**Российская Федерация**

**Ханты-Мансийский автономный округ – Югра**

**Департамент образования и науки**

**Сургутский государственный университет ХМАО**

Политехнический институт

Кафедра автоматики и компьютерных систем

**Пояснительная записка**

к курсовому проекту

по дисциплине «Алгоритмы и структуры данных»

Выполнил: студент группы 609-21,

Павлусенко Е.М.

Принял: старший преподаватель кафедры АиКС

Назаров Е.В.

Сургут

2024 г.

**Задание**

**Класс «Хеш-таблица»**

В соответствии с выбранным вариантом необходимо:

1. Провести анализ предметной области возможного применения проектируемого класса.
2. Провести анализ функциональности проектируемого класса.
3. Разработать интерфейс класса.
4. В соответствии с разработанным интерфейсом спроектировать тестовое приложение.
5. Выполнить проектирование класса, обоснованно выбирая: необходимые поля класса; методы класса, включив в обязательном порядке операции вставки, удаления, изменения, поиска отдельных элементов, входящих в класс.
6. Провести проектирование алгоритмов, лежащих в основе разрабатываемых методов.
7. Реализовать полученное проектное решение.
8. Реализовать тестовое приложение и провести тестирование разработанного и реализованного класса.
9. Провести исследование одной из операций (вставка, удаление, изменение, поиск) над элементами. Построить зависимость времени выполнения операции от числа элементов, над которыми она выполняется.
10. Оценить асимптотическую сложность реализованных алгоритмов вставки, удаления и поиска элементов класса.

**Аннотация**

В пояснительной записке представлен подробный обзор курсового проекта, включая анализ, проектирование, разработку, тестирование, сопровождение и код класса хеш-таблицы, написанного на языке C++. В разделе анализа описана предметная область, проанализированы существующие решения и выведены требования для разрабатываемого класса. В разделе проектирования описан математический аппарат и блок-схемы основных алгоритмов, поведенческая модель класса, представление данных при их обработке. Раздел кодирования описывает реализацию класса, отдельные его поля и функции, и решения, принятые во время разработки. В разделе тестирования проверено соответствие работы класса хеш-таблицы заданию, корректность его работы и поведение объектов класса в различных случаях. Раздел сопровождения представляет собой руководство пользователя для освоения и эффективного использования разработанного класса. В разделе исследования была проанализирована операция удаления ключа из хеш-таблицы и выведена асимптотическая сложность базовых операций разработанного класса.

**Содержание**

[Введение 5](#_Toc166482552)

[1. Анализ 6](#_Toc166482553)

[1.1. Предметные области возможного применения проектируемого класса 6](#_Toc166482554)

[1.2. Обзор существующих решений 6](#_Toc166482555)

[1.3. Функциональность проектируемого класса 8](#_Toc166482556)

[2. Проектирование 9](#_Toc166482557)

[2.1. Анализ процесса хеширования 9](#_Toc166482558)

[2.2. Модель класса 11](#_Toc166482559)

[2.3 Основные алгоритмы (блок-схемы алгоритмов), соответствующие базовым операциям класса 12](#_Toc166482560)

[3. Кодирование 15](#_Toc166482561)

[4. Тестирование 21](#_Toc166482562)

[5. Сопровождение 26](#_Toc166482563)

[6. Исследование 28](#_Toc166482564)

[Заключение 30](#_Toc166482565)

[Список использованных источников 31](#_Toc166482566)

[Приложения 32](#_Toc166482567)

# Введение

В современном мире методы эффективного управления данными являются актуальной темой в информационных технологиях. Помимо сохранения данных, еще одной актуальной проблемой является их эффективное получение из хранилища.

Даже при использовании самого быстрого алгоритма, который выполняет определенную обработку данных, производительность будет низкой, если управление данными не будет оптимизировано. Таким образом, получение данных и передача их алгоритму, а также сохранение результатов его работы становится важным компонентом любого алгоритма.

С течением времени было предложено несколько методов хранения данных и управления ими. Примерами могут служить массивы, связанные списки, деревья и графики. Эти структуры данных отлично подходят для различных целей. Но временные затраты на поиск и получение сохраненных в них данных обычно выше, чем в другой структуре данных - хэш-таблице [4]. Хеш-таблица распределяет элементы в структуре, используя хеш-функцию. Базовые операции с хеш-таблицей имеют среднюю сложность O(1) при условии правильной реализации структуры и хеш-функции.

Ожидаемым результатом этого проекта является реализация класса хеш-таблицы, осуществляющего базовые операции с данными (добавление, поиск и удаление). Хеш-таблица должна менять свой размер в соответствии с количеством элементов в ней. Класс должен обрабатывать все возникающие ошибки.

Применение этого класса обширно, поскольку он может использоваться в различных областях, включая базы данных, кеширование, ассоциативные массивы, сетевое программирование и т.д.

# Анализ

## Предметные области возможного применения проектируемого класса

На основе информации, представленной в источнике [1], в качестве возможных предметных областей проектируемого класса можно выделить следующие:

1. Базы данных

Ускорение поиска данных. Индексация данных с помощью хеш-таблиц позволяет быстро находить нужные записи без необходимости их последовательного просмотра. Также появляется возможность быстрого поиска дубликатов в наборах данных.

2. Кэширование

Хеш-таблицы являются основой многих систем кэширования. Примером может служить кэш веб-браузера, где URL-адреса страниц используются в качестве ключей, а содержимое страниц — в качестве значений. Это позволяет быстро загружать ранее посещенные страницы.

3. Ассоциативные массивы

Во многих языках программирования хеш-таблицы используются для реализации ассоциативных массивов (или словарей), где они позволяют быстро получать доступ к данным по ключу.

4. Сетевое программирование

Хеш-таблицы используются для управления сессиями и хранения данных о соединениях. Это позволяет эффективно управлять большим количеством одновременных соединений.

5. Компиляторы

Компиляторы используют хеш-таблицы для хранения информации о переменных, функциях и других идентификаторах в программах. Это обеспечивает быстрый доступ к информации о переменных при анализе и трансляции кода.

## 1.2. Обзор существующих решений

В результате анализа информации, указанной в источниках [2, 3], могут быть выделены следующие аналоги проектируемого класса:

* Класс HashSet (Множество, основанное на хеш-таблице) из стандартной библиотеки языка Java [2].

Реализует базовые функции добавления, удаления и поиска элемента.

Дополнительные функции: очистка хеш-таблицы; получение количества элементов в ней; проверка на отсутствие элементов в таблице; клонирование хеш-таблицы; итерация по элементам хеш-таблицы.

Хеш-таблица имеет несколько конструкторов. При ее создании можно указать начальный размер хеш-таблицы и коэффициент загрузки. Если они не указываются, то размер хеш-таблицы равен 16, коэффициент загрузки – 0,75. Также можно создать хеш-таблицу, подавая в конструктор коллекцию элементов. В таком случае все элементы из переданной коллекции будут помещены в хеш-таблицу.

Ключевые особенности: хеш-таблица работает с классами, у которых реализованы стандартные методы equals (проверка на равенство объектов класса) и hashCode (возвращает хеш объекта). В классах стандартной библиотеки Java эти функции реализованы, что может упростить работу с хеш-таблицей с данными классами. Однако, нет возможности переписать хеш-функцию стандартного класса, если это потребуется. Также для пользовательских классов нужно обязательно реализовать функции equals и hashCode. Функция клонирования хеш-таблицы, копирует только структуру данных, но не сами данные.

Если к хеш-таблице имеют доступ одновременно несколько потоков, синхронизацию между ними нужно делать вручную.

* Класс Hashtable (Ассоциативный массив) из стандартной библиотеки языка .NET [3].

Данная структура хранит данные в виде “ключ - значение”. Хеширование производится по ключу.

Реализует базовые функции добавления, удаления и поиска элемента по ключу.

Дополнительные функции: отдельный поиск ключа или элемента; очистка хеш-таблицы; получение количества элементов в ней; клонирование хеш-таблицы; итерация по элементам хеш-таблицы.

Хеш-таблица имеет несколько конструкторов. При ее создании можно указать начальный размер хеш-таблицы и коэффициент загрузки. Если они не указываются, то устанавливаются значениями по умолчанию. Коэффициент загрузки – 1. Также можно создать хеш-таблицу, подавая в конструктор коллекцию элементов. В таком случае все элементы из переданной коллекции будут помещены в хеш-таблицу. Из-за специфики реализации можно указать хеш-функцию для ключей.

Ключевые особенности: хеш-таблица работает с классами, у которых реализованы стандартные методы Equals (проверка на равенство объектов класса) и GetHashCode (возвращает хеш объекта). В классах стандартной библиотеки .NET эти функции реализованы, что может упростить работу с хеш-таблицей с данными классами.

Стоит заметить, что хеш-таблица не является типизированной (хранит ключи и значения в виде общего класса Object). Это позволяет хранить в хеш-таблице данные разных видов. Однако, т.к. и ключи, и значения не типизированы, их требуется явно приводить к выбранному типу, что может затруднить разработку с использованием данной структуры данных.

Функция клонирования хеш-таблицы, копирует только структуру данных, но не сами данные.

## 1.3. Функциональность проектируемого класса

Проанализировав функциональность аналогов, представленных в пункте 1.2, можно сделать вывод о том, что в проектируемом классе должны быть реализованы следующие функции (методы):

* Добавление ключа;
* Удаление ключа;
* Поиск ключа;
* Получение количества ключей в хеш-таблице;
* Очистка хеш-таблицы (удаление всех ключей из хеш-таблицы);
* Динамическое изменение размера хеш-таблицы (адаптация хеш-таблицы к количеству ключей в ней и сохранение эффективности выполнения базовых операций);

Также из анализа аналогов можно выделить предполагаемые поля класса:

* Структура данных для хранения ключей;
* Размер структуры данных;
* Количество ключей в хеш-таблице;

# 2. Проектирование

## 2.1. Анализ процесса хеширования

Хеш-таблица обязательно должна иметь структуру данных, в которой будут храниться ключи. Так как хеш-таблица может менять свой размер динамически, требуется определить, в каких случаях требуется это делать и какие переменные для этого понадобятся.

Хеширование – это преобразование ключа записей (данных) в индекс (адрес) их размещения в хеш-таблице. Оно основано на использовании хеш-функций, т.е. функций, осуществляющих преобразование. Основной задачей хеш-функций (и соответственно алгоритмов, лежащих в их основе), является получение из входных значений ключей, в общем случае имеющих произвольное распределение и диапазон значений, равномерно распределенных в заданном диапазоне значений. В основе базовых алгоритмов широко используются операции взятия остатка от деления на величины, характеризующие исходные данные и, например, на размер таблицы, а также генераторы псевдослучайных чисел. Для одинаковых ключей хеш-функция будет возвращать один и тот же индекс. Для различающихся ключей не гарантируется, что полученные через хеш-функцию индексы будут разными. Если индексы одинаковые, то эта ситуация называется коллизией.

Необходимо ввести понятие коэффициента загрузки – соотношение между количеством ключей в хеш-таблице и ее размером.

В результате анализа информации, указанной в источнике [5], могут быть выделены следующие способы разрешения коллизий:

* **Метод цепочек**

Каждая ячейка хеш-таблицы является указателем на список ключей, соответствующих одному и тому же индексу ключа. В случае возникновения коллизий операции поиска или удаления элемента требуют просмотра всех элементов соответствующего ему списка, чтобы найти в нём элемент с заданным ключом. Для добавления элемента нужно добавить элемент в конец или начало соответствующего.

Среднее время работы операции поиска элемента составляет Θ(1 + α), где α — коэффициент загрузки таблицы. Недостаток хеш-таблиц с таким способом разрешения коллизий – дополнительные затраты по памяти и усложнение кэширования хеш-таблицы, т.к. элементы производных списков разбросаны по памяти.

* **Линейное зондирование**

При возникновении коллизии алгоритм последовательно выполняет линейный поиск (обычно с шагом 1) следующей доступной ячейки хеш-таблицы. При удалении элемента требуется заменять его на специальный, заранее определенный элемент.

Линейное зондирование просто реализовать, но оно подвержено кластеризации ключей, что приводит к заметному снижению скорости поиска. Среднее время работы операции поиска элемента составляет (1+1/(1- α))/2 в успешном случае и (1+1/(1- α)^2)/2 в неуспешном. Для эффективной работы данного метода коэффициент загрузки должен быть не более 0.75 [6].

* **Квадратичное зондирование**

При возникновении коллизии алгоритм использует квадратичную функцию для определения следующего индекса ключа. Поиск выполняется до тех пор, пока не будет найден пустая ячейка. При удалении элемента алгоритм действий тот же, что и при линейном зондировании.

Квадратичное зондирование меньше подвержено кластеризации, чем линейное зондирование, но не убирает ее полностью. Среднее время работы операции поиска элемента немного ниже, чем у линейного зондирования. Для того, чтобы избавиться от эффекта кластеризации при квадратичном зондировании полностью, коэффициент загрузки хеш-таблицы должен быть не более 0.5 [6].

* **Двойное хеширование**

При возникновении коллизии алгоритм делает поиск свободной ячейки аналогично линейному пробированию. В отличие от него шаг поиска вычисляется второй, вспомогательной хеш-функцией, а значит, может быть различным для разных ключей. Значения этой хеш-функции должны быть ненулевыми и взаимно-простыми с размером хеш-таблицы. При удалении элемента алгоритм действий тот же, что и при линейном зондировании.

Среднее время работы операции поиска элемента составляет ln(1/(1- α))/α в успешном случае и 1/(1- α) в неуспешном, что ниже, чем в линейном и квадратичном зондировании. Также двойное хеширование не подвержено кластеризации. Недостатками данного способа является необходимость внимательно подбирать хеш-функции, а также дополнительные затраты из-за вычислений со вспомогательной хеш-функцией. Для эффективной работы данного метода коэффициент загрузки должен быть не более 0.75 [6].

При анализе способов разрешения коллизий был выбран метод двойного хеширования как самый эффективный. Максимальный коэффициент загрузки хеш-таблицы – 0.75. Минимальный коэффициент загрузки – 0.25.

Для широкого применения разрабатываемая хеш-таблица должна работать с произвольным типом данных. Исходя из этого требования и выбранного метода разрешения коллизий, в хеш-таблице должна быть предоставлена возможность установить 2 пользовательские хеш-функции, работающие с выбранным типом данных.

## 2.2. Модель класса

Класс хеш-таблицы будет иметь следующие поля:

* Структура, в которой хранятся данные;
* Количество ключей в таблице;
* Размер таблицы (при создании равен 8);
* Значение, представляющее удаленный ключ;
* Значение, представляющее пустой ключ;
* Основная и вспомогательная хеш-функции;

Функции взаимодействия с хеш-таблицей:

* Добавление элемента. Передается добавляемый ключ. Вычисляется основной и вспомогательный хеш ключа. Если ключ уже есть в таблице, функция завершает работу. Если определенная ячейка таблицы уже занята другим ключом, начинается зондирование до тех пор, пока не будет найден либо пустой/удаленный ключ, либо сам ключ. В случае превышения максимального коэффициента загрузки произойдет увеличение размера таблицы в 2 раза.
* Удаление элемента. Передается добавляемый ключ. Вычисляется основной и вспомогательный хеш ключа. Если ключ есть в таблице, он удаляется, функция завершает работу. Если определенная ячейка таблицы уже занята другим ключом, начинается зондирование до тех пор, пока не будет найден либо пустой/удаленный ключ, либо сам ключ. Если текущий коэффициент загрузки ниже минимального коэффициента, произойдет уменьшение размера таблицы в 2 раза.
* Поиск элемента. Передается добавляемый элемент. Вычисляется основной и вспомогательный хеш ключа. Если ключ уже есть в таблице, функция завершает работу и возвращает истину. Если определенная ячейка таблицы уже занята другим ключом, начинается зондирование до тех пор, пока не будет найден либо пустой/удаленный ключ, либо сам ключ. Если искомый ключ не был найден, функция возвращает ложь.
* Получение количества ключей в таблице.
* Очистка хеш-таблицы.

## 2.3 Основные алгоритмы (блок-схемы алгоритмов), соответствующие базовым операциям класса

Блок-схема алгоритма добавления ключа в хеш-таблицу представлена на Рисунке 1.

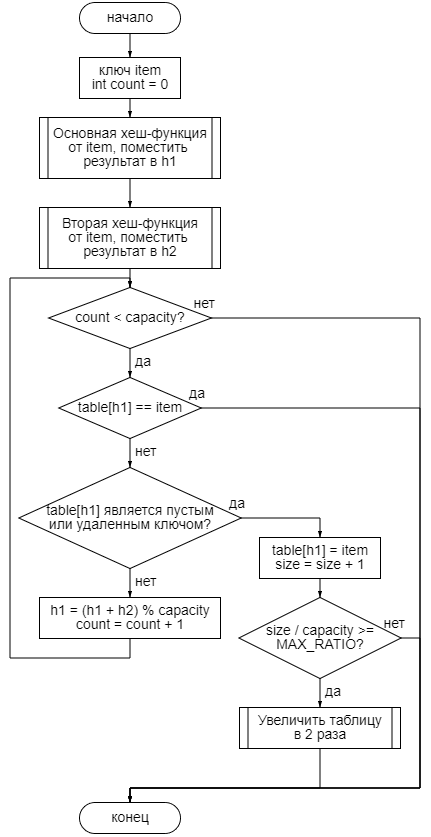


Рисунок 1. Блок-схема алгоритма добавления ключа в хеш-таблицу

Item – передаваемый ключ, count – текущая итерация прохода по таблице, h1 и h2 – числа, получаемые в результате выполнения хеш-функций. Table – структура для хранения ключей, size – количество ключей в ней, capacity – размер таблицы. MAX\_RATIO – максимальный коэффициент загрузки хеш-таблицы.

Блок-схема алгоритма удаления ключа из хеш-таблицы представлена на Рисунке 2.

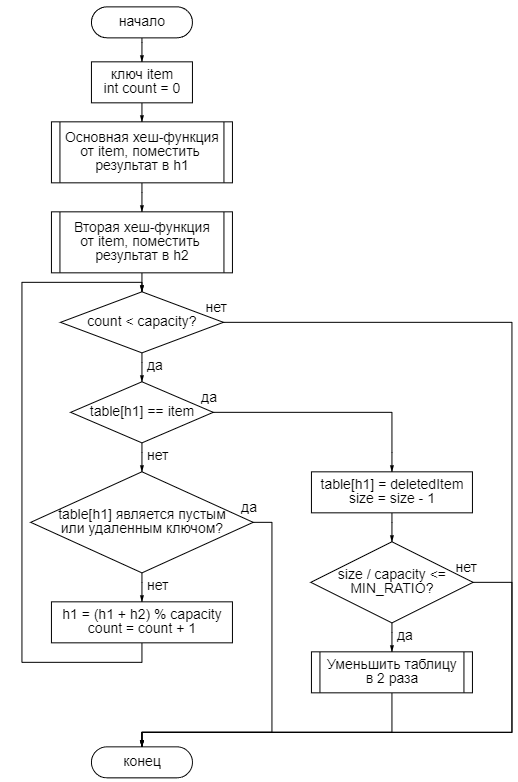


Рисунок 2. Блок-схема алгоритма удаления ключа из хеш-таблицы

Item – передаваемый ключ, count – текущая итерация прохода по таблице, h1 и h2 – числа, получаемые в результате выполнения хеш-функций. Table – структура для хранения ключей, size – количество ключей в ней, capacity – размер таблицы. MIN\_RATIO – минимальный коэффициент загрузки хеш-таблицы, deletedItem – удаленный ключ.

Блок-схема алгоритма поиска ключа в хеш-таблице представлена на Рисунке 3.

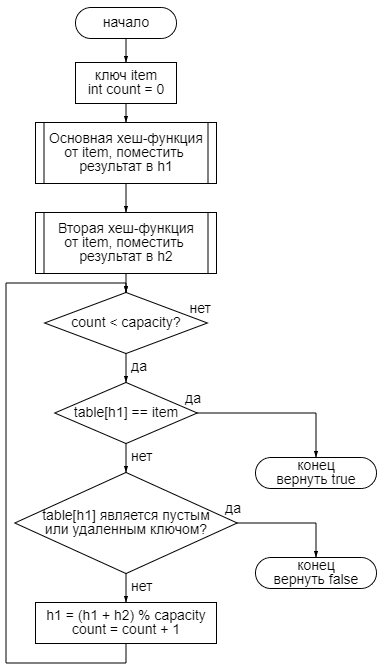
****

Рисунок 3. Блок-схема алгоритма поиска ключа в хеш-таблице

Item – передаваемый ключ, count – текущая итерация прохода по таблице, h1 и h2 – числа, получаемые в результате выполнения хеш-функций. Table – структура для хранения ключей, size – количество ключей в ней, capacity – размер таблицы.

# 3. Кодирование

Так как хеш-таблица должна работать с произвольным типом данных и принимать пользовательские хеш-функции, было принято решение использовать инструмент шаблонов в C++. Он позволяет работать с различными типами данных в функциях и классах.

Шаблонными типами данных являются:

* T – класс ключа
* H1 – класс основной хеш-функции
* HHРрH2 – класс вспомогательной хеш-функции

Для работы с пользовательскими типами требуется, чтобы у класса ключа был реализован оператор ==, а у классов хеш-функций – оператор (), принимающий ключ и размер хеш-таблицы. Данные функции вычисляют индекс ключа в хеш-таблице.

В классе установлены следующие поля и типы:

* Структура для хранения ключей (table) – динамический массив vector с типом ключей T. Позволяет динамически изменять размер и имеет сложность доступа к ключам по индексу O(1).
* Количество ключей в таблице (size) – int.
* Общий размер таблицы (capacity) – int.
* Значение, представляющее удаленный ключ (deletedItem) – Т.
* Значение, представляющее пустой ключ (emptyItem) – Т.
* Основная хеш-функция (hasher) – H1.
* Вспомогательная хеш-функция (hasherProbe) – H2.

Вся логика, описанная в проектировании, реализована в этом классе. Для работы класса была использована библиотека vector. Все поля хранения данных были помещены в приватное пространство класса, так как их произвольное изменение может привести к неправильному функционированию класса. Также все методы класса реализованы в заголовочном файле для правильной работы компилятора С++ с шаблонными типами.

У класса HashTable имеется единственный конструктор (Листинг 1).

*Листинг 1. Конструктор хеш-таблицы*

1. HashTable(const H1& hasher, const H2& hasherProbe, const T& emptyItem, const T& deletedItem) :
2. table(MIN\_CAPACITY, emptyItem),
3. capacity(MIN\_CAPACITY),
4. emptyItem(emptyItem),
5. deletedItem(deletedItem),
6. hasher(hasher),
7. hasherProbe(hasherProbe),
8. size(0) {}

Структура для хранения ключей заполняется пустыми ключами, ее размер по умолчанию равен 8.

У класса также определены константы:

* MIN\_CAPACITY = 8 – минимальный размер хеш-таблицы
* MAX\_RATIO = 0.75 – максимальный коэффициент загрузки хеш-таблицы
* MIN\_RATIO = 0.25 – минимальный коэффициент загрузки хеш-таблицы

Далее представлено описание работы функции добавления ключа в хеш-таблицу (Листинг 2).

*Листинг 2. Функция добавления ключа в хеш-таблицу*

1. bool Add(const T& item){
2. int h1 = hasher(item, capacity);
3. if (table[h1] == emptyItem || table[h1] == deletedItem) {
4. table[h1] = item;
5. size++;
6. if (size / (double)capacity >= MAX\_RATIO) {
7. resize(capacity \* 2);
8. }
9. return true;
10. }
11. if (table[h1] == item) {
12. return false;
13. }
15. int h2 = hasherProbe(item, capacity);
16. for (int i = 1; i != capacity; i++) {
17. h1 = (h1 + h2) % capacity;
18. if (table[h1] == item) {
19. return false;
20. }
21. if (table[h1] == emptyItem || table[h1] == deletedItem) {
22. table[h1] = item;
23. size++;
24. if (size / (double)capacity >= MAX\_RATIO && capacity > MIN\_CAPACITY) {
25. resize(capacity \* 2);
26. }
27. return true;
28. }
29. }
30. return false;
31. }

Функция принимает ссылку на ключ, следовательно работа будет производиться над самой строкой, а не ее локальной копией. Таким образом устроены все функции, доступные пользователю. В начале работы производится попытка вставить ключ без использования вспомогательной хеш-функции. Если был найден пустой или удаленный ключ, то переданный ключ вставляется в таблицу. Если после вставки был пересечен максимальный коэффициент загрузки хеш-таблицы, то размер хеш-таблицы увеличивается в два раза. В случае, если не удалось вставить ключ сразу, вычисляется вспомогательная хеш-функция и производится итерация по таблице с использованием хеш-функций. Алгоритм действий при нахождении пустого или удаленного элемента такой же. Количество итераций равно общему размеру хеш-таблицы.

Функции добавления и удаления ключей возвращают логическое значение. Они используются для того, чтобы понять, была ли операция успешной или нет.

Далее представлено описание работы функции удаления ключа из хеш-таблицы (Листинг 3).

*Листинг 3. Функция удаления ключа из хеш-таблицы*

1. bool Delete(const T& item){
2. int h1 = hasher(item, capacity);
3. if (table[h1] == item) {
4. size--;
5. table[h1] = deletedItem;
6. if(size / (double)capacity <= MIN\_RATIO){
7. resize(capacity / 2);
8. }
9. return true;
10. }
11. if(table[h1] == emptyItem){
12. return false;
13. }
14. int h2 = hasherProbe(item, capacity);
15. for (int i = 1; i != capacity; i++) {
16. h1 = (h1 + h2) % capacity;
17. if (table[h1] == item) {
18. table[h1] = deletedItem;
19. size--;
20. if(size / (double)capacity <= MIN\_RATIO && capacity > MIN\_CAPACITY){
21. resize(capacity / 2);
22. }
23. return true;
24. }
25. if (table[h1] == emptyItem) {
26. return false;
27. }
28. }
29. return false;
30. }

В начале работы производится попытка удалить ключ без использования вспомогательной хеш-функции. Если был найден искомый ключ, то он заменяется удаленным ключом. Если после удаления был пересечен минимальный коэффициент загрузки хеш-таблицы, то размер хеш-таблицы уменьшается в два раза. В случае, если не удалось удалить искомый ключ сразу, вычисляется вспомогательная хеш-функция и производится итерация по таблице с использованием хеш-функций. Алгоритм действий при нахождении искомого ключа такой же. Количество итераций равно общему размеру хеш-таблицы.

При удалении ключа может возникнуть ситуация, когда был пересечен минимальный коэффициент загрузки хеш-таблицы, в то время как размер таблицы равен минимальному. В таком случае изменения размера таблицы не происходит.

Алгоритм работы поиска ключа в хеш-таблице схож с алгоритмом удаления ключа. Однако при нахождении искомого элемента только возвращается значение true, сигнализирующее о том, что элемент присутствует в таблице.

Далее представлено описание работы функции изменения размера таблицы (Листинг 4).

*Листинг 4. Функция изменения размера хеш-таблицы*

1. void resize(int new\_capacity){
2. int oldCapacity = capacity;
3. size = 0;
4. capacity = new\_capacity;
5. std::vector<T> oldTable(new\_capacity, emptyItem);
6. std::swap(oldTable, table);
7. for(int i = 0; i < oldCapacity; i++){
8. if(oldTable[i] != deletedItem && oldTable[i] != emptyItem){
9. Add(oldTable[i]);
10. }
11. }
12. }

Данная функция находится в приватном пространстве класса, т.е. пользователь не имеет права вызывать данную функцию.

В функцию передается новый размер хеш-таблицы. Старый размер сохраняется. Обновляются количество ключей и размер хеш-таблицы. Также создается динамический массив с новым размером. Он меняется местами с тем массивом, который был ранее у хеш-таблицы. Из старого массива все непустые и неудаленные ключи помещаются в хеш-таблицу с новым массивом с использованием функции добавления ключа. Динамические массивы имеют реализованный деструктор, поэтому дополнительных действий для удаления старого массива производить не нужно.

Представлено описание работы функции очистки хеш-таблицы (Листинг 5).

*Листинг 5. Функция очистки хеш-таблицы*

1. void Clear(){
2. capacity = MIN\_CAPACITY;
3. table.resize(MIN\_CAPACITY);
4. table.clear();
5. for(int i = 0; i < capacity; i++){
6. table[i] = emptyItem;
7. }
8. size = 0;
9. }

Размер хеш-таблицы становится равным минимальным, внутренний массив также уменьшается до этого размера и очищается. Все его элементы устанавливаются пустыми ключами.

# 4. Тестирование

Для тестирования разработанного класса будет использоваться специальное приложение (см. Приложение 2), основанное на тестируемом классе. Для упрощения тестирования хеш-таблицы в качестве ключей был выбран примитивный тип данных int.

Пользовательский интерфейс выглядит следующим образом (Рисунок 4):

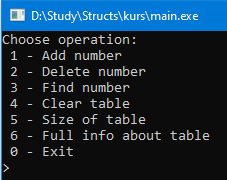


Рисунок 4. Внешний вид пользовательского интерфейса

Назначение элементов интерфейса тестового приложения:

1. Добавление числа в хеш-таблицу. Приложение запрашивает ввод числа для его добавления в хеш-таблицу. Число добавляется в хеш-таблицу в случае его отсутствия в ней. В противном случае будет выведено сообщение о наличии числа в хеш-таблице.

2. Удаление числа из хеш-таблицы. Приложение запрашивает ввод числа для его удаления из хеш-таблицы. Число удаляется из хеш-таблицы в случае его наличия в ней. В противном случае будет выведено сообщение о отсутствии числа в хеш-таблице.

3. Поиск числа в хеш-таблице. Приложение запрашивает ввод числа для его поиска в хеш-таблице. Затем выводится сообщение о наличии или отсутствии введенного числа в хеш-таблице.

4. Очистка хеш-таблицы.

5. Вывод количества ключей в хеш-таблице.

6. Вывод расширенной информации о хеш-таблице. Данная команда выводит количество ключей в таблице, ее размер, текущий коэффициент загрузки и распределение ключей (Рисунок 5). Команда предназначена для проверки работы хеш-таблицы.

0. Завершение работы тестового приложения. После будет предложено нажать любую кнопку для закрытия консоли.

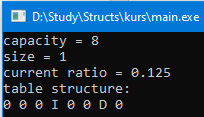


Рисунок 5. Вывод информации о хеш-таблице

Классификация ключей по символам:

* 0 – пустой ключ
* D – удаленный ключ
* I – стандартный ключ, записанный в таблицу посредством вставки

После создания хеш-таблица находится в следующем состоянии (Рисунок 6):

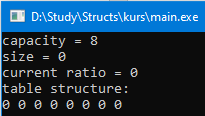


Рисунок 6. Информация о хеш-таблице после создания

В хеш-таблицу были добавлены следующие числа: 77 64 53 24 29. Все числа были добавлены успешно, хеш-таблица выглядит следующим образом (Рисунок 7):

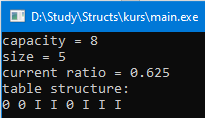


Рисунок 7. Хеш-таблица после добавления 5 ключей

Далее была произведена проверка функции, возвращающей количество ключей в текущей хеш-таблице. Результат представлен на Рисунке 8.



Рисунок 8. Количество ключей в текущей хеш-таблице

При попытке вставить в хеш-таблицу ключ, который уже там находится, программа выведет соответствующее сообщение (Рисунок 9):

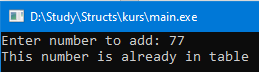


Рисунок 9. Попытка вставить имеющийся ключ в хеш-таблицу

Далее была произведена проверка функции, проверяющей наличие ключа в хеш-таблице для существующего (64, Рисунок 10) и несуществующего (65, Рисунок 11) ключа.

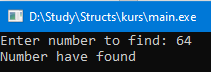


Рисунок 10. Успешный поиск для существующего ключа

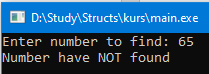


Рисунок 11. Неуспешный поиск для несуществующего ключа

После добавления еще одного элемента (11) таблица увеличивается в два раза из-за достижения максимального коэффициента загрузки (Рисунок 12).

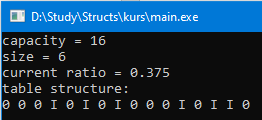


Рисунок 12. Увеличенная после добавления ключа хеш-таблица

Распределение ключей в хеш-таблице также изменилось.

После удаления ключа 77 хеш-таблица имеет следующий вид (Рисунок 13):

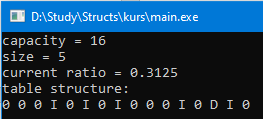


Рисунок 13. Хеш-таблица с удаленным ключом

При попытке удалить несуществующий ключ (удаленный 77) программа выведет сообщение об ошибке (Рисунок 14).

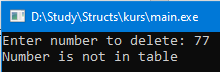


Рисунок 14. Попытка удалить несуществующий ключ

Было произведено удаление еще одного ключа (53), и хеш-таблица уменьшилась в два раза из-за достижения минимального коэффициента загрузки (Рисунок 15).

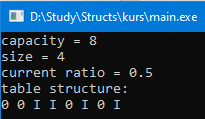


Рисунок 15. Уменьшенная после удаления ключа хеш-таблица

При удалении ключей и достижении минимального коэффициента загрузки при минимальном размере хеш-таблицы ее размер уменьшаться не будет (Рисунок 16).

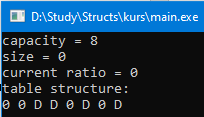


Рисунок 16. Хеш-таблица со всеми удаленными ключами

Из данного примера может возникнуть потребность в рехешировании ключей хеш-таблицы, когда удаленных ключей становится гораздо больше, чем стандартных.

При использовании функции очистки хеш-таблицы (“Clear table”) хеш-таблица становится такой же, как и при ее создании (Рисунок 6).

Таким образом было проведено тестирование разработанного класса. Класс соответствует заданию на проектирование и готов к эксплуатации.

# 5. Сопровождение

Реализация класса представлена в Приложении 1.

**Поля класса**

* vector<T> table – динамический массив для хранения ключей;
* int size – количество ключей в таблице;
* int capacity – размер массива;
* T emptyItem – значение, представляющее пустой ключ;
* T deletedItem – значение, представляющее удаленный ключ;
* H1 hasher – основная хеш-функция;
* H2 hasherProbe – вспомогательная хеш-функция.

Примечание: данные поля закрыты для редактирования и чтения. Некоторые из них можно задать при создании, другие можно прочитать, используя специализированные функции.

**Создание объекта класса**

Класс имеет конструктор HashTable(const H1& hasher, const H2& hasherProbe, const T& emptyItem, const T& deletedItem) и конструктор копирования (созданный по умолчанию).

Требования к классам:

* Классы (структуры) H1 и H2 должны иметь реализованный оператор (), принимающий аргументы item (ссылка на ключ) и int capacity (размер таблицы). Он должен возвращать индекс ячейки в хеш-таблице. Ошибки выхода за пределы массива не обрабатываются.
* Класс T должен иметь реализованный оператор ==. Для типов из стандартной библиотеки C++ данный оператор уже реализован. При желании использовать конструктор копирования пользователю нужно удостовериться, чтобы при копировании объектов класса T не возникало утечек памяти, если таковые могут возникнуть.

В качестве специальных значений, определяющих пустой и удаленный ключ, следует использовать те, которые не могут использоваться для решения пользовательской задачи. Также значения для пустого и удаленного ключа должны быть разными.

После создания хеш-таблицы разработанным конструктором она имеет 0 ключей и размер = 8. Размер хеш-таблицы всегда равен степени двойки, поэтому нужно внимательно подходить к разработке хеш-функций.

**Методы класса**

* Add – добавление ключа в хеш-таблицу. Передается сам ключ. Возвращает true, если ключ был успешно добавлен, false, если таковой уже находится в хеш-таблице.
* Contains – проверка наличия ключа в хеш-таблице. Передается требуемый ключ. Возвращает логическое значение.
* Delete – удаление ключа из хеш-таблицы. Передается требуемый ключ. Возвращает true, если ключ был успешно удален, false, если таковой не был найден в хеш-таблице.
* Size – получение количества ключей в хеш-таблице.
* Clear – очистка хеш-таблицы.
* DebugInfo - Вывод полной информации о хеш-таблице в консоль. Пример вывода функции представлен на Рисунке 17.

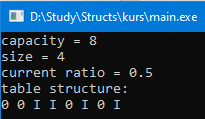


Рисунок 17. Пример работы функции DebugInfo.

**Пользовательский модуль**

Для ознакомления с возможностями класса предлагается использовать модуль, описанный в Приложении 2.

Меню программы выглядит следующим образом (Рисунок 18):

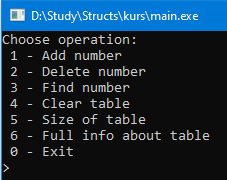


Рисунок 18. Внешний вид интерфейса пользовательского модуля

# 6. Исследование

В данной работе будет проводиться исследование операции удаления ключа из хеш-таблицы. В качестве типа ключей был выбран int. Код для исследования представлен в Приложении 3.

Таблица определенного размера предварительно заполняется случайными уникальными числами. Эти же числа вставляются в массив чисел. Затем 25 раз из массива выбирается случайное число и производится его удаление из хеш-таблицы. Измеряется среднее время удаления (мс) из 25 итераций и выводится на консоль. Данный алгоритм выполняется для 10 размеров хеш-таблицы. Пример выполнения программы представлен на Рисунке 19.

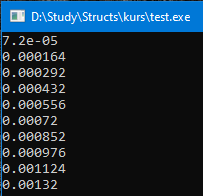


Рисунок 19. Работа программы из Приложения 3

Также было измерено среднее количество дополнительных проходов по таблице при удалении. Результаты измерений представлены в Таблице 1. График зависимости времени удаления от количества ключей представлен в Рисунке 20.

Таблица 1. Среднее время удаления и среднее количество проходов

|  |  |  |
| --- | --- | --- |
| Количество ключей | Среднее время удаления (мс) | Среднее количество проходов при удалении |
| 100000 | 0,000072 | 0,2 |
| 200000 | 0,000164 | 0,44 |
| 300000 | 0,000292 | 0,4 |
| 400000 | 0,000432 | 0,32 |
| 500000 | 0,000556 | 0,36 |
| 600000 | 0,000720 | 0,52 |
| 700000 | 0,000852 | 0,48 |
| 800000 | 0,000976 | 0,4 |
| 900000 | 0,001124 | 0,48 |
| 1000000 | 0,001320 | 0,44 |

Рисунок 20. График зависимости времени удаления от количества ключей

Как видно из графика, время удаления линейно зависит от количества ключей в хеш-таблице. При этом среднее количество дополнительных проходов ключей остается примерно одним и тем же (0,4). Это позволяет сделать вывод, что время выполнения операции удаления не зависит от спроектированной логики класса хеш-таблицы. Можно предположить, что время увеличивается либо из-за математических операций, используемых в хеш-функциях и методах хеш-таблицы, либо из-за оператора доступа по индексу в динамическом массиве. Несмотря на линейное увеличение времени удаления, оно остается очень низким.

**Асимптотическая сложность реализованных базовых алгоритмов**

Добавление ключа. В случае, если без использования вспомогательной хеш-функции найден пустой ключ или тот же, что и добавляемый, сложность составляет O(1) (Лучший случай). В худшем случае, если потребовалось использовать вспомогательную функцию и сделать максимум проходов по таблице, сложность составит O(N). Стоит отметить, что возможность возникновения худшего случай будет зависеть от реализации хеш-функций и самих ключей. При правильном подборе хеш-функций худший случай труднодостижим, а сложность в среднем случае равна О(1), что подтверждается экспериментом, выполненным ранее.

Алгоритмы удаления и поиска ключа схожи с алгоритмом добавления ключа. В лучшем и среднем случае сложность составляет О(1), в худшем – О(N).

# Заключение

В рамках курсового проекта был разработан класс хеш-таблицы. Реализованный класс обладает гибкостью и отличается функциональностью, позволяя пользователю задавать свой тип ключей и свои хеш-функции для ключей. Класс реализует алгоритмы вставки, удаления и поиска ключа, а также очистку и вывод количества ключей в хеш-таблице.

Тестирование показало, что логика класса реализована корректно.

Разработанный класс может найти применение в образовательных целях для изучения хеш-таблиц, в базах данных, ассоциативных массивах и любом другом программном обеспечении, где требуется доступ к данным за время O(1).

В качестве перспектив развития проекта можно:

* Добавить рехеширование хеш-таблицы, когда количество удаленных ключей гораздо больше количества реальных ключей.
* Добавить обработку ошибок выхода за пределы массива после вычисления пользовательских хеш-функций.
* Добавить метод, возвращающий итератор хеш-таблицы. Это позволит пользователю просмотреть все ключи, которые в данный момент находятся в хеш-таблице.
* Рассмотреть установку минимальной и максимальной границ коэффициента загрузки хеш-таблицы пользователем.

# Список использованных источников

1. Что такое хеширование и хеш | Функции, таблицы, пароли и файлы. URL: [https://pro32.com/ru/article/chto-takoe-kheshirovanie-i-khesh-funktsii-tablitsy-paroli-i-fayly/#:~:text=Области%20применения%20хеш-таблиц%3A,доступ%20к%20данным%20по%20ключу.](https://pro32.com/ru/article/chto-takoe-kheshirovanie-i-khesh-funktsii-tablitsy-paroli-i-fayly/%23:~:text=Области%20применения%20хеш-таблиц%3A,доступ%20к%20данным%20по%20ключу.)
2. Документация к классу HashSet из стандартной библиотеки языка Java. URL: <https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html>
3. Документация к классу Hashtable из стандартной библиотеки языка .NET. URL: <https://learn.microsoft.com/en-us/dotnet/api/system.collections.hashtable?view=netcore-3.0>
4. Хеш-таблица. URL: <https://ru.wikipedia.org/wiki/Хеш-таблица>
5. Разрешение коллизий в хеш-таблицах. URL: <https://www.educative.io/answers/hash-table-collision-resolution>
6. Сравнительный анализ методов линейного зондирования, квадратичного зондирования и двойного хеширования для выявления коллизии в хэш-таблице. URL:

<https://www.academia.edu/6876268/COMPARATIVE_ANALYSIS_OF_LINEAR_PROBING_QUADRATIC_PROBING_AND_DOUBLE_HASHING_TECHNIQUES_FOR_RESOLVING_COLLUSION_IN_A_HASH_TABLE>

# Приложения

Приложение 1. Класс хеш-таблицы

1. #include<vector>
2. #include<iostream>
3. #define MIN\_CAPACITY 8
4. #define MAX\_RATIO 0.75
5. #define MIN\_RATIO 0.25
6. template <class T, class H1, class H2>
7. class HashTable{
8. private:
9. std::vector<T> table;
10. int size;
11. int capacity;
12. T deletedItem;
13. T emptyItem;
14. H1 hasher;
15. H2 hasherProbe;
16. void resize(int new\_capacity){
17. int oldCapacity = capacity;
18. size = 0;
19. capacity = new\_capacity;
20. std::vector<T> oldTable(new\_capacity, emptyItem);
21. std::swap(oldTable, table);
22. for(int i = 0; i < oldCapacity; i++){
23. if(oldTable[i] != deletedItem && oldTable[i] != emptyItem){
24. Add(oldTable[i]);
25. }
26. }
27. }
28. public:
29. HashTable(const H1& hasher, const H2& hasherProbe, const T& emptyItem, const T& deletedItem) :
30. table(MIN\_CAPACITY, emptyItem),
31. capacity(MIN\_CAPACITY),
32. emptyItem(emptyItem),
33. deletedItem(deletedItem),
34. hasher(hasher),
35. hasherProbe(hasherProbe),
36. size(0) {}
38. bool Add(const T& item){
39. int h1 = hasher(item, capacity);
40. if (table[h1] == emptyItem || table[h1] == deletedItem) {
41. table[h1] = item;
42. size++;
43. if (size / (double)capacity >= MAX\_RATIO) {
44. resize(capacity \* 2);
45. }
46. return true;
47. }
48. if (table[h1] == item) {
49. return false;
50. }
52. int h2 = hasherProbe(item, capacity);
53. for (int i = 1; i != capacity; i++) {
54. h1 = (h1 + h2) % capacity;
55. if (table[h1] == item) {
56. return false;
57. }
58. if (table[h1] == emptyItem || table[h1] == deletedItem) {
59. table[h1] = item;
60. size++;
61. if (size / (double)capacity >= MAX\_RATIO) {
62. resize(capacity \* 2);
63. }
64. return true;
65. }
66. }
67. return false;
68. }
69. bool Contains(const T& item) const {
70. int h1 = hasher(item, capacity);
71. if(table[h1] == item){
72. return true;
73. }
74. if(table[h1] == emptyItem){
75. return false;
76. }
77. int h2 = hasherProbe(item, capacity);
78. int i = 1;
79. while(i < capacity && table[h1] != emptyItem){
80. h1 = (h1 + h2) % capacity;
81. if (table[h1] == item && table[h1] != deletedItem)
82. return true;
83. i++;
84. }
85. return false;
86. }
87. bool Delete(const T& item){
88. int h1 = hasher(item, capacity);
89. if (table[h1] == item) {
90. size--;
91. table[h1] = deletedItem;
92. if(size / (double)capacity <= MIN\_RATIO && capacity > MIN\_CAPACITY){
93. resize(capacity / 2);
94. }
95. return true;
96. }
97. if(table[h1] == emptyItem){
98. return false;
99. }
100. int h2 = hasherProbe(item, capacity);
101. for (int i = 1; i != capacity; i++) {
102. h1 = (h1 + h2) % capacity;
103. if (table[h1] == item) {
104. table[h1] = deletedItem;
105. size--;
106. if(size / (double)capacity <= MIN\_RATIO && capacity > MIN\_CAPACITY){
107. resize(capacity / 2);
108. }
109. return true;
110. }
111. if (table[h1] == emptyItem) {
112. return false;
113. }
114. }
115. return false;
116. }
117. int Size() const { return size; }
118. void Clear(){
119. capacity = MIN\_CAPACITY;
120. table.resize(MIN\_CAPACITY);
121. table.clear();
122. for(int i = 0; i < capacity; i++){
123. table[i] = emptyItem;
124. }
125. size = 0;
126. }
127. void DebugInfo() const {
128. std::cout << "capacity = " << capacity << "\n";
129. std::cout << "size = " << size << "\n";
130. std::cout << "current ratio = " << size / (double)capacity << "\n";
131. std::cout << "table structure:\n";
132. for(int i = 0; i < capacity; i++){
133. if(table[i] == emptyItem) {
134. std::cout << "0 ";
135. continue;
136. }
137. if(table[i] == deletedItem) {
138. std::cout << "D ";
139. continue;
140. }
141. std::cout << "I ";
142. }
143. std::cout << "\n\n";
144. }
145. };

Приложение 2. Код тестового приложения

1. #include<iostream>
2. #include<limits>
3. #include"table.h"
4. using namespace std;
5. struct Hasher {
6. int operator()(const int &item, int capacity) const {
7. return (item \* 1091) % capacity;
8. }
9. };
10. struct HasherProbe {
11. int operator()(const int &item, int capacity) const {
12. return (item \* 1039) % capacity;
13. }
14. };
15. void clearConsole(){
16. cin.clear();
17. cin.ignore(numeric\_limits<streamsize>::max(), '\n');
18. }
19. int inputNumber(){
20. int k;
21. while(!(cin >> k)){
22. cout << "Wrong number! Try again: ";
23. clearConsole();
24. }
25. return k;
26. }
27. int main(){
28. Hasher h1;
29. HasherProbe h2;
30. int emptyItem = INT\_MIN, deletedItem = INT\_MAX;
31. HashTable<int, Hasher, HasherProbe> table(h1, h2, emptyItem, deletedItem);
33. int k = -1, number;
34. while(1){
35. cout << "Choose operation:\n 1 - Add number\n 2 - Delete number\n 3 - Find number"
36. "\n 4 - Clear table\n 5 - Size of table\n 6 - Full info about table\n 0 - Exit\n> ";
37. k = inputNumber();
38. system("cls");
39. if(k == 0) break;
40. switch(k) {
41. case 1:
42. cout << "Enter number to add: ";
43. number = inputNumber();
44. if(table.Add(number)){
45. cout << "Number was added to table\n\n";
46. } else {
47. cout << "This number is already in table\n\n";
48. }
50. break;
51. case 2:
52. cout << "Enter number to delete: ";
53. number = inputNumber();
54. if(table.Delete(number)){
55. cout << "Number was deleted from table\n\n";
56. } else {
57. cout << "Number is not in table\n\n";
58. }
60. break;
61. case 3:
62. cout << "Enter number to find: ";
63. number = inputNumber();
64. if(table.Contains(number)){
65. cout << "Number have found\n\n";
66. } else {
67. cout << "Number have NOT found\n\n";
68. }
69. break;
70. case 4:
71. table.Clear();
72. cout << "Table was cleared\n\n";
73. break;
74. case 5:
75. cout << "Size is " << table.Size() << "\n\n";
76. break;
77. case 6:
78. table.DebugInfo();
79. break;
80. default:
81. cout << "Error. Wrong number.\n\n";
82. break;
83. }
84. }
85. system("pause");
86. return 0;
87. }

Приложение 3. Код для исследования операции удаления

1. #include<iostream>
2. #include<chrono>
3. #include<random>
4. #include"table.h"
5. using namespace std;
6. struct Hasher {
7. int operator()(const int &item, int capacity) const {
8. return item % capacity;
9. }
10. };
11. struct HasherProbe {
12. int operator()(const int &item, int capacity) const {
13. return capacity - 1 - (item % (capacity - 1));
14. }
15. };
16. int randnum(int min, int max) {
17. random\_device rd;
18. mt19937 gen(rd());
19. uniform\_int\_distribution<int> distribution(min, max);
20. return distribution(gen);
21. }
22. void generateKeys(HashTable<int, Hasher, HasherProbe> &table, int \*arr, int n, int min, int max){
23. int key, i = 0;
25. while(i < n){
26. key = randnum(min, max);
27. if(table.Add(key)){
28. arr[i] = key;
29. i++;
30. }
31. }
32. }
33. void measureDeleteTime(HashTable<int, Hasher, HasherProbe> &table, int \*arr, int size, int min, int max){
34. chrono::duration<double, milli> total;
35. int item;
36. total = total.zero();
37. for(int n = size/10; n < size + 1; n += size/10){
38. generateKeys(table, arr, n, min, max);
39. for(int i = 0; i < 25; i++){
41. item = arr[randnum(0, n - 1)];
42. auto start = chrono::high\_resolution\_clock::now();
43. table.Delete(item);
44. auto end = chrono::high\_resolution\_clock::now();
45. chrono::duration<double, milli> duration = end - start;
46. total += duration;
47. }
48. table.Clear();
49. cout << total.count() / 25 << "\n";
50. }
51. }
52. #define N 1000000
53. int main(){
54. Hasher h1;
55. HasherProbe h2;
56. int emptyItem = INT\_MIN, deletedItem = INT\_MAX;
57. HashTable<int, Hasher, HasherProbe> table(h1, h2, emptyItem, deletedItem);
58. int \*keys = new int[N];
59. measureDeleteTime(table, keys, N, 0, 10000000);
60. system("pause");
61. return 0;
62. }