

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 полиномов»**

Выполнил: студент группы 3822Б1ФИ2

_____/Миронов А. И./
Подпись

Проверил: к.т.н, доцент каф. ВВиСП

_____/Кустикова В.Д./
Подпись

Нижний Новгород
2024

Содержание

Введение.....	3
1 Постановка задачи.....	4
2 Руководство пользователя.....	5
2.1 Приложение для демонстрации работы таблиц.....	5
3 Руководство программиста	8
3.1 Описание алгоритмов	8
3.1.1 Неупорядоченная таблица.....	8
3.1.2 Упорядоченная таблица.....	10
3.1.3 Хэш таблица.....	12
3.2 Описание программной реализации	15
3.2.1 Схема наследования классов.....	15
3.2.2 Описание класса Table	15
3.2.3 Описание класса ScanTable	17
3.2.4 Описание класса SortedTable	18
3.2.5 Описание класса HashTable.....	20
Заключение	23
Литература	24
Приложения	25
Приложение А. Реализация структуры Record	25
Приложение Б. Реализация класса Table	25
Приложение В. Реализация класса ScanTable	26
Приложение Г. Реализация класса SortedTable.....	28
Приложение Д. Реализация класса HashTable.....	29

Введение

Лабораторная работа направлена на сравнительный анализ и реализацию трех типов таблиц: упорядоченной, неупорядоченной и хэш таблицы. В ходе работы были разработаны и проанализированы три таблицы, каждая из которых имеет свои сильные и слабые стороны. Результаты работы помогут лучше понять принципы работы и эффективность различных типов таблиц в зависимости от задачи.

1 Постановка задачи

Цель:

Целью работы является изучение особенностей и преимуществ каждого типа таблиц, а также определение области их применения.

Задачи:

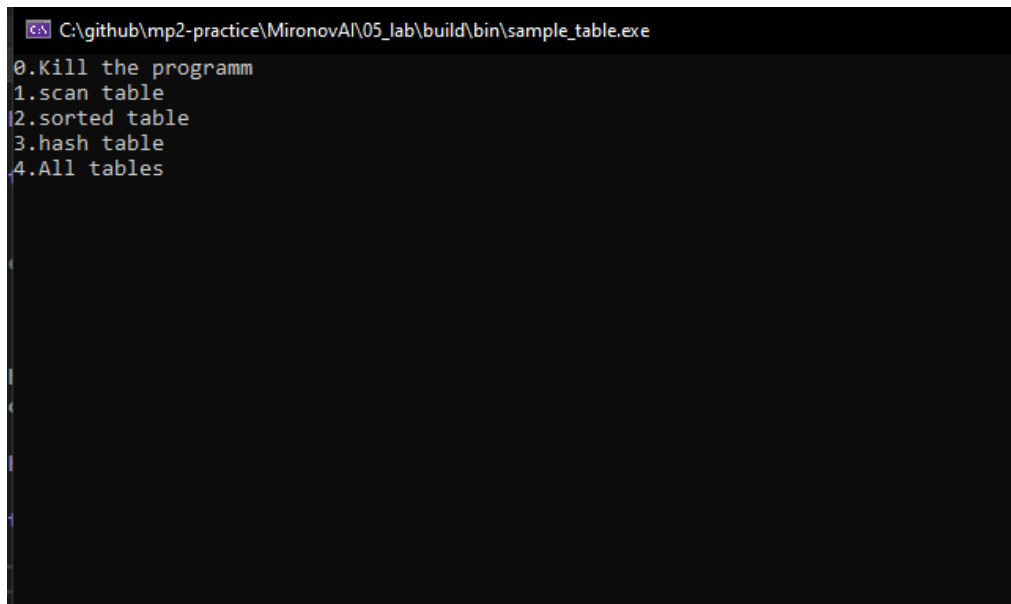
1. Изучение основных принципов работы с таблицами.
2. Создание упорядоченной, неупорядоченной и хэш таблиц.
3. Написание программы на C++.
4. Анализ времени выполнения программы и оценка эффективности использования различных таблиц для задачи работы с полиномами.
5. Тестирование программы на различных входных данных, включая выражения с разными операциями и скобками.

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

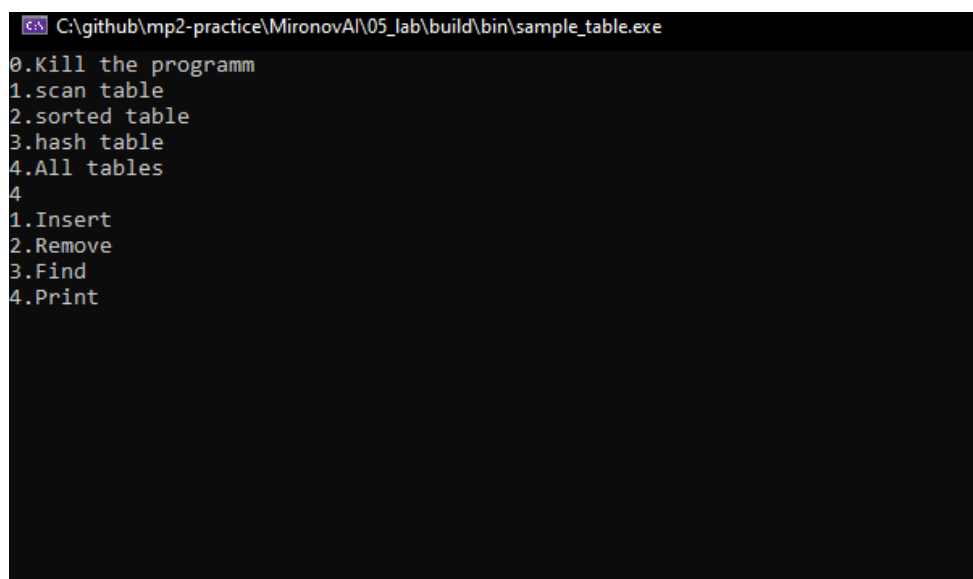
1. Запустите приложение с названием sample_table.exe. В результате появится окно, показанное ниже и вам будет предложено ввести число от 0 до 4. Это число повлияет на выбор таблиц, с которыми будет работать алгоритм (рис. 1).



```
C:\github\mp2-practice\MironovAI\05_lab\build\bin\sample_table.exe
0.Kill the programm
1.scan table
2.sorted table
3.hash table
4.All tables
```

Рис. 1. Основное окно программы

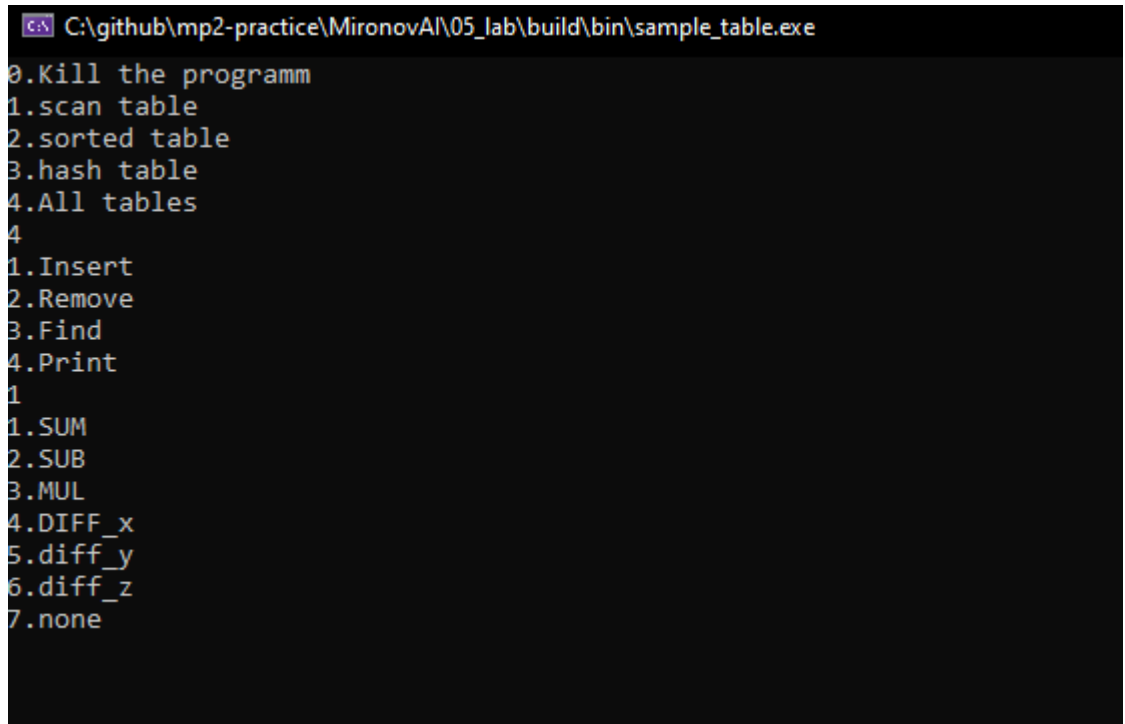
2. Далее вам будет предложено ввести число от 1 до 4. Это число повлияет на выбор операции, которая будет применена алгоритмом (рис. 2).



```
C:\github\mp2-practice\MironovAI\05_lab\build\bin\sample_table.exe
0.Kill the programm
1.scan table
2.sorted table
3.hash table
4.All tables
4
1.Insert
2.Remove
3.Find
4.Print
```

Рис. 2. Выбор операции

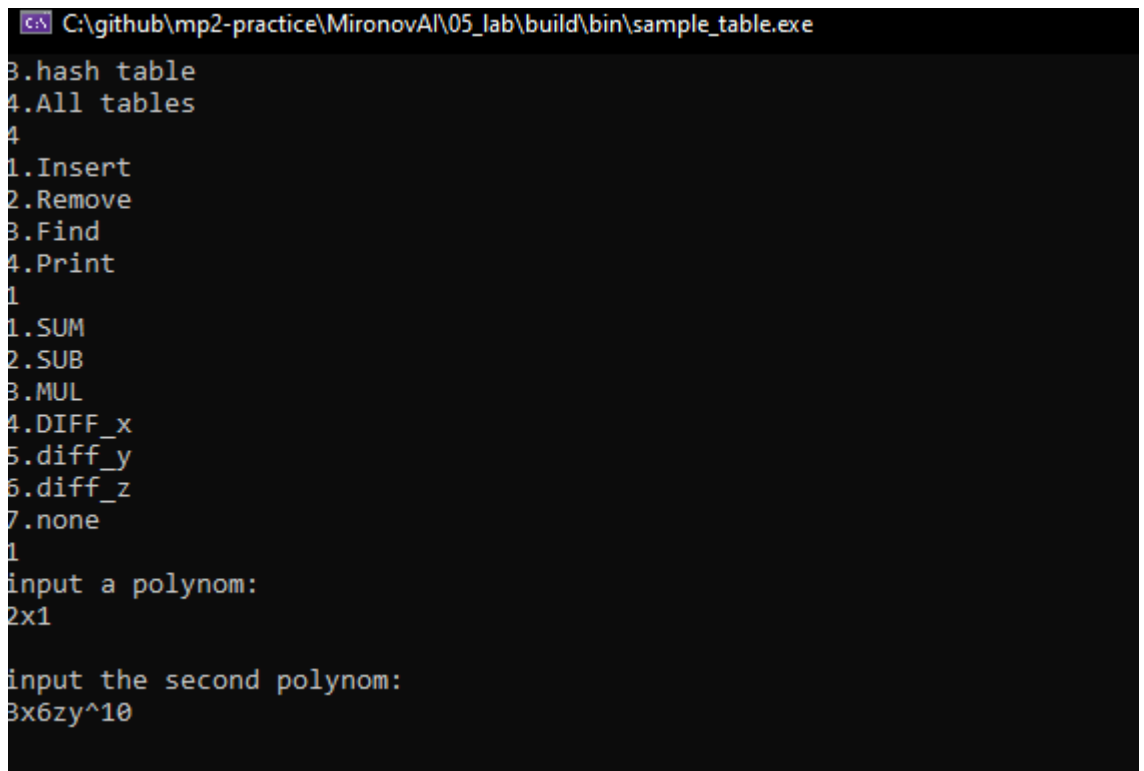
3. Далее будет предложено ввести операцию над полиномами. (Рис. 3).



```
C:\github\mp2-practice\MironovAI\05_lab\build\bin\sample_table.exe
0.Kill the programm
1.scan table
2.sorted table
3.hash table
4.All tables
4
1.Insert
2.Remove
3.Find
4.Print
1
1.SUM
2.SUB
3.MUL
4.DIFF_x
5.diff_y
6.diff_z
7.none
```

Рис. 3. Выбор операции над полиномами

4. Далее будет предложено ввести сами полиномы. Если вы выбрали бинарную операцию над полиномами, будет необходимо ввести два полинома. Если вы выбрали операцию «Print», то сразу будет выведено содержимое таблиц, полином вводить нельзя (Рис. 4).



```
C:\github\mp2-practice\MironovAI\05_lab\build\bin\sample_table.exe
3.hash table
4.All tables
4
1.Insert
2.Remove
3.Find
4.Print
1
1.SUM
2.SUB
3.MUL
4.DIFF_x
5.diff_y
6.diff_z
7.none
1
input a polynom:
2x1
input the second polynom:
3x6zy^10
```

Рис. 4. Ввод полиномов

5. После всех действий будет выведены результаты: “OK” в случае, если операция была выполнена успешно, иначе – название таблицы и метод, в котором возникло исключение (Рис. 5).

```
C:\github\mp2-practice\MironovAI\05_lab\build\bin\sample_table.exe
4.All tables
4
1.Insert
2.Remove
3.Find
4.Print
1
1.SUM
2.SUB
3.MUL
4.DIFF_x
5.diff_y
6.diff_z
7.none
1
input a polynom:
2x1

input the second polynom:
3x6zy^10

OK
OK
OK
0.Kill the programm
1.scan table
2.sorted table
3.hash table
4.All tables
```

Рис. 5. Результат работы демонстрации алгоритмов таблиц

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 элемента

Алгоритм:

1. Каждая запись таблицы последовательно сравнивается с ключом вставляемой записи до тех пор, пока не будет найдена запись с ключом или не просмотрим все элементы таблицы.
2. Если запись была найдена, то вставка не производится, иначе элемент вставляется в конец таблицы.

Пример:

2	4	
3	5	

Операция добавления элемента 3 с ключом 3:

2	4	3
3	5	3

Операция удаления элемента

Алгоритм:

1. Каждая запись таблицы последовательно сравнивается с ключом вставляемой записи до тех пор, пока не будет найдена запись с ключом или не просмотрим все элементы таблицы.
2. Если запись была найдена, то этот элемент удаляется и все элементы справа сдвигаются на один элемент влево, иначе удаление не производится.

Пример:

2	4	
---	---	--

3	5	
---	---	--

Операция удаления элемента с ключом 2:

4		
5		

Операция поиска элемента

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	
3	5	

Операция поиска элемента с ключом 2:

2
3

Операция проверки на пустоту

Операция проверки на пустоту реализована при помощи переменной, хранящей количество элементов в таблице. Возвращается результат сравнения количества элементов с нулём.

Пример:

2	4	
3	5	

Операция проверки на пустоту:

Результат: false

Операция проверки на полноту

Операция проверки на полноту реализована при помощи переменной, хранящей количество элементов в таблице. Возвращается результат сравнения количества элементов с максимальным размером таблицы.

Пример:

2	4	1
3	5	1

Операция проверки на полноту:

Результат: true

3.1.2 Упорядоченная 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, причём в отсортированном по ключам по не убыванию, порядке, имеет указатель на последний не занятый элемент (в случае полноты он будет выходить за пределы массива). Поддержка элементов в отсортированном порядке позволяет быстрее искать элемент в таблице при помощи алгоритма «Бинарный поиск».

Операция добавления

Операция добавления элемента реализуется при помощи поиска алгоритма «Бинарный поиск», который находит позицию для искомого элемента. Если структура хранения полна, то в этом случае возникает исключение.

Алгоритм:

1. Определяем место для вставки элемента при помощи алгоритма «Бинарный поиск».
2. Если запись была найдена, то вставка не производится, иначе элемент вставляется на это место, предварительно сдвигаются все элементы начиная с этого места на 1 элемент вправо.

Пример:

2	4	
3	5	

Операция добавления элемента 3 с ключом 3:

2	3	4
---	---	---

3	3	5
---	---	---

Операция удаления

Алгоритм:

1. Ищется элемент при помощи алгоритма «Бинарный поиск».
2. Если запись была найдена, то этот элемент удаляется и все элементы справа сдвигаются на один элемент влево, иначе удаление не производится.

Пример:

2	4	
3	5	

Операция удаления элемента с ключом 2:

4		
5		

Операция поиска

Операция удаления элемента реализуется при помощи алгоритма «Бинарный поиск», который находит элемент за оптимальное время. Если искомый элемент не был найден, то функция вернёт нулевой указатель, иначе вернет указатель на элемент.

Пример:

2	4	
3	5	

Операция поиска элемента с ключом 2:

2
3

Операция проверки на пустоту

Операция проверки на пустоту реализована при помощи переменной, хранящей количество элементов в таблице.

Пример:

2	4	
3	5	

Операция проверки на пустоту:

Результат: false

Операция проверки на полноту

Операция проверки на полноту реализована при помощи переменной, хранящей количество элементов в таблице.

Пример:

2	4	1
3	5	1

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 необходим для подхода избежаний коллизий «открытое перемешивание». Кроме того, в такой таблице реализована функция хэширования, которая, в среднем, должна ускорить поиск элемента. Структура данных хранит элементы хаотичном порядке, согласно хэш функции. В избежание коллизий в таблице реализован подход «открытое перемешивание».

Операция добавления

Операция добавления элемента реализуется при помощи хэш функции, которая считает позицию для искомого элемента. Если структура хранения полна, то в этом случае возникает исключение.

Алгоритм:

1. Определяем место для вставки элемента при помощи функции хэширования. Если в ячейке уже находится элемент, то мы используем линейный сдвиг, пока не найдём свободное место.
2. Если запись была найдена, то вставка не производится, иначе элемент вставляется на это место.

Пример:

2		4
3		5

Операция добавления элемента 3 с ключом 3:

2	3	4
3	3	5

Операция удаления

Операция добавления элемента реализуется при помощи хэш функции, которая считает позицию для искомого элемента. Если искомый элемент не был найден, возникает исключение, иначе – удаляем элемент.

Алгоритм:

1. Определяем место для вставки элемента при помощи функции хэширования. Если в ячейке уже находится другой элемент, то мы используем линейный сдвиг, пока не найдём место, в котором не было элемента, или пока не найдём сам элемент.
2. Если запись была найдена, то этот элемент удаляется, иначе удаление не производится.
3. При удалении ячейки меняется на фиктивную запись.

Пример:

2		4
3		5

Операция добавления ключом 4:

2		
3		

Операция поиска

Операция добавления элемента реализуется при помощи хэш функции,

Алгоритм:

1. Считается значение хэш функции для ключа.
2. Пока не найден элемент или мы не нашли место, в котором ещё не было элементов, используем линейный сдвиг – увеличиваем позицию на некоторую константу, которая по умолчанию равна 13.

Пример:

2		4
---	--	---

3		5
---	--	---

Операция поиска элемента с ключом 2:

2
3

Операция проверки на пустоту

Операция проверки на пустоту реализована при помощи переменной, хранящей количество элементов в таблице.

Пример:

2		4
3		5

Операция проверки на пустоту:

Результат: false

Операция проверки на полноту

Операция проверки на полноту реализована при помощи переменной, хранящей количество элементов в таблице.

Пример:

2	4	1
3	5	1

Операция проверки на полноту:

Результат: true

3.2 Описание программной реализации

3.2.1 Схема наследования классов

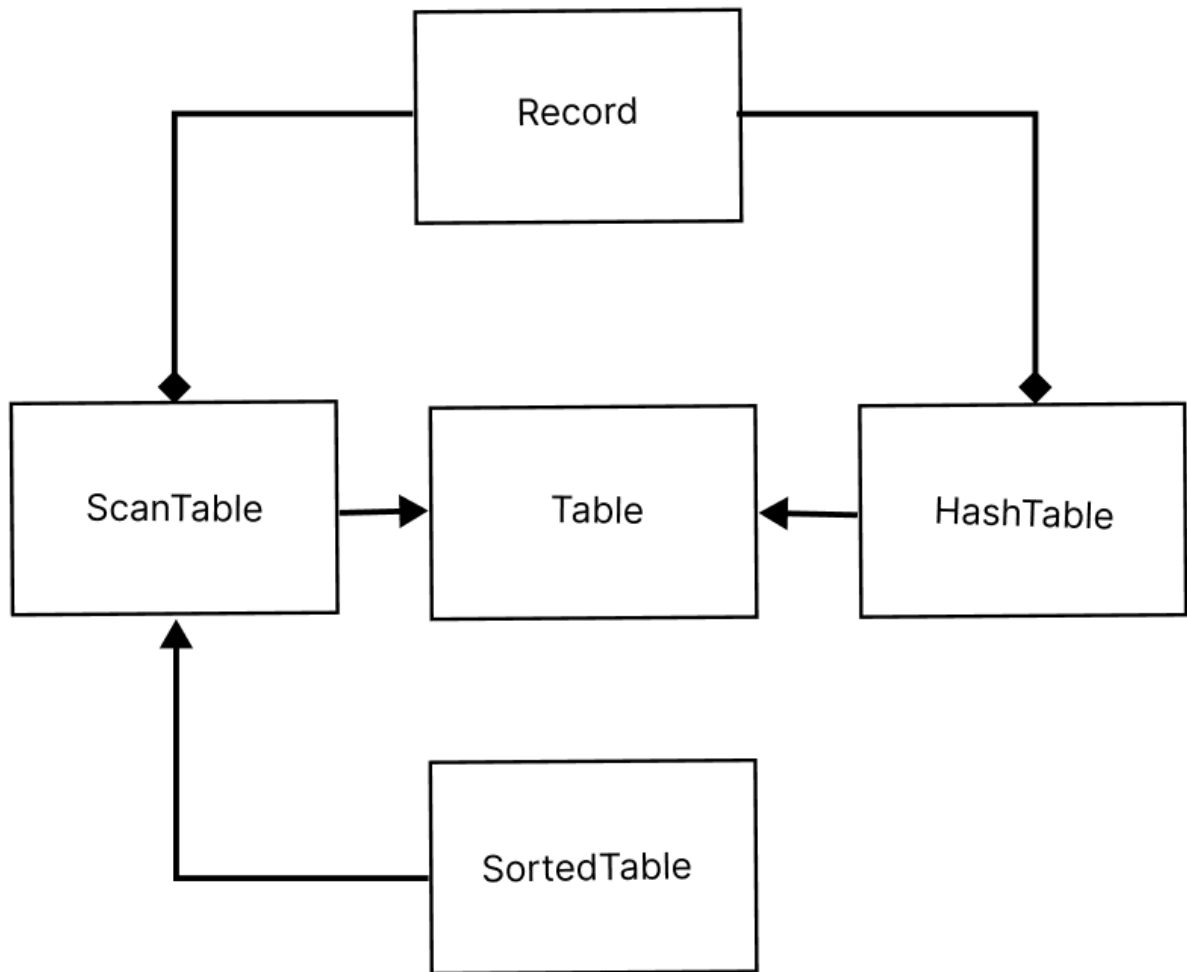


Рис. 6. Схема наследования классов

3.2.2 Описание класса Table

```
template <class Key, class Value>
class Table {
protected:
    int size;
    int max_size;
    int curr;

public:

    bool empty() const noexcept;
    bool full() const noexcept;
    bool reset() noexcept;
    bool next() noexcept;
    int get_size() const noexcept;
    int get_max_size() const noexcept;
    virtual Record<Key, Value>* find(const Key& key) = 0;
    virtual void insert(const Key& key, const Value& value) = 0;
    virtual void remove(const Key& key) = 0;
};
```

Назначение: абстрактный класс для представления различных видов таблиц. Таблицы будут использовать структуру записей **Record**. В структуре хранятся два поля – ключ и значение и определены операции сравнения структуры. В данной задаче структура аналогична **std::pair**, за исключением наименования полей.

Поля:

size – количество элементов в таблице.

max_size – максимальное число элементов.

curr – индекс текущего элемента (изначально 0).

Методы:

bool empty() const noexcept;

Назначение: проверка на пустоту.

Входные данные: отсутствуют.

Выходные данные: true, если таблица пуста, иначе false.

bool full() const noexcept;

Назначение: проверка на полноту.

Входные данные: отсутствуют.

Выходные данные: true, если таблица полна, иначе false.

bool reset() const noexcept;

Назначение: ставит индекс текущего элемента на начало.

Входные данные: отсутствуют.

Выходные данные: true, если таблица не пуста, иначе false.

bool next() const noexcept;

Назначение: сдвиг индекса текущего элемента вправо.

Входные данные: отсутствуют.

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

int get_size() const noexcept;

Назначение: получение количества элементов.

Входные данные: отсутствуют.

Выходные данные: количество элементов.

int get_max_size() const noexcept;

Назначение: получение максимального количества элементов.

Входные данные: отсутствуют.

Выходные данные: максимальное количество элементов.

3.2.3 Описание класса ScanTable

```
template <class Key, class Value>
class ScanTable : public Table<Key, Value> {
protected:
    Record<Key, Value>** recs;

public:
    ScanTable(int _max_size=DefaultSize) noexcept;
    ScanTable(const ScanTable<Key, Value>& table) noexcept;
    virtual ~ScanTable();

    virtual Record<Key, Value>* find(const Key& key);
    virtual void insert(const Key& key, const Value& value);
    virtual void remove(const Key& key);
    friend ostream& operator<<(ostream& buf, const ScanTable& table);
};
```

Назначение: неупорядоченной таблицы.

Класс наследуется от абстрактного класса **Table**, что позволяет использовать все определенные в нём поля и методы. **DefaultSize** равен 101 по умолчанию.

Поля:

recs** – массив указателей на записи.

Методы:

ScanTable(int _max_size=DefaultSize) noexcept;

Назначение: конструктор по умолчанию, с параметрами.

Входные параметры:

max_size – максимальный размер таблицы.

Выходные параметры: отсутствуют.

ScanTable(const ScanTable<Key, Value>& table) noexcept;

Назначение: конструктор копирования.

Входные параметры:

table – таблица, которую копируем.

Выходные параметры: отсутствуют.

virtual ~ScanTable();

Назначение: деструктор.

Входные параметры отсутствуют.

Выходные параметры: отсутствуют.

```
virtual Record<Key, Value>* find(const Key& key);
```

Назначение: поиск 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 Key& key, const Value& value);
```

Назначение: вставка элемента в таблицу.

Входные параметры:

key – ключ;

value – элемент, который хотим вставить.

Выходные параметры отсутствуют.

```
virtual void remove(const Key& key);
```

Назначение: удаление элемента из таблицы.

Входные параметры:

key – ключ, по которому удаляем.

Выходные параметры отсутствуют.

```
friend ostream& operator<<(ostream& buf, const ScanTable& table);
```

Назначение: оператор вывода таблицы.

Входные параметры:

buf – буфер;

table – таблица, которую выводим в буфер.

Выходные параметры: ссылка на буфер.

3.2.4 Описание класса SortedTable

```
template <class Key, class Value>  
class SortedTable : public ScanTable<Key, Value> {  
private:  
    void sort();  
  
public:  
    SortedTable(int _max_size=DefaultSize) noexcept;  
    SortedTable(const ScanTable<Key, Value>& table) noexcept;  
    SortedTable(const SortedTable<Key, Value>& table) noexcept;  
  
    void remove(const Key& _key);
```

```

    Record<Key, Value>* find(const Key& key);
    void insert(const Key& _key, const Value& _data);
};

```

Назначение: упорядоченной таблицы.

Класс наследуется от класса **ScanTable**, что позволяет использовать все определенные в нём поля и методы. Это наследование позволяет использовать общий оператор вывода и метод удаления элемента. В свою очередь, **ScanTable** наследуется от **Table**, поэтому в классе также будут определены и его методы.

DefaultSize равен 101 по умолчанию.

Поля отсутствуют.

Методы:

```
SortedTable(int _max_size=DefaultSize) noexcept;
```

Назначение: конструктор по умолчанию, с параметрами.

Входные параметры:

max_size – максимальный размер таблицы.

Выходные параметры: отсутствуют.

```
SortedTable(const SortedTable<Key, Value>& table) noexcept;
```

Назначение: конструктор копирования.

Входные параметры:

table – таблица, которую копируем.

Выходные параметры: отсутствуют.

```
SortedTable(const ScanTable<Key, Value>& table) noexcept;
```

Назначение: конструктор с параметрами, копирующий данные неупорядоченной таблицы.

Входные параметры:

table – таблица, которую копируем.

Выходные параметры: отсутствуют.

```
virtual ~SortedTable();
```

Назначение: деструктор.

Входные параметры отсутствуют.

Выходные параметры: отсутствуют.

```
virtual Record<Key, Value>* find(const Key& key);
```

Назначение: поиск 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 remove(const Key& key);

Назначение: удаление элемента из таблицы.

Входные параметры:

key – ключ, по которому удаляем.

Выходные параметры отсутствуют.

virtual void insert(const Key& key, const Value& value);

Назначение: вставка элемента в таблицу.

Входные параметры:

key – ключ;

value – элемент, который хотим вставить.

Выходные параметры отсутствуют.

void sort();

Назначение: сортировка таблицы.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

3.2.5 Описание класса HashTable

```
template <class Key, class Value>
class HashTable : public Table<Key, Value>
{
protected:
    Record<Key, Value>** recs;
    Record<Key, Value>* pMark = new Record<Key, Value>();
    size_t step;

    virtual size_t hash(const Key& key) const;
public:
    HashTable<Key, Value>(int _max_size=DefaultSize, int step_=-1) noexcept;
    HashTable<Key, Value>(const HashTable& table) noexcept;
    virtual ~HashTable<Key, Value>();

    Record<Key, Value>* find(const Key& key);
    void insert(const Key& key, const Value& value);
    void remove(const Key& key);
    void next(int pos);
    friend ostream& operator<<(ostream& buf, const HashTable& table);
};
```

Назначение: хэш таблицы.

Класс наследуется от абстрактного класса **Table**, что позволяет использовать все определенные в нём поля и методы. **DefaultSize** равен 101 по умолчанию.

Поля:

recs – массив указателей на записи.

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

step – линейный шаг в методе «открытое перемешивание».

Методы:

HashTable<Key, Value>(int _max_size=DefaultSize, int step_=-1) noexcept;

Назначение: конструктор по умолчанию, с параметрами.

Входные параметры:

max_size – максимальный размер таблицы.

Выходные параметры: отсутствуют.

HashTable(const HashTable<Key, Value>& table) noexcept;

Назначение: конструктор копирования.

Входные параметры:

table – таблица, которую копируем.

Выходные параметры: отсутствуют.

virtual ~HashTable();

Назначение: деструктор.

Входные параметры отсутствуют.

Выходные параметры: отсутствуют.

Record<Key, Value>* find(const Key& key);

Назначение: поиск 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(const Key& key, const Value& value);

Назначение: вставка элемента в таблицу.

Входные параметры:

key – ключ;

value – элемент, который хотим вставить.

Выходные параметры отсутствуют.

void remove(const Key& key) ;

Назначение: удаление элемента из таблицы.

Входные параметры:

key – ключ, по которому удаляем.

Выходные параметры отсутствуют.

friend ostream& operator<<(ostream& buf, const ScanTable& table) ;

Назначение: оператор вывода таблицы.

Входные параметры:

buf – буфер;

table – таблица, которую выводим в буфер.

Выходные параметры: ссылка на буфер.

virtual size_t hash(const Key& key) const;

Назначение: функция хэширования.

Входные параметры:

key – ключ, по которому ищем элемент.

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

Заключение

В ходе выполнения лабораторной работы была реализована и исследована работа трех типов таблиц: упорядоченной, неупорядоченной и хэш-таблицы для работы с классом полиномов.

Результаты исследования показали, что каждая из таблиц имеет свои преимущества и недостатки. Упорядоченная таблица обеспечивает быстрый поиск и вставку элементов, но требует дополнительных операций для поддержания порядка элементов. Неупорядоченная таблица имеет простую реализацию и быструю вставку элементов, но поиск элементов происходит медленнее. Хэш-таблица обеспечивает быстрый поиск и вставку элементов, но может иметь коллизии и требует дополнительных операций для разрешения конфликтов.

В целом, выбор типа таблицы зависит от конкретной задачи и требований к производительности. Для работы с классом полиномов упорядоченная таблица может быть наиболее подходящей, если необходимо часто выполнять операции поиска и вставки полиномов в порядке возрастания степени. Неупорядоченная таблица может быть использована, если необходимо быстро добавлять новые полиномы, а поиск полиномов происходит редко. Хэш-таблица может быть использована, если необходимо быстро выполнять операции поиска и вставки полиномов, а коллизии разрешаются эффективно.

В ходе выполнения лабораторной работы были получены навыки работы с различными типами таблиц, а также были изучены преимущества и недостатки каждого типа таблицы. Результаты исследования могут быть использованы для разработки эффективных алгоритмов и структур данных для работы с классом полиномов.

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/Хеш-таблица>]
4. Хэш таблица [<https://neerc.ifmo.ru/wiki/index.php?title=Хеш-таблица>]

Приложения

Приложение А. Реализация структуры Record

```
template <class Key, class Value>
Record<Key, Value>::Record() :data(Value()), key(Key())
{
}
template <class Key, class Value>
Record<Key, Value>::Record(const Key& _key, const Value& _data) : key(_key)
{
    data = _data;
}

template <class Key, class Value>
Record<Key, Value>::Record(const Record<Key, Value>& record)
{
    key = record.key;
    data = record.data;
}

template <class Key, class Value>
Record<Key, Value>::~~Record()
{
}

template <class Key, class Value>
bool Record<Key, Value>::operator==(const Record<Key, Value>& record) const
{
    return key == record.key && data == *record.data;
}

template <class Key, class Value>
bool Record<Key, Value>::operator==(const Key& key) const
{
    return this->key == key;
}

template <class Key, class Value>
bool Record<Key, Value>::operator<(const Record<Key, Value>&record) const
{
    return key < record.key;
}

template <class Key, class Value>
bool Record<Key, Value>::operator>(const Record<Key, Value>& record) const
{
    return key > record.key;
}
```

Приложение Б. Реализация класса Table

```
template <class Key, class Value>
bool Table<Key, Value>::full() const noexcept
{
    return (size == max_size);
}
```

```

}

template <class Key, class Value>
bool Table<Key, Value>::empty() const noexcept
{
    return (size == 0);
}

template <class Key, class Value>
bool Table<Key, Value>::reset() noexcept
{
    if (!empty()) {
        curr = 0;
        return true;
    }
    else {
        curr = -1;
        return false;
    }
}

template <class Key, class Value>
bool Table<Key, Value>::next() noexcept
{
    if (curr < max_size && !empty())
    {
        curr++;
        return true;
    }
    else {
        return false;
    }
}

template <class Key, class Value>
int Table<Key, Value>::get_size() const noexcept
{
    return size;
}

template <class Key, class Value>
int Table<Key, Value>::get_max_size() const noexcept
{
    return max_size;
}

```

Приложение В. Реализация класса ScanTable

```

template <class Key, class Value>
ScanTable<Key, Value>::ScanTable(int _max_size) noexcept
{
    if (_max_size <= 0)
    {
        max_size = -1;
        return;
    }
    max_size = _max_size;
    size = 0;
    curr = 0;
    recs = new Record<Key, Value>*[max_size];
}

```

```

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

template <class Key, class Value>
ScanTable<Key, Value>::ScanTable(const ScanTable<Key, Value>& table) noexcept
:
    max_size(table.max_size), size(table.size), curr(table.curr)
{
    recs = new Record<Key, Value>*[max_size];
    for (int i = 0; i < max_size; ++i)
    {
        if (table.recs[i])
        {
            recs[i] = new Record<Key, Value>(*table.recs[i]);
        }
    }
}

template <class Key, class Value>
ScanTable<Key, Value>::~~ScanTable()
{
    if (max_size == -1) return;
    for (int i = 0; i < max_size; ++i)
    {
        if (recs[i]) delete recs[i];
    }
    if (recs != nullptr) delete recs;
}

template <class Key, class Value>
void ScanTable<Key, Value>::insert(const Key& _key, const Value& _data)
{
    if (this->full())
    {
        throw string("Table is full\n");
    }

    Record<Key, Value>* exist = find(_key);
    if (exist)
    {
        exist->data = _data;
    }
    else
    {
        recs[size++] = new Record<Key, Value>(_key, _data);
    }
}

template <class Key, class Value>
void ScanTable<Key, Value>::remove(const Key& _key)
{
    if (this->empty())
    {
        throw string("Table is empty\n");
    }
    Record<Key, Value>* exist = find(_key);
    if (!exist) throw string("Key doesn't exist\n");

    delete recs[curr];

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

```

```

        {
            recs[i] = recs[i + 1];
        }
        --size;
    }

template <class Key, class Value>
Record<Key, Value>* ScanTable<Key, Value>::find(const Key& key)
{
    for (int i = 0; i < size; ++i)
    {
        if (recs[i]->key == key)
        {
            curr = i;
            return recs[i];
        }
    }
    return nullptr;
}

```

Приложение Г. Реализация класса SortedTable

```

template <class Key, class Value>
SortedTable<Key, Value>::SortedTable(int _max_size) noexcept :
ScanTable(_max_size) {}

template <class Key, class Value>
SortedTable<Key, Value>::SortedTable(const ScanTable<Key, Value>& table)
noexcept : ScanTable(st)
{
    this->sort();
}

template <class Key, class Value>
SortedTable<Key, Value>::SortedTable(const SortedTable<Key, Value>& table)
noexcept :
    size(table.size), max_size(table.max_size), curr(table.curr)
{
    recs = new Record<Key, Value>*[max_size];
    for (int i = 0; i < count; i++) {
        recs[i] = Record<Key, Value>(*recs[i]);
    }
}

template <class Key, class Value>
Record<Key, Value>* SortedTable<Key, Value>::find(const Key& key)
{
    int l = 0, r = size - 1;
    while (l <= r) {

        int mid = (l + r) / 2;

        if (recs[mid]->key == key)
        {
            curr = mid;
            return recs[mid];
        }
        if (recs[mid]->key < key)
        {
            curr = mid;
            l = mid + 1;
        }
    }
}

```

```

        }
        else
        {
            r = mid - 1;
        }
    }
    return nullptr;
}

template <class Key, class Value>
void SortedTable<Key, Value>::insert(const Key& _key, const Value& _data) {

    if (this->full()) {
        throw string("Table is full\n");
    }

    Record<Key, Value>* exist = find(_key);
    if (exist)
    {
        exist->data = _data;
    }
    else
    {
        if (recs[curr] != nullptr) if (recs[curr]->key < _key) curr++;
        for (int i = size - 1; i >= curr; i--)
        {
            recs[i + 1] = recs[i];
        }
        recs[curr] = new Record<Key, Value>(_key, _data);
        size++;
    }
}

template <class Key, class Value>
void SortedTable<Key, Value>::sort()
{
    for (int i = 0; i < size; ++i)
    {
        for (int j = i + 1; j < size; ++j)
        {
            if (recs[i] > recs[j])
            {
                Record<Key, Value>* t = recs[i];
                recs[i] = recs[j];
                recs[j] = t;
            }
        }
    }
}

```

Приложение Д. Реализация класса HashTable

```

template <class Key, class Value>
size_t HashTable<Key, Value>::hash(const Key& key) const
{
    std::hash<Key> hasher;
    return hasher(key) % max_size;
}

template <class Key, class Value>
HashTable<Key, Value>::HashTable(int _max_size) noexcept
{

```

```

        if (_max_size <= 0)
        {
            max_size = -1;
            return;
        }
        max_size = _max_size;
        size = 0;
        curr = 0;
        recs = new Record<Key, Value>*[max_size];
        for (int i = 0; i < _max_size; ++i) recs[i] = nullptr;

        step = (max_size == 13) ? 11 : 13;
    }

template <class Key, class Value>
HashTable<Key, Value>::HashTable(const HashTable& table) noexcept
{
    this->max_size = table.max_size;
    this->step = table.step;
    curr = table.curr;
    recs = new Record<Key, Value>*[max_size];
    for (int i = 0; i < max_size; ++i)
    {
        if (table.recs[i])
        {
            recs[i] = new Record<Key, Value>(table.recs[i]);
        }
    }
}

template <class Key, class Value>
HashTable<Key, Value>::~~HashTable()
{
    for (int i = 0; i < max_size; ++i)
    {
        if (recs[i] == pMark) recs[i] = nullptr;
        else if (recs[i] != nullptr) delete recs[i];
    }
    delete pMark;
}

template <class Key, class Value>
Record<Key, Value>* HashTable<Key, Value>::find(const Key& key)
{
    int hs = hash(key), t = (hs + step) % max_size, c = 1;
    curr = hs;

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

```

```

        {
            return nullptr;
        }
        t = (t + step) % max_size;
        ++c;
    }
    if (recs[curr] != pMark && recs[curr] != nullptr) next(curr);
    return nullptr;
}

template <class Key, class Value>
void HashTable<Key, Value>::insert(const Key& key, const Value& value)
{
    if (size == max_size)
    {
        throw string("Table is full\n");
    }
    Record<Key, Value>* exist = find(key);

    if (!exist)
    {
        recs[curr] = new Record<Key, Value>(key, value);
        size++;
    }
    else
    {
        exist->data = value;
    }
}

template <class Key, class Value>
void HashTable<Key, Value>::remove(const Key& key)
{
    if (size == 0)
    {
        throw string("Table is empty\n");
    }
    Record<Key, Value>* exist = find(key);

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

template <class Key, class Value>
void HashTable<Key, Value>::next(int pos)
{
    if (size == max_size) curr = 0;
    int new_pos = (pos + step % max_size);
    while (new_pos != pos && (recs[new_pos] != pMark && recs[new_pos] !=
nullptr))
    {
        new_pos = (new_pos + step) % max_size;
    }
    curr = new_pos;
}

```