

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:

**«Реализация и использование различных структур  
таблиц для хранения и обработки полином»**

**Выполнил:** студентка группы  
3822Б1ФИ2

\_\_\_\_\_/Ясакова Т.Е. /  
Подпись

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

Подпись

Нижний Новгород  
2024

# Содержание

Введение.....	4
1 Постановка задачи.....	5
2 Руководство пользователя.....	6
2.1 Приложение для демонстрации работы таблиц.....	6
3 Руководство программиста .....	10
3.1 Описание алгоритмов .....	10
3.1.1 Неупорядоченная таблица.....	10
3.1.2 Упорядоченная таблица.....	12
3.1.3 Хэш-таблица .....	14
3.2 Описание программной реализации .....	16
3.2.1 Описание класса TabRecord .....	16
3.2.2 Описание класса Table.....	18
3.2.3 Описание класса ScanTable .....	19
3.2.4 Описание класса SortedTable .....	21
3.2.5 Описание класса HashTable.....	22
3.2.6 Описание класса ArrayHashTable .....	23
Заключение .....	26
Литература .....	27
Приложения .....	28
Приложение А. Реализация класса TabRecord .....	28
Приложение Б. Реализация класса Table .....	29
Приложение В. Реализация класса ScanTable .....	30
Приложение Г. Реализация класса SortedTable.....	32
Приложение Д. Реализация класса HashTable.....	34
Приложение Е. Реализация класса ArrayHashTable.....	34
Приложение Ж. Sample_table.....	38



## **Введение**

Лабораторная работа направлена на изучение и реализацию трех типов таблиц: упорядоченной, неупорядоченной и хэш-таблицы.

В таблицах будут содержаться полиномы, реализованные на базе предыдущей лабораторной работы. Таблицы должны обеспечить возможность работы над полиномами, например, сложение, умножение, вычитание, взятие производной внутри самих таблиц.

# 1 Постановка задачи

## **Цель:**

Цель работы – научиться работать с записями таблицы. Роль записей в таблицах будут играть полиномы на линейных односвязных списках, которые были реализованы в предыдущей лабораторной работе. Ключ к записи – это строковое представление полинома.

## **Задачи:**

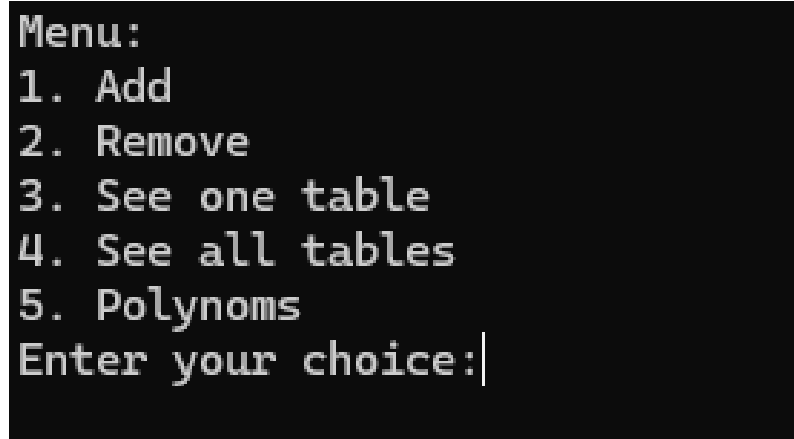
1. Изучение основных принципов работы с таблицами.
2. Создание упорядоченной, неупорядоченной и хэш-таблиц.
3. Предоставить возможность хранить, добавлять, удалять в таблицах и еще сортировать в упорядоченной.
4. Написание программы на C++.
5. Тестирование программы на различных входных данных.

Результатом выполнения лабораторной работы станет полнофункциональная реализация алгоритмов работы с таблицами, которые могут быть использованы для различных задач.

## 2 Руководство пользователя

### 2.1 Приложение для демонстрации работы таблиц

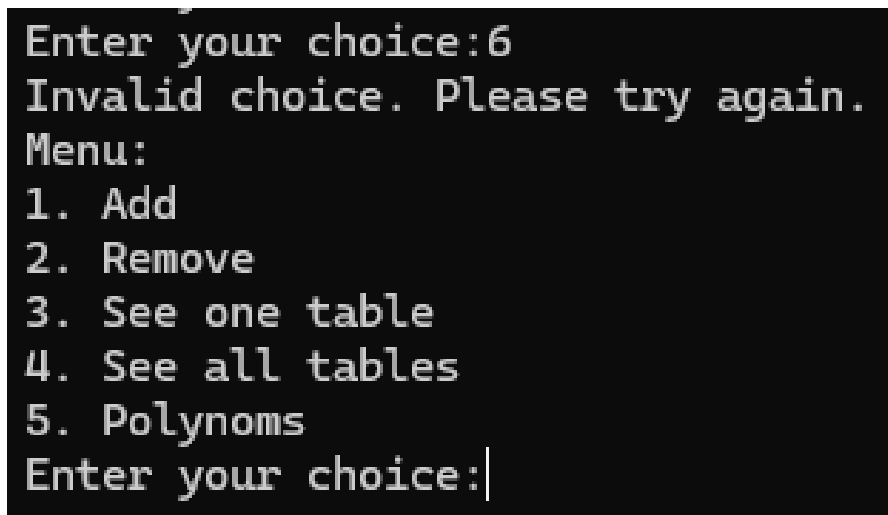
1. Запустите приложение с названием `sample_table.exe`. В результате появится окно, показанное ниже и вам будет предложено ввести число от 1 до 5. Это число повлияет на выбор операции (рис. 1).



```
Menu:
1. Add
2. Remove
3. See one table
4. See all tables
5. Polynoms
Enter your choice:|
```

Рис. 1. Основное окно программы

2. При вводе неверной команды, программа попросит ввести корректно (рис. 2).



```
Enter your choice:6
Invalid choice. Please try again.
Menu:
1. Add
2. Remove
3. See one table
4. See all tables
5. Polynoms
Enter your choice:|
```

Рис. 2. Проверка на корректный ввод

3. При выборе команды 1, нужно ввести полином и добавить его либо в одну из трех таблиц или во все (рис. 3):

```

Enter your choice:1
Enter polynom:x+1
1. Add to one table
2. Add to all tables
0. CANCEL
Enter:1
TABLES
1)Scan Table
2)Sorted Table
3)Array Hash Table
0)CANCEL
Enter:1

```

Рис. 3. Добавление полинома в таблицу

4. При выборе команды 2, нужно ввести полином, которых хотите удалить, выбрать из какой таблицы удалять (рис. 4).

```

Enter your choice:2
Enter polynom:x+1
1. Remove from one table
2. Remove from all tables
0. CANCEL
Enter:1
TABLES
1)Scan Table
2)Sorted Table
3)Array Hash Table
0)CANCEL
Enter:3
Table is empty

```

Рис. 4. Удаление полинома из таблицы

5. При выборе команды 3, нужно выбрать, какую из трех таблиц, требуется вывести (рис. 5).

```

Enter your choice:3
TABLES
1)Scan Table
2)Sorted Table
3)Array Hash Table
0)CANCEL
Enter:3
x+1      | 1.00+x
-x+1     | 1.00-x

```

Рис. 5. Вывод одной таблицы

6. При выборе команды 4 будут выведены все таблицы (рис. 6):

```

Enter your choice:4
ScanTable
x+1      | 1.00+x
-x+1     | 1.00-x

SortedTable
-x+1     | 1.00-x
x+1      | 1.00+x

ArrayHashTable
x+1      | 1.00+x
-x+1     | 1.00-x

```

Рис. 6. Вывод таблиц

7. При выборе команды 5, нужно ввести полиномы, которые есть в таблицах, с которыми вы хотите работать. Будут представлены операции, которые можно произвести над полиномами. Результат можно добавить в таблицу (рис. 7):



```
Enter your choice:5
Enter first polynomial: x+1
Enter second polynomial: -x+1
1)+
2)-
3)*
4)dx
5)dy
6)dz
Enter:4
1.00
1. Add to one table
2. Add to all tables
0. CANCEL
Enter:2
```

Рис. 7. Операции над полиномами

## 3 Руководство программиста

### 3.1 Описание алгоритмов

#### 3.1.1 Неупорядоченная таблица

Неупорядоченная таблица представлена массивом указателей на записи, где каждая запись представляет собой пару ключ-значение. Неупорядоченная таблица поддерживает операции поиска, операции и удаления, проверки на пустоту и полноту. Структура данных хранит элементы в первых  $n$  ячейках массива записей, имеет указатель на последний не занятый элемент (в случае полноты он будет выходить за пределы массива).

##### Операция добавления

Операция добавления элемента реализуется при помощи указателя на первый свободный элемент. В свободную ячейку вставляется переданная запись, количество записей в таблице увеличивается. Если этот элемент уже был в таблице, запись перезаписывается.

Пример:

2	4	
one	two	

Операция добавления элемента five с ключом 3:

2	4	3
one	two	five

##### Операция удаления

Операция удаления элемента реализуется при помощи перебора всех элементов, пока не будет найден искомый. Удаляем найденный элемент, уменьшая количество элементов в таблице. На его место ставим последний элемент.

Пример:

2	4	
one	two	

Операция удаления элемента с ключом 2:

4		
two		

## Операция поиска

Операция поиска элемента реализуется при помощи перебора всех элементов, пока не будет найден искомый. Если искомый элемент не был найден, то функция вернёт нулевой указатель, иначе вернет указатель на элемент.

Пример:

2	4	
one	two	

Операция поиска элемента с ключом 2:

2
one

## Операция проверки на пустоту

Операция проверки на пустоту реализована при помощи переменной, отвечающей за текущее количество записей в таблице. Если она равна нулю, то таблица пуста, иначе в ней имеются записи.

Пример:

2	4	
one	two	

Операция проверки на пустоту:

Результат: false

## Операция проверки на полноту

Операция проверки на полноту реализована при помощи переменной, отвечающей за текущее количество записей в таблице. Если она равна максимальному количеству записей в таблице, то таблица полна.

Пример:

2	4	1
one	two	five

Операция проверки на полноту:

Результат: true

### 3.1.2 Упорядоченная таблица

Упорядоченная таблица представлена массивом указателей на записи, где каждая запись представляет собой пару ключ-значение. Упорядоченная таблица поддерживает операции поиска, операции и удаления, проверки на пустоту и полноту. Структура данных хранит элементы в первых  $n$  ячейках массива записей в отсортированном по ключам по не убыванию. Поддержка элементов в отсортированном порядке позволяет быстрее искать элемент в таблице при помощи алгоритма «Бинарный поиск».

#### Функция хэширования

Алгоритм возвращает хэш-код ключа, используя стандартную хэш-функцию. Затем полученный хэш-код делится на максимальное число элементов таблицы, что позволяет определить индекс ячейки хэш-таблицы, где будет храниться информация с данным ключом. Этот процесс обеспечивает равномерное распределение данных по таблице и быстрый доступ к записям.

#### Операция добавления

Операция добавления элемента реализуется при помощи поиска алгоритма «Бинарный поиск», который находит позицию для искомого элемента и добавляет туда переданную запись, увеличивая количество элементов в таблице. Все элементы, имеющие индекс больше, сдвигаются вправо. Если этот элемент уже был в таблице, запись перезаписывается.

Пример:

2	4	
one	two	

Операция добавления элемента two с ключом 3:

2	3	4
one	two	five

#### Операция удаления

Операция удаления элемента реализуется при помощи алгоритма «Бинарный поиск», который находит элемент за оптимальное время. Элемент удаляется и уменьшается текущее количество записей в таблице. Если элемент не был последним в таблице, все записи, у которых позиция больше, чем у искомого, будут сдвинуты влево.

Пример:

2	4	
one	two	

Операция удаления элемента с ключом 2:

4		
two		

### Операция поиска

Операция поиска элемента в упорядоченной таблице реализуется с помощью «Бинарного поиска». Алгоритм «Бинарного поиска» в отсортированных таблицах использует стратегию деления интервала пополам, что позволяет эффективно находить элементы. Сравнивая ключ с целевым значением и последовательно сужая интервал поиска, алгоритм быстро находит искомый элемент и возвращает указатель на него или возвращает нулевой указатель, если элемент не найден.

Пример:

2	4	
one	two	

Операция поиска элемента с ключом 2:

2
one

### Операция проверки на пустоту

Операция проверки на пустоту реализована при помощи переменной, отвечающей за текущее количество записей в таблице. Если она равна нулю, то таблица пуста, иначе в ней имеются записи.

Пример:

2	4	
one	two	

Операция проверки на пустоту:

Результат: false

### Операция проверки на полноту

Операция проверки на полноту реализована при помощи переменной, отвечающей за текущее количество записей в таблице. Если она равна максимальному количеству записей в таблице, то таблица полна.

Пример:

2	4	1
one	two	five

Операция проверки на полноту:

Результат: true

### 3.1.3 Хэш-таблица

Хэш-таблица представлена массивом указателей на записи, где каждая запись представляет собой пару ключ-значение и наличием указателя на фиктивный элемент, который будет означать, что текущий элемент был удалён. Этот указатель нужен для обработки коллизий с помощью подхода «открытое перемешивание». В таблице реализована функция хэширования, которая должна ускорить поиск элемента. Структура данных хранит элементы хаотичном порядке, согласно хэш-функции.

#### Операция добавления

Операция добавления элемента реализуется при помощи хэш-функции, которая считает позицию для искомого элемента. Если текущая ячейка занята каким-то другим элементом, то проверяются все последующие элементы таблицы, используя линейный сдвиг, пока не найдем пустую ячейку, либо не вернёмся обратно. Если структура хранения не полна, то добавляем новый элемент в пустую ячейку. Если этот элемент уже был в таблице, запись перезаписывается.

Пример:

2		4
one		two

Операция добавления элемента five с ключом 3:

2	3	4
one	five	two

#### Операция удаления

Операция удаления элемента реализуется при помощи алгоритма «Бинарный поиск», который находит элемент за оптимальное время. Если запись найдена, она удаляется, освобождая память, и количество записей в таблице уменьшается.

Пример:

2		4
one		two

Операция удаления элемента с ключом 4:

2		
one		

### Операция поиска

Операция поиска элемента реализуется при помощи хэш-функции, которая считает позицию для искомого элемента. Если текущая ячейка занята каким-то другим элементом, то проверяются все последующие элементы таблицы, используя линейный сдвиг, пока либо не упрёмся в ячейку, в которой никто не был, либо не вернёмся обратно. Если искомым элемент не был найден, возвращаем пустой указатель, иначе – указатель на запись.

Пример:

2		4
one		two

Операция поиска элемента с ключом 2:

2
one

### Операция проверки на пустоту

Операция проверки на пустоту реализована при помощи переменной, отвечающей за текущее количество записей в таблице. Если она равна нулю, то таблица пуста, иначе в ней имеются записи.

Пример:

2	4	
one	two	

Операция проверки на пустоту:

Результат: false

## Операция проверки на полноту

Операция проверки на полноту реализована при помощи переменной, отвечающей за текущее количество записей в таблице. Если она равна максимальному количеству записей в таблице, то таблица полна.

Пример:

2	4	1
one	two	five

Операция проверки на полноту: результат: true

## 3.2 Описание программной реализации

### 3.2.1 Описание класса TabRecord

```
template <typename TKey, typename TData>
class TabRecord {
protected:
    TKey key;
    TData* data;
public:
    TabRecord() : key(), data(nullptr) {}
    TabRecord(const TKey& key, TData* data);
    TabRecord(const TabRecord<TKey, TData>& record);
    ~TabRecord();
    TKey GetKey() const;
    TData* GetData() const;
    const TabRecord<TKey, TData>& operator=(const TabRecord<TKey, TData>&
record);
    bool operator==(const TabRecord<TKey, TData>& record) const;
    bool operator!=(const TabRecord<TKey, TData>& record) const;
};
```

Назначение: представление записи в таблицы.

Поля:

**data**— указатель на массив типа **TData**.

**key** — ключ к записи.

Методы:

**TabRecord()** ;

Назначение: конструктор по умолчанию.

Входные параметры: отсутствуют.

Выходные параметры: новая запись с инициализированными значениями.

**TabRecord(const TKey& key, TData\* data);**

Назначение: инициализация значения **data** записи и ключа **key** к этой записи.



Входные параметры: **d** – значение **data**, **k** – значение **key**.

Выходные параметры: новая запись с инициализированным значением **data** и **key**.

**TabRecord(const TabRecord<TKey, TData>& record);**

Назначение: конструктор копирования.

Входные параметры: **record** – запись, на основе которой создаем копию.

Выходные параметры: отсутствуют.

**~TabRecord();**

Назначение: деструктор.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TKey GetKey() const;**

Назначение: получение значения ключа.

Входные параметры: отсутствуют.

Выходные параметры: ключ.

**TData\* GetData() const;**

Назначение: получение значения данных записи.

Входные параметры: отсутствуют.

Выходные параметры: данные.

**const TabRecord<TKey, TData>& operator=(const TabRecord<TKey, TData>& record);**

Назначение: оператор присваивания.

Входные параметры: **record** – ссылка на другой объект, из которого будут скопированы данные.

Выходные параметры: ссылка на объект.

**bool operator==(const TabRecord<TKey, TData>& record) const;**

Назначение: сравнение на равенство.

Входные параметры: **record** – ссылка на другой объект для сравнения.

Выходные параметры: **true** – объекты равны, **false** – иначе.

**bool operator!=(const TabRecord<TKey, TData>& record) const;**

Назначение: сравнение на неравенство.

Входные параметры: **record** – ссылка на другой объект для сравнения.

Выходные параметры: **true** – объекты не равны, **false** – иначе.

### 3.2.2 Описание класса Table

```
template <typename TKey, typename TData>
class Table {
protected:
    int count;
    int maxSize;
    int currPos;
public:
    Table(int maxSize);
    virtual ~Table() {}
    virtual TabRecord<TKey, TData>* Find(TKey key) = 0;
    virtual void Insert(TKey key, TData* data) = 0;
    virtual void Remove(TKey key) = 0;
    virtual TabRecord<TKey, TData>* GetCurrent() const = 0;
    virtual bool IsFull() const;
    virtual bool IsEmpty() const;
    virtual void Reset();
    virtual void Next();
    virtual bool IsEnded() const;
};
```

Назначение: абстрактный класс для представления различных видов таблиц

Поля:

**count** – количество элементов в таблице.

**maxSize** – максимальное число элементов в таблицы.

**currPos** – индекс текущего элемента.

Методы:

**Table(int maxSize);**

Назначение: конструктор с параметрами.

Входные параметры: **maxSize** – максимальное количество элементов в таблице.

Выходные параметры: отсутствуют.

**virtual ~Table();**

Назначение: деструктор.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**virtual TTabRecord <TKey, TData>\* Find(const TKey& key) = 0;**

Назначение: поиск записи в таблице по ключу.

Входные параметры: **key** – ключ, который ищем.

Выходные параметры: указатель на запись. Если запись не была найдена, вернётся нулевой указатель.

**virtual void Insert(const TKey& key, const TData& data) = 0;**

Назначение: вставка записи в таблицу.

Входные параметры: **key** – ключ, **data** – запись, которую хотим вставить.

Выходные параметры: отсутствуют.

**virtual void Remove(const TKey& key) = 0;**

Назначение: удаление записи из таблицы.

Входные параметры: **k** – ключ.

Выходные параметры: отсутствуют.

```
virtual TabRecord<TKey, TData>* GetCurrent() const = 0;
```

Назначение: получение текущей записи в таблице.

Входные параметры: отсутствуют.

Выходные параметры: указатель на запись.

```
virtual bool IsFull() const;
```

Назначение: проверка таблицы на полноту.

Входные параметры отсутствуют:

Выходные параметры: **true** – если таблица полна, **false** – в противном случае.

```
virtual bool IsEmpty() const;
```

Назначение: проверка на пустоту.

Входные параметры: отсутствуют.

Выходные параметры: **true** – если таблица пуста, **false** – в противном случае.

```
virtual bool Reset();
```

Назначение: ставит индекс текущего элемента на начало.

Входные параметры: отсутствуют.

Выходные параметры: **false**, если индекс вышел за пределы таблицы, иначе **true**.

```
virtual bool Next();
```

Назначение: переход к следующей записи.

Входные параметры: отсутствуют.

Выходные параметры: **false**, если индекс вышел за пределы таблицы, иначе **true**.

```
virtual bool IsEnded() const;
```

Назначение: проверяет, дошли ли мы до конца таблицы.

Входные параметры: отсутствуют.

Выходные параметры: **true** – если дошли (больше), **false** – в противном случае.

### 3.2.3 Описание класса ScanTable

```
template <typename TKey, typename TData>
class ScanTable : public Table<TKey, TData> {
protected:
    TabRecord<TKey, TData>** recs;
public:
    ScanTable(int _maxSize);
    ScanTable(const ScanTable<TKey, TData>& table);
    virtual ~ScanTable();
    virtual TabRecord<TKey, TData>* Find(TKey key) override;
    virtual void Insert(TKey key, TData* data) override;
    virtual void Remove(TKey key) override;
    TabRecord<TKey, TData>* GetCurrent() const override;
```

```

    bool IsEnded() const;
    friend std::ostream& operator<<(std::ostream& out, const ScanTable<TKey,
TData>& t);
};

```

Назначение: представление просматриваемой неупорядоченной таблицы

Поля:

**recs** – массив указателей на записи.

Методы:

```

ScanTable(int _maxSize);

```

Назначение: конструктор с параметром.

Входные параметры: **\_maxSize** – максимальное количество элементов в таблице.

Выходные параметры: отсутствуют.

```

ScanTable(const TScanTable<TKey, TData>& table);

```

Назначение: конструктор копирования

Входные параметры: **table** – ссылка на существующую таблицу, на основе которой будет создан новый.

Выходные параметры: отсутствуют.

```

Virtual ~ScanTable();

```

Назначение: деструктор.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

```

virtual TTabRecord <TKey, TData>* Find(const TKey& key) override;

```

Назначение: поиск записи в таблице по ключу.

Входные параметры: **key** – ключ, который ищем.

Выходные параметры: указатель на запись. Если запись не была найдена, вернётся нулевой указатель.

```

virtual void Insert(const TKey& key, const TData& data) override;

```

Назначение: вставка записи в таблицу.

Входные параметры: **key** – ключ, **data** – запись, которую хотим вставить.

Выходные параметры: отсутствуют.

```

virtual void Remove(const TKey& key) override;

```

Назначение: удаление записи из таблицы.

Входные параметры: **key** – ключ.

Выходные параметры: отсутствуют.

```

TabRecord<TKey, TData>* GetCurrent() const override;

```

Назначение: получение текущую запись в таблице.

Входные параметры: отсутствуют.

Выходные параметры: указатель на текущую запись в таблице.

```
bool IsEnded() const;
```

Назначение: проверить, завершена ли таблица.

Входные параметры: отсутствуют.

Выходные параметры: **true** – если завершена, **false** – в противном случае..

```
friend      std::ostream&      operator<<(std::ostream&      out,      const
TScanTable<TKey, TData>& t);
```

Назначение: оператор вывода.

Входные параметры: **out** – ссылка на объект типа **ostream**, который представляет выходной поток, **t** – ссылка на объект типа **TScanTable<TKey, TData>** который будет выводиться.

Выходные параметры: ссылка на объект типа **ostream**.

### 3.2.4 Описание класса SortedTable

```
template <typename TKey, typename TData>
class SortedTable : public ScanTable<TKey, TData> {
private:
    void QuickSort(TabRecord<TKey, TData>** arr, int left, int right);
public:
    SortedTable(int maxSize);
    SortedTable(const ScanTable<TKey, TData>& table);
    TabRecord<TKey, TData>* Find(TKey key) override;
    void Insert(TKey key, TData* data) override;
    void Remove(TKey key);
};
```

Назначение: представление упорядоченной таблицы.

Поля: отсутствуют.

Методы:

```
void QuickSort(TabRecord<TKey, TData>** arr, int left, int right);
```

Назначение: сортировка массива по заданному ключу.

Входные параметры: **arr** – массив указателей, **left** – индекс левой границы сортируемого участка массива, **right** – индекс правой границы.

Выходные параметры: отсутствуют.

```
SortedTable(int maxSize);
```

Назначение: конструктор с параметром.

Входные параметры: **maxSize** – максимальное количество элементов в таблице.

Выходные параметры: отсутствуют.

```
SortedTable(const TScanTable<TKey, TData>& table);
```

Назначение: конструктор с параметрами, копирующий данные неупорядоченной таблицы.

Входные параметры: **table** – таблица, которую копируем.

Выходные параметры: отсутствуют.

```
TabRecord <TKey, TData>* Find(TKey key) override;
```

Назначение: поиск записи в таблице по ключу.

Входные параметры: **key** – ключ, который ищем.

Выходные параметры: указатель на запись. Если запись не была найдена, вернётся нулевой указатель.

```
void Insert(TKey key, TData* data) override;
```

Назначение: вставка записи в таблицу.

Входные параметры: **key** – ключ, **data** – запись, которую хотим вставить.

Выходные параметры: отсутствуют.

```
void Remove(TKey key) ;
```

Назначение: удаление записи из таблицы.

Входные параметры: **key** – ключ.

Выходные параметры: отсутствуют

### 3.2.5 Описание класса **HashTable**

```
template <typename TKey, typename TData>  
class HashTable : public TTable<TKey, TData>  
{  
protected:  
    virtual size_t hash_func(const TKey& key) = 0;  
public:  
    THashTable(int n) : Table<TKey,TData>(n) ;  
};
```

Назначение: абстрактный базовый класс для хэш-таблиц

Поля: отсутствуют.

Методы:

```
virtual size_t hash_func(const TKey& key) = 0;
```

Назначение: функция хэширования.

Входные параметры: **key** – ключ, по которому ищем элемент.

Выходные параметры: хэш, первый возможный индекс элемента в таблице.

```
THashTable(int n) : TTable<TKey,TData>(n) ;
```

Назначение: конструктор с параметрами.

Входные параметры: **n** – максимальный размер таблицы.

Выходные параметры: отсутствуют.

### 3.2.6 Описание класса ArrayHashTable

```
template <typename TKey, typename TData>
class ArrayHashTable : public HashTable<TKey, TData> {
private:
    TabRecord<TKey, TData>** recs;
    TabRecord<TKey, TData>* pMark;
    int freePos;
    int hashStep;
    void GetNextPos(int pos);
    virtual size_t hashFunc(const TKey& key) const override;
public:
    ArrayHashTable(int n, int step);
    ArrayHashTable(const ArrayHashTable& ahtable);
    ~ArrayHashTable();
    TabRecord<TKey, TData>* Find(const TKey key);
    void Insert(TKey key, TData* data);
    void Remove(TKey key);
    void Next();
    void Reset();
    TabRecord<TKey, TData>* GetCurrent() const;
    friend ostream& operator<<(ostream& out, const ArrayHashTable<TKey, TData>&
t);
};
```

Назначение: представление хэш-таблиц с помощью метода открытого перемешивания

Поля:

**recs** – массив указателей на записи.

**pMark** - фиктивная запись, которая означает, что элемент в ячейке был удалён.

**freePos** – индекс свободной позиции.

**hashStep** - линейный шаг в методе «открытое перемешивание».

Методы:

**void GetNextPos(int pos);**

Назначение: получение следующей позиции с учетом заданной.

Входные параметры: **pos** – текущий индекс в хэш-таблице, для которого нужно рассчитать следующую позицию.

Выходные параметры: следующая позиция.

**virtual size\_t hash\_func(const TKey& key) const override;**

Назначение: функция хэширования.

Входные параметры: **key**– ключ, по которому ищем элемент.

Выходные параметры: хэш, первый возможный индекс элемента в таблице.

**ArrayHashTable(int n, int step);**

Назначение: конструктор с параметрами.

Входные параметры: **n** – максимальный размер таблицы, **step** - линейный шаг.

Выходные параметры: отсутствуют.

**ArrayHashTable(const ArrayHashTable& ahtable);**

Назначение: конструктор копирования.

Входные параметры: **ahtable** – ссылка на существующую таблицу, на основе которой будет создана новая.

Выходные параметры: отсутствуют.

**~ArrayHashTable()** ;

Назначение: деструктор.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TabRecord <TKey, TData>\* Find(const TKey& key) ;**

Назначение: поиск записи в таблице по ключу.

Входные параметры: **key** – ключ, который ищем.

Выходные параметры: указатель на запись. Если запись не была найдена, вернётся нулевой указатель.

**void Insert(TKey key, TData\* data) override;**

Назначение: вставка записи в таблицу.

Входные параметры: **key** – ключ, **data** – запись, которую хотим вставить.

Выходные параметры: отсутствуют.

**void Remove(TKey key) override;**

Назначение: удаление записи из таблицы.

Входные параметры: **key** – ключ.

Выходные параметры: отсутствуют.

**void Next()** ;

Назначение: переход к следующей записи.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**void Reset()** ;

Назначение: ставит индекс текущего элемента на начало.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TabRecord<TKey, TData>\* GetCurrent() const;**

Назначение: получение записи.

Входные параметры: отсутствуют.

Выходные параметры: данные.



```
friend std::ostream& operator<<(std::ostream& out, const TArrayHashTable<TKey,
TData>& t);
```

Назначение: оператор вывода.

Входные параметры: **out** – ссылка на объект типа **ostream**, который представляет выходной поток, **t** – ссылка на объект типа **TArrayHashTable <TKey, TData>** который будет выводиться.

Выходные параметры: ссылка на объект типа **ostream**.

## Заключение

В рамках данной лабораторной работы была разработана и реализована структура данных для работы с записями таблиц. Были созданы классы **SortedTable**, **ScanTable**, **ArrayHashTable** предоставляющие функционал для работы с упорядоченной, просматриваемой, хэш-таблицы соответственно.

Таким образом, результатом выполнения лабораторной работы стала реализация структуры данных для работы с записями в таблицах, позволяющей удобно и эффективно выполнять различные операции над ними. В качестве записей выступали полиномы, реализованные в прошлой лабораторной работе. Программа позволяет выполнять арифметические операции над ними в таблицах, вставлять, удалять и выводить полиномы.

## **Литература**

1. Неупорядоченные таблицы [<https://studfile.net/preview/7081338/page:6/>].
2. Упорядоченные таблицы [<https://studfile.net/preview/1047385/>].
3. Хэш-таблица [<https://ru.wikipedia.org/wiki/Хэш-таблица>]
4. Хэш-таблица [<https://neerc.ifmo.ru/wiki/index.php?title=Хэш-таблица>]

# Приложения

## Приложение А. Реализация класса TabRecord

```
#ifndef _TABRECORD_H
#define _TABRECORD_H
#include <iostream>
#include <iomanip>
using namespace std;
template <typename TKey, typename TData>
class TabRecord {
protected:
    TKey key;
    TData* data;
public:
    TabRecord() : key(), data(nullptr) {}
    TabRecord(const TKey& _key, TData* _data);
    TabRecord(const TabRecord<TKey, TData>& record);
    ~TabRecord();
    TKey GetKey() const;
    TData* GetData() const;
    const TabRecord<TKey, TData>& operator=(const TabRecord<TKey, TData>&
record);
    bool operator==(const TabRecord<TKey, TData>& record) const;
    bool operator!=(const TabRecord<TKey, TData>& record) const;
    friend ostream& operator<<(std::ostream& out, const TabRecord < TKey,
TData>& record)
    {
        out << left << setw(10) << record.GetKey() << " | " << left <<
setw(10) << *record.GetData();
        return out;
    }
};

template <typename TKey, typename TData>
TabRecord<TKey, TData>::TabRecord(const TKey& _key, TData* _data) : key(_key)
{
    data = new TData(*_data);
}

template <typename TKey, typename TData>
TabRecord<TKey, TData>::TabRecord(const TabRecord<TKey, TData>& record) :
key(record.key) {
    data = new TData(*record.data);
}

template<typename TKey, typename TData>
TabRecord<TKey, TData>::~~TabRecord() {
    delete data;
}

template <typename TKey, typename TData>
TKey TabRecord<TKey, TData>::GetKey() const {
    return key;
}

template <typename TKey, typename TData>
TData* TabRecord<TKey, TData>::GetData() const {
    return data;
}

template<typename TKey, typename TData>
```

```

const TabRecord<TKey, TData>& TabRecord<TKey, TData>::operator=(const
TabRecord& record) {
    if (this == &record) {
        return *this;
    }
    key = record.key;
    data = new TData(*record.data);
    return *this;
}

template<typename TKey, typename TData>
bool TabRecord<TKey, TData>::operator==(const TabRecord<TKey, TData>& record)
const {
    return (key == record.GetKey() && *data == *record.GetData());
}

template<typename TKey, typename TData>
bool TabRecord<TKey, TData>::operator!=(const TabRecord<TKey, TData>& record)
const {
    return !(*this == record);
}

#endif

```

## Приложение Б. Реализация класса Table

```

#ifndef _TABLE_H
#define _TABLE_H

#include "TabRecord.h"
#include <iostream>
using namespace std;
template <typename TKey, typename TData>
class Table {
protected:
    int count;
    int maxSize;
    int currPos;
public:
    Table(int maxSize);
    virtual ~Table() {}
    virtual TabRecord<TKey, TData>* Find(TKey key) = 0;
    virtual void Insert(TKey key, TData* data) = 0;
    virtual void Remove(TKey key) = 0;
    virtual TabRecord<TKey, TData>* GetCurrent() const = 0;
    virtual bool IsFull() const;
    virtual bool IsEmpty() const;
    virtual void Reset();
    virtual void Next();
    virtual bool IsEnded() const;
};

template <typename TKey, typename TData>
Table<TKey, TData>::Table(int maxSize) : count(0), maxSize(maxSize), currPos(-
1) {}

template <typename TKey, typename TData>
bool Table<TKey, TData>::IsFull() const {
    return (count == maxSize);
}

template <typename TKey, typename TData>

```

```

bool Table<TKey, TData>::IsEmpty() const {
    return (count == 0);
}

template <typename TKey, typename TData>
void Table<TKey, TData>::Reset() {
    currPos = 0;
}

template <typename TKey, typename TData>
void Table<TKey, TData>::Next() {
    if (IsEnded()) throw ("table is ended");
    ++currPos;
}

template <typename TKey, typename TData>
bool Table<TKey, TData>::IsEnded() const {
    return (currPos == maxSize - 1);
}

#endif

```

## Приложение В. Реализация класса ScanTable

```

template <typename TKey, typename TData>
class ScanTable : public Table<TKey, TData> {
protected:
    TabRecord<TKey, TData>** recs;

public:
    ScanTable(int _maxSize);
    ScanTable(const ScanTable<TKey, TData>& table);
    virtual ~ScanTable();

    virtual TabRecord<TKey, TData>* Find(TKey key) override;
    virtual void Insert(TKey key, TData* data) override;
    virtual void Remove(TKey key) override;
    TabRecord<TKey, TData>* GetCurrent() const override;
    bool IsEnded() const;
    friend std::ostream& operator<<(std::ostream& out, const ScanTable<TKey,
TData>& t) {
        if (t.IsEmpty()) {
            out << "Table is empty" << endl;
            return out;
        }
        for (int i = 0; i < t.count; ++i)
        {
            if (t.recs[i] != nullptr) out << *(t.recs[i]);
        }
        return out;
    }
};

template <typename TKey, typename TData>
ScanTable<TKey, TData>::ScanTable(int _maxSize) : Table<TKey, TData>(_maxSize)
{
    if (_maxSize <= 0) {
        throw "table size must be greater than 0";
    }
    currPos = 0;
    maxSize = _maxSize;
    recs = new TabRecord<TKey, TData>[_maxSize];
    for (int i = 0; i < _maxSize; ++i) {

```

```

        recs[i] = nullptr;
    }
}

template <typename TKey, typename TData>
ScanTable<TKey, TData>::ScanTable(const ScanTable<TKey, TData>& table) :
Table<TKey, TData>(table.maxSize) {
    if (table.IsEmpty())
    {
        recs = nullptr;
        return;
    }
    count = table.count;
    currPos = 0;
    recs = new TabRecord<TKey, TData>*[maxSize];

    for (int i = 0; i < maxSize; ++i)
    {
        if (table.recs[i])
        {
            recs[i] = new TabRecord<TKey, TData>(*table.recs[i]);
        }
    }
}

template <typename TKey, typename TData>
ScanTable<TKey, TData>::~~ScanTable() {
    if (recs != nullptr) {
        for (int i = 0; i < count; i++)
            if (recs[i] != nullptr) delete recs[i];
        delete recs;
    }
}

template <typename TKey, typename TData>
TabRecord<TKey, TData>* ScanTable<TKey, TData>::Find(TKey key) {
    TabRecord<TKey, TData>* res = nullptr;

    for (int i = 0; i < count; i++) {
        if (recs[i]->key == key) {
            currPos = i;
            res = recs[i];
            break;
        }
    }

    return res;
}

template <typename TKey, typename TData>
void ScanTable<TKey, TData>::Insert(TKey key, TData* data) {
    if (IsFull()) {
        throw "Table is full";
    }
    if (Find(key) != nullptr) {
        throw("key is already exist");
    }

    recs[count++] = new TabRecord<TKey, TData>(key, data);
}

template <typename TKey, typename TData>
void ScanTable<TKey, TData>::Remove(TKey key) {

```

```

    if (IsEmpty())
        throw "ERROR: Table is empty.";

    if (Find(key) != nullptr) {
        delete recs[currPos];
        if (count - 1 != currPos) {
            recs[currPos] = new TabRecord<TKey, TData>(*recs[count - 1]);
            delete recs[count - 1];
        }
        else {
            recs[currPos] = nullptr;
        }
        count--;
    }
    else {
        throw "ERROR: key not found.";
    }
}

```

```

template <typename TKey, typename TData>
TabRecord<TKey, TData>* ScanTable<TKey, TData>::GetCurrent() const {
    if (currPos >= 0 && currPos < count) {
        return recs[currPos];
    }
    throw("Out of range");
}

```

```

template <typename TKey, typename TData>
bool ScanTable<TKey, TData>::IsEnded() const {
    return currPos >= count;
}
#endif

```

## Приложение Г. Реализация класса SortedTable

```

#ifndef _SORT_TABLE_H
#define _SORT_TABLE_H

#include "ScanTable.h"

template <typename TKey, typename TData>
class SortedTable : public ScanTable<TKey, TData> {
private:
    void QuickSort(TabRecord<TKey, TData>** arr, int left, int right);
public:
    SortedTable(int maxSize);
    SortedTable(const ScanTable<TKey, TData>& table);

    TabRecord<TKey, TData>* Find(TKey key) override;
    void Insert(TKey key, TData* data) override;
    void Remove(TKey key);
};

template <typename TKey, typename TData>
SortedTable<TKey, TData>::SortedTable(int    maxSize)    :    ScanTable<TKey,
TData>(maxSize) {}

template <typename TKey, typename TData>
SortedTable<TKey, TData>::SortedTable(const ScanTable<TKey, TData>& table) :
ScanTable<TKey, TData>(table) {

```



```

        if (!IsEmpty()) {
            QuickSort(recs, 0, count - 1);
        }
    }

template <typename TKey, typename TData>
TabRecord<TKey, TData>* SortedTable<TKey, TData>::Find(TKey key) {
    int left = 0, right = count - 1;
    TabRecord<TKey, TData>* search = nullptr;

    while (left <= right) {
        int mid = (right + left) / 2;

        if (recs[mid]->key == key) {
            search = recs[mid];
            right = mid;
            left = mid + 1;
        }
        else if (recs[mid]->key < key) left = mid + 1;
        else right = mid - 1;
    }
    if (right != -1)
        if (search == nullptr)
            currPos = right + 1;
        else
            currPos = right;
    else
        currPos = 0;
    return search;
}

template <typename TKey, typename TData>
void SortedTable<TKey, TData>::Insert(TKey key, TData* data) {
    if (IsFull())
        throw "ERROR: Table is full.";

    if (Find(key) != nullptr) {
        throw "Key repeat, it's not good";
    }
    for (int i = count - 1; i >= currPos; i--) {
        recs[i + 1] = recs[i];
    }
    recs[currPos] = new TabRecord<TKey, TData>(key, data);

    count++;
}

template <typename TKey, typename TData>
void SortedTable<TKey, TData>::Remove(TKey key) {
    if (IsEmpty()) {
        throw "ERROR: Table is empty.";
    }

    TabRecord<TKey, TData>* rec = Find(key);
    if (rec == nullptr) {
        throw "ERROR: Key not found.";
    }
    delete rec;
    for (int i = currPos; i < count - 1; i++) {
        recs[i] = recs[i + 1];
    }
}

```

```

        count--;
    }

template <typename TKey, typename TData>
void SortedTable<TKey, TData>::QuickSort(TabRecord<TKey, TData>** arr, int
left, int right) {
    if (left >= right) return;

    int i = left;
    int j = right;

    TabRecord<TKey, TData>* middle = arr[(left + right) / 2];

    while (i <= j) {
        while (arr[i]->GetKey() < middle->GetKey())
            i++;
        while (arr[j]->GetKey() > middle->GetKey())
            j--;
        if (i <= j) {
            TabRecord<TKey, TData>* temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i++;
            j--;
        }
    }

    QuickSort(arr, left, j);
    QuickSort(arr, i, right);
}

#endif

```

## Приложение Д. Реализация класса HashTable

```

#ifndef _HASH_TABLE_H
#define _HASH_TABLE_H

#include "Table.h"

template <typename TKey, typename TData>
class HashTable : public Table<TKey, TData> {
protected:
    virtual size_t hashFunc(const TKey& key) const = 0;

public:
    HashTable(int n) : Table<TKey, TData>(n) {}
};

#endif

```

## Приложение Е. Реализация класса ArrayHashTable

```

#ifndef _ARRAY_HASH_TABLE_H
#define _ARRAY_HASH_TABLE_H
#include <HashTable.h>

template <typename TKey, typename TData>
class ArrayHashTable : public HashTable<TKey, TData> {
private:
    TabRecord<TKey, TData>** recs;
    TabRecord<TKey, TData>* pMark;

```

```

    int freePos;
    int hashStep;
    void GetNextPos(int pos);
    virtual size_t hashFunc(const TKey& key) const override;
public:
    ArrayHashTable(int n, int step);
    ArrayHashTable(const ArrayHashTable& ahtable);
    ~ArrayHashTable();

    TabRecord<TKey, TData>* Find(const TKey key);
    void Insert(TKey key, TData* data);
    void Remove(TKey key);
    void Next();
    void Reset();
    TabRecord<TKey, TData>* GetCurrent() const;
    friend ostream& operator<<(ostream& out, const ArrayHashTable<TKey, TData>&
t) {
        if (t.IsEmpty()) {
            out << "Table is empty";
            return out;
        }
        for (int i = 0; i < t.maxSize; i++) {
            if (t.recs[i] && t.recs[i] != t.pMark)
                out << *(t.recs[i]);
        }
        return out;
    }

};

template <typename TKey, typename TData>
ArrayHashTable<TKey, TData>::ArrayHashTable(int n, int step) : HashTable<TKey,
TData>(n) {
    maxSize = n;
    hashStep = step;
    recs = new TabRecord<TKey, TData>*[n];
    pMark = new TabRecord<TKey, TData>();

    for (int i = 0; i < n; ++i) {
        recs[i] = nullptr;
    }

    freePos = -1;
    count = 0;
    currPos = 0;
}

template <typename TKey, typename TData>
ArrayHashTable<TKey, TData>::ArrayHashTable(const ArrayHashTable& ahtable) :
HashTable<TKey, TData>(ahtable.maxSize) {
    hashStep = ahtable.hashStep;
    currPos = ahtable.currPos;
    recs = new TabRecord<TKey, TData>*[maxSize];
    pMark = new TabRecord<TKey, TData>();

    for (int i = 0; i < maxSize; ++i) {
        if (ahtable.recs[i] == nullptr) {
            recs[i] = nullptr;
        }
        else if (ahtable.recs[i] == ahtable.pMark) {
            recs[i] = pMark;
        }
    }
}

```

```

        else {
            recs[i] = new TabRecord<TKey, TData>(*ahtable.recs[i]);
        }
    }
}

template <typename TKey, typename TData>
void ArrayHashTable<TKey, TData>::Insert(TKey key, TData* data) {
    if (IsEnded())
        throw "Table is full\n";
    TabRecord<TKey, TData>* tmp = Find(key);

    if (!tmp)
    {
        recs[freePos] = new TabRecord<TKey, TData>(key, data);
        count++;
    }
    else
    {
        tmp->data = data;
    }
}

template <typename TKey, typename TData>
void ArrayHashTable<TKey, TData>::Remove(TKey key) {
    if (IsEmpty())
    {
        throw string("Table is empty\n");
    }
    TabRecord<TKey, TData>* exist = Find(key);

    if (!exist)
    {
        throw string("Wrong key\n");
    }
    count--;
    delete recs[currPos];
    recs[currPos] = pMark;
}

template <typename TKey, typename TData>
TabRecord<TKey, TData>* ArrayHashTable<TKey, TData>::Find(const TKey key) {
    int hs = hashFunc(key), t = (hs + hashStep) % maxSize, c = 1;
    freePos = hs;

    if (recs[hs] == nullptr)
    {
        freePos = hs;
        return nullptr;
    }
    if (recs[hs]->key == key && recs[hs] != pMark)
    {
        currPos = hs;
        return recs[hs];
    }
    while (recs[t] != nullptr && t != hs && c < maxSize)
    {
        if (recs[t]->key == key)
        {
            currPos = t;
            return recs[t];
        }
        if (recs[t] == nullptr)

```

```

        {
            freePos = t;
            return nullptr;
        }
        t = (t + hashStep) % maxSize;
        ++c;
    }
    if (recs[freePos] != pMark && recs[freePos] != nullptr) {
        GetNextPos(freePos);
    }
    return nullptr;
}

template <typename TKey, typename TData>
ArrayHashTable<TKey, TData>::~ArrayHashTable()
{
    for (int i = 0; i < maxSize; ++i) {
        if (recs[i] != nullptr) {
            delete recs[i];
        }
    }
    delete[] recs;
    delete pMark;
}

template <typename TKey, typename TData>
TabRecord<TKey, TData>* ArrayHashTable<TKey, TData>::GetCurrent() const {
    if (currPos == -1 || currPos >= maxSize || recs[currPos] == nullptr) {
        return nullptr;
    }

    return recs[currPos];
}

template <typename TKey, typename TData>
void ArrayHashTable<TKey, TData>::GetNextPos(int pos) {
    if (count == maxSize) currPos = 0;
    int new_pos = (pos + hashStep % maxSize);
    while (new_pos != pos && (recs[new_pos] != pMark && recs[new_pos] !=
nullptr))
    {
        new_pos = (new_pos + hashStep) % maxSize;
    }
    currPos = new_pos;
}

template <typename TKey, typename TData>
void ArrayHashTable<TKey, TData>::Reset() {
    currPos = 0;
    while (!IsEnded() && (recs[currPos] == nullptr || recs[currPos] == pMark))
    {
        currPos++;
    }
}

template <typename TKey, typename TData>
void ArrayHashTable<TKey, TData>::Next() {
    if (IsEnded()) throw ("table is ended");
    currPos++;
    while (!IsEnded() && (recs[currPos] == nullptr || recs[currPos] == pMark))
    {
        currPos++;
    }
}

```

```

}

template<class TKey, class TData>
size_t ArrayHashTable<TKey, TData>::hashFunc(const TKey& key) const
{
    return std::hash<TKey>{}(key) % maxSize;
}

template <typename string, typename TPolynom>
class PolynomialHashTable : public ArrayHashTable<string, TPolynom> {
    size_t hashFunc(const string& key) const {
        uint64_t hashValue = 0;
        for (char ch : key) {
            hashValue += ch;
        }
        return (hashValue % maxSize);
    }
};

public:
    PolynomialHashTable(int n, int step) : ArrayHashTable<string, TPolynom>(n,
step) {}
};

#endif

```

## Приложение Ж. Sample\_table

```

#ifndef _INTRO_H
#define _INTRO_H
#include "tpolynom.h"
#include "SortTable.h"
#include "ArrayHashTable.h"
using namespace std;

class Tables {
private:
    ScanTable<string, TPolynom > scanTable;
    SortedTable<string, TPolynom> sortedTable;
    PolynomialHashTable<string, TPolynom> arrayHashTable;

    void poly_ops();
    int choose_table();
    void remove(const string& str);
    void remove_one(const string& str);
    void add(const string& str);
    void add_one(const string& str);
    TabRecord<string, TPolynom>* find(const string& str);
public:
    Tables() :scanTable(10), sortedTable(10), arrayHashTable(10, 1) {}
    void menu();
};

int Tables::choose_table() {
    int choice;
    cout << "TABLES" << endl;
    cout << "1)Scan Table" << endl;
    cout << "2)Sorted Table" << endl;
    cout << "3)Array Hash Table" << endl;
    cout << "0)CANCEL" << endl;
    cout << "Enter:";
    if (!(cin >> choice)) {
        cout << "Invalid input." << endl;
        return 0;
    }
}

```

```

    }

    return choice;
}

void Tables::add_one(const string& str) {
    TPolynom* polynom = new TPolynom(str);
    switch (choose_table()) {
        case 0: return;
        case 1: {
            scanTable.Insert(str, polynom);
            break;
        }
        case 2: {
            sortedTable.Insert(str, polynom);
            break;
        }
        case 3: {
            arrayHashTable.Insert(str, polynom);
            break;
        }
    }
}

void Tables::add(const string& str) {
    int choice;
    cout << "1. Add to one table" << endl;
    cout << "2. Add to all tables" << endl;
    cout << "0. CANCEL" << endl;
    cout << "Enter:";
    cin >> choice;
    switch (choice) {
        case 0: return;
        case 1: { add_one(str); break; }
        case 2: {
            TPolynom* polynom = new TPolynom(str);
            scanTable.Insert(str, polynom);
            sortedTable.Insert(str, polynom);
            arrayHashTable.Insert(str, polynom);
            break;
        }
    }
}

void Tables::remove_one(const string& str) {
    switch (choose_table()) {
        case 0: return;
        case 1: {
            scanTable.Remove(str);
            break;
        }
        case 2: {
            sortedTable.Remove(str);
            break;
        }
        case 3: {
            arrayHashTable.Remove(str);
            break;
        }
    }
}

void Tables::remove(const string& str) {

```

```

int choice;
cout << "1. Remove from one table" << endl;
cout << "2. Remove from all tables" << endl;
cout << "0. CANCEL" << endl;
cout << "Enter:";
cin >> choice;
switch (choice) {
case 0: return;
case 1: { remove_one(str); break; }
case 2: {
    scanTable.Remove(str);
    sortedTable.Remove(str);
    arrayHashTable.Remove(str);
    break;
}
}
}

void Tables::poly_ops() {
    string pol_name1, pol_name2;
    cout << "Enter first polynomial: ";
    cin >> pol_name1;

    TabRecord<string, TPolynom>* rec1 = find(pol_name1);
    if (rec1 != nullptr) {
        cout << "Enter second polynomial: ";
        cin >> pol_name2;

        TabRecord<string, TPolynom>* rec2 = find(pol_name2);
        if (rec2 != nullptr) {
            TPolynom p1 = *rec1->GetData();
            TPolynom p2 = *rec2->GetData();

            int choice;
            cout << "1)+ " << endl;
            cout << "2)- " << endl;
            cout << "3)* " << endl;
            cout << "4)dx" << endl;
            cout << "5)dy" << endl;
            cout << "6)dz" << endl;
            cout << "Enter:";
            cin >> choice;
            switch (choice) {
            case 0: return;
            case 1: {
                cout << (p1 + p2).ToString() << "\n";
                add((p1 + p2).ToString());
                break; }
            case 2: {
                cout << (p1 - p2).ToString() << "\n";
                add((p1 - p2).ToString());
                break; }
            case 3: {
                cout << (p1 * p2).ToString() << "\n";
                add((p1 * p2).ToString());
                break; }
            case 4: {
                cout << (p1.dx()).ToString() << "\n";
                add((p1.dx()).ToString());
                break; }
            case 5: {
                cout << (p1.dy()).ToString() << "\n";
                add((p1.dy()).ToString());

```



```

        break; }
    case 6: {
        cout << (p1.dz()).ToString() << "\n";
        add((p1.dz()).ToString());
        break; }

    }

}

}

}

TabRecord<string, TPolynom>* Tables::find(const string& str) {
    TabRecord<string, TPolynom>* record = nullptr;

    if ((record = scanTable.Find(str)) != nullptr) {
        return record;
    }
    if ((record = sortedTable.Find(str)) != nullptr) {
        return record;
    }
    if ((record = arrayHashTable.Find(str)) != nullptr) {
        return record;
    }

    return nullptr;
}

void Tables::menu() {
    int choice;
    do {
        cout << "Menu:" << endl;
        cout << "1. Add" << endl;
        cout << "2. Remove" << endl;
        cout << "3. See one table" << endl;
        cout << "4. See all tables" << endl;
        cout << "5. Polynoms" << endl;
        cout << "Enter your choice:";
        cin >> choice;
        switch (choice) {
            case 0: break;
            case 1: {
                cout << "Enter polynom:";
                string pol_name;
                cin >> pol_name;
                add(pol_name);
                break;
            }
            case 2: {
                cout << "Enter polynom:";
                string pol_name;
                cin >> pol_name;
                remove(pol_name);
                break;
            }
            case 3: {
                switch (choose_table()) {
                    case 0: return;
                    case 1: {
                        cout << scanTable;
                        break;
                    }
                    case 2: {

```

```

        cout << sortedTable;
        break;
    }
    case 3: {
        cout << arrayHashTable;
        break;
    }
    }
    break;
}
case 4: {
    cout << "ScanTable" << endl;
    cout << scanTable << endl;
    cout << "SortedTable" << endl;
    cout << sortedTable << endl;
    cout << "ArrayHashTable" << endl;
    cout << arrayHashTable << endl;
    break;
}
case 5: { poly_ops(); break; }
default:
    cout << "Invalid choice. Please try again." << endl;
    break;
}
} while (choice != 0);
}

#endif

int main() {
    try {
        Tables example;
        example.menu();
    }
    catch (char* exp) {
        cout << exp << endl;
    }
    return 0;
}

```