МИНИСТЕРСТВО НАУКИ И ВЫСШЕГООБРАЗОВАНИЯ

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:

**«СТАТИСТИЧЕСКАЯ ОБРАБОТКА РЕЗУЛЬТАТОВ ЭКЗАМЕНАЦИОННОЙ СЕССИИ (ТАБЛИЦЫ)»**

**Выполнил(а):** студент группы 3822Б1ФИ1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Суворов Д.И. /

Подпись

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

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

Подпись

Нижний Новгород  
2024

**Содержание**

[Введение 3](#_Toc168454543)

[1 Постановка задачи 4](#_Toc168454544)

[1 Руководство пользователя 5](#_Toc168454545)

[1.1 Приложение для демонстрации работы таблиц 5](#_Toc168454546)

[2 Руководство программиста 7](#_Toc168454547)

[2.1 Описание программной алгоритмов 7](#_Toc168454548)

[2.1.1 Просматриваемая таблица 7](#_Toc168454549)

[2.1.1 Сортированная таблица 10](#_Toc168454550)

[2.1.2 ХЭШ таблица 13](#_Toc168454551)

[2.2 Описание программной реализации 18](#_Toc168454552)

[2.2.1 Схема наследования классов 18](#_Toc168454553)

[2.2.2 Описание класса TabRecord 18](#_Toc168454554)

[2.2.3 Описание класса Table 19](#_Toc168454555)

[2.2.4 Описание класса ScanTable 21](#_Toc168454556)

[2.2.5 Описание класса SortedTable 22](#_Toc168454557)

[2.2.6 Описание класса HashTable 23](#_Toc168454558)

[2.2.7 Описание класса ArrayHashTable 23](#_Toc168454559)

[Заключение 25](#_Toc168454560)

[Список литературы 26](#_Toc168454561)

[Приложения 27](#_Toc168454562)

[Приложение А. Реализация класса TabRecord 27](#_Toc168454563)

[Приложение Б. Реализация класса Table 27](#_Toc168454564)

[Приложение В. Реализация класса ScanTable 28](#_Toc168454565)

[Приложение Г. Реализация класса SortedTable 30](#_Toc168454566)

[Приложение Д. Реализация класса ArrayHashTable 32](#_Toc168454567)

# Введение

Таблицы – широко используемый инструмент в программном обеспечении, который представляет собой структуру данных, организованную в виде наборов пар "ключ-значение". Такая организация позволяет эффективно хранить и управлять данными, обеспечивая быстрый доступ к значениям по соответствующим ключам.

Таблицы позволяют структурировано хранить данные, а также иметь возможность поиска необходимых, вставки, удаления записей о данный по их уникальному ключу. Благодаря этому таблицы используются во многих областях программирования и компьютерных наук, таких как базы данных, обработка больших данных, кэширование и т.п. Именно поэтому понимание и умение работать с таблицами является важным навыком для разработчиков программного обеспечения, который позволяет создавать более эффективные и производительные приложения. В рамках данной лабораторной работы мы рассмотрим основные аспекты работы с таблицами, их реализацию и применение на практике.

# **Постановка задачи**

Цель – реализовать структуры данных для хранения таблиц разных видов.

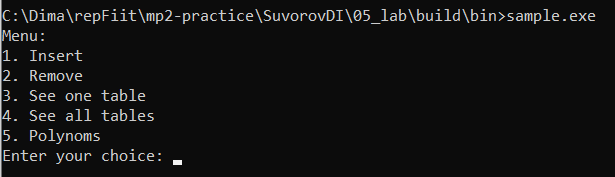
Задачи:

1. Внимательно изучить предметную область.
2. Реализовать класс Table.
3. На его основе реализовать классы ScanTable, SortedTable, ArrayHashRable.
4. Провести тестирование разработанных классов для проверки их корректной работы.
5. Сделать выводы о проделанной работе.

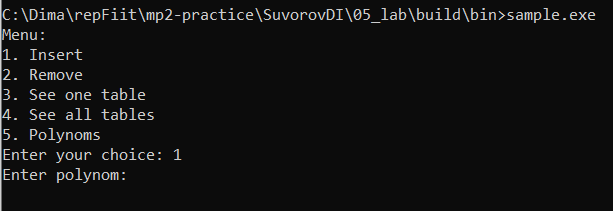
# Руководство пользователя

## Приложение для демонстрации работы таблиц

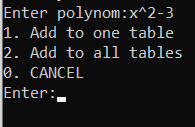
1. Запустите приложение с названием sampl.exe. В результате появится окно, показанное ниже. В стартовом окне пользователю предоставляется выбор действия, которое необходимо выполнить с таблицами: добавление, удаление, вывод одной таблицы, вывод всех таблиц, указание на действие, которое хочется протестировать с полиномами, хранящимися в таблицах. (рис. 1).



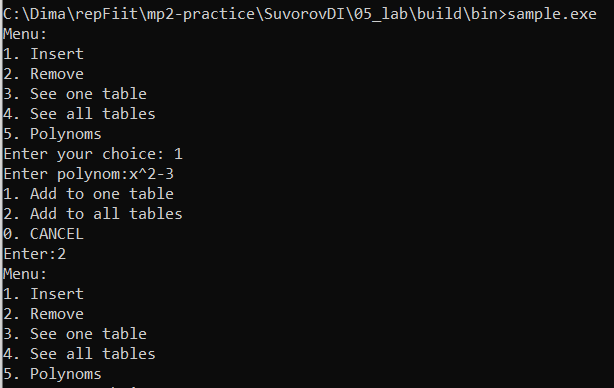
1. Начальное окно программы. Выбор действия
2. После ввода выбора (пусть для однозначности был выбран вариант 1) появится новый вывод в консоль приложения. Программа ожидает ввода строкового полинома (рис. 2).



1. Ввод полинома в таблицы
2. После ввода полинома необходимо выбрать как добавить элемент (в определённую одну таблицу или во все единовременно) (рис. 3).



1. Выбор формата вставки
2. После выбора формата вставки в консоли ещё раз выводится главное меню программы (цикл программы продолжается) (рис. 4).



1. Повторение цикла приложения

# Руководство программиста

## Описание программной алгоритмов

Таблица — динамическая структура данных, состоящая из последовательности записей, каждая из которых состоит из пары «ключ-значение». Каждая зпись в таблице имеет уникальный ключ.

Таблицы поддерживают три основных операции: поиск, вставка, удаление:

Рассмотрим эти операции для трёх видов таблиц: просматриваемая таблица, сортированная таблица, хэш таблица.

### Просматриваемая таблица

Просматриваемая таблица – это последовательность записей, в которой записи располагаются в порядке вставки.

* **Операция поиска записи**

1. Если последовательность записей пуста, то элемент не найден.
2. Проходим по последовательности записей от первой до последней.
3. Ключ каждой рассматриваемой записи сравниваем с искомым ключом.
4. Если ключи совпали, то элемент найден. Заканчиваем перебор записей.
5. Если перебрав все записи искомый ключ не совпал ни с одним из их ключей, то запись не найдена.

* **Операция вставки записи**

1. Проверяем, есть ли в таблице свободное место. Если нет, то завершаем вставку.
2. Проверяем, нет ли в таблице записи с ключом, равным ключу вставляемой записи. Если есть, то завершаем алгоритм (ключи в таблице должны быть уникальны).
3. Если оба условия ложны, то создаем новую запись с ключом и данными, которые мы хотим вставить в просматриваемую таблицу.
4. Добавляем созданную запись на позиуию с индексом SIZE.

Пример:

Таблица:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
|  | **54** | **3452** |
|  | **10** | **8654** |
| **. . .** | **. . .** | **. . .** |
|  | **26** | **100** |
| **SIZE - 1** | **87** | **653** |
|  |  |  |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

Необходимо вставить запись с ключом с ключом 111 и значением 4321.

Запись:

|  |  |
| --- | --- |
| **KEY** | **VALUE** |
| **111** | **4321** |

Вставка в конец таблицы на позицию SIZE:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
|  | **54** | **3452** |
|  | **10** | **8654** |
| **. . .** | **. . .** | **. . .** |
|  | **26** | **100** |
|  | **87** | **653** |
| **SIZE - 1** | **111** | **4321** |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

* **Операция удаления записи**

1. Если таблица пустая, то завершить удаление.
2. Если таблица не содержит записи с ключом, который равен ключу, по которому необходимо удалить запись.
3. Находим индекс, по котором лежит запись с подходящим ключом.
4. Удаляем запись по найденному индексу, на освободившееся место вставляем последнюю запись таблицы и уменьшаем количество.

Пример:

Таблица:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
|  | **54** | **3452** |
|  | **10** | **8654** |
| **. . .** | **. . .** | **. . .** |
|  | **26** | **100** |
| **SIZE - 1** | **87** | **653** |
|  |  |  |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

Необходимо удалить запись с ключом 10.

Поиск записи по ключу:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
|  | **54** | **3452** |
| **2** | **10** | **8654** |
| **. . .** | **. . .** | **. . .** |
|  | **26** | **100** |
| **SIZE - 1** | **87** | **653** |
|  |  |  |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

Удаляем запись:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
|  | **54** | **3452** |
|  |  |  |
| **. . .** | **. . .** | **. . .** |
|  | **26** | **100** |
| **SIZE - 1** | **87** | **653** |
|  |  |  |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

На освободившееся место устанавливаем последнюю запись и уменьшаем количество записей:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
|  | **54** | **3452** |
| **2** | **87** | **653** |
| **. . .** | **. . .** | **. . .** |
| **SIZE - 1** | **26** | **100** |
|  |  |  |
|  |  |  |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

### Сортированная таблица

Сортированная таблица – это отсортированная по ключам последовательность записей.

* **Операция поиска записи**

1. Выполняем бинарный поиск по ключам, который возвращает индекс записи в таблице, где должна располагаться запись с искомым ключом.
2. Если не нашлось подходящих записей, то завершаем поиск с отрицательным исходом, запоминается индекс ячейки справа от той, где должна была находится искомая запись, если бы она была в таблице.
3. Если запись нашлась, то завершаем поиск с положительным исходом.

* **Операция вставки записи**

1. Если в таблице нет свободного места, то завершаем вставку.
2. Проверяем, нет ли в таблице записи с ключом, равным ключу вставляемой записи. Если есть, то завершаем алгоритм (ключи в таблице должны быть уникальны).
3. Выполняем алгоритм поиска и запоминаем индекс записи, по которому бинарный поиск предполагал найти запись с нашим ключом.
4. Выполняем перепаковку от последнего элемента до найденного индекса.
5. Создаем новую запись с ключом и данными, которые необходимы для вставки.
6. В освободившееся место вставляем новую запись.

Пример:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
| **1** | **26** | **100** |
| **2** | **54** | **3452** |
|  | **22** | **1532** |
| **. . .** | **. . .** | **. . .** |
| **SIZE - 2** | **84** | **8538** |
| **SIZE - 1** | **87** | **653** |
|  |  |  |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

Необходимо вставить новую запись с ключом 30 и значением 321.

В результате поиска запись с ключом 30 ожидалась под индексом 1.

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
| **1** | **26** | **100** |
| **2** | **54** | **3452** |
|  | **22** | **1532** |
| **. . .** | **. . .** | **. . .** |
| **SIZE - 2** | **84** | **8538** |
| **SIZE - 1** | **87** | **653** |
|  |  |  |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

Выполняем перепаковку:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
| **1** | **26** | **100** |
| **2** |  |  |
| **3** | **54** | **3452** |
|  | **22** | **1532** |
| **. . .** | **. . .** | **. . .** |
| **SIZE - 2** | **84** | **8538** |
| **SIZE - 1** | **87** | **653** |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

В освободившееся место вставляем созданную запись:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
| **1** | **26** | **100** |
| **2** | **30** | **321** |
| **3** | **54** | **3452** |
|  | **22** | **1532** |
| **. . .** | **. . .** | **. . .** |
| **SIZE - 2** | **84** | **8538** |
| **SIZE - 1** | **87** | **653** |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

* **Операция удаления записи**

1. Если в таблице нет записей, то завершаем удаление.
2. Если в таблице нет элемента с искомым ключом, то завершаем удаление.
3. Выполняем алгоритм поиска искомой записи.
4. Удаляем найденную запись.
5. Производим перепаковку влево с записи с следующим индексом, после удаленной записи до конца.

Пример:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
| **1** | **26** | **100** |
| **2** | **30** | **321** |
| **3** | **54** | **3452** |
|  | **22** | **1532** |
| **. . .** | **. . .** | **. . .** |
| **SIZE - 2** | **84** | **8538** |
| **SIZE - 1** | **87** | **653** |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

Пусть необходимо удалить запись в ключом 54.

Производим поиск по ключу и удаляем запись.

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
| **1** | **26** | **100** |
| **2** | **30** | **321** |
|  |  |  |
|  | **22** | **1532** |
| **. . .** | **. . .** | **. . .** |
| **SIZE - 2** | **84** | **8538** |
| **SIZE - 1** | **87** | **653** |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

Производим перепаковку:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **13** | **2574** |
| **1** | **26** | **100** |
| **2** | **30** | **321** |
|  | **22** | **1532** |
| **. . .** | **. . .** | **. . .** |
| **SIZE - 2** | **84** | **8538** |
| **SIZE - 1** | **87** | **653** |
|  |  |  |
| **. . .** | **. . .** | **. . .** |
|  |  |  |
| **MAX\_SIZE** |  |  |

### ХЭШ таблица

ХЭШ таблица построена, как и прошлые виды таблиц, на последовательности записей. Однако положение записей определяется иначе. Если в просматриваемой таблице при вставке новая запись занимала последнее место в последовательности, в сортированной таблице новая запись занимала такое место, чтобы поддержать отсортированный порядок ключей, то в ХЭШ таблице индекс записи определяется ХЭШ функцией.

ХЭШ функция – это функция, которая по введённому ключу выдает значение индекса в последовательности записей. По этому индексу должен располагаться элемент с заданным ключом. Однако разные ключи могу получать от ХЭШ функции одинаковые значения индекса. Такая ситуация называется «коллизией». Для разрешения этой проблемы, если по полученному от ХЭШ функции индексу в последовательности уже находится другая запись, мы будем делать шаг на hash\_step записей далее (hash\_step лучше взять взаимно простым к MAX\_SIZE). То есть при коллизии новый индекс рассчитывается следующим образом: NewInd = (hash\_ind + hash\_step) % MAX\_SIZE.

Что делать если один из элементов, с которым у текущего есть коллизии был удален из таблицы. Чтобы текущий элемент был достижим, при удалении пустую освободившуюся запись заполнить фиктивными значениями, которые будут сигнализировать о том, что фактической записи нет, но она была, и была удалена.

Важно отметить, что в ХЭШ таблице заметна отсутствие непрерывности в расположении записей.

* **Операция поиска**
  1. Передать искомый ключ в ХЭШ функцию.
  2. Если по полученному индексу от ХЭШ функции в последовательности записей лежит запись с искомым ключом, то завершить поиск.
  3. Иначе возникла коллизия и необходимо получить NewInd сделав шаг.
  4. Проверяем таким образом каждый новый индекс, пока не найдем запись с ключом, соответствующим искомому, либо пока не просмотрим все элементы таблицы или не дойдём до индекса, у которого нет записи.

Пример:

Пусть ХЭШ функция выглядит следующим образом:

hash\_ind = key % MAX\_SIZE

MAX\_SIZE = 10

hash\_step = 3

Таблица содержит значения:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **20** | **1235** |
| **1** |  |  |
| **2** |  |  |
| **3** | **13** | **539** |
| **4** |  |  |
| **5** | **45** | **980** |
| **6** | **6** | **234** |
| **7** |  |  |
| **8** |  |  |
| **9** | **43** | **14236** |

Необходимо найти запись с ключом 43.

Передав ключ в ХЭШ функцию получаем индекс 3.

Но по индексу 3 находится запись с ключом 13.

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **20** | **1235** |
| **1** |  |  |
| **2** |  |  |
| **3** | **13** | **539** |
| **4** |  |  |
| **5** | **45** | **980** |
| **6** | **6** | **234** |
| **7** |  |  |
| **8** |  |  |
| **9** | **43** | **14236** |

Делаем шаг: новый индекс = (3 + 3) % 10 = 6

По полученному индексу лежит запись с ключом 6.

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **20** | **1235** |
| **1** |  |  |
| **2** |  |  |
| **3** | **13** | **539** |
| **4** |  |  |
| **5** | **45** | **980** |
| **6** | **6** | **234** |
| **7** |  |  |
| **8** |  |  |
| **9** | **43** | **14236** |

Значит делаем ещё один шаг: новый индекс = (6 + 3) % 10 = 9.

По полученному индексу находится запись с искомым ключом. Запись найдена.

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **20** | **1235** |
| **1** |  |  |
| **2** |  |  |
| **3** | **13** | **539** |
| **4** |  |  |
| **5** | **45** | **980** |
| **6** | **6** | **234** |
| **7** |  |  |
| **8** |  |  |
| **9** | **43** | **14236** |

В процессе операции поиска также запоминается свободное место (это либо пустая ячейка без записи, либо фиктивное значение при удалении).

* **Операция вставки**

1. Если количество записей равно MAX\_SIZE, то завершить вставку.
2. Выполнить поиск по ключу записи, которую необходимо вставить.
3. Если запись с таким ключом уже есть, то завершить вставку
4. Создать по заданному ключу и значению новую запись
5. Вставить её по индексу свободного места, найденного при выполнении поиска.

Пример:

* **Операция произведения полиномов**
  1. Создаем новый пустой список для хранения умноженных мономов.
  2. Итерируем по мономам первого полинома.
  3. Внутри этого цикла итерируем также по мономам второго полинома.
  4. Умножаем каждую пару мономов и добавляем ненулевые результаты в новый список.
  5. Если в результате умножения нет ненулевых мономов, добавляем нулевой моном.
  6. После завершения операции умножения возвращаем новый полином.

Пример:

Пусть дана следующая ХЭШ таблица:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **20** | **1235** |
| **1** |  |  |
| **2** |  |  |
| **3** | **13** | **539** |
| **4** |  |  |
| **5** | **45** | **980** |
| **6** | **6** | **234** |
| **7** |  |  |
| **8** |  |  |
| **9** | **43** | **14236** |

Необходимо вставить запись с ключом 53 и значением 133

Выполняем поиск:

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **20** | **1235** |
| **1** |  |  |
| **FREE 2** |  |  |
| **3** | **13** | **539** |
| **4** |  |  |
| **5** | **45** | **980** |
| **6** | **6** | **234** |
| **7** |  |  |
| **8** |  |  |
| **9** | **43** | **14236** |

Оранжевым обозначены записи, с которыми в процессе поиска возникли коллизии.

Зеленым обозначено свободно место, найденное операцией поиска.

Далее вставляем запись по индексу FREE.

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **20** | **1235** |
| **1** |  |  |
| **2** | **53** | **133** |
| **3** | **13** | **539** |
| **4** |  |  |
| **5** | **45** | **980** |
| **6** | **6** | **234** |
| **7** |  |  |
| **8** |  |  |
| **9** | **43** | **14236** |

* **Операция удаления записи**
  1. Если таблица не имеет записей, то завершаем удаление.
  2. Выполняем поиск записи по ключу, запись с которым необходимо удалить.
  3. Если записи с этим ключом нет, завершаем удаление.
  4. Получаем от операции поиска индекс искомой записи.
  5. Удаляем запись и на её место добавляем фиктивные значения, для указания того факта, что по этому индексу был удалён элемент.

Пример:

Пусть дана ХЭШ таблица, в которой необходимо удалить элемент с ключом 53.

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **20** | **1235** |
| **1** |  |  |
| **2** | **53** | **133** |
| **3** | **13** | **539** |
| **4** |  |  |
| **5** | **45** | **980** |
| **6** | **6** | **234** |
| **7** |  |  |
| **8** |  |  |
| **9** | **43** | **14236** |

Выполняем поиск.

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **20** | **1235** |
| **1** |  |  |
| **2** | **53** | **133** |
| **3** | **13** | **539** |
| **4** |  |  |
| **5** | **45** | **980** |
| **6** | **6** | **234** |
| **7** |  |  |
| **8** |  |  |
| **9** | **43** | **14236** |

В результате операции поиска получаем индекс записи, которую необходимо удалить.

Удаляем и заполняем поля фиктивными значениями.

|  |  |  |
| --- | --- | --- |
|  | **KEY** | **VALUE** |
| **0** | **20** | **1235** |
| **1** |  |  |
| **2** | **-1** | **0** |
| **3** | **13** | **539** |
| **4** |  |  |
| **5** | **45** | **980** |
| **6** | **6** | **234** |
| **7** |  |  |
| **8** |  |  |
| **9** | **43** | **14236** |

## Описание программной реализации

### Схема наследования классов

**Table**

**HashTable**

**ArrayHashTable**

**ScanTable**

**SortedTable**

1. Схема наследования классов

На рис. 6 показаны отношения между классами:

Абстрактный класс Table описывает общие параметры таблиц.

Класс ScanTable описывает просматриваемую таблицу.

Класс SortedTable описывает сортированную таблицу.

Класс HashTable описывает общий вид ХЭШ таблиц.

Класс ArrayHashTable описывает ХЭШ таблицу построенную на массиве.

### Описание класса TabRecord

template <typename TKey, typename TData>

struct TabRecord

{

TKey key;

TData \*data;

TabRecord();

TabRecord(TKey key, TData\* data);

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

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

TData\* GetData() const;

friend std::ostream& operator<<(std::ostream& out, const TabRecord<TKey, TData>& r);

};

Поля:

* key – ключ записи.
* data –указатель на значение записи.

Методы:

* TabRecord();

Назначение: конструктор по умолчанию.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

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

Назначение: конструктор с параметрами.

Входные параметры: key – ключ создаваемой записи, **data** – указатель на значение создаваемой записи.

Выходные параметры: отсутствуют.

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

Назначение: операция сравнения.

Входные параметры: **r** – ссылка на запись, с которой мы сравниваем текущую.

Выходные параметры: логический результат сравнения.

* friend std::ostream& operator<<(std::ostream& out, const TabRecord<TKey, TData>& r);

Назначение: операция вывода.

Входные параметры: ссылка на поток вывода и ссылка на выводимый объект записи.

Выходные параметры: ссылка на поток вывода.

* TData\* GetData() const;

Назначение: геттер для значения записи.

Входные параметры: отсутствуют.

Выходные параметры: указатель на значение записи.

* friend std::ostream& operator<<(std::ostream& out, const TabRecord<TKey, TData>& r);

Назначение: операция вывода.

Входные параметры: ссылка на поток вывода и ссылка на выводимый объект записи.

Выходные параметры: ссылка на поток вывода.

### 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;

bool IsFull() const noexcept;

bool IsEmpty() const noexcept;

void Reset() noexcept;

virtual bool IsEnded() const noexcept;

virtual void Next();

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

};

Поля:

* count – количество записей в таблице.
* maxSize – максимально возможное число записей в таблице.
* currPos – индекс текущего элемента.

Методы:

* Table(int maxSize);

Назначение: конструктор по умолчанию.

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

Выходные параметры: отсутствуют.

* virtual ~Table();

Назначение: создание списка с заданным начальным узлом

Входные параметры: \_pFirst – указатель на первый узел списка.

Выходные параметры: новый объект класса TList.

* virtual TabRecord<TKey, TData> \*Find(TKey key) = 0;

Назначение: поиск записи в таблице по ее ключу.

Входные параметры: key – ключ, пок которому производится поиск.

Выходные параметры: указать на найденную запись.

* virtual void Insert(TKey с, TData \*data) = 0;

Назначение: освобождение памяти списка при удалении объекта.

Входные параметры: key, data – ключ и указатель на значение записи, которая будет помещаться в таблицу.

Выходные параметры: отсутствуют.

* virtual void Remove(TKey key) = 0;

Назначение: удаление записи по переданному ключу.

Входные параметры: key – ключ записи на удаление.

Выходные параметры: отсутствуют.

* bool IsFull() const noexcept;

Назначение: определяет есть ли свободное место в таблице.

Входные параметры: отсутствует.

Выходные параметры: логическое значение true, если нет места, false – иначе.

* bool IsEmpty() const noexcept;

Назначение: определяет пустая таблица или нет.

Входные параметры: отсутствует

Выходные параметры: логическое значение true, если пусто, false – иначе.

* void Reset() noexcept;

Назначение: устанавливает поле currPos.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

* virtual bool IsEnded() const noexcept;

Назначение: определяет достигли ли мы конца таблицы.

Входные параметры: отсутствуют.

Выходные параметры: лог. Значение: true, еслиcurrPos достигнул конец, false – иначе.

* virtual void Next();

Назначение: сдвиг текущего элемента на следующего.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

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

Назначение: геттер для текущего элемента.

Входные параметры: отсутствуют.

Выходные параметры: указатель на текущую запись.

### 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> &t);

~ScanTable() override;

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

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

void Remove(TKey key) override;

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

bool IsEnded() const noexcept override;

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

};

Поля:

* recs – двойной указатель на записи.

Методы:

* ScanTable(int maxSize);

Назначение: конструктор с параметром, создает пустую просматриваемую таблицу.

Входные параметры: maxSize – максимально возможный размер создаваемой таблицы.

Выходные параметры: отсутствуют.

* ScanTable(const ScanTable<TKey, TData> &t);

Назначение: конструктор копирования, создает копию существующей просматриваемой таблицы.

Входные параметры: t – ссылка на существующую просматриваемую таблицу.

Выходные параметры: отсутствуют.

* ~ScanTable() override;

Назначение: деструктор для просматриваемой таблице, освобождает память от массива записей.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

* virtual ~THeadRingList();

Назначение: виртуальный деструктор, освобождает выделенную память при уничтожении объектов производных классов.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

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

Назначение: вставляет новый узел с данными в начало списка.

Входные параметры: out – ссылка на поток вывода, t – ссылка на выводимую просматриваемую таблицу.

Выходные параметры: ссылка на поток вывода.

### Описание класса SortedTable

template <typename TKey, typename TData>

class SortedTable : public ScanTable<TKey, TData>

{

private:

void QuickSort(TabRecord<TKey, TData> \*\*rec\_arr, int size);

public:

SortedTable(int maxSize);

SortedTable(const ScanTable<TKey, TData>\*);

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

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

void Remove(TKey key) override;

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

};

Методы:

* SortedTable(int maxSize);

Назначение: конструктор с параметром, создает пустую сортированную таблицу с заданным максимально возможным количеством записей.

Входные параметры: maxSize - максимально возможное количество записей.

Выходные параметры: отсутствуют.

* SortedTable(const ScanTable<TKey, TData>\* st);

Назначение: конструктор копирования, создает сортированную копию существующей просматриваемой таблицы.

Входные параметры: st – ссылка на существующую просматриваемую таблицу.

Выходные параметры: отсутствуют.

* void QuickSort(TabRecord<TKey, TData> \*\*rec\_arr, int size);

Назначение: алгоритм быстрой сортировки, для сортировки просматриваемой таблицы при приведении её к сортированной.

Входные параметры: rec\_arr –двойной указатель на записи, size – размер части массива для сортировки.

Выходные параметры: отсутствуют.

### Описание класса HashTable

template <typename TKey, typename TData>

class HashTable : public Table<TKey, TData>

{

protected:

virtual int hashFunc(const TKey key);

public:

HashTable(int n);

};

Методы:

* virtual int hashFunc(const TKey key);

Назначение: ХЭШ функция, возвращающая по ключу индекс в массиве записей.

Входные параметры: key – ключ записи.

Выходные параметры: целое число – индекс в массиве записей.

* HashTable(int n);

Назначение: создает объект HashTable с указанным максимальным количеством записей.

Входные параметры: n – максимальное количество записей.

Выходные параметры: отсутствуют.

### Описание класса ArrayHashTable

template <typename TKey, typename TData>

class ArrayHashTable : public HashTable<TKey, TData>

{

protected:

TabRecord<TKey, TData>\*\* recs;

TabRecord<TKey, TData>\* pMark;

int freePos;

int hashStep;

int GetNextPos(int h\_val);

public:

ArrayHashTable(int n, int step);

ArrayHashTable(const ArrayHashTable<TKey, TData> &ahtable);

~ArrayHashTable();

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

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

void Remove(TKey key);

bool Reset();

void Next();

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

friend std::ostream& operator<<(std::ostream& out, const ArrayHashTable<TKey, TData>& aht);

};

Поля:

* recs – двойной указатель на записи.
* pMark – указатель на фиктивную запись для удалённых записей.
* freePos – индекс свободного места для вставки.
* hashStep – шаг, совершаемый для обработки коллизии.

Методы:

* ArrayHashTable(int n, int step);

Назначение: конструктор с параметрами, создает пустую ХЭШ таблицу с заданным максимальным количеством записей и шагом.

Входные параметры: n – максимальное количество записей, step – шаг.

Выходные параметры: отсутствуют.

* ArrayHashTable(const ArrayHashTable<TKey, TData> &ahtable);

Назначение: конструктор копирования, создает новую ХЭШ таблицу на основе уже созданной.

Входные параметры: ahtable – копируемая ХЭШ таблица.

Выходные параметры: отсутствуют.

* ~ArrayHashTable();

Назначение: деструктор. Освобождает память от массива записей и от фаиктивной записи pMark.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

* int GetNextPos(int h\_val);

Назначение: Получение следующей позиции при коллизии, используя hashStep.

Входные параметры: h\_val – текущее значение индекса.

Выходные параметры: целое число – следующее значение индекса.

# Заключение

В результате данной лабораторной работе удалось изучить и реализовать 3 популярных вида таблиц. Была освоена работа с ХЭШ таблицей и ХЭШ функцией. Были реализованы и протестированы операции поиска, вставки и удаления для каждой таблицы. Цель лабораторной работы выполнена.

# Список литературы

1. Хеш-таблицы [https://habr.com/ru/articles/509220/].
2. Лекция «Организация доступа по имени» Сысоев А. В. [https://cloud.unn.ru/s/S9AiMJ45HCKJRNs].

# Приложения

# Приложение А. Реализация класса TabRecord

template <typename TKey, typename TData>

struct TabRecord

{

TKey key;

TData \*data;

TabRecord() : key(TKey()), data(nullptr) { }

TabRecord(TKey key, TData\* data) : key(key), data(data) { }

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

bool operator==(const TabRecord<TKey, TData>& r) const

{ return (key == r.key) && (\*data == \*r.data); }

TData\* GetData() const { return data; }

friend std::ostream& operator<<(std::ostream& out, const TabRecord<TKey, TData>& r)

{

out << "KEY: " << r.key << " -> VALUE:\n" << \*r.data << "\n";

return out;

}

};

template <typename TKey, typename TData>

const TabRecord<TKey, TData>& TabRecord<TKey, TData>::operator=(const TabRecord<TKey, TData>& r)

{

if (this == &r)

return \*this;

key = r.key;

data = new TData(r.data);

return \*this;

}

## Приложение Б. Реализация класса Table

template <typename TKey, typename TData>

class Table

{

protected:

int count;

int maxSize;

int currPos;

public:

Table(int maxSize)

{

if (maxSize < 0)

throw std::exception("negative\_table\_max\_size\n");

this->maxSize = maxSize;

count = 0;

currPos = -1;

}

virtual ~Table() { };

virtual TabRecord<TKey, TData> \*Find(TKey key) = 0;

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

virtual void Remove(TKey key) = 0;

bool IsFull() const noexcept

{

return count == maxSize;

}

bool IsEmpty() const noexcept

{

return count == 0;

}

void Reset() noexcept

{

currPos = 0;

}

virtual bool IsEnded() const noexcept

{

return currPos == maxSize; // general case

}

virtual void Next()

{

if (IsEnded())

throw std::exception("table is ended");

currPos += 1;

}

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

};

## 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> &t);

~ScanTable() override {

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

{

delete recs[i];

}

delete[] recs;

}

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

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

void Remove(TKey key) override;

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

{

if (IsEmpty())

throw std::exception("Table\_is\_empty\n");

return recs[currPos];

}

bool IsEnded() const noexcept override

{

return currPos == count;

}

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

{

if (t.IsEmpty())

{

out << "EMPTY\_TABLE\n";

return out;

}

ScanTable<TKey, TData> tmp(t);

while (!tmp.IsEnded())

{

out << \*tmp.GetCurrent();

tmp.Next();

}

return out;

}

};

template <typename TKey, typename TData>

ScanTable<TKey, TData>::ScanTable(int size) : Table<TKey, TData>(size)

{

recs = new TabRecord<TKey, TData>\*[size];

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

{

recs[i] = nullptr;

}

}

template <typename TKey, typename TData>

ScanTable<TKey, TData>::ScanTable(const ScanTable<TKey, TData>& t) : Table<TKey, TData>(t.maxSize)

{

if (t.IsEmpty())

{

recs = new TabRecord<TKey, TData>\* [maxSize];// recs[i] = nullptr;

return;

}

count = t.count;

currPos = 0;

recs = new TabRecord<TKey, TData>\* [maxSize];

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

{

recs[i] = new TabRecord<TKey, TData>(t.recs[i]->key, t.recs[i]->data);

}

}

template <typename TKey, typename TData>

TabRecord<TKey, TData>\* ScanTable<TKey, TData>::Find(TKey key)

{

if (IsEmpty())

return nullptr;

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

{

if (recs[i]->key == key)

{

currPos = i;

return recs[i];

}

}

return nullptr;

}

template <typename TKey, typename TData>

void ScanTable<TKey, TData>::Insert(TKey key, TData\* data)

{

if (IsFull())

throw std::exception("table is full\n");

if (Find(key) != nullptr)

throw std::exception("record with this key already existed\n");

TabRecord<TKey, TData>\* res = new TabRecord<TKey, TData>(key, data);

currPos = count;

recs[count++] = res;

}

template <typename TKey, typename TData>

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

{

if (IsEmpty())

throw std::exception("table is empty\n");

if (Find(key) == nullptr)

throw std::exception("no such element\n");

delete recs[currPos];

recs[currPos] = recs[--count];

}

## Приложение Г. Реализация класса SortedTable

template <typename TKey, typename TData>

class SortedTable : public ScanTable<TKey, TData>

{

private:

void QuickSort(TabRecord<TKey, TData> \*\*rec\_arr, int size);

public:

SortedTable(int maxSize) : ScanTable(maxSize) {};

SortedTable(const ScanTable<TKey, TData>\*);

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

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

void Remove(TKey key) override;

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

{

if (IsEmpty())

throw std::exception("Table\_is\_empty\n");

return recs[currPos];

}

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

{

if (t.IsEmpty())

{

out << "EMPTY\_TABLE\n";

return out;

}

ScanTable<TKey, TData> tmp(t);

while (!tmp.IsEnded())

{

out << \*tmp.GetCurrent();

tmp.Next();

}

return out;

}

};

template <typename TKey, typename TData>

void SortedTable<TKey, TData>::QuickSort(TabRecord<TKey, TData> \*\*rec\_arr, int size)

{

int i = 0;

int j = size - 1;

TabRecord<TKey, TData> \*mid = rec\_arr[size / 2];

do

{

while (recs[i]->key < mid->key)

{

i++;

}

while (recs[j]->key > mid->key)

{

j--;

}

if (i <= j)

{

TabRecord<TKey, TData> \*tmp = rec\_arr[i];

rec\_arr[i] = rec\_arr[j];

rec\_arr[j] = tmp;

i++;

j--;

}

} while (i <= j);

if (j > 0)

{

QuickSort(rec\_arr, j + 1);

}

if (i < size)

{

QuickSort(&rec\_arr[i], size - i);

}

}

template <typename TKey, typename TData>

SortedTable<TKey, TData>::SortedTable(const ScanTable<TKey, TData>\* t) : ScanTable<TKey, TData>(\*t)

{

if (IsEmpty())

return;

this->QuickSort(this->recs, this->count);

}

template <typename TKey, typename TData>

TabRecord<TKey, TData> \*SortedTable<TKey, TData>::Find(TKey key)

{

int left = 0, right = count - 1;

TabRecord<TKey, TData>\* record = nullptr;

while (left <= right)

{

int middle = (left + right) / 2;

if (recs[middle]->key == key)

{

left = middle + 1;

right = middle;

record = recs[middle];

}

else if (recs[middle]->key > key)

right = middle - 1;

else

left = middle + 1;

}

currPos = right;

return record;

}

template <typename TKey, typename TData>

void SortedTable<TKey, TData>::Insert(TKey key, TData \*data)

{

if (IsFull())

throw std::exception("table\_is\_full\n");

if (Find(key) != nullptr)

throw std::exception("record with this key already existed\n");

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 std::exception("table\_is\_empty\n");

TabRecord<TKey, TData>\* res = Find(key);

if (res == nullptr)

throw std::exception("no\_such\_element\n");

delete res;

for (int i = currPos; i < count - 1; i++)

recs[i] = recs[i + 1];

count--;

}

## Приложение Д. Реализация класса ArrayHashTable

template <typename TKey, typename TData>

class ArrayHashTable : public HashTable<TKey, TData>

{

protected:

TabRecord<TKey, TData>\*\* recs;

TabRecord<TKey, TData>\* pMark;

int freePos;

int hashStep;

int GetNextPos(int h\_val) { return (h\_val + hashStep) % maxSize; }

public:

ArrayHashTable(int n, int step);

ArrayHashTable(const ArrayHashTable<TKey, TData> &ahtable);

~ArrayHashTable();

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

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

void Remove(TKey key);

bool Reset();

void Next();

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

{

if (IsEmpty())

throw std::exception("Table\_is\_empty\n");

return recs[currPos];

}

friend std::ostream& operator<<(std::ostream& out, const ArrayHashTable<TKey, TData>& aht)

{

if (aht.IsEmpty())

{

out << "the table is empty\n";

return out;

}

ArrayHashTable<TKey, TData> tmp(aht);

tmp.Reset();

while (!tmp.IsEnded())

{

out << \*tmp.GetCurrent();

tmp.Next();

}

return out;

}

};

template <typename TKey, typename TData>

ArrayHashTable<TKey, TData>::ArrayHashTable(int n, int step) : HashTable(n) {

hashStep = step;

recs = new TabRecord<TKey, TData>\*[n];

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

{

recs[i] = nullptr;

}

pMark = new TabRecord<TKey, TData>();

freePos = -1;

}

template <typename TKey, typename TData>

ArrayHashTable<TKey, TData>::~ArrayHashTable()

{

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

{

if (recs[i] != nullptr && recs[i] != pMark) delete recs[i];

}

delete pMark;

delete[] recs;

}

template <typename TKey, typename TData>

ArrayHashTable<TKey, TData>::ArrayHashTable(const ArrayHashTable<TKey, TData>& ahtable) :

ArrayHashTable<TKey, TData>(ahtable.maxSize, ahtable.hashStep) {

if (ahtable.IsEmpty())

{

return;

}

count = ahtable.count;

freePos = ahtable.freePos;

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]->key, ahtable.recs[i]->data); }

}

}

template <typename TKey, typename TData>

TabRecord<TKey, TData>\* ArrayHashTable<TKey, TData>::Find(const TKey key)

{

TabRecord<TKey, TData> \*res = nullptr;

int hash\_ind = hashFunc(key);

freePos = -1;

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

{

if (recs[hash\_ind] == nullptr)

{

if (freePos == -1)

freePos = hash\_ind;

break;

}

else if (recs[hash\_ind] == pMark)

{

if (freePos == -1)

freePos = hash\_ind;

}

else if (recs[hash\_ind]->key == key)

{

res = recs[hash\_ind];

currPos = hash\_ind;

break;

}

hash\_ind = GetNextPos(hash\_ind);

}

return res;

}

template <typename TKey, typename TData>

void ArrayHashTable<TKey, TData>::Insert(TKey key, TData \*data)

{

if (IsFull()) throw std::exception("table\_is\_full\n");

TabRecord<TKey, TData>\* res = Find(key);

if (freePos == -1)

throw std::exception("there are no places\n");

if (res != nullptr)

throw std::exception("there is already such key. Key must have unique value \n");

currPos = freePos;

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

count++;

}

template <typename TKey, typename TData>

void ArrayHashTable<TKey, TData>::Remove(TKey key)

{

if (IsEmpty()) throw std::exception("table is empty\n");

TabRecord<TKey, TData>\* res = Find(key);

if (res == nullptr) throw std::exception("no such element\n");

delete recs[currPos];

recs[currPos] = pMark;

count--;

}

template <typename TKey, typename TData>

bool ArrayHashTable<TKey, TData>::Reset()

{

currPos = 0;

while (!IsEnded())

{

if (recs[currPos] != nullptr && recs[currPos] != pMark)

break;

currPos++;

}

return IsEnded();

}

template <typename TKey, typename TData>

void ArrayHashTable<TKey, TData>::Next()

{

if (IsEnded())

throw std::exception("table is ended\n");

while (!IsEnded())

{

currPos++;

if (recs[currPos] != nullptr && recs[currPos] != pMark)

break;

}

}