable() – деструктор класса;
8. void insert(const Data& d) –метод, отвечающий за вставку полинома в таблицу;
9. Data\* find(const string key) – метод, находящий полином по ключу, возвращает указатель на объект структуры Data;
10. void print() – метод печати таблицы.

**Класс HashTableSomething содержит одно поле в pritate:**

1. Data\* Tab[1000] – массив указателей на объекты структуры Data.

**Класс HashTableSomething содержит следующие методы в модификаторе доступа public:**

1. HashTableSomething() – конструктор класса по умолчанию;
2. ~ HashTableSomething() – деструктор класса;
3. void insert(const Data& d) –метод, отвечающий за вставку полинома в таблицу;
4. Data\* find(const string key) – метод, находящий полином по ключу, возвращает указатель на объект структуры Data;
5. void print() – метод печати таблицы;
6. int HashIt(string key) – хеш-функция.

**Класс HashTableOnList содержит одно поле в private:**

1. link\* Tab[1000] – массив указателей на объекты структуры link.

**Класс HashTableOnList содержит следующие методы в модификаторе доступа public:**

1. THashTableOnList() – конструктор класса по умолчанию;
2. ~ THashTableOnList() – деструктор класса;
3. void insert(const Data& d) –метод, отвечающий за вставку полинома в таблицу;
4. Data\* find(const string key) – метод, находящий полином по ключу, возвращает указатель на объект структуры Data;
5. void print() – метод печати таблицы.

**Класс Interface содержит в private:**

1. BaseTable\* Base – указатель на объект суперкласса BaseTable;
2. BaseTable\* tables[6] – массив указателей на объекты суперкласса BaseTable;
3. string namesOfTables[6] – массив названий всех видов реализованных таблиц;
4. string tecName – имя текущей таблицы.

**Класс Interface содержит следующие методы в модификаторе доступа public:**

1. Interface() – конструктор класса по умолчанию;
2. void countPolynomInPoint(Data data, int x, int y, int z) – метод, производящий подсчет в точке с координатами, поступающими в качестве аргумента;
3. string controlPolynom(const string& str) – метод, отвечающий за проверку строки, введенной пользователем;
4. int inputControl(int left\_board, int right\_board, void (\*print)(const string&), const string& str) – метод, реализующий выбор пункта меня, принимает в качестве аргумента границы и ссылку на функцию печати пунктов меню;
5. void tableSelection() – метод, реализующий выбор текущей таблицы;
6. void menu() – метод, реализующий главное меню программы;
7. void addPolinom() – метод, позволяющий добавить новый полином сразу во все таблицы;
8. void searchAndMakeOperations() –метод, отвечающий за поиск полинома по ключу, в случае положительного результата предлагает пользователю произвести действия над найденным полиномом;
9. string controlKey(const string& key) – контроль правильности введенного пользователем ключа;
10. void print() – метод печати таблицы;
11. void deleteByKey() – метод, отвечающий за удаление полинома по ключу сразу из всех таблиц;
12. void calculate() – метод, производящий операции над полиномами, такие как: сложение, вычитание и умножение;
13. void calPol() – метод, позволяющий ввести арифметическое выражение из ключей, уже имеющихся в таблице, сохраняет новый полином сразу во все таблицы.

## Описание структуры программы

Программа содержит два основных проекта:

1. base
2. gtest

Проект base содержит модули:

* “Main.cpp”;
* “BaseTable.h” с объявлением структуры Data и класса BaseTable;
* “BaseTable.cpp” с реализацией класса BaseTable;
* “HashTableOnList.h” с объявлением структуры link и класса HashTableOnList;
* “HashTableOnList.cpp” с реализацией класса HashTableOnList;
* “HashTableSomething.h” с объявлением класса HashTableSomething;
* “HashTableSomething.cpp” с реализацией класса HashTableSomething;
* “Interface.h” с объявлением класса Interface;
* “Interface.cpp” с реализацией класса Interface;
* “LineTable.h” с объявлением класса LineTable;
* “LineTable.cpp” с реализацией класса LineTable;
* “OrderedTableOnArray.h” с объявлением класса OrderedTableOnArray;
* “OrderedTableOnArray.cpp” с реализацией класса OrderedTableOnArray;
* “Polynom.h” с объявлением класса Polynom;
* “Polynom.cpp” с реализацией методов класса Polynom с вспомогательными функциями;
* “Postfix.h” с объявлением структуры Monom и класса Postfix;
* “Postfix.cpp” с реализацией класса Postfix;
* “SearchTreeTable.h” с объявлением структуры TreeNode и класса SearchTreeTable;
* “SearchTreeTable.cpp” с реализацией класса SearchTreeTable;
* “Stack.h” с объявлением и реализацией класса Stack;
* “UnorderedTableOnList .h” с объявлением структуры Link и класса UnorderedTableOnList ;
* “ UnorderedTableOnList .cpp” с реализацией класса UnorderedTableOnList .

Проект gtest содержит необходимую структуру для работы тестов Google Test и набор тестов, проверяющих правильность реализации основных классов.

## Описание алгоритмов

**Алгоритм реализации списка мономов**

На вход поступает строка с введенным пользователем полиномом. В цикле посимвольно каждый элемент строки распознается как знак, число или константа. В зависимости от этого формируется моном с коэффициентом, именем и степенью, представленной в линейном виде.

**Алгоритм вычитания полиномов**

На вход поступает вычитаемое в виде объекта класса Polynom. Далее создаются итераторы, указывающие на начала уменьшаемого и вычитаемого, и объект класса Polynom, куда будет записан ответ. Далее в цикле каждый элемент вычитаемого сравнивается с каждым элементом уменьшаемого. Если находятся мономы с одинаковыми именами, то их разница отправляется в результат. Все оставшиеся мономы уменьшаемого заносятся в результат с изначальным знаком, а оставшиеся мономы вычитаемого с противоположным.

**Алгоритм сложения полиномов**

На вход поступает слагаемое в виде объекта класса Polynom. Далее создаются итераторы, указывающие на начала двух слагаемых, и объект класса Polynom, куда будет записан ответ. Далее в цикле каждый элемент одного слагаемого сравнивается с каждым элементом другого. Если находятся мономы с одинаковыми именами, то их сумма отправляется в результат. Все оставшиеся мономы обоих слагаемых заносятся в результат с изначальным знаком.

**Алгоритм произведения полиномов**

На вход поступает множитель в виде объекта класса Polynom. Далее создаются итераторы, указывающие на начала двух множителей, и объект класса Polynom, куда будет записан ответ. Далее в цикле каждый элемент одного множителя умножается на каждый элемент другого. Если находятся мономы с одинаковыми именами, то в результат отправляется их сумма. Все остальные мономы, получившиеся в результате перемножения мономов, также отправляются в результат.

**Алгоритм вычисления в точке**

На вход поступают значения неизвестных x, y и z. Создается переменная, которая будет возвращена функцией в виде ответа. Далее в цикле вычисляется значение каждого монома и приплюсовывается к общей сумме. Для этого берется степень каждого монома и делится нацело на 100, находя степень переменной x, далее степень делится нацело на 10 и берется остаток от деления на 10, чтобы найти степень y, а также находится степень z, значение которой равно остатку от деления степени полинома на 10. После этого коэффициент монома умножается на значения переменных, возведенных в нужную степень. Таким образом формируется итоговый ответ.

**Алгоритмы для реализации неупорядоченной линейной таблицы на массиве:**

1. **Вставка**

Ищем по имени полином:

* + если такое имя уже есть в таблице, то алгоритм завершает свою работу.
  + если такого имени нет, то в свободную ячейку памяти записываем указатель на объект полинома (в конец массива). При отсутствии места массив перезаписывается с удвоением размера.

1. **Удаление**

Ищем по имени полином:

* + если такое имени нет, то алгоритм завершает свою работу.
  + если такое имя есть, то вместо найденного полинома записывается последний, а размер массива уменьшается.

Алгоритм завершает свою работу.

1. **Поиск**

А) Проходим по массиву и пока не дошли до конца массива, сравниваем имя объекта полинома и переданный ключ:

* + Если они равны, то нашли нужный полином. Алгоритм прекращает свою работу.
  + Если они не равны, то продолжаем идти по массиву, пока не найдем.

Б) Если дошли до конца массива и не нашли полином, то возвращаем признак отсутствия элемента. Алгоритм прекращает свою работу.

**Алгоритмы для реализации неупорядоченной линейной таблицы на листе:**

1. **Вставка**

Ищем по имени полином:

* + если такое имя уже есть в таблице, то алгоритм завершает свою работу.
  + если такого имени нет, то создаем новое звено и заполняем его данным. Добавляем звено в начало списка.

1. **Удаление**

Ищем по имени полином:

* + если такое имени нет, то алгоритм завершает свою работу.
  + если такое имя есть, удаляем полином, используя стандартное удаление звена списка.

Алгоритм завершает свою работу.

1. **Поиск**

Имеем указатель на первое звено в списке.

А) Проходим по списку и пока не дошли до конца, сравниваем имя объекта полинома и переданный ключ:

* + Если они равны, то нашли нужный полином. Алгоритм прекращает свою работу.
  + Если они не равны, то продолжаем идти по списку, пока не найдем.

Б) Если дошли до конца списка и не нашли полином, то возвращаем признак отсутствия элемента. Алгоритм прекращает свою работу.

**Алгоритмы для реализации упорядоченной таблицы на массиве:**

1. **Вставка**

Ищем по имени полином в упорядоченном массиве:

* + если такое имя уже есть в таблице, то алгоритм завершает свою работу.
  + если такого имени нет, то ищем позицию для вставки и вставляем полином, смещая записи, начиная с той позиции, куда будем вставлять новую запись, при отсутствии места массив увеличивает свой размер.

1. **Удаление**

Ищем по имени полином в упорядоченном массиве:

* + если такое имени нет, то алгоритм завершает свою работу.
  + если такое имя есть, то производим перезаписывание элементов до индекса найденного элемента.

Алгоритм завершает свою работу.

1. **Поиск**

А) Проходим по массиву и пока не дошли до конца массива, сравниваем имя объекта полинома и переданный ключ:

* + Если они равны, то нашли нужный полином. Возвращаем указатель на найденный элемент. Алгоритм прекращает свою работу.
  + Если они не равны, то продолжаем идти по массиву, пока не найдем.

Б) Если дошли до конца массива и не нашли полином, то возвращаем признак отсутствия элемента. Алгоритм прекращает свою работу.

**Алгоритмы для реализации таблицы на поисковом дереве:**

1. **Вставка**

Ищем по имени полином:

* + если такое имя есть, то алгоритм завершает свою работу;
  + если такого имени нет, то сравниваем с последним элементом в поиске и вставляем по лексикографическому порядку в один из указателей, затем пересчитываются разницы высот для всех родительских вершин, и при необходимости происходит балансировка дерева.

Алгоритм завершает свою работу.

1. **Удаление**

Ищем по имени полином:

* + если такое имени нет, то алгоритм завершает свою работу.
  + если такое имя есть, удаляем этот элемент.

Алгоритм завершает свою работу.

1. **Поиск**

Ищем по имени полином, сравнивая с корневым элементом:

* + если меньше, то идем по указателю на меньший элемент
  + если больше, то идем по указателю на больший элемент
  + в случае равенства передается ссылка на полином.

Алгоритм завершает свою работу.

**Алгоритмы для реализации хеш-таблицы с открытой адресацией:**

1. **Вставка**

С помощью хэш-функции и исходного ключа вычисляем производный ключ, если не нашли:

* + если найденное место свободно, то элемент вставляется в эту позицию
  + иначе: происходит смещение позиции, новая позиция вычисляется по формуле (k+p)%N, k – текущая позиция, p – смещение, N – размер массива

Если нашли: алгоритм завершает свою работу.

1. **Удаление**

С помощью хэш-функции и исходного ключа вычисляем производный ключ:

* + Если нашли такой, то алгоритм завершает свою работу.
  + Если не нашли, то продолжаем проверять элементы в таблице, пока не встретим удаленный, или не инициализированный элемент, или не повторим операцию кол-во раз равное размеру таблицы (пройдем все элементы).

Алгоритм завершает свою работу.

1. **Поиск**

Вычисляем хеш по переданному ключу:

* + если значение ключа в ячейке равно переданному, то возвращаем указатель на текущий элемент
  + если не совпало, то начинаем проверять элементы далее, пока не встретим удаленный, или не инициализированный элемент, или не повторим операцию кол-во раз равное размеру таблицы (пройдем все элементы).

Если элемент не был найдет, возвращаем нулевой указатель.

# Результаты тестов

Для выявления ошибок в работе программы были реализованы тесты на основе фреймворка для написания автоматических тестов Google Test.

Примеры тестов, проверяющих класс List (с полным списком можно ознакомиться в приложении):

TEST(List, can\_create\_list)

{

ASSERT\_NO\_THROW(List<int> list);

}

TEST(List, can\_create\_list\_with\_argument)

{

List<int> list1;

ASSERT\_NO\_THROW(List<int> list2(list1));

}

TEST(List, can\_equate\_list)

{

List<int> list1, list2;

list1.PushBack(5);

ASSERT\_NO\_THROW(list2 = list1);

}

TEST(List, can\_compare\_two\_lists\_with\_value)

{

List<int> list1, list2;

list1.PushBack(10);

list2.PushBack(10);

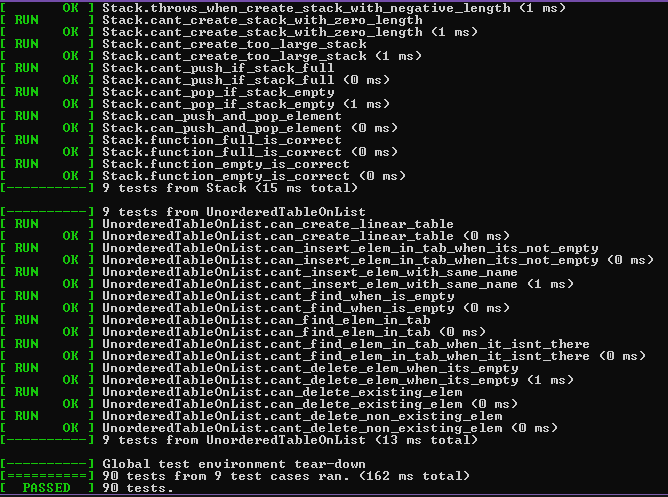
EXPECT\_TRUE(list1 == list2);

}

Список модульных тестов:

1. test\_hashtableonlist.cpp
2. test\_hashtablesomething.cpp
3. test\_linetable.cpp
4. test\_list.cpp
5. test\_orderedtableonarray.cpp
6. test\_polynom.cpp
7. test\_searchtreetable.cpp
8. test\_stack.cpp
9. test\_unorderedtableonlist .cpp

Написанные тесты программа прошла успешно (рис. 12).



**Рис. 12 Фрагмент модульных тестов**

# Заключение

В лабораторной работе был разработан программный комплекс, включающий в себя базовый класс для работы с таблицами, а также наследуемые от него классы таблиц.

Таблицы являются важным и широко распространенным в практике типом структур данных. На практике мы обратили внимание на эффективность в организации доступа по имени для управления информацией в привычной для человека форме.

Просматриваемые таблицы являются одним из самых простых способов организации таблиц, данный подход является эффективным при небольшом наборе имеющихся записей.

Упорядоченные таблицы представляют собой эффективный способ организации таблиц при большом количестве имеющихся записей.

Представление таблиц при помощи деревьев поиска обеспечивает сложность в среднем для всех операций обработки (поиска, вставки и удаления). Максимальная сложность обработки деревьев поиска имеет порядок N. Сбалансированные деревья поиска обеспечивает сложность порядка для любых вариантов исходных данных. Но идеально сбалансированные деревья требуют больших накладных расходов для балансировки.

Использование хеширование позволяет разрабатывать эффективные способы представления таблиц. Эффективность обработки таблиц с вычислимым входом зависит не от количества записей, а от степени заполненности структуры хранения. А метод цепочек обеспечивает получение структуры хранения таблицы с динамическим распределением памяти

В ходе работы также написаны тесты, позволяющие проверить работоспособность программы. Программа успешно прошла тестирование.

# Литература

1. Барышева И.В., Мееров И.Б., Сысоев А.В., Шестакова Н.В. Под редакцией Гергеля В.П. Лабораторный практикум. Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2017. – 105с.
2. Полищук А.П., Семериков С.А. Методы вычислений в классах языка С++: Учебное пособие. – Кривой Рог: Издательский отдел КГПИ, 1999. – 350 с., ил..
3. Сайт: habr.com – Самые популярные структуры данных – <https://habr.com/ru/post/489494/>
4. Сайт: cpp.mazurok.com – Многочлены - <https://cpp.mazurok.com/tag/%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%D1%87%D0%BB%D0%B5%D0%BD%D1%8B/>
5. Сайт: best programmer – 9 структур данных C++ – <https://bestprogrammer.ru/izuchenie/9-struktur-dannyh-c-kotorye-nuzhno-znat-na-sobesedovanii-po-kodirovaniyu>

# Приложение

**Модуль List.h**

#include <iostream>

#include <string>

#include <iostream>

#include <fstream>

const int MAX\_SIZE\_LIST = 1000;

using namespace std;

template <class type>

class List

{

private:

    template <class type>

    struct Node

    {

        Node\* pNext;

        type data;

        Node(type \_data = {}, Node\* \_pNext = nullptr)

        {

            pNext = \_pNext;

            data = \_data;

        }

    };

    int numberOfElements;

    Node <type>\* pFirst;

    Node <type>\* pCurrent;

public:

    class iterator

    {

    private:

        Node<type>\* pointer;

    public:

        iterator()

        {

            pointer = new Node<type>;

        }

        iterator(const iterator& it)

        {

            pointer = it.pointer;

        }

        Node<type>\* operator->() const

        {

            return pointer;

        }

        bool operator == (const iterator& it) const

        {

            if (pointer == it.pointer)

                return true;

            else return false;

        }

        bool operator != (const iterator& it) const

        {

            return !(\*this == it);

        }

        type operator\*()const

        {

            if (pointer == nullptr) { throw("nullptr"); }

            return pointer->data;

        }

        iterator& operator ++()

        {

            this->pointer = this->pointer->pNext;

            return \*this;

        }

        iterator& operator=(const iterator& temp)

        {

            pointer = temp.pointer;

            return \*this;

        }

        friend class List;

    };

    List()

    {

        pFirst = nullptr;

        pCurrent = nullptr;

        numberOfElements = 0;

    }

    List(const List<type>& lst)

    {

        this->pFirst = nullptr;

        this->numberOfElements = 0;

        Node<type>\* count = lst.pFirst;

        while (count != nullptr)

        {

            this->PushBack(count->data);

            count = count->pNext;

        }

    }

    List<type>& operator = (const List<type>& lst)

    {

        Node<type>\* count = lst.pFirst;

        if (!this->IsEmpty())

            this->Clear();

        while (count != nullptr)

        {

            this->PushBack(count->data);

            count = count->pNext;

        }

        return \*this;

    }

    bool operator==(const List& lst)const

    {

        if (this->numberOfElements != lst.numberOfElements)

            return false;

        else

        {

            Node<type>\* left = this->pFirst;

            Node<type>\* right = lst.pFirst;

            while (right != nullptr)

            {

                if (left->data != right->data)

                    return false;

                left = left->pNext;

                right = right->pNext;

            }

            return true;

        }

    }

    bool operator!=(const List& lst)const

    {

        return !(\*this == lst);

    }

    void PushBack(type tmp)

    {

        if (IsEmpty())

        {

            pFirst = new Node<type>(tmp);

            pCurrent = pFirst;

            numberOfElements++;

        }

        else

        {

            if (numberOfElements == MAX\_SIZE\_LIST)

                throw("Polynom full");

            pCurrent->pNext = new Node<type>(tmp);

            pCurrent = pCurrent->pNext;

            numberOfElements++;

        }

    }

    type PopFront()

    {

        type val = pFirst->data;

        Node<type>\* tmp = pFirst;

        pFirst = pFirst->pNext;

        delete tmp;

        numberOfElements--;

        return val;

    }

    void Insert(iterator it, type val)

    {

        Node<type>\* count = pFirst;

        Node<type>\* prev = nullptr;

        Node<type>\* insert = new Node<type>(val);

        int pos = 0;

        while (count != it.pointer)

        {

            prev = count;

            count = count->pNext;

            pos++;

        }

        if (pos == 0)

        {

            Node<type>\* tmp = pFirst;

            insert->pNext = tmp;

            pFirst = insert;

            numberOfElements++;

        }

        else

        {

            insert->pNext = count;

            prev->pNext = insert;

            numberOfElements++;

        }

    }

    void Erase(iterator it)

    {

        Node<type>\* count = pFirst;

        Node <type>\* prev = nullptr;

        int pos = 0;

        while (count != it.pointer)

        {

            prev = count;

            count = count->pNext;

            pos++;

        }

        if (pos == 0)

        {

            Node<type>\* tmp = pFirst;

            pFirst = pFirst->pNext;

            delete tmp;

            numberOfElements--;

        }

        else

        {

            prev->pNext = count->pNext;

            delete count;

            numberOfElements--;

        }

    }

    void Clear()

    {

        while (numberOfElements != 0)

            PopFront();

    }

    iterator Begin()

    {

        iterator it;

        it.pointer = pFirst;

        return it;

    }

    iterator End()

    {

        iterator it;

        if (IsEmpty()) {

            it.pointer = pFirst;

            return it;

        }

        it.pointer = pCurrent->pNext;

        return it;

    }

    iterator penultimate()

    {

        iterator it;

        if (IsEmpty()) {

            it.pointer = pFirst;

            return it;

        }

        it.pointer = pCurrent;

        return it;

    }

    int GetSize()

    {

        return numberOfElements;

    }

    bool IsEmpty()

    {

        if (pFirst == nullptr)

            return true;

        else return false;

    }

    Node<type>\* GetHead() { return pFirst; }

    ~List()

    {

        Clear();

    }

    friend class iterator;

};

**Модуль Polynom.h**

#ifndef \_\_POLYNOM\_H\_\_

#define \_\_POLYNOM\_H\_\_

#include "List.h"

const int DEG\_MAX = 10;

struct Monom

{

    double koef;

    int deg;

    string name;

    Monom(double \_k = 1, int \_d = 0)

    {

        koef = \_k;

        deg = \_d;

    }

    Monom(double \_k, int \_d, string \_name)

    {

        koef = \_k;

        deg = \_d;

        name = \_name;

    }

    Monom operator +(const Monom& mn)

    {

        if (this->deg != mn.deg)

            throw "incorrect addition";

        Monom tmp;

        tmp.koef = this->koef + mn.koef;

        tmp.deg = this->deg;

        tmp.name = this->name;

        return tmp;

    }

    Monom operator -(const Monom& mn)

    {

        if (this->deg != mn.deg)

            throw "incorrect subtraction";

        Monom tmp;

        tmp.koef = this->koef - mn.koef;

        tmp.deg = this->deg;

        tmp.name = this->name;

        return tmp;

    }

    Monom operator \*(const Monom& mn)

    {

        Monom tmp;

        tmp.koef = this->koef \* mn.koef;

        tmp.deg = this->deg + mn.deg;

        int a = tmp.deg / 100, b = (tmp.deg / 10) % 10, c = tmp.deg % 10;

        string \_a, \_b, \_c;

        \_a = to\_string(a);

        \_b = to\_string(b);

        \_c = to\_string(c);

        string \_name = "";

        if ((a != 0) && (a != 1))

            \_name = \_name + "x" + \_a;

        if ((b != 0) && (b != 1))

            \_name = \_name + "y" + \_b;

        if ((c != 0) && (c != 1))

            \_name = \_name + "z" + \_c;

        if (a == 1)

            \_name = \_name + "x";

        if (b == 1)

            \_name = \_name + "y";

        if (c == 1)

            \_name = \_name + "z";

        tmp.name = \_name;

        return tmp;

    }

    int getDegX()

    {

        return deg / 100;

    }

    int getDegY()

    {

        return (deg / 10) % 10;

    }

    int getDegZ()

    {

        return deg % 10;

    }

};

class Polynom

{

private:

    string prefix;

    List<Monom> polynom;

    bool IsSign(char op);

    bool IsVariable(char op);

    int GetDegree(char p);

public:

    Polynom(List<Monom> \_polynom) {

        this->polynom = \_polynom;

    }

    Polynom(string \_str = "") :prefix(\_str) {}

    Polynom(const Polynom& pl);

    void Convert();

    int CalculateInPoint(int \_x, int \_y, int \_z);

    Polynom operator+(Polynom& pl);

    Polynom operator-(Polynom& pl);

    Polynom operator\*(Polynom& pl);

    Polynom& operator\*(const double& scal)

    {

        auto it = this->polynom.Begin();

        for (auto it = this->polynom.Begin(); it != this->polynom.End(); ++it)

        {

            it->data.koef = it->data.koef \* (scal);

        }

        return \*this;

    }

    Polynom& operator=(const Polynom& pl);

    friend ostream& operator<<(ostream& os, Polynom& pl);

    friend istream& operator>>(istream& is, Polynom& pl);

    bool operator == (const Polynom& pl)const;

    Polynom Sort();

    string GetPrefix();

    Polynom& IntegrateDX(Polynom& p1);

    Polynom& IntegrateDY(Polynom& p1);

    Polynom& IntegrateDZ(Polynom& p1);

    string nameNewMonom(int \_deg);

    Polynom& DerivativeDX(Polynom& p1);

    Polynom& DerivativeDY(Polynom& p1);

    Polynom& DerivativeDZ(Polynom& p1);

    string CreateString();

};

#endif

**Модуль Polynom.cpp**

#include "polynom.h"

#include "windows.h"

bool Polynom::IsSign(char op)

{

    if (op == '+' || op == '-')

        return true;

    else return false;

}

bool Polynom::IsVariable(char op)

{

    if (op == 'x' || op == 'y' | op == 'z')

        return true;

    else return false;

}

int Polynom::GetDegree(char p)

{

    if (int(p) > 49 && int(p) < 58)

        return int(p - 48);

    else return 1;

}

Polynom::Polynom(const Polynom& pl)

{

    this->prefix = pl.prefix;

    this->polynom = pl.polynom;

}

void Polynom::Convert()

{

    cout << "Convert" << endl;

    int begin = 0;

    bool sign = true;

    Monom monom;

    string num;

    List<Monom>::iterator it;

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

    {

        monom = {};

        if (prefix[i] == '+')

        {

            sign = true;

            i++;

        }

        else if (prefix[i] == '-')

        {

            sign = false;

            i++;

        }

        if (!IsSign(prefix[i]) && !IsVariable(prefix[i]))

        {

            while (!IsVariable(prefix[i]) && i != prefix.size() && !IsSign(prefix[i]))

            {

                num += prefix[i];

                i++;

            }

            monom.koef = atof(num.c\_str());

            num = "";

        }

        begin = i;

        if (IsVariable(prefix[i]))

        {

            int x = 0, y = 0, z = 0, sum;

            while (!IsSign(prefix[i]) && i != prefix.size())

            {

                switch (prefix[i])

                {

                case 'x':

                {

                    x = GetDegree(prefix[i + 1]);

                    i++;

                    break;

                }

                case 'y':

                {

                    y = GetDegree(prefix[i + 1]);

                    i++;

                    break;

                }

                case 'z':

                {

                    z = GetDegree(prefix[i + 1]);

                    i++;

                    break;

                }

                default:

                    i++;

                }

            }

            i--;

            sum = x \* 100 + y \* 10 + z;

            monom.name = prefix.substr(begin, i - begin + 1);

            monom.deg = sum;

            if (!sign)

                monom.koef = monom.koef \* (-1);

        }

        if (polynom.IsEmpty())

        {

            polynom.PushBack(monom);

            continue;

        }

        for (it = polynom.Begin(); it != polynom.End(); ++it)

        {

            if (monom.deg == it->data.deg)

            {

                it->data.koef = it->data.koef + monom.koef;

                if (it->data.koef == 0)

                    polynom.Erase(it);

                break;

            }

            if (monom.deg > it->data.deg)

            {

                polynom.Insert(it, monom);

                break;

            }

        }

        if (it == polynom.End())

            polynom.PushBack(monom);

    }

}

int Polynom::CalculateInPoint(int \_x, int \_y, int \_z)

{

    int result = 0;

    for (auto it = polynom.Begin(); it != polynom.End(); ++it)

    {

        int x = it->data.deg / 100, y = (it->data.deg / 10) % 10, z = it->data.deg % 10;

        result += (pow(\_x, x) \* pow(\_y, y) \* pow(\_z, z)) \* it->data.koef;

    }

    return result;

}

Polynom Polynom::Sort()

{

    if (this->polynom.IsEmpty())

        throw "polynom is empty";

    Polynom res;

    List<Monom>::iterator right, left, max;

    for (right = this->polynom.Begin(); right != this->polynom.End(); ++right)

    {

        max->data = right->data;

        for (left = right; left != this->polynom.End(); ++left)

        {

            if (right->data.deg < left->data.deg)

            {

                max->data = left->data;

                left->data = right->data;

                right->data = max->data;

            }

        }

    }

    return \*this;

}

Polynom Polynom::operator+(Polynom& pl)

{

    if (this->polynom.IsEmpty() || pl.polynom.IsEmpty())

        throw "polynom is empty";

    Polynom result;

    List<Monom>::iterator right, left;

    bool check = false;

    for (right = pl.polynom.Begin(); right != pl.polynom.End(); ++right)

    {

        check = false;

        for (left = this->polynom.Begin(); left != this->polynom.End(); ++left)

            if (right->data.deg == left->data.deg)

            {

                if ((left->data.koef + right->data.koef) != 0)

                    result.polynom.PushBack(\*(left)+\*(right));

                check = true;

            }

        if (check == false)

            result.polynom.PushBack(\*right);

    }

    check = false;

    for (left = this->polynom.Begin(); left != this->polynom.End(); ++left)

    {

        check = false;

        right = pl.polynom.Begin();

        for (right = pl.polynom.Begin(); right != pl.polynom.End(); ++right)

            if (left->data.deg == right->data.deg)

                check = true;

        if (check == false)

            result.polynom.PushBack(\*left);

    }

    if (result.polynom.IsEmpty())

    {

        right = pl.polynom.Begin();

        right->data.koef = 0;

        right->data.name = "";

        right->data.deg = 0;

        result.polynom.PushBack(\*right);

    }

    return result.Sort();

}

Polynom Polynom::operator-(Polynom& pl)

{

    if (this->polynom.IsEmpty() || pl.polynom.IsEmpty())

        throw "polynom is empty";

    Polynom result;

    List<Monom>::iterator right, left;

    bool check = false;

    for (right = pl.polynom.Begin(); right != pl.polynom.End(); ++right)

    {

        check = false;

        for (left = this->polynom.Begin(); left != this->polynom.End(); ++left)

            if (right->data.deg == left->data.deg)

            {

                if ((left->data.koef - right->data.koef) != 0)

                {

                    result.polynom.PushBack(\*(left)-\*(right));

                }

                check = true;

            }

        if (check == false)

        {

            int koef = right->data.koef;

            right->data.koef = (-1) \* right->data.koef;

            result.polynom.PushBack(\*right);

            right->data.koef = koef;

        }

    }

    check = false;

    for (left = this->polynom.Begin(); left != this->polynom.End(); ++left)

    {

        check = false;

        right = pl.polynom.Begin();

        for (right = pl.polynom.Begin(); right != pl.polynom.End(); ++right)

            if (left->data.deg == right->data.deg)

                check = true;

        if (check == false)

            result.polynom.PushBack(\*left);

    }

    return result.Sort();

}

Polynom Polynom::operator\*(Polynom& pl)

{

    if (this->polynom.IsEmpty() || pl.polynom.IsEmpty())

        throw "polynom is empty";

    Polynom result;

    List<Monom>::iterator right, left;

    for (right = this->polynom.Begin(); right != this->polynom.End();)

    {

        for (left = pl.polynom.Begin(); left != this->polynom.End();)

        {

            if ((right->data.koef \* left->data.koef) != 0)

                result.polynom.PushBack((\*(right)) \* (\*(left)));

            ++left;

        }

        ++right;

    }

    while (left != pl.polynom.End())

    {

        result.polynom.PushBack(\*left);

        ++left;

    }

    return result.Sort();

}

Polynom& Polynom::operator=(const Polynom& pl)

{

    this->polynom = pl.polynom;

    this->prefix = pl.prefix;

    return \*this;

}

bool Polynom::operator==(const Polynom& pl)const

{

    if (this->prefix == pl.prefix)

        return true;

    else return false;

}

ostream& operator<<(ostream& os, Polynom& pl)

{

    if (pl.polynom.IsEmpty()) {

        os << pl.prefix;

    }

    else

    {

        for (auto it = pl.polynom.Begin(); it != pl.polynom.End(); ++it)

        {

            if (it->data.deg != 0)

            {

                if (it != pl.polynom.Begin() && it->data.koef > 0)

                    os << '+';

                if ((it->data.koef != 1) && (it->data.koef != -1))

                {

                    os << it->data.koef;

                    os << it->data.name;

                }

                if (it->data.koef == 1)

                    os << it->data.name;

                if (it->data.koef == -1)

                {

                    os << '-';

                    os << it->data.name;

                }

            }

            else

            {

                if (it != pl.polynom.Begin() && it->data.koef > 0)

                    os << '+';

                os << it->data.koef;

            }

        }

    }

    return os;

}

istream& operator>>(istream& is, Polynom& pl)

{

    for (auto it = pl.polynom.Begin(); it != pl.polynom.End(); ++it)

    {

        if ((it->data.koef != 1) && (it->data.name != "\0"))

        {

            is >> it->data.koef;

            is >> it->data.name;

        }

        if (it->data.name == "\0")

            is >> it->data.koef;

        if (it->data.koef == 1)

            is >> it->data.name;

    }

    return is;

}

string Polynom::GetPrefix()

{

    return this->prefix;

}

Polynom& Polynom::IntegrateDX(Polynom& p1)

{

    List <Monom> result;

    for (auto it = p1.polynom.Begin(); it != p1.polynom.End(); ++it)

    {

        cout << "������ � IntegrateDX ������� result " << endl;

        int degMonom = it->data.deg + 100;

        int newDegX = degMonom / 100;

        double koefpolynom = it->data.koef;

        double newKoefpolynom = koefpolynom / newDegX;

        string nameMonom = p1.nameNewMonom(degMonom);

        cout << degMonom << " " << newKoefpolynom << endl;

        Monom newMonom(newKoefpolynom, degMonom, nameMonom);

        result.PushBack(newMonom);

    }

    this->polynom = result;

    return \*this;

}

Polynom& Polynom::IntegrateDY(Polynom& p1)

{

    List <Monom> result;

    for (auto it = p1.polynom.Begin(); it != p1.polynom.End(); ++it)

    {

        cout << "������ � IntegrateDY ������� result " << endl;

        int degMonom = it->data.deg + 10;

        int newDegY = (degMonom / 10) % 10;

        double koefpolynom = it->data.koef;

        double newKoefpolynom = koefpolynom / newDegY;

        string nameMonom = p1.nameNewMonom(degMonom);

        cout << degMonom << " " << newKoefpolynom << endl;

        Monom newMonom(newKoefpolynom, degMonom, nameMonom);

        result.PushBack(newMonom);

    }

    this->polynom = result;

    return \*this;

}

Polynom& Polynom::IntegrateDZ(Polynom& p1)

{

    List <Monom> result;

    for (auto it = p1.polynom.Begin(); it != p1.polynom.End(); ++it)

    {

        cout << "������ � IntegrateDZ ������� result " << endl;

        int degMonom = it->data.deg + 1;

        int newDegZ = degMonom % 10;

        double koefpolynom = it->data.koef;

        double newKoefpolynom = koefpolynom / newDegZ;

        string nameMonom = p1.nameNewMonom(degMonom);

        cout << degMonom << " " << newKoefpolynom << endl;

        Monom newMonom(newKoefpolynom, degMonom, nameMonom);

        result.PushBack(newMonom);

    }

    this->polynom = result;

    return \*this;

}

string Polynom::nameNewMonom(int \_deg)

{

    int a = \_deg / 100;

    int b = (\_deg / 10) % 10;

    int c = \_deg % 10;

    string \_name = "";

    if ((a != 0) && (a != 1))

        \_name = \_name + "x" + to\_string(a);

    if (a == 1)

        \_name = \_name + "x";

    if ((b != 0) && (b != 1))

        \_name = \_name + "y" + to\_string(b);

    if (b == 1)

        \_name = \_name + "y";

    if ((c != 0) && (c != 1))

        \_name = \_name + "z" + to\_string(c);

    if (c == 1)

        \_name = \_name + "z";

    return \_name;

}

Polynom& Polynom::DerivativeDX(Polynom& p1)

{

    List <Monom> result;

    for (auto it = p1.polynom.Begin(); it != p1.polynom.End(); ++it)

    {

        cout << "������ � DerivativeDX ������� result " << endl;

        double newKoef = it->data.koef \* (it->data.deg / 100);

        int newDegMonom = it->data.deg - 100;

        string nameMonom = p1.nameNewMonom(newDegMonom);

        cout << newDegMonom << " " << newKoef << endl;

        Monom newMonom(newKoef, newDegMonom, nameMonom);

        result.PushBack(newMonom);

    }

    this->polynom = result;

    return \*this;

}

Polynom& Polynom::DerivativeDY(Polynom& p1)

{

    List <Monom> result;

    for (auto it = p1.polynom.Begin(); it != p1.polynom.End(); ++it)

    {

        cout << "������ � DerivativeDY ������� result " << endl;

        double newKoef = it->data.koef \* ((it->data.deg / 10) % 10);

        int newDegMonom = it->data.deg - 10;

        string nameMonom = p1.nameNewMonom(newDegMonom);

        cout << newDegMonom << " " << newKoef << endl;

        Monom newMonom(newKoef, newDegMonom, nameMonom);

        result.PushBack(newMonom);

    }

    this->polynom = result;

    return \*this;

}

Polynom& Polynom::DerivativeDZ(Polynom& p1)

{

    List <Monom> result;

    for (auto it = p1.polynom.Begin(); it != p1.polynom.End(); ++it)

    {

        double newKoef = it->data.koef \* (it->data.deg % 10);

        int newDegMonom = it->data.deg - 1;

        string nameMonom = p1.nameNewMonom(newDegMonom);

        cout << newDegMonom << " " << newKoef << endl;

        Monom newMonom(newKoef, newDegMonom, nameMonom);

        result.PushBack(newMonom);

    }

    this->polynom = result;

    return \*this;

}

string Polynom::CreateString()

{

    string Result;

    int powX, powY, powZ;

    double mulMonom;

    auto it = this->polynom.Begin();

    if (it->pNext != nullptr)

    {

        powX = it->data.deg / 100;

        powY = (it->data.deg / 10) % 10;

        powZ = it->data.deg % 10;

        mulMonom = it->data.koef;

        if (mulMonom == -1)

            Result += '-';

        else

            if (mulMonom != 1)

                Result += to\_string(mulMonom);

        if (powX)

            if (powX != 1)

                Result = Result + "x" + to\_string(powX);

            else

                Result += 'x';

        if (powY)

            if (powY != 1)

                Result = Result + "y" + to\_string(powY);

            else

                Result += 'y';

        if (powZ)

            if (powZ != 1)

                Result = Result + "z" + to\_string(powZ);

            else

                Result += 'z';

        while (it->pNext != nullptr)

        {

            ++it;

            powX = it->data.deg / 100;

            powY = (it->data.deg / 10) % 10;

            powZ = it->data.deg % 10;

            mulMonom = it->data.koef;

            if (mulMonom == -1)

                Result += '-';

            if (mulMonom > 0)

                Result += '+';

            if ((mulMonom != 1) && (mulMonom != -1))

                Result += to\_string(mulMonom);

            if (powX)

                if (powX != 1)

                    Result = Result + "x" + to\_string(powX);

                else

                    Result += 'x';

            if (powY)

                if (powY != 1)

                    Result = Result + "y" + to\_string(powY);

                else

                    Result += 'y';

            if (powZ)

                if (powZ != 1)

                    Result = Result + "z" + to\_string(powZ);

                else

                    Result += 'z';

        }

    }

    return Result;

}

**Модуль BaseTable.h**

#ifndef \_\_BASETABLE\_H\_\_

#define \_\_BASETABLE\_H\_\_

#include <vector>

#include "Polynom.h"

struct Data

{

    std::string key;

    std::string polynomString;

    Polynom \_polynom;

};

class BaseTable

{

public:

    virtual Data\* find(const std::string key) = 0;

    virtual void insert(const Data& data) = 0;

    virtual void deleteByKey(const std::string& key) = 0;

    virtual void print() = 0;

    virtual ~BaseTable() {}

};

#endif

**Модуль HashTableOnList.h**

#pragma once

#ifndef \_\_HASH\_TABLE\_ON\_LIST\_H\_\_

#define \_\_HASH\_TABLE\_ON\_LIST\_H\_\_

#include "BaseTable.h"

using namespace std;

struct link

{

    Data data;

    link\* pNext;

};

class HashTableOnList : public BaseTable

{

    link\* Tab[1000];

public:

    HashTableOnList()

    {

        for (int i = 0; i < 1000; i++) { Tab[i] = nullptr; }

    }

    ~HashTableOnList();

    Data\* find(const string key);

    void insert(const Data& data);

    void deleteByKey(const string& key);

    void print();

};

#endif

**Модуль HashTableOnList.cpp**

#include "HashTableOnList.h"

#include "BaseTable.cpp"

int HashIt(string key)

{

    int Hash = 0;

    for (unsigned int i = 0; i < key.size(); i++) { Hash += (int)key[i]; }

    Hash \*= key.size();

    return Hash % 1000;

}

HashTableOnList::~HashTableOnList()

{

    for (int i = 0; i < 1000; i++)

        if (Tab[i] != nullptr)

            delete Tab[i];

}

Data\* HashTableOnList::find(const string key)

{

    int hash = HashIt(key);

    if (Tab[hash] == nullptr)

        return nullptr;

    else

    {

        link\* p = Tab[hash];

        while (p != nullptr)

        {

            if (p->data.key == key)

                return &p->data;

            p = p->pNext;

        }

    }

    return nullptr;

}

void HashTableOnList::insert(const Data& data)

{

    if (find(data.key) != nullptr)

        throw exception();

    int hash = HashIt(data.key);

    link\* p = new link;

    p->pNext = nullptr;

    p->data = data;

    if (Tab[hash] == nullptr)

    {

        Tab[hash] = p;

        return;

    }

    else

    {

        link\* pp = Tab[hash];

        while (pp->pNext != nullptr)

            pp = pp->pNext;

        pp->pNext = p;

        return;

    }

}

void HashTableOnList::deleteByKey(const string& key)

{

    if (find(key) == nullptr)

        throw exception();

    int hash = HashIt(key);

    link\* p = Tab[hash];

    link\* pp = p;

    if (p->data.key == key)

    {

        pp = p->pNext;

        delete p;

        Tab[hash] = pp;

        p = pp;

        return;

    }

    p = p->pNext;

    while (p->data.key != key)

    {

        if (p->data.key == key)

        {

            while (pp->pNext != p)

                pp = pp->pNext;

            pp->pNext = p->pNext;

            delete p;

            p = pp;

            return;

        }

        p = p->pNext;

    }

}

void HashTableOnList::print()

{

    int k = 0;

    link\* p;

    std::cout << "\t��� ������� �� ����� " << std::endl;

    for (int i = 0; i < 1000; i++)

    {

        p = Tab[i];

        while (p != nullptr)

        {

            cout << p->data.key << ":\t" << p->data.\_polynom << endl;

            p = p->pNext;

            k++;

        }

    }

    if (k == 0) { cout << "Empty" << endl; }

}

**Модуль HashTableSomething.h**

#pragma once

#ifndef \_\_HASHTABLIN\_H\_\_

#define \_\_HASHTABLIN\_H\_\_

#include "BaseTable.h"

class HashTableSomething : public BaseTable

{

    Data\* Tab[1000];

public:

    HashTableSomething()

    {

        for (int i = 0; i < 1000; i++)

        {

            Tab[i] = nullptr;

        }

    }

    ~HashTableSomething();

    Data\* find(const std::string key);

    void insert(const Data& data);

    void deleteByKey(const string& key);

    void print();

    int HashIt(string key)

    {

        int Hash = 0;

        for (unsigned int i = 0; i < key.size(); i++) { Hash += (int)key[i]; }

        Hash \*= key.size();

        return Hash % 1000;

    }

};

#endif

**Модуль HashTableSomething.cpp**

#include "HashTableSomething.h"

#include "BaseTable.cpp"

HashTableSomething::~HashTableSomething()

{

    for (int i = 0; i < 1000; i++) {

        if (Tab[i] != nullptr) { delete Tab[i]; }

    }

}

Data\* HashTableSomething::find(const std::string key) {

    int SearchHash = HashIt(key);

    if (Tab[SearchHash] == nullptr) {

        for (int i = 0; i < 10; i++) {

            SearchHash = (SearchHash + 7) % 1000;

            if (Tab[SearchHash] != nullptr)

                if (Tab[SearchHash]->key == key)

                    return Tab[SearchHash];

        }

        return nullptr;

    }

    else {

        if (Tab[SearchHash]->key == key) { return Tab[SearchHash]; }

        else {

            for (int i = 0; i < 10; i++) {

                SearchHash = (SearchHash + 7) % 1000;

                if (Tab[SearchHash] != nullptr) {

                    if (Tab[SearchHash]->key == key) { return Tab[SearchHash]; }

                }

            }

            return nullptr;

        }

    }

}

void HashTableSomething::insert(const Data& data)

{

    if (find(data.key) != nullptr) { throw exception(); }

    int InsertHash = HashIt(data.key);

    if (Tab[InsertHash] == nullptr) {

        Tab[InsertHash] = new Data;

        Tab[InsertHash]->key = data.key;

        Tab[InsertHash]->polynomString = data.polynomString;

        Tab[InsertHash]->\_polynom = data.\_polynom;

        return;

    }

    else {

        if (Tab[InsertHash]->polynomString == data.polynomString) { throw exception(); }

        for (int i = 0; i < 10; i++) {

            InsertHash = (InsertHash + 7) % 1000;

            if (Tab[InsertHash] == nullptr) {

                Tab[InsertHash] = new Data;

                Tab[InsertHash]->key = data.key;

                Tab[InsertHash]->polynomString = data.polynomString;

                Tab[InsertHash]->\_polynom = data.\_polynom;

                return;

            }

            else { if (Tab[InsertHash]->polynomString == data.polynomString) { throw exception(); } }

        }

        throw exception();

    }

}

void HashTableSomething::deleteByKey(const string& key) {

    if (find(key) == nullptr) { throw exception(); }

    Data\* p = find(key);

    int deleteHash = HashIt(p->key);

    delete Tab[deleteHash];

    Tab[deleteHash] = nullptr;

}

void HashTableSomething::print()

{

    cout << "\t���-�������" << endl;

    for (int i = 0; i < 1000; i++)

        if (Tab[i] != nullptr) { cout << Tab[i]->key << ":\t" << Tab[i]->polynomString << endl; }

}

**Модуль TInterface.h**

#ifndef \_\_INTERFACE\_H\_\_

#define \_\_INTERFACE\_H\_\_

#include "LineTable.h"

#include "UnorderedTableOnList.h"

#include "OrderedTableOnArray.h"

#include "SearchTreeTable.h"

#include "HashTableSomething.h"

#include "HashTableOnList.h"

#include <conio.h>

#include <iostream>

#include "Postfix.h"

class Interface

{

    BaseTable\* Base;

    BaseTable\* tables[6];

    string namesOfTables[6] = { "�������� ������� �� �������","�������� ������� �� ������","������������� ������� �� �������",

        "������� �� ��������� ������","���-������� � �������� �������������� (�� �������)","���-������� �� �������� (����� �������)" };

    string tecName;

public:

    Interface()

    {

        tables[0] = new LineTable();

        tables[1] = new UnorderedTableOnList();

        tables[2] = new OrderedTableOnArray();

        tables[3] = new SearchTreeTable();

        tables[4] = new HashTableSomething();

        tables[5] = new HashTableOnList();

        Base = tables[0];

        tecName = namesOfTables[0];

    }

    void countPolynomInPoint(Data data, int x, int y, int z);

    string controlPolynom(const string& str);

    int inputControl(int left\_board, int right\_board, void (\*print)(const string&), const string& str);

    void tableSelection();

    void menu();

    void addPolynom();

    void searchAndMakeOperations();

    string controlKey(const string& key);

    void print();

    void deleteByKey();

    void calculate();

    void calPol();

    void exitApp();

};

#endif

**Модуль LineTable.h**

#ifndef \_\_LINETABLE\_H\_\_

#define \_\_LINETABLE\_H\_\_

#include "BaseTable.h"

class LineTable : public BaseTable

{

    Data\* dat;

    int size;

    int capacity;

    void repacking();

public:

    LineTable() : BaseTable()

    {

        capacity = 1000;

        size = 0;

        dat = new Data[capacity];

    }

    void deleteByKey(const string& key);

    void insert(const Data& d);

    Data\* find(const string key);

    void print();

    ~LineTable();

};

#endif

**Модуль LineTable.cpp**

#include "LineTable.h"

void LineTable::repacking()

{

    capacity += 1000;

    Data\* temp = new Data[capacity];

    if (temp == nullptr)

    {

        throw 5;

    }

    for (int i = 0; i < size; i++)

    {

        temp[i] = dat[i];

    }

    delete[] dat;

    dat = temp;

}

void LineTable::deleteByKey(const string& key)

{

    if (size == 0)

    {

        throw exception();

    }

    Data\* d = find(key);

    if (d != nullptr)

    {

        Data temp = dat[size - 1];

        dat[size - 1] = \*d;

        \*d = temp;

        size--;

    }

    else

        throw exception();

}

void LineTable::insert(const Data& d)

{

    if (size == capacity)

    {

        repacking();

    }

    if (find(d.key) == nullptr)

    {

        dat[size] = d;

        size++;

    }

    else

        throw exception();

}

Data\* LineTable::find(const string key)

{

    for (int i = 0; i < size; i++)

    {

        if (dat[i].key == key)

        {

            return &dat[i];

        }

    }

    return nullptr;

}

void LineTable::print()

{

    std::cout << "\t �������� �� ������� " << std::endl;

    for (int i = 0; i < size; i++)

        cout << dat[i].key << ":\t" << dat[i].polynomString << endl;

    if (size == 0)

        cout << "Empty" << endl;

}

LineTable::~LineTable() { delete[] dat; }

**Модуль** **OrderedTableOnArray.h**

#pragma once

#ifndef \_\_ORDEREDTABLEONARRAY\_H\_\_

#define \_\_ORDEREDTABLEONARRAY\_\_

#include <string>

#include <iostream>

#include "BaseTable.h"

using namespace std;

class OrderedTableOnArray : public BaseTable

{

    Data\* Tab[1000];

    int size;

public:

    OrderedTableOnArray() :BaseTable() {

        for (int i = 0; i < 1000; i++) { Tab[i] = nullptr; }

        size = 0;

    }

    ~OrderedTableOnArray();

    Data\* find(const string key);

    void insert(const Data& data);

    void deleteByKey(const string& key);

    void print();

    int Compare(string k1, string k2);

};

#endif

**Модуль OrderedTableOnArray.cpp**

#include "OrderedTableOnArray.h"

OrderedTableOnArray::~OrderedTableOnArray()

{

    for (int i = 0; i < 1000; i++)

        if (Tab[i] != nullptr)

            delete Tab[i];

}

Data\* OrderedTableOnArray::find(const string key)

{

    int midd, left = 0, right = size - 1;

    while (left <= right)

    {

        midd = (left + right) / 2;

        if (Compare(Tab[midd]->key, key) == 0)

            return Tab[midd];

        if (Compare(Tab[midd]->key, key) < 0)

            left = midd + 1;

        if (Compare(Tab[midd]->key, key) > 0)

            right = midd - 1;

    }

    return nullptr;

}

void OrderedTableOnArray::insert(const Data& data)

{

    if (find(data.key) != nullptr) { throw 3; }

    string h = "";

    if (Tab[0] != nullptr)

        h = Tab[0]->key;

    int j = 0;

    while (j < size && (Compare(h, data.key) < 0))

    {

        j++;

        if (Tab[j] != nullptr)

            h = Tab[j]->key;

        else

            h = "";

    }

    if (Tab[j] == nullptr)

    {

        Tab[j] = new Data;

        Tab[j]->key = data.key;

        Tab[j]->\_polynom = data.\_polynom;

        Tab[j]->polynomString = data.polynomString;

    }

    else

    {

        Tab[size] = new Data;

        for (int i = size; i > j; i--)

            Tab[i] = Tab[i - 1];

        Tab[j] = new Data;

        Tab[j]->key = data.key;

        Tab[j]->\_polynom = data.\_polynom;

        Tab[j]->polynomString = data.polynomString;

    }

    size++;

    //throw exception();

}

void OrderedTableOnArray::deleteByKey(const string& key)

{

    if (find(key) == nullptr)

        throw exception();

    int midd, left = 0, right = size - 1;

    while (left <= right)

    {

        midd = (left + right) / 2;

        if (Compare(Tab[midd]->key, key) == 0)

        {

            delete Tab[midd];

            for (int i = midd; i < size; i++)

                Tab[i] = Tab[i + 1];

            size--;

            return;

        }

        if (Compare(Tab[midd]->key, key) < 0)

            left = midd + 1;

        if (Compare(Tab[midd]->key, key) > 0)

            right = midd - 1;

    }

}

void OrderedTableOnArray::print()

{

    int k = 0;

    string h;

    cout << "\t��������������� �������� ������� " << endl;

    for (int i = 0; i < size; i++)

    {

        cout << Tab[i]->key << ":\t" << Tab[i]->\_polynom << endl;

        k++;

    }

    if (k == 0)

        cout << "Empty" << endl;

}

int OrderedTableOnArray::Compare(string k1, string k2)

{

    string alf = "abcdefghijklmnopqrstuvwxyz";

    unsigned int i = 0;

    int check1 = 0, check2 = 0;

    while (i < k1.length() && i < k2.length())

    {

        if (k1[i] == k2[i])

            i++;

        else

        {

            for (unsigned int k = 0; k < alf.length(); k++)

            {

                if (k1[i] == alf[k])

                {

                    check1 = k + 1;

                    break;

                }

                if (k2[i] == alf[k])

                {

                    check2 = k + 1;

                    break;

                }

            }

            if (check1 > 0)

                return -1;

            if (check2 > 0)

                return 1;

        }

    }

    if (i == k1.length()) {

        if (i == k2.length()) { return 0; }

        else { return -1; }

    }

    else { return 1; }

}

**С реализацией остальных классов и тестов можно ознакомиться по ссылке: https://github.com/NastyaMortina/Lab\_1.Sem\_4/**