МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Хеш-таблицы с цепочками

Студент гр. 8381 _____ Киреев К.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Киреев К.А.

Группа 8381

Тема работы: Хеш-таблицы с цепочками

Исходные данные: необходимо провести исследование алгоритма вставки в хеш-таблицу с цепочками, включающее генерацию входных данных, использование их для измерения количественных характеристик алгоритмов, сравнение экспериментальных результатов с теоретическими.

Содержание пояснительной записки:
«Содержание», «Введение», «Задание», «Описание программы», «Тестирование», «Исследование», «Заключение», «Список использованных источников».

Предполагаемый объем пояснительной записки:
Не менее 40 страниц.

Дата выдачи задания:

Дата сдачи реферата:

Дата защиты реферата:

Студент _____ Киреев К.А,

Преподаватель _____ Жангиров Т.Р.

АННОТАЦИЯ

В ходе выполнения курсовой работы была разработана программа с GUI, позволяющая исследовать алгоритм вставки в хеш-таблицу с цепочками с двумя различными хеш-функциями. Программа обладает следующей функциональностью: создание файлов, содержащих входные данные для хеш-таблицы при среднем и худшем случае вставки, создание хеш-таблицы по заданному файлу, хеш-функции и/или длине ключа; удаление и добавление элементов в хеш-таблицу. В результате исследования было выявлено, что выбор хеш-функции, количества элементов, а также длины ключей элементов хеш-таблицы значительно влияют на скорость работы алгоритма в среднем случае.

SUMMARY

In the course of the course work, a program was developed with a GUI that allows you to explore the algorithm for inserting into a hash table with chains with two different hash functions. The program has the following functionality: creating files containing input data for a hash table with an average and worst case of insertion, creating a hash table for a given file, hash function and / or key length; delete and add items to the hash table. As a result of the study, it was revealed that the choice of the hash function, the number of elements, and also the key lengths of the hash table elements significantly affect the speed of the algorithm in the average case.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы

Реализация и экспериментальное машинное исследование алгоритмов кодирования (Фано-Шеннона, Хаффмана), быстрого поиска на основе БДП или хеш-таблиц, сортировок.

Основные задачи

Генерация входных данных, использование их для измерения количественных характеристик структур данных, алгоритмов, действий, сравнение экспериментальных результатов с теоретическими.

Методы решения

Разработка программы велась на базе операционной системы Windows 10 в среде разработки QtCreator. Для создания графической оболочки использовался редактор интерфейса в QtCreator. Для реализации хеш-таблицы был создан класс Hash, включающий в себя методы для создания и обработки хеш-таблицы.

1. ЗАДАНИЕ

Необходимо провести исследование алгоритма вставки в хеш-таблицу с цепочками в среднем и худшем случаях.

Исследование должно содержать:

1. Анализ задачи, цели, технологию проведения и план экспериментального исследования.
2. Генерацию представительного множества реализаций входных данных (с заданными особенностями распределения (для среднего и для худшего случаев)).
3. Выполнение исследуемых алгоритмов на сгенерированных наборах данных. При этом в ходе вычислительного процесса фиксируется как характеристики (например, время) работы программы, так и количество произведенных базовых операций алгоритма.
4. Фиксацию результатов испытаний алгоритма, накопление статистики.
5. Представление результатов испытаний, их интерпретацию и сопоставление с теоретическими оценками.

2. ОПИСАНИЕ ПРОГРАММЫ

2.1. Описание интерфейса пользователя

Интерфейс данной программы позволяет пользователю взаимодействовать с программой через специальные виджеты. В интерфейсе представлены виджеты для выбора файла для создания хеш-таблицы, генерации представительного множества входных данных для среднего и худшего случая, панель для базовых операций с хеш-таблицей. Основные виджеты для создания и сохранения хеш-таблицы и их назначение представлены в табл. 1.

Таблица 1 – Основные виджеты панели создания и сохранения хеш-таблицы

Класс объекта	Название виджета	Назначение
QTextEdit	inputht	Поле ввода файла для создания/сохранения хеш-таблицы
QPushButton	openfile	Выбор файла для создания хеш-таблицы
QPushButton	savefile	Выбор файла для сохранения хеш-таблицы
QPushButton	create	Создание хеш-таблицы по заданному файлу
QTextEdit	output	Вывод хеш-таблицы
QTextEdit	info	Вывод данных для хеш-таблицы

Основные виджеты панели создания хеш-таблицы и их назначение представлены в табл. 2.

Таблица 2 – Основные виджеты панели генерации представительного множества входных данных

Класс объекта	Название виджета	Назначение
QPushButton	averagecase	Генерация множества для среднего

Окончание таблицы 2

QPushButton	worstcase	Генерация множества для худшего случая
QLabel	label_2	Название поля ввода числа элементов
QLabel	label_3	Название поля ввода длины ключа
QLineEdit	sizeac	Поле ввода количества элементов для генерации
QLineEdit	len	Поле ввода длины ключа

Основные виджеты панели базовых операций с хеш-таблицей и их назначение представлены в табл. 3.

Таблица 3 – Основные виджеты панели базовых операций с хеш-таблицей

Класс объекта	Название виджета	Назначение
QPushButton	deleteItem	Удаление элемента из хеш-таблицы
QPushButton	pushButton	Добавление элемента в хеш-таблицу
QLineEdit	input	Поле ввода элемента
QLabel	label	Название поля ввода элемента

2.2. Описание основных функций для обработки хеш-таблицы

Для реализации хеш-таблицы был создан класс Hash, который содержит указатель на массив списков для реализации хеш-таблицы с цепочками, количество блоков и элементов хеш-таблицы. Основные методы класса Hash представлены в табл. 4.

Таблица 4 – Основные методы класса Hash

Метод	Назначение
void insertItem(string key);	Вставка элемента с ключом key в хеш-таблицу

Окончание таблицы 4

void deleteItem(string key);	Удаление элемента с ключом key из хеш-таблицы
int hashFunction(string x);	Возвращает индекс хеш-таблицы для заданного элемента x, полученный с помощью первой хеш-функции.
int hashFunction2(string x);	Возвращает индекс хеш-таблицы для заданного элемента x, полученный с помощью второй хеш-функции.
void displayHash(QTextEdit *&output);	Отображение хеш-таблицы в поле QTextEdit output
void displayInfo(QTextEdit *&info);	Отображение информации о хеш-таблице в поле QTextEdit info
void clearHashTable();	Очистка хеш-таблицы и возврат к первоначальному количеству блоков
void resize();	Увеличение размера хеш-таблицы

Для соединения ui с методами хеш-таблицы были созданы функции, представленные в processing.cpp. Основные функции приведены в табл. 5.

Таблица 5 – Основные функции processing.cpp

Функция	Назначение
void deleteItemFromHashTable(QTextEdit *&output, QLineEdit *&input, QTextEdit *&info);	Удаление элемента из хеш-таблицы
void insertItemToHashTable(QTextEdit *&output, QLineEdit *&input, QTextEdit *&info);	Добавление элемента в хеш-таблицу
void createHashTable(QTextEdit *&output, QTextEdit *&in, QTextEdit *&info);	Создание хеш-таблицы по заданному файлу
void generateAverageCase(QTextEdit *&output, QLineEdit *&sizeac, QLineEdit *&len);	Генерация представительного множества входных данных для среднего случая
void generateWorstCase(QTextEdit *&in, QLineEdit *&sizeac, QLineEdit *&len);	Генерация представительного множества входных данных для худшего случая первой хеш-функции

Окончание таблицы 5

void generateWorstCase2(QTextEdit *&in, QLineEdit *&sizeac, QLineEdit *&len);	Генерация представительного множества входных данных для худшего случая второй хеш-функции
string generateRandomString(int len);	Генерация случайной строки

Основные слоты mainwindow.cpp приведены в табл. 6.

Таблица 6 – Основные слоты mainwindow.cpp

Слот	Назначение
void on_create_clicked();	Создание хеш-таблицы
void on_openfile_clicked();	Выбор файла для чтения
void on_savefile_clicked();	Выбор файла для записи
void on_deleteItem_clicked();	Удаление элемента из хеш-таблицы
void on_averagecase_clicked();	Генерация представительного множества входных данных для среднего случая
void on_pushButton_clicked();	Добавление элемента в хеш-таблицу
void on_worstcase_clicked();	Генерация представительного множества входных данных для худшего случая

2.3. Описание алгоритма разрешения коллизий

Технология сцепления элементов состоит в том, что элементы множества с одним и тем же хеш-значением связываются в *цепочку-список*. В позиции index хранится указатель на голову списка тех элементов, у которых хеш-значение ключа равно index. Если таких элементов в таблице нет, то позиция index содержит NULL. Такой способ разрешения коллизий позволяет хранить множества с потенциально бесконечным пространством ключей, снимая ограничения на размер множества. Поэтому этот способ организации хеш-таблицы называется *открытым* или *внешним хешированием*.

Каждый элемент хранит указатель на голову списка или NULL. Используется массив списков. При хешировании включаемого ключа получаем место в массиве (фактически попадаем в нужный нам список) и включаем в данный список наши данные.

Пример коллизии с разрешением методом цепочек представлен на рис. 1.

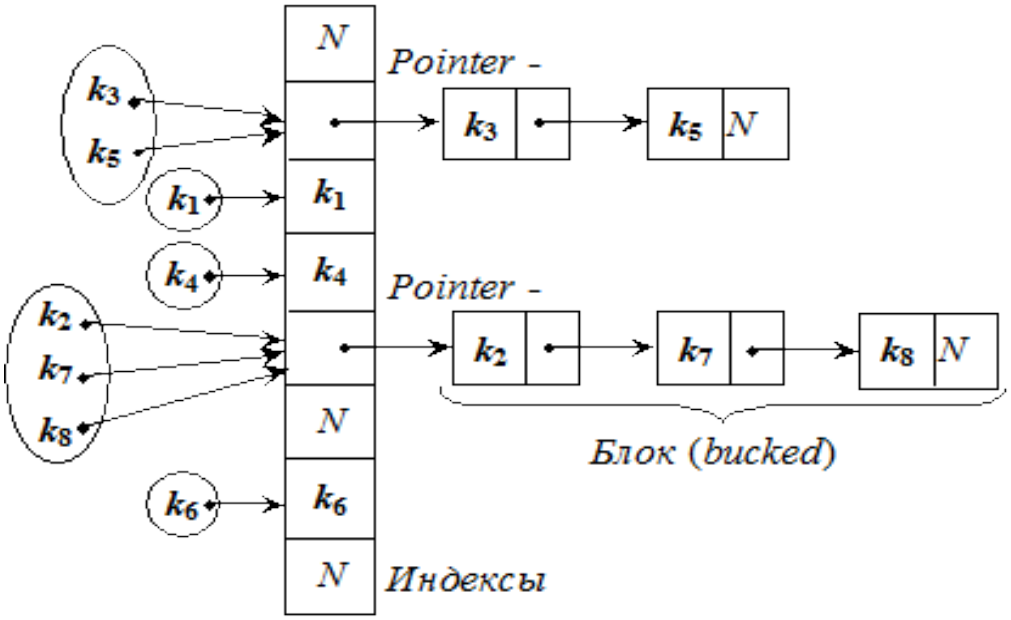

Рисунок 1 – Пример разрешения коллизий методом цепочек

2.4 Описание используемых хеш-функций

В качестве первой хеш-функции используется аддитивный метод, в котором ключом является символьная строка. В хеш-функции строка преобразуется в целое суммированием всех символов и возвращается остаток от деления на размер таблицы SIZE. Коллизии возникают в строках, состоящих из одинакового набора символов, например, *abc* и *cab*.

Данный метод можно несколько модифицировать, получая результат, суммируя только первый и последний символы строки-ключа.

Для второй функции был выбран хороший и широко используемый способ определения хеша строки s длины n, представленный на рис. 2:

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

Рисунок 2 – Полиномиальная скользящая хеш-функция

где p и m - некоторые выбранные положительные числа. Это называется полиномиальной скользящей хеш-функцией. Целесообразно сделать p простым числом, примерно равным количеству символов во входном алфавите.

Очевидно, m должно быть большим числом, поскольку вероятность столкновения двух случайных строк составляет около $\frac{1}{m}$. Хороший выбор для m - это большое простое число.

3. ТЕСТИРОВАНИЕ

3.1. Вид программы

Программа представляет собой окно с графическим интерфейсом. Вид программы представлен на рис. 3.

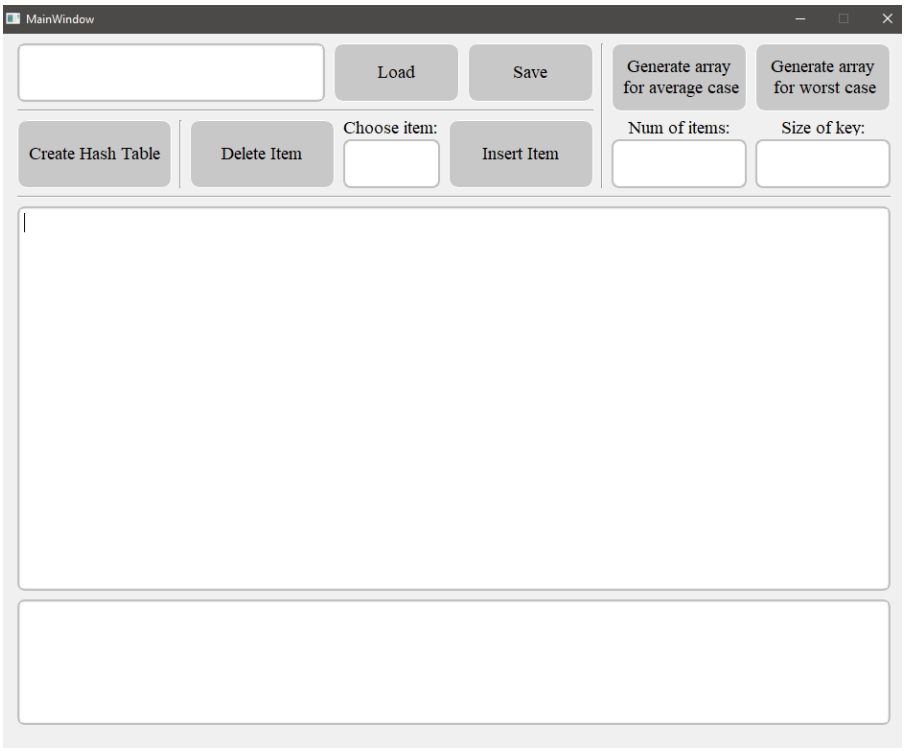

Рисунок 3 – Вид программы

Вначале нужно ввести файл для создания хеш-таблицы, количество элементов и длину ключа. После генерации множества по нажатию кнопки создается хеш-таблица. После создания хеш-таблицы в файл вывода и в поле текстового вывода помещается информация о хеш-таблице: количество коллизий, число хранимых элементов, число блоков хеш-таблицы, коэффициент заполнения хеш-таблицы, максимальная длина цепочки в хеш-таблице, а также общее чисто итераций для вставок. Вид программы после создания хеш-таблицы представлен на рис. 4.

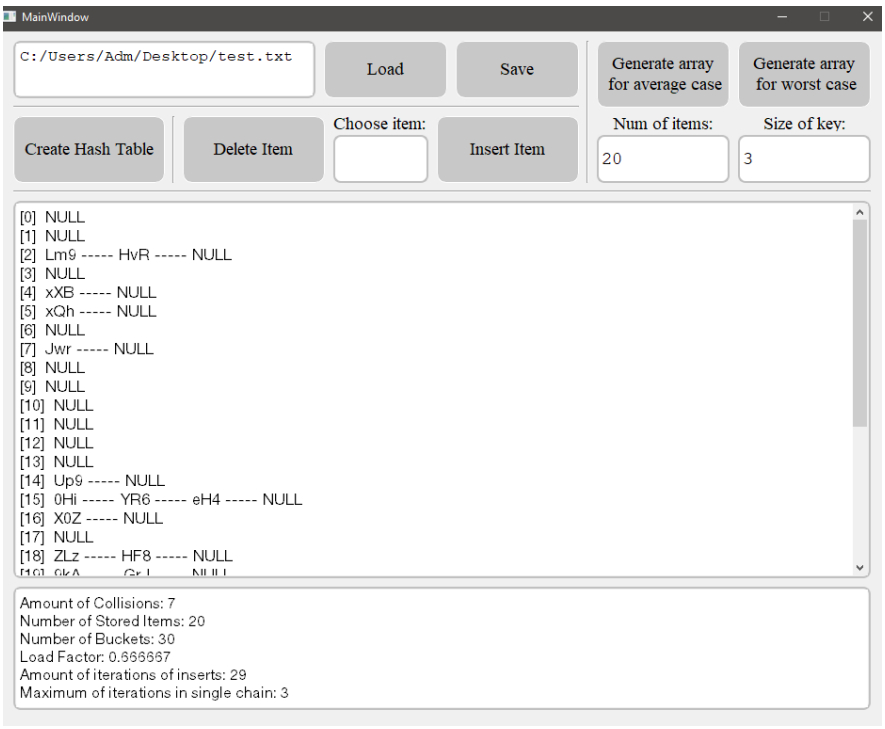

Рисунок 4 – Вид программы после создания хеш-таблицы

3.2. Тестирование генерации представительного множества данных

Генерация файла была протестирована для различной длины ключа, числа элементов, а также для худшего случая хеш-функций. Входные параметры: K - длина ключа, N – число пар. Хеш-функцию необходимо изменять в самой программе. Результаты тестирования приведены в табл. 7.

Таблица 7 – Тестирование генерации файла

Хеш-функция	Худший/средний случай	N	K	Содержание сгенерированного множества
hashFunction	Средний случай	3	5	SwxYU ic2WX TuNMh
hashFunction	Средний случай	4	3	KMB alL 65n xyo

Окончание таблицы 7

hashFunction	Худший случай	3	5	7lskR s7lRk Rks7l
hashFunction2	Средний случай	3	5	74snu XbdrN QRrQA
hashFunction2	Средний случай	5	3	uuj smW 6wz Ud6 NR4
hashFunction2	Худший случай	3	5	MD7th MD7th MD7th

Из таблицы видно, что количество пар соответствует N, а длина ключа равна K. В качестве символов используются английские строчные и заглавные буквы и цифры. Правильность подбора худшего случая проще всего оценить для хеш-функции hashFunction, которая считается как сумма ASCII кодов всех символов ключа. Из таблицы видно, что ключи в худшем случае состоят из одинаковых символов, это значит, что сумма их кодов всегда равна, и их длина одинакова. Таким образом, все ключи будут давать одинаковое значение хеш-функции.

3.3. Тестирование создания хеш-таблицы

Хеш-таблица создается по сгенерированному множеству, выбор хеш-функции осуществляется непосредственно в самой программе. Изначальный размер хеш-таблицы H равен 15. По мере заполнения хеш-таблицы ее размер может увеличиться. Результаты тестирования представлены в табл. 8.

Таблица 8 – Тестирование создания хеш-таблицы

Хеш-функция	Случай	Содержимое файла	Данные о таблице
1	Средний случай	G08GP FgK2l iGDss GPAUa frwXP	Amount of Collisions: 1 Number of Stored Items: 5 Number of Buckets: 15 Load Factor: 0.333333 Amount of iterations of inserts: 6 Maximum of iterations in single chain: 2
1	Средний случай	gf79kqdVXv eRbn9EhMFo fsJK1XcgT4 y2BP5ydtuW xUacFGDMex	Amount of Collisions: 0 Number of Stored Items: 5 Number of Buckets: 15 Load Factor: 0.333333 Amount of iterations of inserts: 5 Maximum of iterations in single chain: 1
1	Худший случай	3BfQOJ5wtk OQJt5wB3kf QJOwt35fkB BfJtkw3QO5 5Qk3OJwftB	Amount of Collisions: 4 Number of Stored Items: 5 Number of Buckets: 15 Load Factor: 0.333333 Amount of iterations of inserts: 15 Maximum of iterations in single chain: 5
2	Средний случай	pLua6 mrK5c H8v3d BIR5g Nb3aL	Amount of Collisions: 0 Number of Stored Items: 5 Number of Buckets: 15 Load Factor: 0.333333 Amount of iterations of inserts: 5 Maximum of iterations in single chain: 1

Окончание таблицы 8

2	Средний случай	ZEfDWliVX7 T2xe9EnGYc EIU6tW1sPA 320PEAHZqd BJobyDQLB4	Amount of Collisions: 1 Number of Stored Items: 5 Number of Buckets: 15 Load Factor: 0.333333 Amount of iterations of inserts: 6 Maximum of iterations in single chain: 2
2	Худший случай	tHTnZ tHTnZ tHTnZ tHTnZ tHTnZ	Amount of Collisions: 4 Number of Stored Items: 5 Number of Buckets: 15 Load Factor: 0.333333 Amount of iterations of inserts: 15 Maximum of iterations in single chain: 5

Из таблицы видно, что в худших случаях происходят множественные коллизии, элементы расположены последовательно друг за другом, число итераций в худшем случаи вставки намного превышает число итераций для средних случаев. В общем случае элементы располагаются случайно и имеют индексы от 0 до $N - 1$.

4. ИССЛЕДОВАНИЕ

4.1. План экспериментального исследования.

Для проведения исследования сложности алгоритма вставки в хеш-таблицу необходимо выяснить параметры, от которых зависит эффективность алгоритма. Так как важной частью алгоритма является хеширование, то следует исследовать алгоритм с разными способами хеширования, определяя для каждого зависимость числа итераций от выбранного параметра. После накопления данных необходимо провести сравнение результатов для различных хеш-функций и сделать выводы об эффективности алгоритмов хеширования. Кроме того, объединив всю статистику, следует сделать выводы о сложности алгоритма вставки в среднем и худшем случаях.

План проведения исследования:

1. Сбор информации о зависимости числа итераций в среднем случае
   1.1. Зависимость от длины ключа
      1.1.1. Для hashFunction
      1.1.2. Для hashFunction 2
   1.2.Зависимость от количества элементов
      1.2.1. Для hashFunction
      1.2.2. Для hashFunction2
2. Сбор информации о зависимости количества коллизий от числа элементов для вставки и длины ключа в среднем случае
      2.1.1. Для hashFunction
      2.1.2. Для hashFunction2
3. Анализ собранной информации, сравнение эффективности хеш-функций, выводы о зависимостях эффективности алгоритма от различных параметров
4. Сбор информации о зависимости числа итераций в худшем случае
   4.1.Для hashFunction
   4.2.Для hashFunction2

5. Анализ собранной информации, сравнение экспериментальных значений с теоретическими значениями, выводы о сложности алгоритма вставки в среднем и в худшем случаях

4.2. Технология проведения исследования

Исследование производится путем генерации представительного множества данных для среднего или худшего случая, далее на основе обработанных данных строятся графики зависимостей с помощью Google Sheets. Для исследования берутся два показателя – число итераций всех вставок и число итераций худшей вставки, то есть такой цепочки, которая имеет наибольшую длину. Пример построения графика зависимости числа итераций всех вставок от числа элементов представлено на рис. 5.

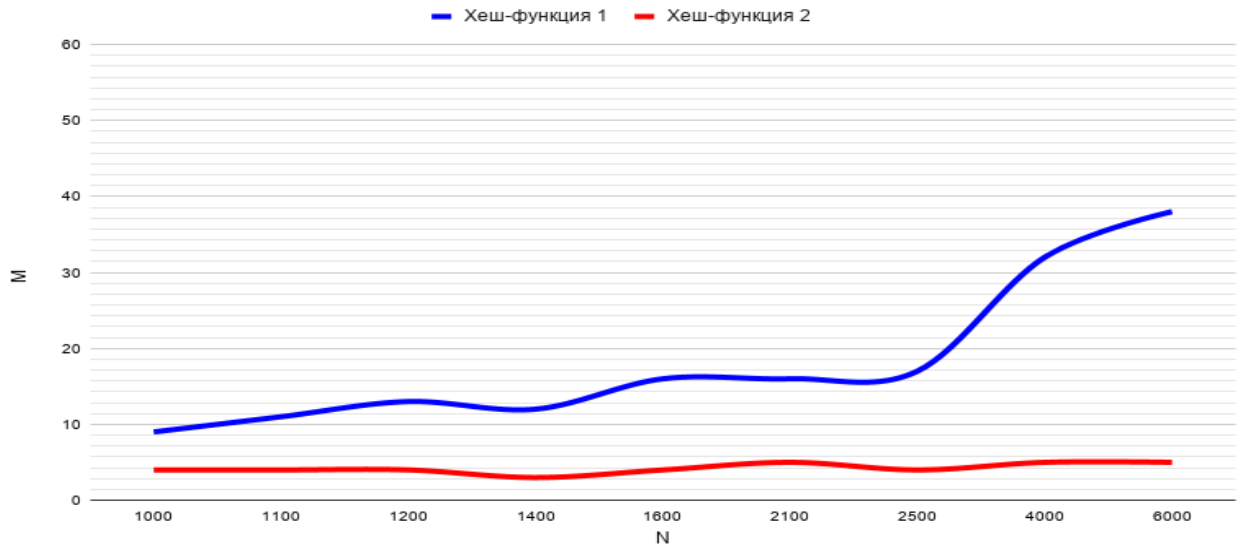

Рисунок 5 – Пример графика зависимости

Обозначения параметров исследования приведены в табл. 9.

Таблица 9 – Основные параметры

Обозначение	Параметр
I	Количество всех итераций вставки в цепочку

Окончание таблицы 9

K	Длина ключа
N	Количество элементов хеш-таблицы
M	Максимальная число итераций вставки/ максимальная длина цепочки
C	Количество коллизий

4.3. Исследование зависимостей от длины ключа

Был проведен ряд тестов, где переменным параметром бралась максимальная длина ключа K, а постоянным параметром – количество элементов N. В данном тестировании N = 1500. Значения параметров для тестирования №1 приведены в табл. 10.

Таблица 10 – Параметры тестирования №1

Ключ	Key1	Key2	Key3	Key4	Key5	Key6	Key7	Key8	Key9	Key10
Длина	2	4	6	8	10	13	16	19	23	27

В результате обработки значений был построен график зависимости $I(K)$ для каждой из хеш-функций, представленный на рис. 6.

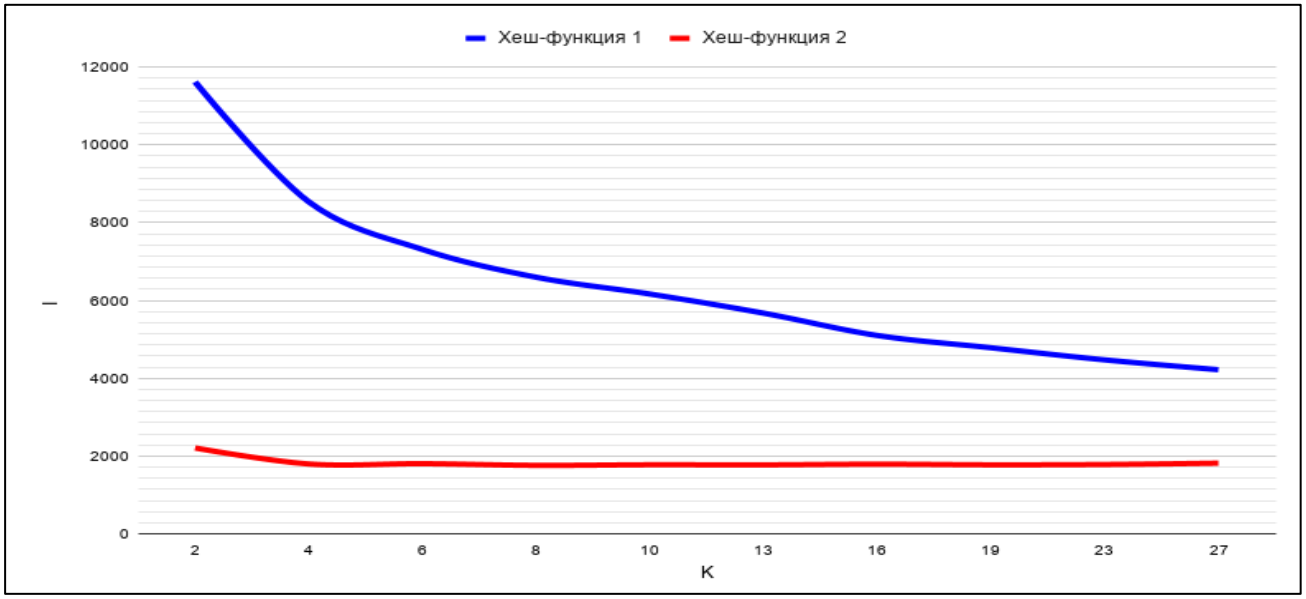

Рисунок 6 – График зависимости $I(K)$ для тестирования №1

Также был построен график зависимости $M(K)$ для каждой из хеш-функций, представленный на рис. 7.

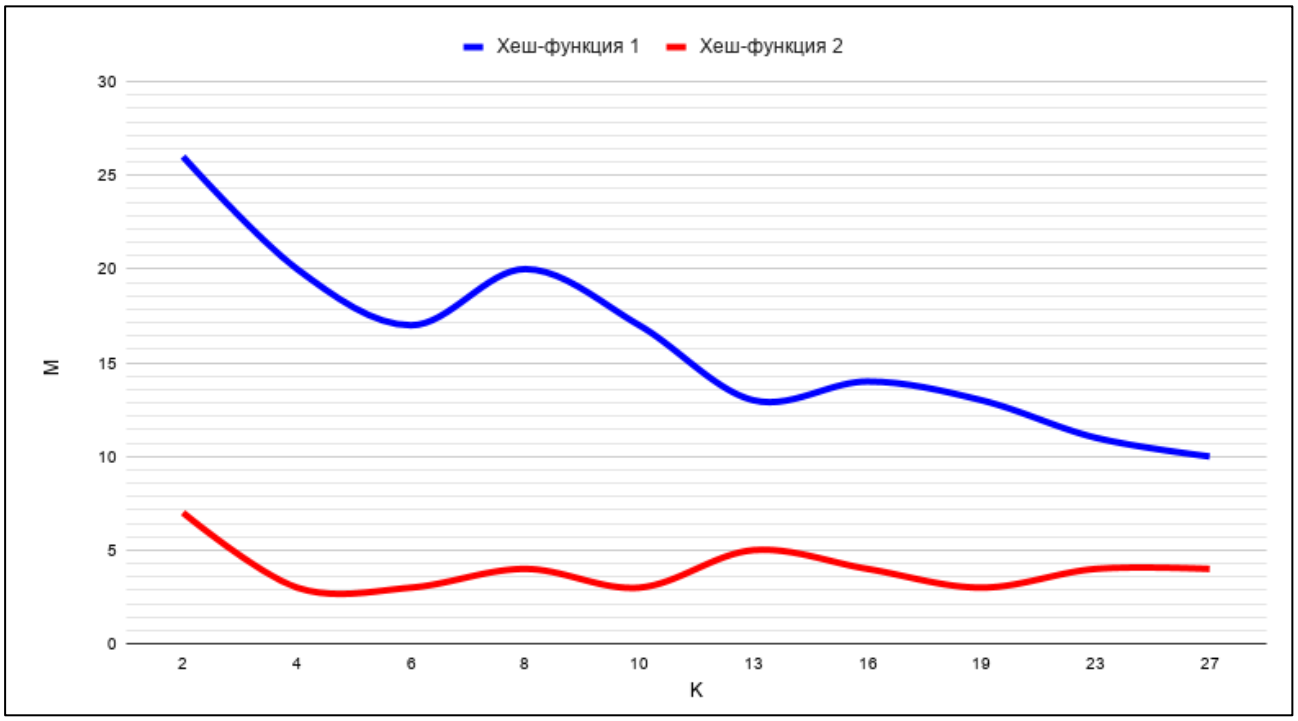

Рисунок 7 – График зависимости $M(K)$ для тестирования №1

В следующем тестировании N = 10000. Значения параметров для тестирования №2 представлено в табл. 11.

Таблица 11 – Параметры тестирования №2

Ключ	Key1	Key2	Key3	Key4	Key5	Key6	Key7	Key8	Key9	Key10	Key11
Длина	50	60	70	100	130	170	210	250	300	350	400

В результате обработки значений был построен график зависимости $I(K)$ для каждой из хеш-функций, представленный на рис. 8.

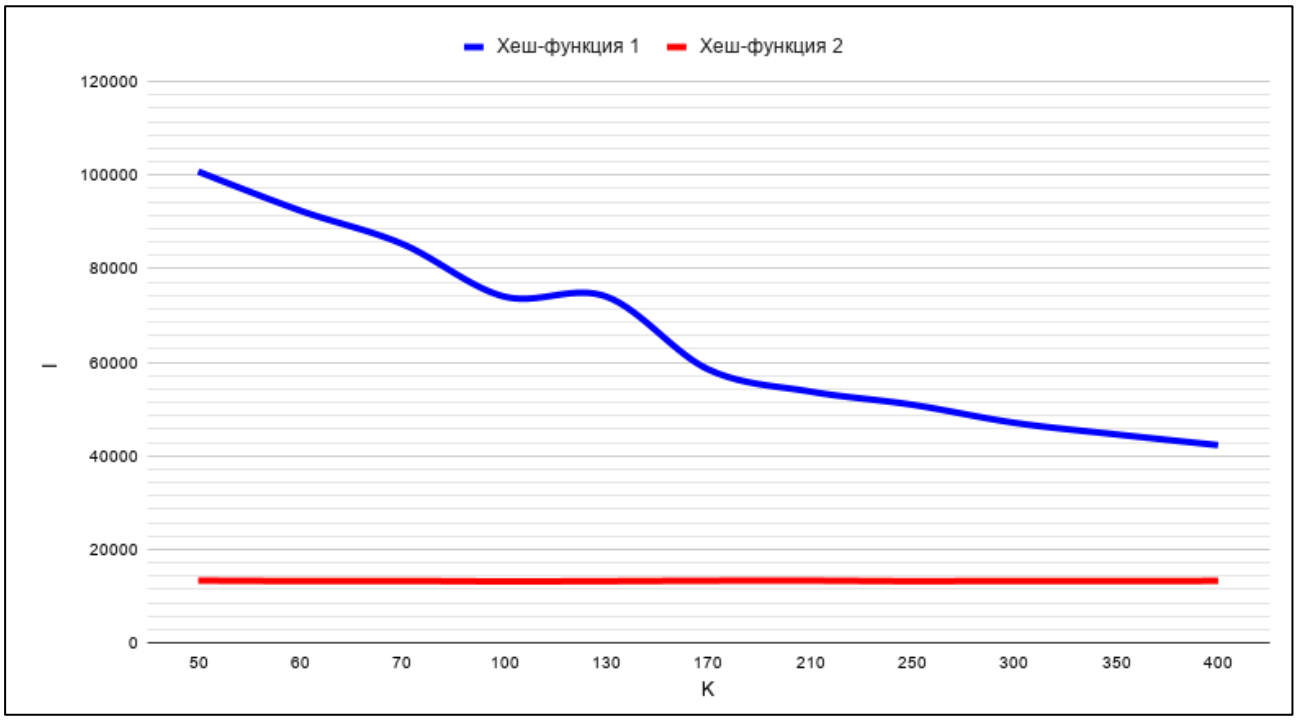

Рисунок 8 – График зависимости $I(K)$ для тестирования №2

Также был построен график зависимости $M(K)$ для каждой из хеш-функций, представленный на рис. 9.

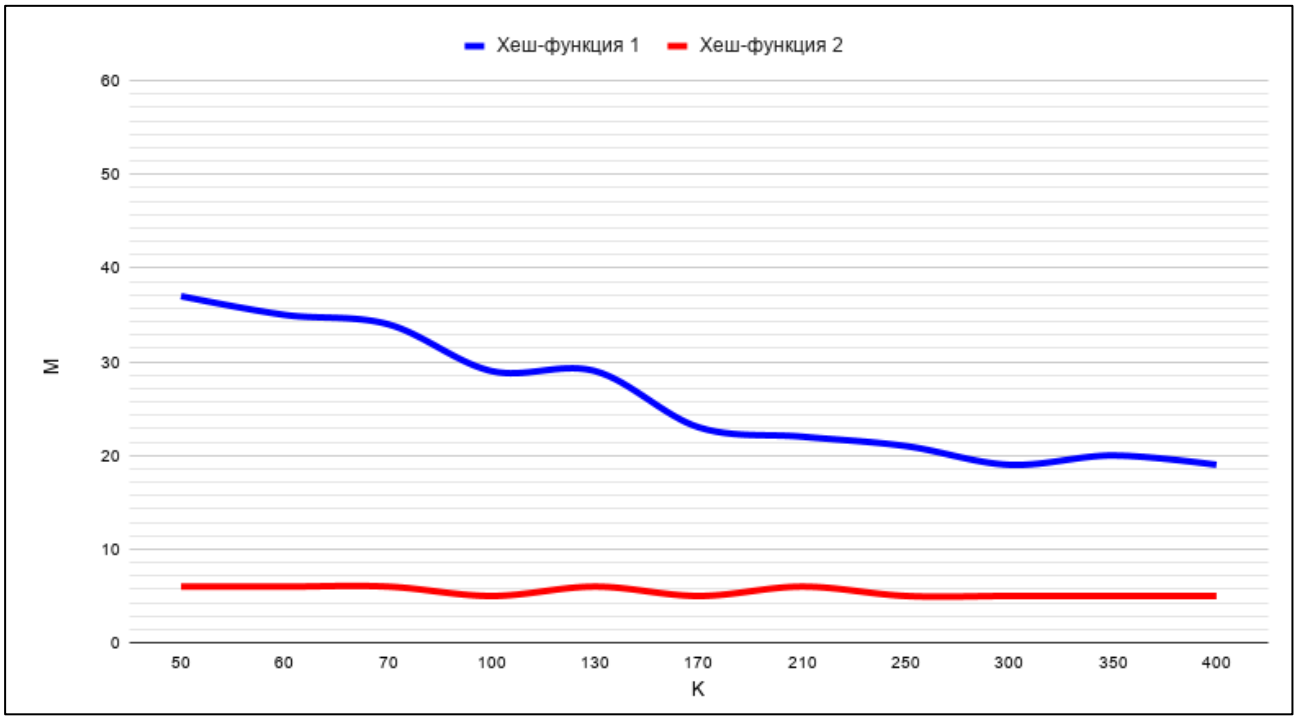

Рисунок 9 – График зависимости $M(K)$ для тестирования №2

Из графиков можно увидеть, что эффективность первой хеш-функции зависит от длины ключей, а вторая ведет себя стабильнее при изменении длины

ключей. При малых значениях K возникают худшие случаи вставки, где M в несколько раз больше, чем K. При хешировании функцией hashFunction при небольших значениях K заметен спад числа итераций и уменьшение длины максимальной цепочки (M << N), также при большем значении числа K, при его последующем увеличении заметен спад числа итераций и длины максимальной цепочки. В случае функции hashFunction2 при больших и небольших значениях K сохраняется примерно одинаковое число итераций и длина максимальной цепочки.

В целом можно сделать вывод, что для первой хеш-функции особенно важна длина ключа. При увеличении длины ключа, уменьшается число итераций и происходит уменьшении длины максимальной цепочки. Однако вторая хеш-функция ведет себя стабильнее при изменении длины ключа, так как число итераций и длина максимальной цепочки держатся на примерно одинаковом уровне.

4.4. Исследование зависимостей от числа элементов

Был проведен ряд тестов, где переменным бралось число элементов N, постоянным же параметром являлась длина ключа. В данном тестировании K = 15. Значения параметров для тестирования №3 приведены в табл. 12.

Таблица 12 – Параметры тестирования №3

Элементы	N1	N2	N3	N4	N5	N6	N7	N8
Количество	10	20	30	50	70	90	120	150

В результате обработки значений был построен график зависимости $I(N)$ для каждой из хеш-функций, представленный на рис. 10.

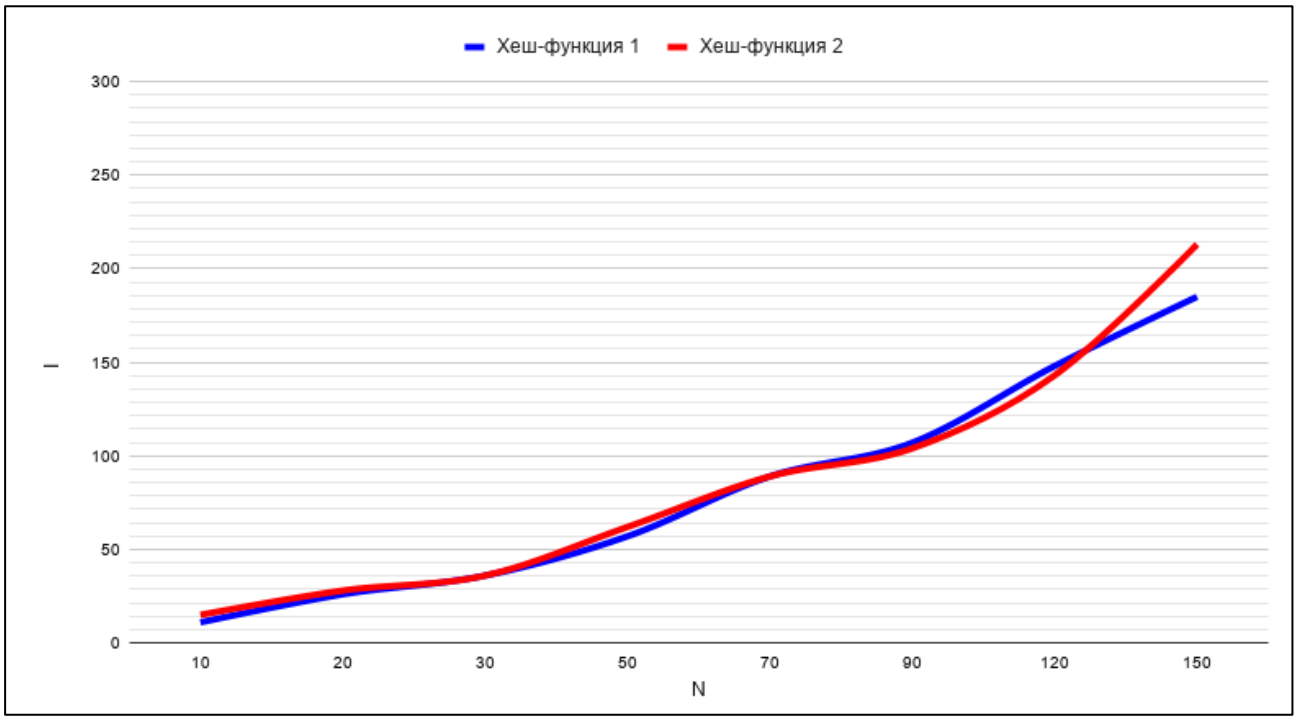

Рисунок 10 – График зависимости $I(N)$ для тестирования №3

Также был построен график зависимости $M(N)$ для каждой из хеш-функций, представленный на рис. 11.

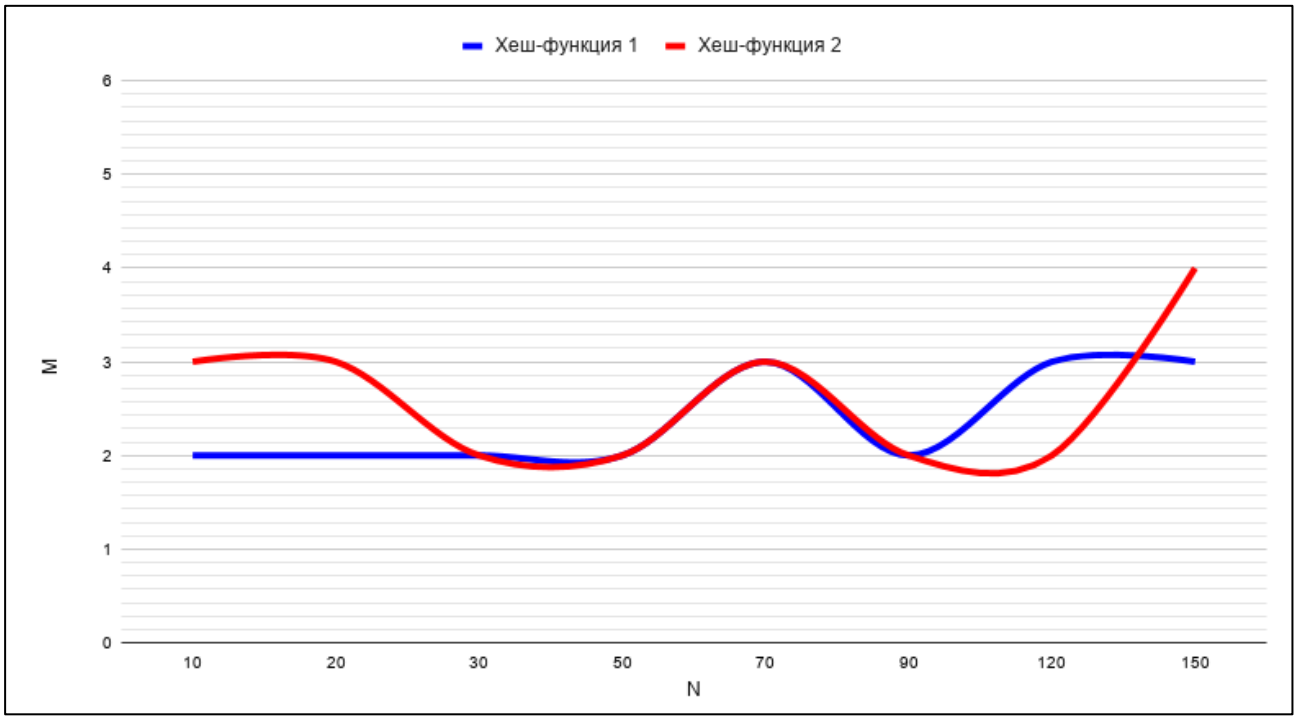

Рисунок 11 – График зависимости $M(N)$ для тестирования №3

В следующем тестировании K = 15. Значения параметров для тестирования №4 приведены в табл. 13.

Таблица 13 – Параметры тестирования №4

Элементы	N1	N2	N3	N4	N5	N6	N7	N8	N9
Количество	1000	1100	1200	1400	1600	2100	2500	4000	6000

В результате обработки значений был построен график зависимости $I(N)$ для каждой из хеш-функций, представленный на рис. 12.

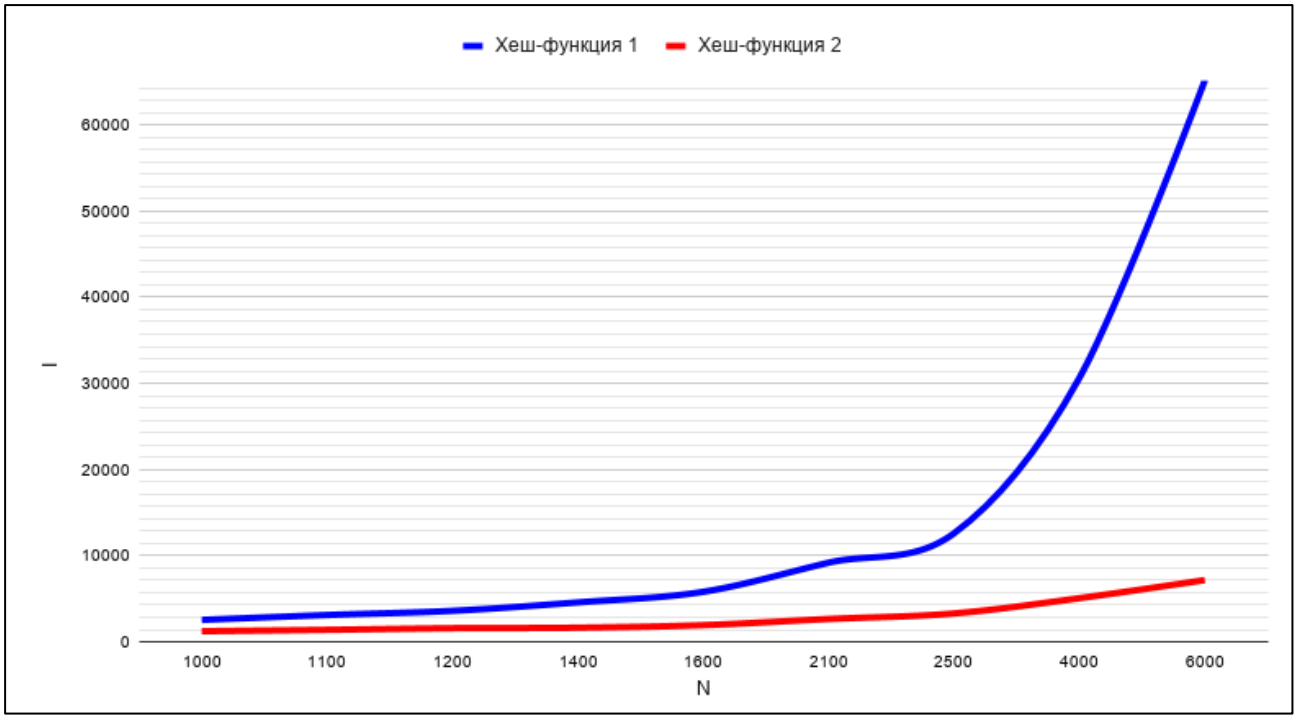

Рисунок 12 – График зависимости $I(N)$ для тестирования №4

Также был построен график зависимости $M(N)$ для каждой из хеш-функций, представленный на рис. 13.

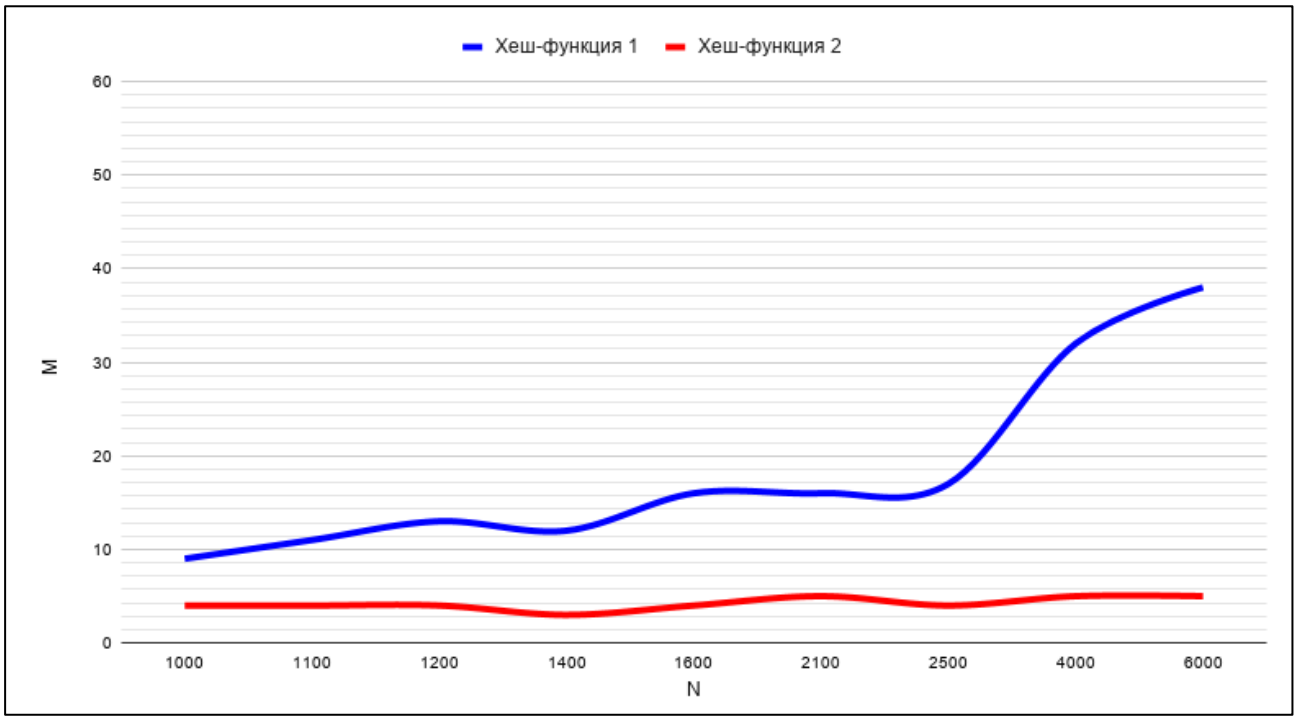

Рисунок 13 – График зависимости $M(N)$ для тестирования №4

Из тестирований №3 и №4 видно, что при N > 2500 количество итераций для хеш-функции hashFunction значительно возрастает в сравнении с hashFunction2, что говорит о множественных коллизиях. Также можно отметить, что зависимость $I(N)$ hashFunction является нелинейной, а для hashFunction2 линейной. Зависимость $M(N)$, при значении N > 1000, hashFunction также превышает аналогичную для hashFunction2, причем максимальная длина цепочки hashFunction2 держится на примерно одинаковом значении. Однако при малом количестве элементов зависимость $M(N)$ для обеих функций схожа.

В следующем тестировании №5 выбран K = 25, а диапазон значений количества элементов был увеличен. Значения параметров для тестирования №5 приведены в табл. 14.

Таблица 14 – Параметры тестирования №5

Элементы	N1	N2	N3	N4	N5	N6	N7
Количество	20	60	100	150	200	250	500

Окончание таблицы 14

Элементы	N8	N9	N10	N11	N12	N13	N14	N15	N16	N17
Количество	750	1000	3000	5000	10000	15000	20000	30000	50000	80000

В результате обработки значений был построен график зависимости $I(N)$ для каждой из хеш-функций, представленный на рис. 14.

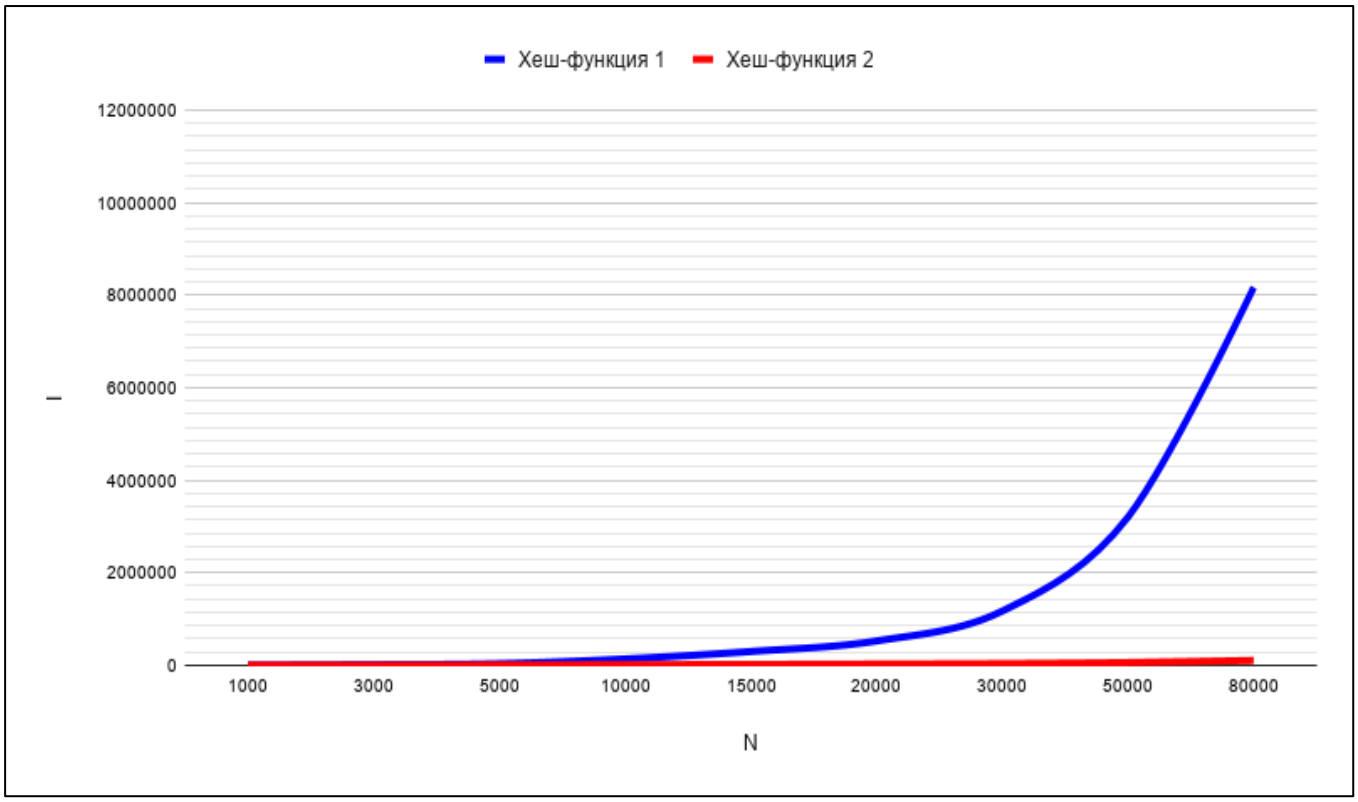

Рисунок 14 – График зависимости $I(N)$ для тестирования №5

Также был построен график зависимости $M(N)$ для каждой из хеш-функций, представленный на рис. 15.

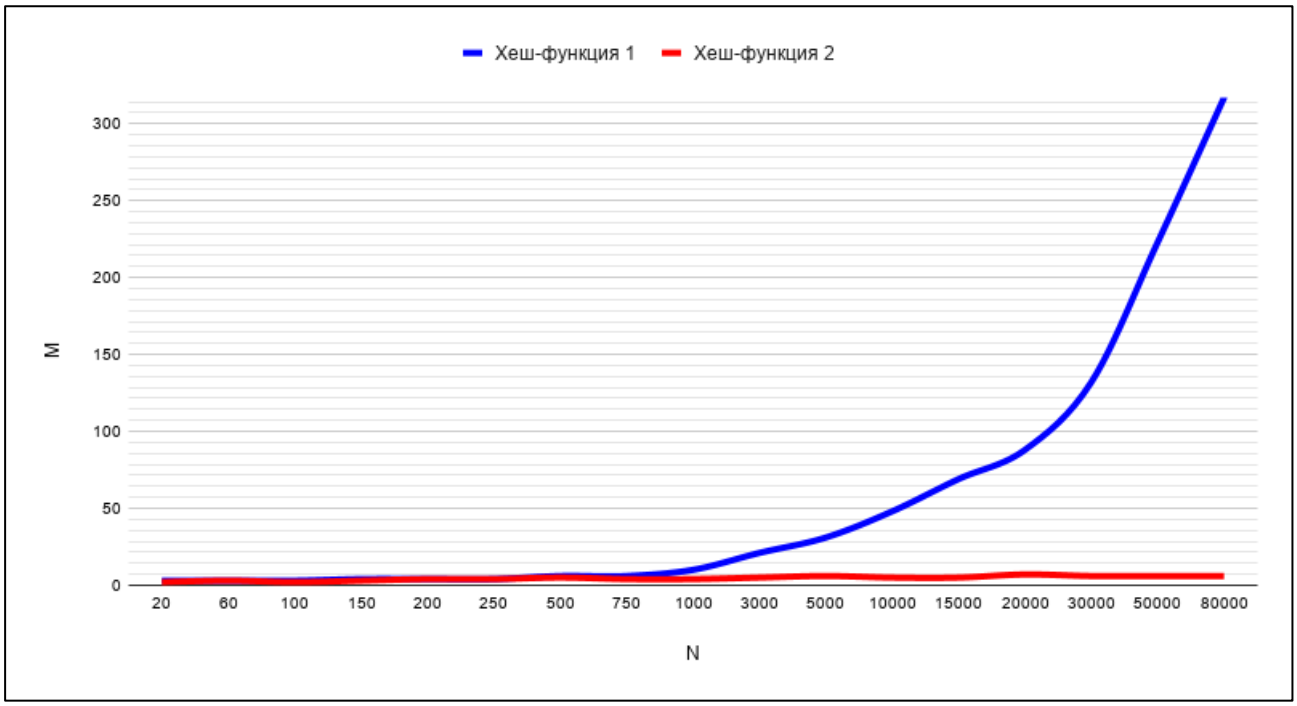

Рисунок 15 – График зависимости $M(N)$ для тестирования №5

Для удобства исследования графиков были построены дополнительные графики для половины значений, так как при N > 1000 уже видно изменения зависимости функций. В результате был построен график зависимости $I(N)$ для каждой из хеш-функций для значений N ≤ 1000, представленный на рис. 16.

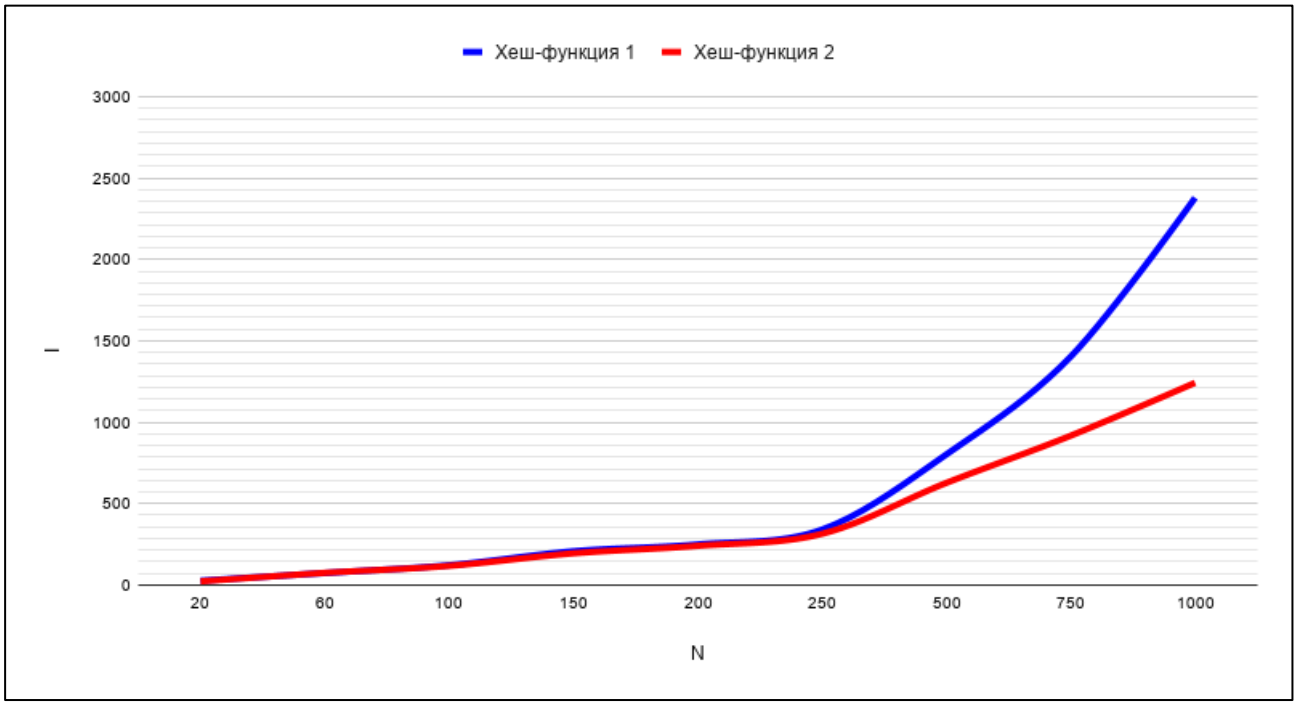

Рисунок 16 – График зависимости $I(N)$ для N ≤ 1000 тестирования №5

Также был построен график зависимости $M(N)$ для каждой из хеш-функций для значений N ≤ 1000, представленный на рис. 17.

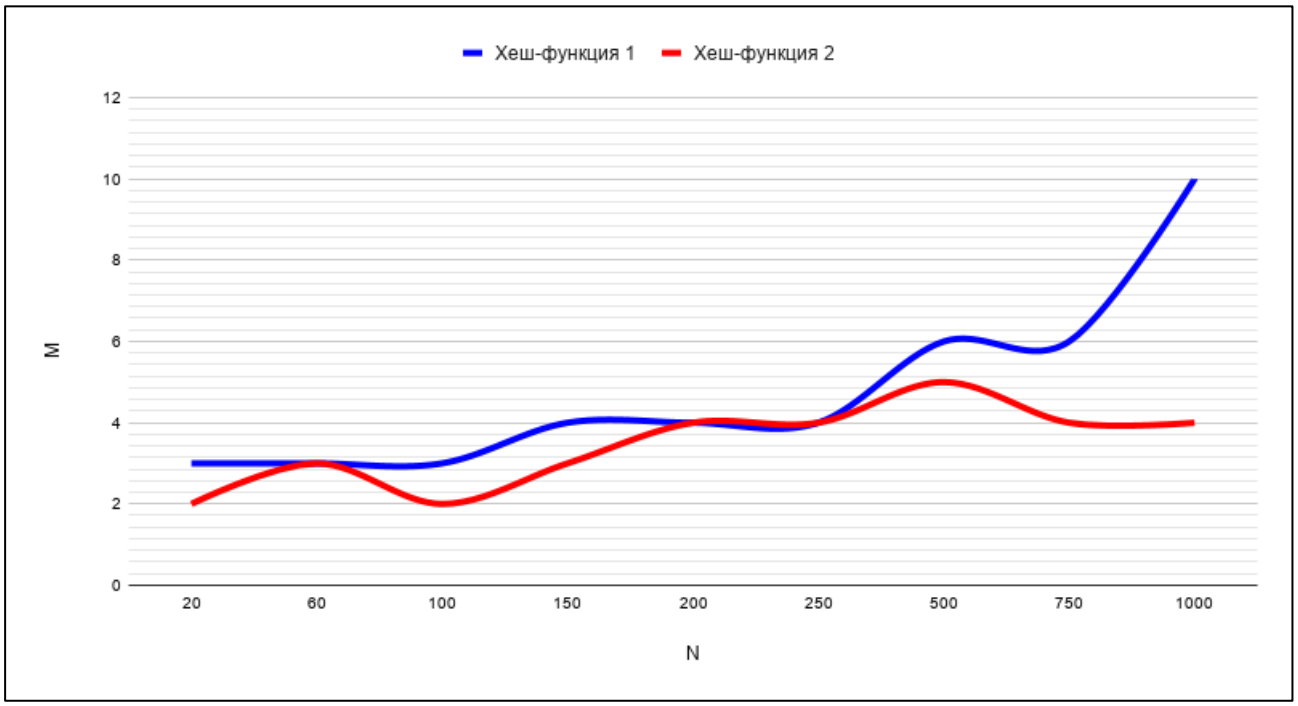

Рисунок 17 – График зависимости $M(N)$ для N ≤ 1000 тестирования №5

Как видно из графиков, зависимость $I(N)$ для хеш-функции hashFunction2 близка к линейной, а зависимость максимальной длины цепочки $M(N)$ держится на примерно одинаковом уровне. Также можно заметить, что зависимость $I(N)$ для хеш-функции hashFunction нелинейна и с N = 250 количество итераций начинает резко возрастать, а зависимость максимальной длины цепочки $M(N)$ с некоторого момента начинает значительно возрастать, а конкретно после N = 750. Стоит отметить, что в тестировании №5 количество итераций hashFunction и hashFunction2 возрастает одинаково до значения N = 250, а максимальная длина цепочки также возрастает одинаково до того же значения, после которого начинаются отличия в данных.

В следующих тестированиях №6 и №7 исследуется влияние длины ключа для одинаковых больших значений N. В тестировании №6 выбран K = 50. Значения параметров для тестирования №6 приведены в табл. 15.

Таблица 15 – Параметры тестирования №6

Элементы	N1	N2	N3	N4	N5	N6	N7	N8
Количество	5000	7000	10000	15000	20000	27000	34000	44000
Элементы	N9	N10	N11					
Количество	54000	65000	80000					

В результате обработки значений был построен график зависимости $I(N)$ для каждой из хеш-функций, представленный на рис. 18.

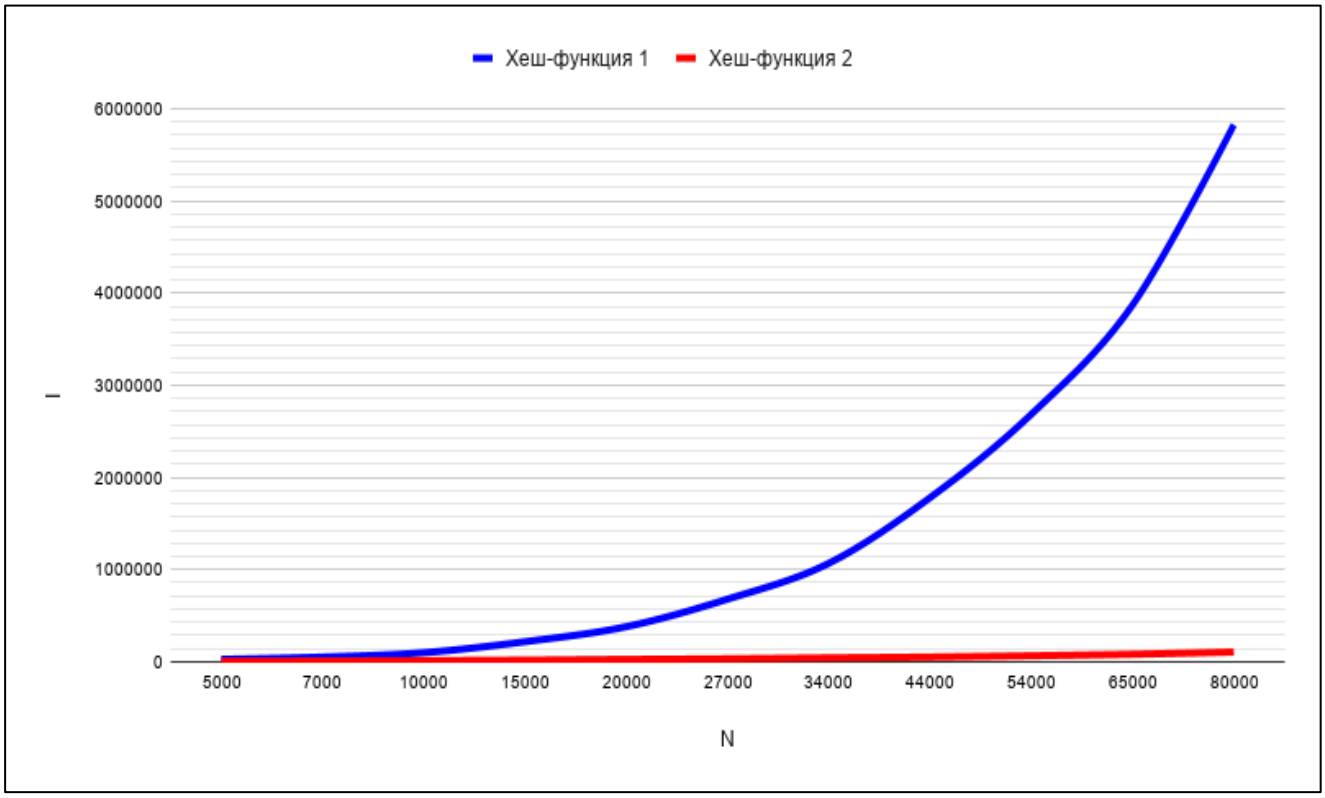

Рисунок 18 – График зависимости $I(N)$ для тестирования №6

Также был построен график зависимости $M(N)$ для каждой из хеш-функций, представленный на рис. 19.

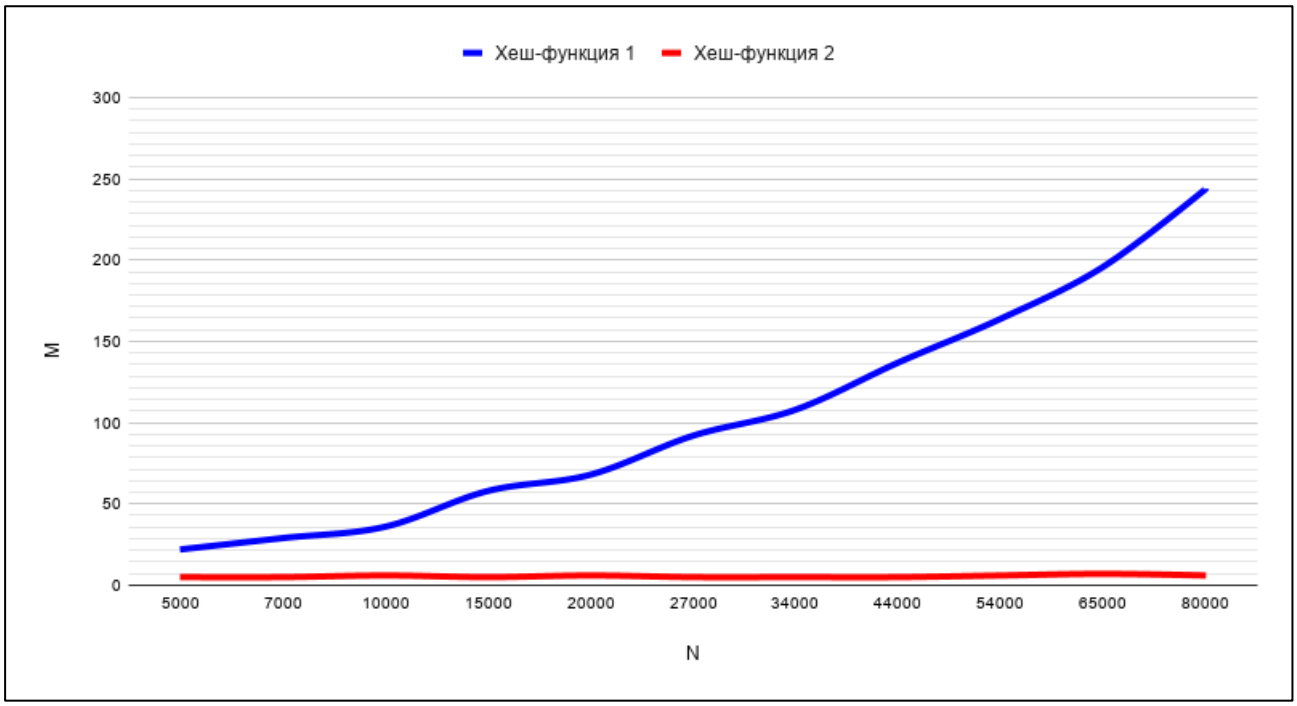

Рисунок 19 – График зависимости $M(N)$ для тестирования №6

В следующем тестировании K = 150. Значения параметров для тестирования №7 схожи с параметрами для тестирования №6 и приведены в табл. 15.

В результате обработки значений был построен график зависимости $I(N)$ для каждой из хеш-функций, представленный на рис. 20.

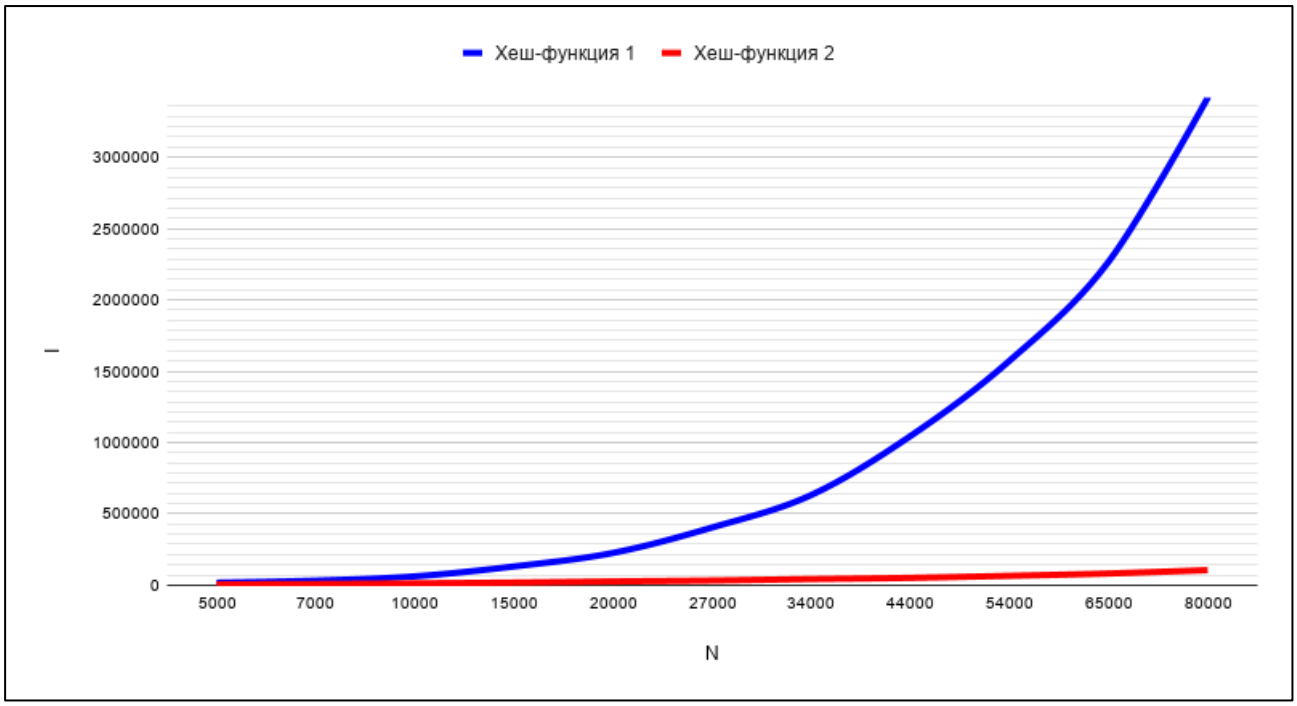

Рисунок 20 – График зависимости $I(N)$ для тестирования №7

Также был построен график зависимости $M(N)$ для каждой из хеш-функций, представленный на рис. 21.

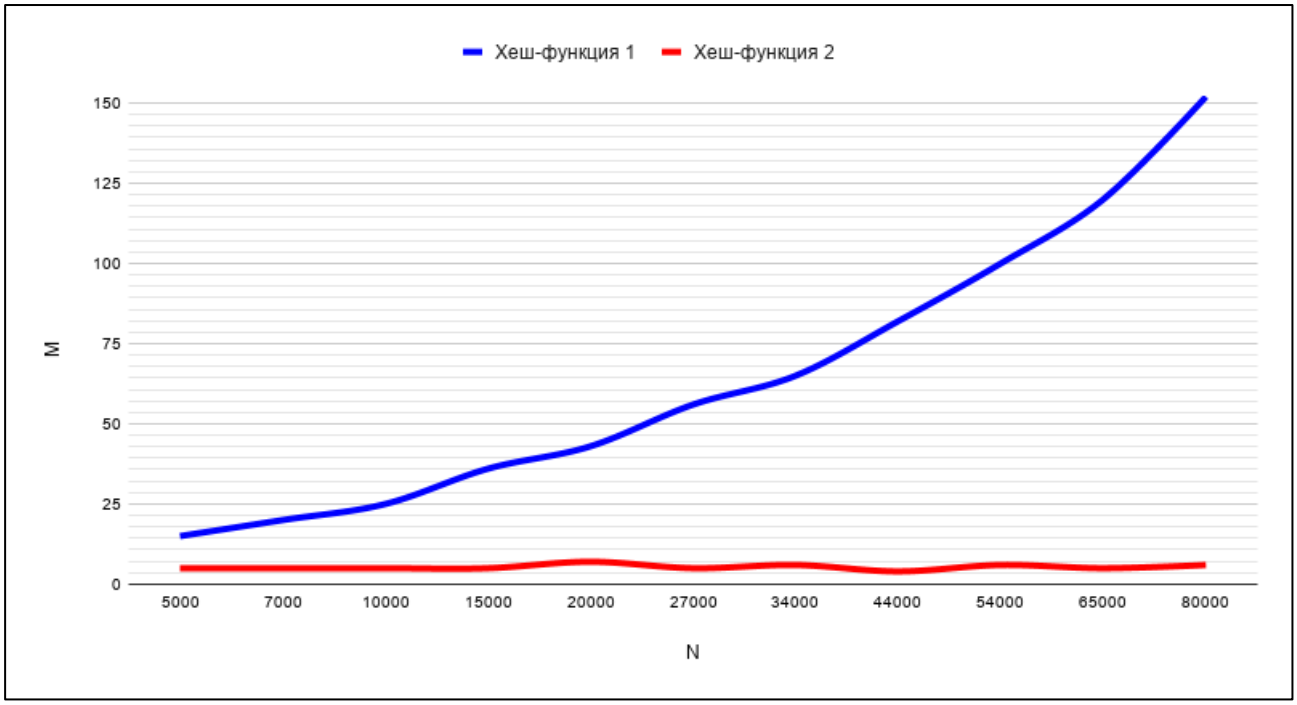

Рисунок 21 – График зависимости $M(N)$ для тестирования №7

Из тестирований №6 и №7 видно, что большую роль в зависимости $I(N)$ и $M(N)$ играет длина ключа K. При достаточно большом значении K возникает меньше коллизий и происходит меньше итераций, чем при небольшом K, что видно в тестировании №6 и №7. Однако для хеш-функции hashFunction и значения K есть определенное значение, при котором зависимость перестает быть линейной ввиду множественных коллизий. Стоит отметить, что для хеш-функции hashFunction2 длина ключа не играет роли при подсчете количества итераций вставки.

Также был построен график зависимости $I(N)$ для хеш-функции hashFunction при разных значениях длины ключа K, представленный на рис. 22.

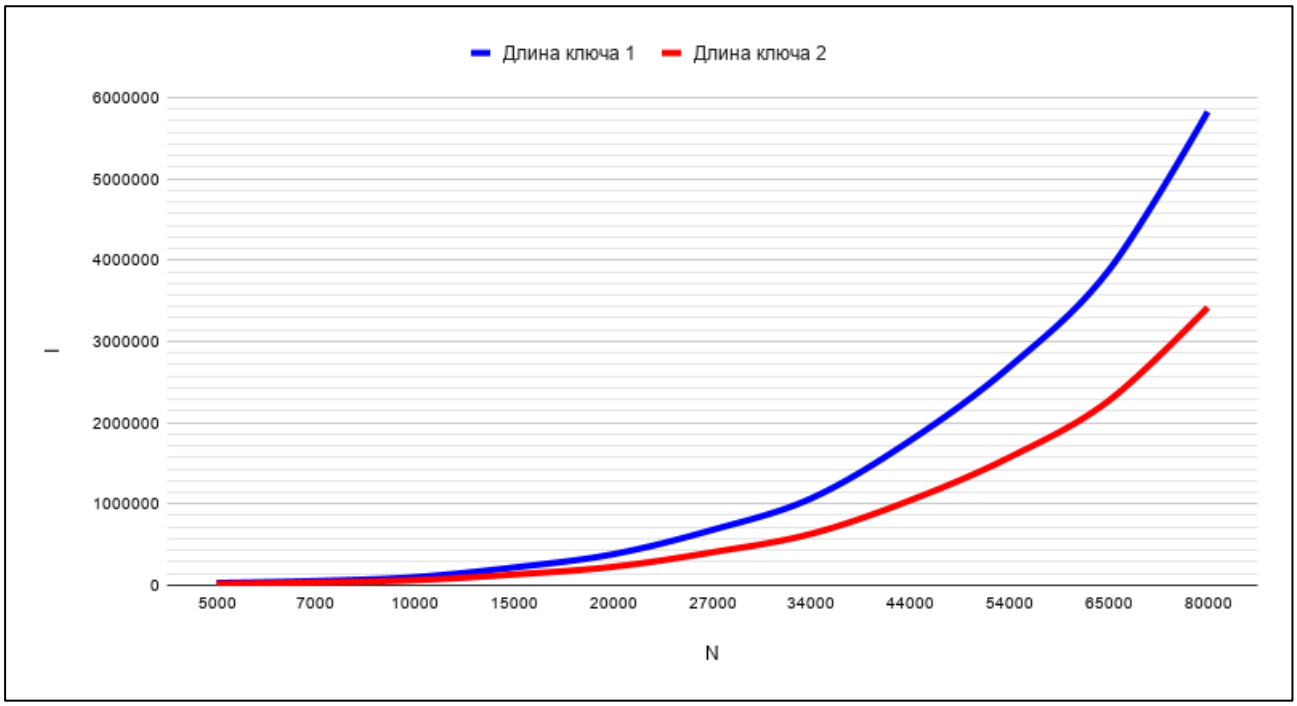

Рисунок 22 – График зависимости $I(N)$ для разных K для hashFunction

Также был построен график зависимости $I(N)$ для хеш-функции hashFunction2 при разных значениях длины ключа K, представленный на рис. 23.

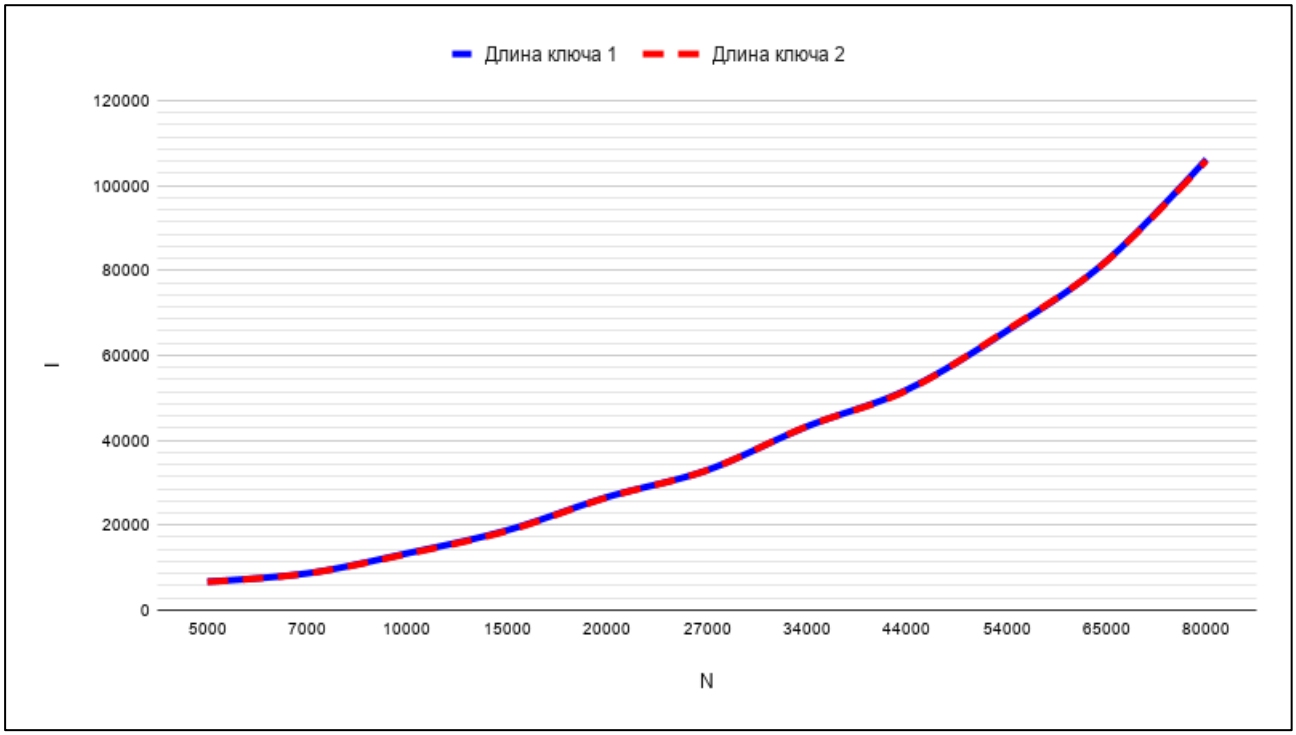

Рисунок 23 – График зависимости $I(N)$ для разных K для hashFunction2

4.5. Исследование зависимостей количества коллизий от числа элементов для вставки и длины ключа

Был проведен ряд тестов, где переменным бралась длина ключа K и исследуемая хеш-функция, постоянным же параметром являлся массив количества элементов. В данном тестировании №8 длины ключей брались относительно небольших значений: K = 7, K = 31, K = 89. Значения параметров для тестирования №8 приведены в табл. 16.

Таблица 16 – Параметры тестирования №8

Элементы	N1	N2	N3	N4	N5	N6	N7	N8
Количество	10	20	30	80	130	180	500	1000
Элементы	N9	N10	N11					
Количество	2000	5000	10000					

В результате обработки значений был построен график зависимости $C(N)$ для первой хеш-функции, представленный на рис. 24.

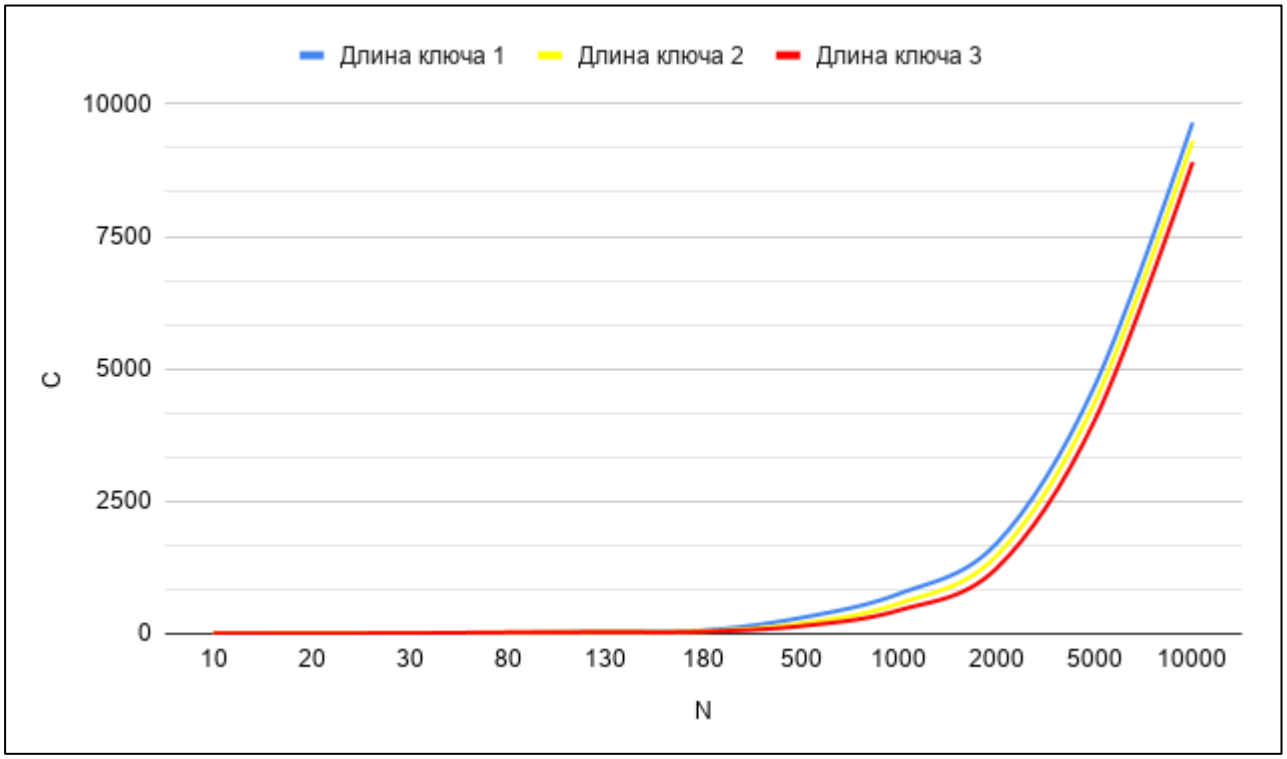

Рисунок 24 – График зависимости $C(N)$ при разных K для hashFunction

Также был построен график зависимости $M(N)$ для первой хеш-функции, представленный на рис. 25.

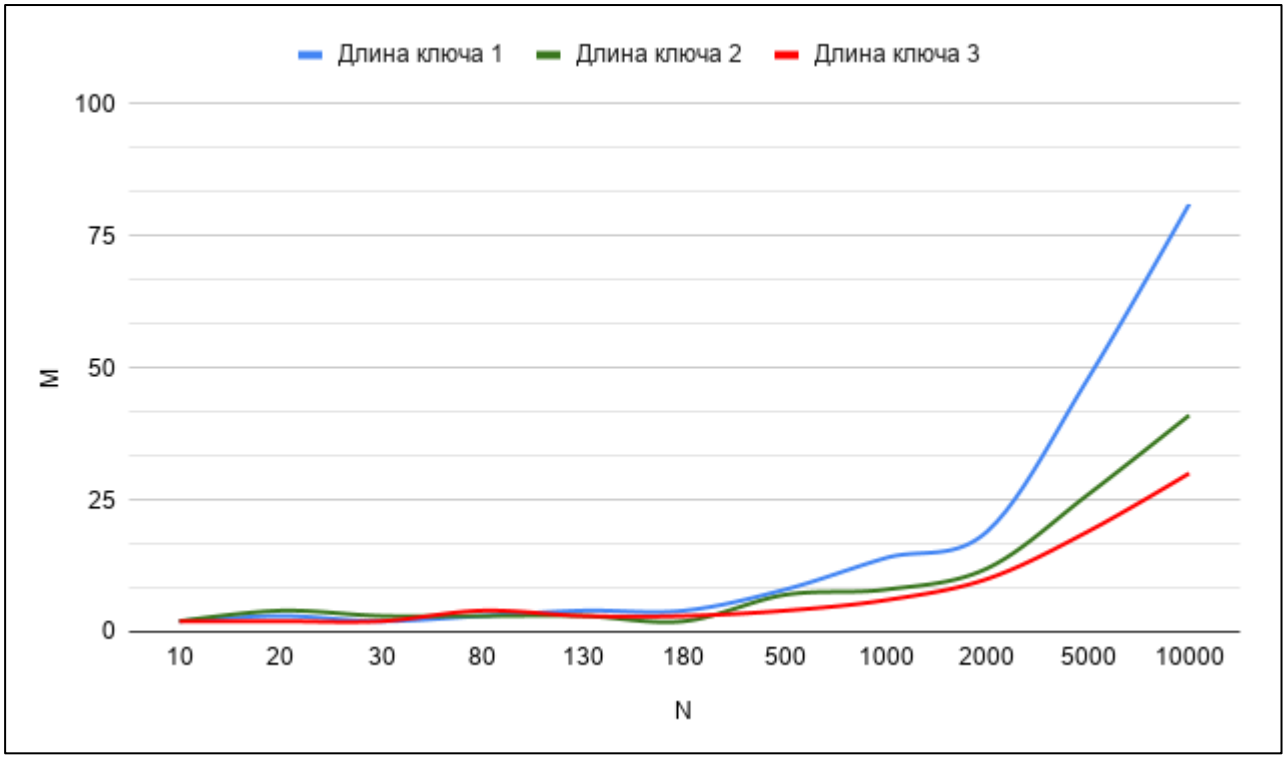

Рисунок 25 – График зависимости $M(N)$ для тестирования №8

Также был построен график зависимости $C(N)$ для второй хеш-функции, представленный на рис. 26.

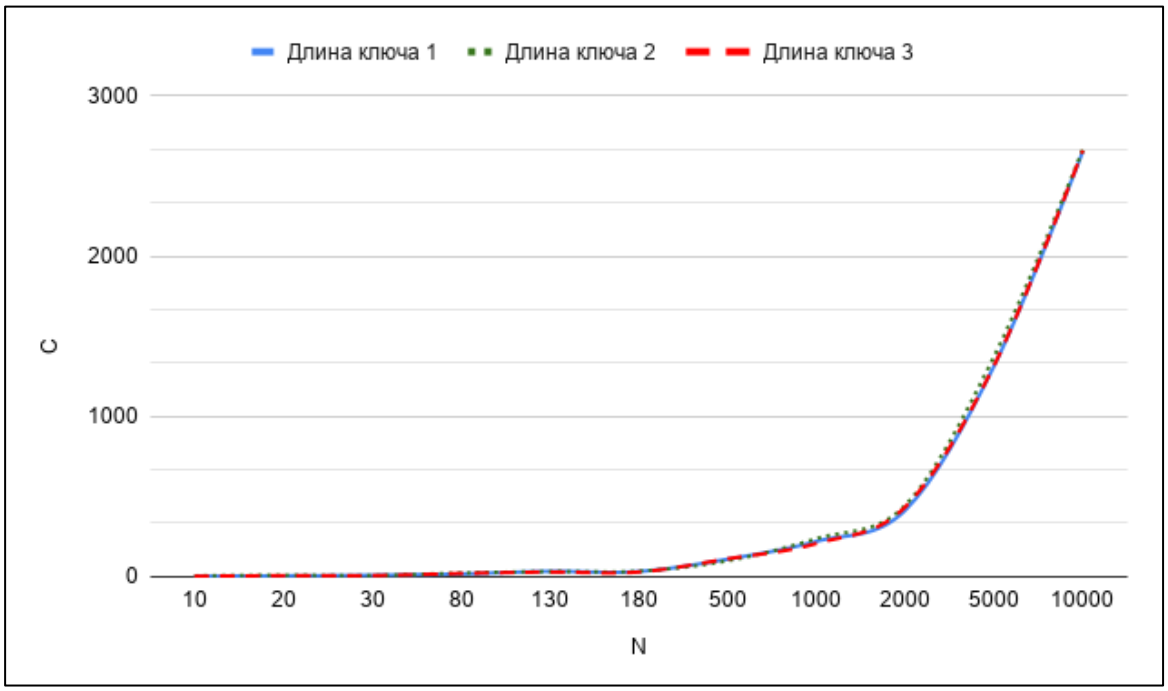

Рисунок 26 – График зависимости $C(N)$ при разных K для hashFunction2

Также был построен график зависимости $M(N)$ для второй хеш-функции, представленный на рис. 27.

Рисунок 27 – График зависимости $M(N)$ для тестирования №8

В следующем тестировании №9 длины ключей брались относительно больших значений: K = 100, K = 200, K = 300. Значения параметров для тестирования №9 приведены в табл. 17.

Таблица 17 – Параметры тестирования №9

Элементы	N1	N2	N3	N4	N5	N6	N7	N8
Количество	5000	10000	15000	25000	35000	50000	65000	85000

В результате обработки значений был построен график зависимости $C(N)$ для первой хеш-функции, представленный на рис. 28.

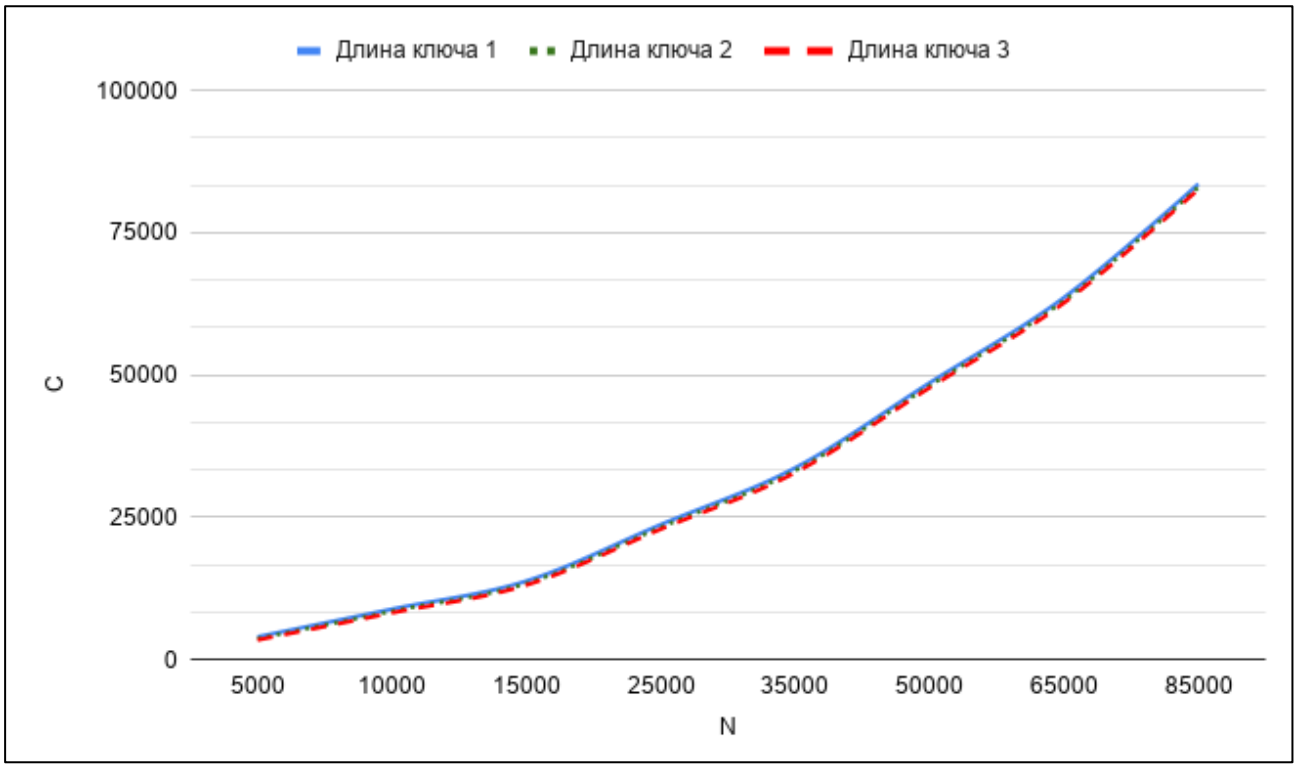

Рисунок 28 – График зависимости $C(N)$ при разных K для hashFunction

Также был построен график зависимости $M(N)$ для первой хеш-функции, представленный на рис. 29.

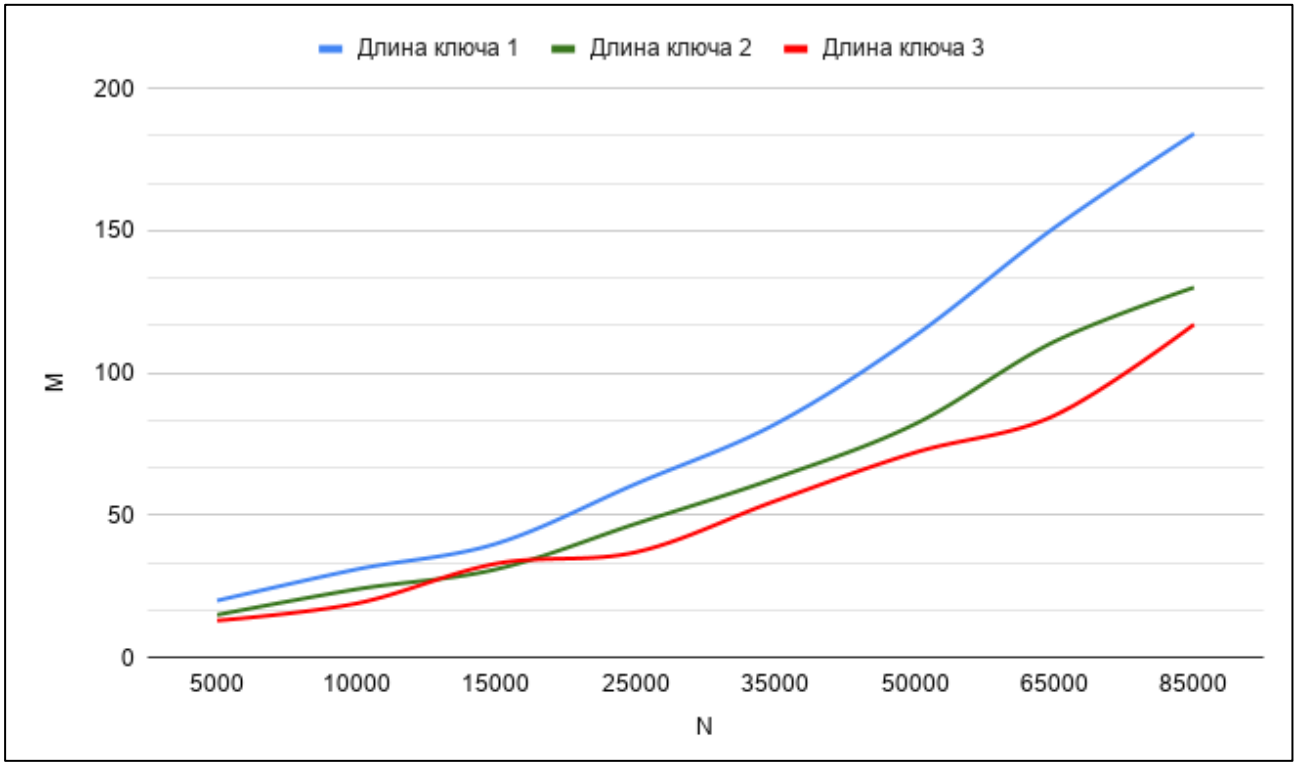

Рисунок 29 – График зависимости $M(N)$ для тестирования №9

Также был построен график зависимости $C(N)$ для второй хеш-функции, представленный на рис. 30

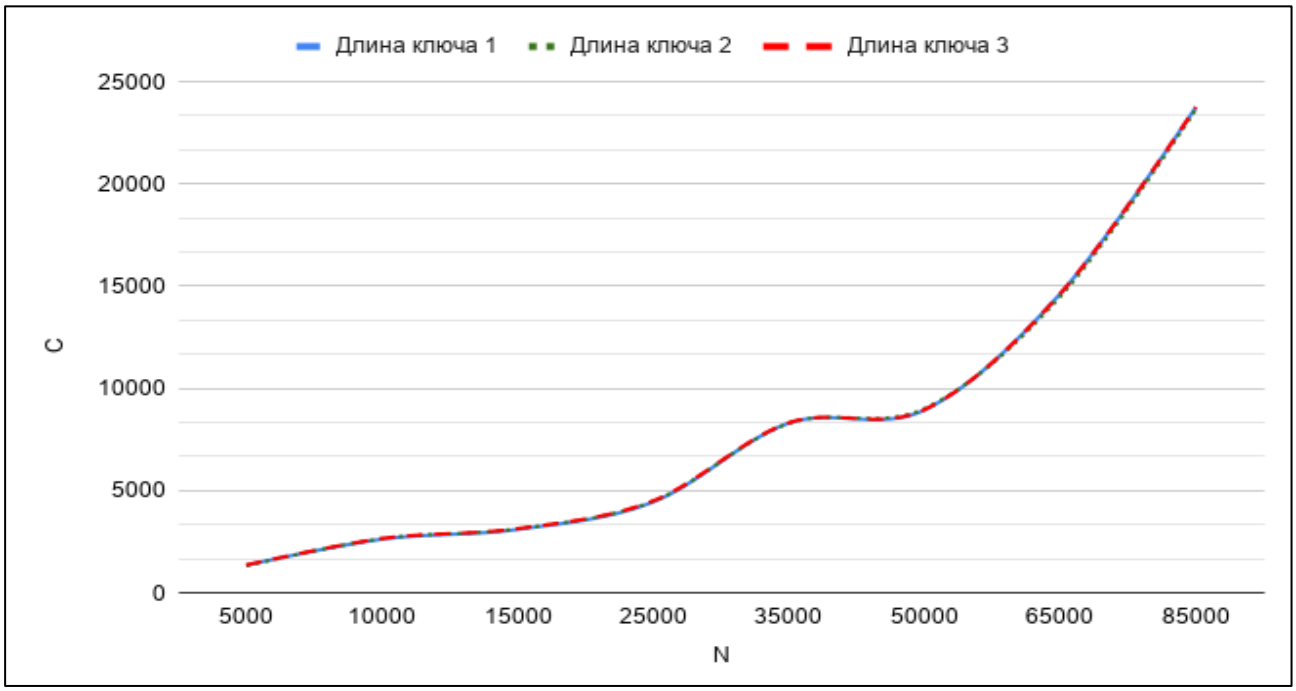

Рисунок 30 – График зависимости $C(N)$ при разных K для hashFunction2

Также был построен график зависимости $M(N)$ для второй хеш-функции, представленный на рис. 31.

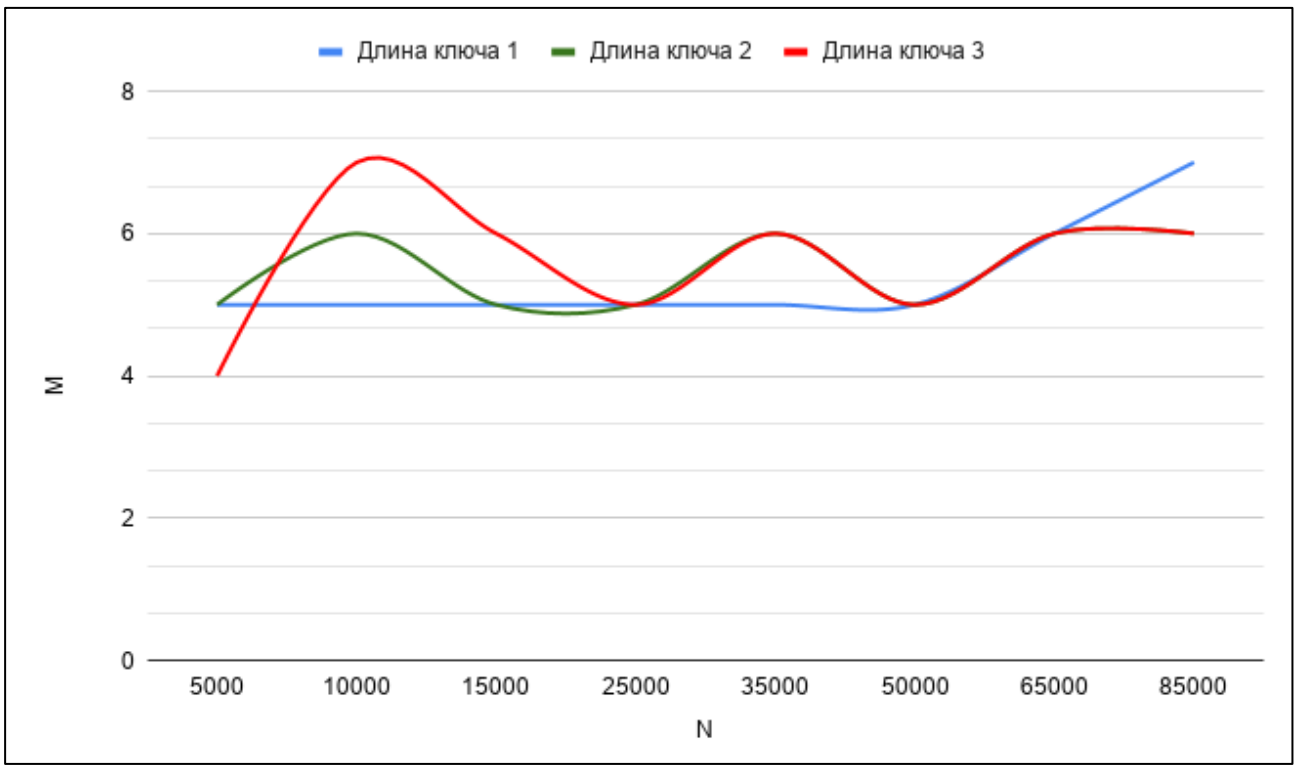

Рисунок 31 – График зависимости $M(N)$ для тестирования №9

Из тестирований №8 и №9 видно, что большую роль в зависимости $C(N)$ и $M(N)$ играет длина ключа K. Из графиков видно, что при большем значении длины ключа и для хеш-функции hashFunction, и для хеш-функции hashFunction2 идет спад количества коллизий и уменьшается максимальная длина одной цепочки, хотя в некоторых случаях зависимость одинаковая для среднего и длинного ключа.

4.6. Вывод об эффективности хеш-функций

Наиболее эффективной хеш-функция из исследуемых оказалась хеш-функция hashFunction2. В тестах функция обеспечивала линейную зависимость $I(N)$, а также $M \ll N$, что гарантирует сложность алгоритма вставки одного элемента $O(1)$, а сложность алгоритма вставки N элементов $O(N)$. Однако в худших ситуациях обе хеш-функции снижают эффективность алгоритма. В таких случаях нельзя гарантировать сложность алгоритма вставки одного элемента $O(1)$.

4.7. Исследование худшего случая вставки

Для каждой хеш-функции можно сгенерировать такие входные данные, что каждый ключ после хеширования будет давать одинаковое число, из-за чего при каждой вставке будет иметь место худший случай.

Значения параметров для тестирования №10 приведены в табл. 18.

Таблица 18 – Параметры тестирования №10

Элементы	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10
Количество	20	60	100	150	200	250	500	750	1000	3000
Элементы	N11	N12	N13	N14	N15	N16	N17			
Количество	5000	10000	15000	20000	30000	50000	80000			

В результате тестирования программой был построен график зависимости $I(N)$ для обеих хеш-функций. Графики представлены на рис. 32.

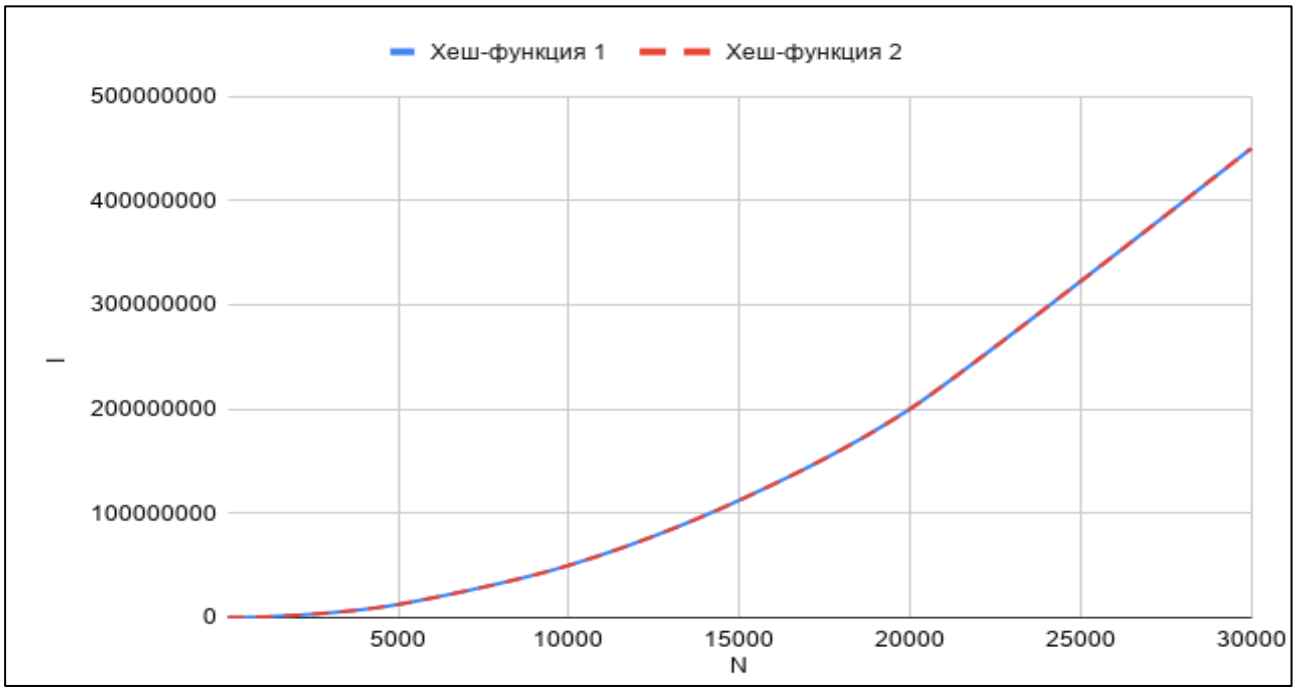

Рисунок 32 – График зависимости $I(N)$ для тестирования №10

Также был построен график зависимости $C(N)$ для каждой из хеш-функций, представленный на рис. 33.

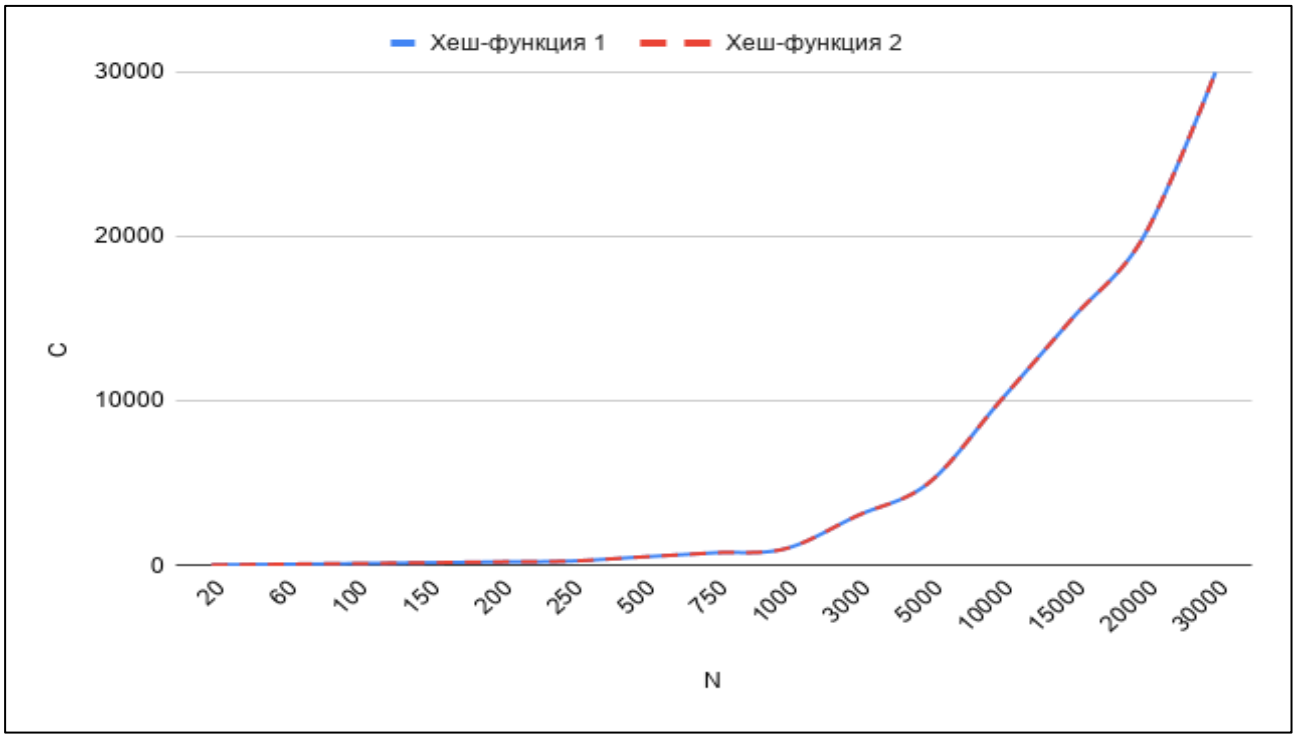

Рисунок 33 – График зависимости $C(N)$ для тестирования №10

Также был построен график зависимости $M(N)$ для каждой из хеш-функций, представленный на рис. 34.

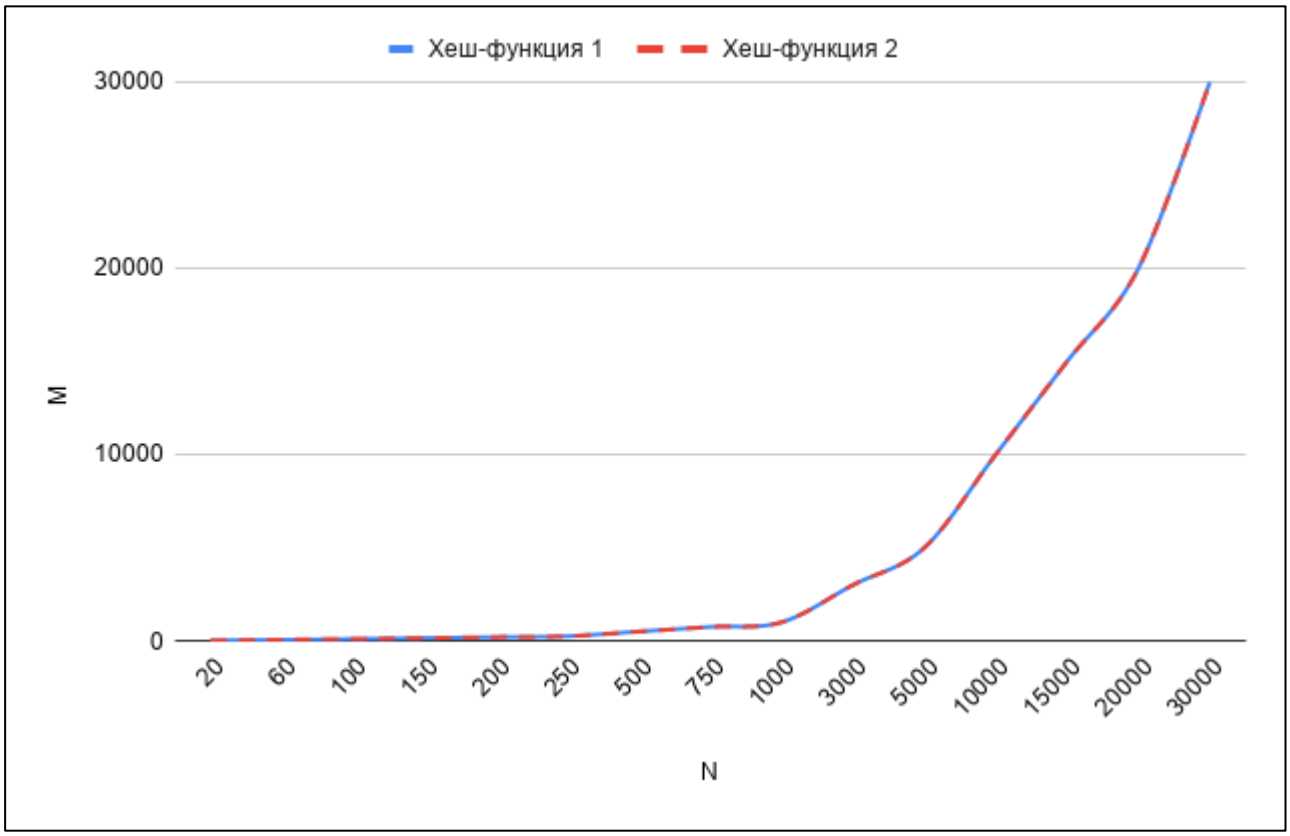

Рисунок 34 – График зависимости $M(N)$ для тестирования №10

Из рис. 34 видно, что зависимость $M(N)$ линейная для обеих хеш-функций, причем $M = N$, что и соответствует худшему случаю вставки. Сложность алгоритма вставки одного в худшем элемента $O(N)$. Зависимость $I(N)$ всех хеш-функций одинаковая и нелинейная. Чтобы точнее оценить ее, были взяты значения I при некоторых N, указанные в табл. 18. Так как при каждой вставке ожидается худший случай, то теоретически число итераций равно сумме целых чисел от 1 до N, что показано в табл.19.

Таблица 19 – Соответствие $I(N)$ для тестирования №10

N	I	$\sum_{i=1}^{N} i$
100	5050	5050

Окончание таблицы 19

500	125250	125250
1000	500500	500500
5000	12502500	12502500

Из таблицы 19 видно, что число итераций действительно равно сумме всех чисел от 1 до N, а значит для алгоритма вставки N элементов сложность $O(N^2)$.

4.8. Выводы об исследовании алгоритма

Исследование показало, что эффективность алгоритма вставки зависит от хеш-функции, а также от длины ключа. Теоретически в среднем случае сложность алгоритма вставки одного элемента $O(1)$. Среди исследуемых хеш-функций более устойчивой к параметрам, то есть чаще всего обеспечивающей условия среднего случая, оказалась хеш-функция с названием hashFunction2, а более неустойчивой, то есть реже всего обеспечивающей условия среднего случая, - hashFunction. Выбор наиболее эффективной хеш-функции позволяет обеспечить быструю работу хеш-таблицы с большими объемами различных данных, не занимая при этом большое количество памяти.

Исследование показало, что в худшем случае для любых хеш-функций сложность алгоритма вставки одного элемента $O(N)$, так как $M = N$ при любых N. В то же время, сложность алгоритма вставки N элементов $O(N^2)$, а точное число итераций высчитывается по формуле $I = \sum_{i=1}^{N} i$.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была разработана программа, которая обладает следующей функциональностью: создание файла, содержащего входные данные для хеш-таблицы, создание хеш-таблицы по заданному файлу, хеш-функции, размеру хеш-таблицы и длине ключа. С помощью программы было проведено исследование среднего и худшего случаев алгоритма вставки в хеш-таблицу с различными хеш-функциями. В ходе исследования была исследована зависимость эффективности алгоритма от различных параметров хеш-таблицы, а также от выбора хеш-функции. В результате было выявлено, что на эффективность значительно влияет как выбор алгоритма хеширования, так и количество элементов, а также длина ключа. При этом в худшем случае сложность алгоритма вставки $O(N)$, а в среднем случае $O(1)$. При выборе неэффективной хеш-функции при широком диапазоне параметров может наблюдаться худший случай, а при выборе эффективной хеш-функции уменьшается количество коллизий и реже возникают худшие случаи вставки в хеш-таблицу.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Introduction to algorithms. Second edition. M.: Thomas H. Cormen, 2002. 1296 с.
2. Hashing witch chaining // GeeksforGeeks URL: https://www.geeksforgeeks.org/c-program-hashing-chaining/ (дата обращения: 21.12.2019)
3. Хеширование // URL: https://github.com/ilyamoskalev/HashTable#mc (дата обращения: 22.12.2019)
4. IMPLEMENTING OUR OWN HASH TABLE WITH SEPARATE CHAINING IN JAVA // GeeksforGeeks URL: https://www.geeksforgeeks.org/implementing-our-own-hash-table-with-separate-chaining-in-java/ (дата обращения: 22.12.2019)
5. String Hashing // CP-Algorithms URL: https://cp-algorithms.com/string/string-hashing.html (дата обращения: 23.12.2019)
6. Введение в хеш-таблицы // Bits of Mind. URL: https://bitsofmind.wordpress.com/2008/07/28/introduction_in_hash_tables/ (дата обращения: 21.12.2019)

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД ПРОГРАММЫ. MAINWINDOW.CPP

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <hash.h>
#include <processing.h>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    ui->inputht->setPlainText("C:/Users/Adm/Desktop/test.txt");
    ui->sizeac->setText("1");
    ui->len->setText("1");
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_create_clicked()
{
    createHashTable(ui->output, ui->inputht, ui->info);
}

void MainWindow::on_deleteItem_clicked()
{
    deleteItemFromHashTable(ui->output, ui->input, ui->info);
}


void MainWindow::on_averagecase_clicked()
{
    generateAverageCase(ui->inputht, ui->sizeac, ui->len);
}

void MainWindow::on_openfile_clicked()
{
    QString    fileName    =    QFileDialog::getOpenFileName(this,    tr("load"),
QDir::homePath(), tr("*.txt"));
    ui->inputht->setPlainText(fileName);
}


void MainWindow::on_savefile_clicked()
```

```
{
    QString    filePath    =    QFileDialog::getSaveFileName(this,    tr("save"),
QDir::homePath(), tr("*.txt"));
        if (QString::compare(filePath, QString()) != 0)
        {
            ofstream ff(qPrintable(filePath));
            ff << qPrintable(ui->output->toPlainText());
            ff.close();
        }
}

void MainWindow::on_pushButton_clicked()
{
    insertItemToHashTable(ui->output, ui->input, ui->info);
}

void MainWindow::on_worstcase_clicked()
{
    generateWorstCase(ui->inputht, ui->sizeac, ui->len);
}
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД ПРОГРАММЫ. MAINWINDOW.H

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void on_create_clicked();

    void on_openfile_clicked();

    void on_savefile_clicked();

    void on_deleteItem_clicked();

    void on_averagecase_clicked();

    void on_pushButton_clicked();

    void on_worstcase_clicked();

private:
    Ui::MainWindow *ui;
};
#endif // MAINWINDOW_H
```

ПРИЛОЖЕНИЕ В

ИСХОДНЫЙ КОД ПРОГРАММЫ. MAIN.CPP

```
#include "mainwindow.h"

#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

ПРИЛОЖЕНИЕ Г

ИСХОДНЫЙ КОД ПРОГРАММЫ. HASH.CPP

```
#include "hash.h"
#include <processing.h>
long int count_iterations = 0;
long int max_it = 0;

Hash::Hash(int n)
{
    this->BUCKET = n; // constructor
    this->ITEMS = 0;
    table = new list<string>[BUCKET];
}

void Hash::insertItem(string value) // inserts a key into hash table
{
    int index = hashFunction(value);
    count_iterations++;
    for (auto x : table[index])
        count_iterations++;
    table[index].push_back(value);

    if(table[index].size() > max_it)
        max_it = table[index].size();
    ITEMS++;
    if((float)ITEMS/BUCKET > 0.7)
        resize();
}

void Hash::deleteItem(string key) // deletes a key from hash table
{
    // get the hash index of key
    int index = hashFunction(key);

    // find the key in (index)th list
    list <string> :: iterator i;
    for (i = table[index].begin(); i != table[index].end(); i++)
        if (*i == key)
            break;

    // if key is found in hash table, remove it
    if (i != table[index].end())
    {
        table[index].erase(i);
        ITEMS--;
    }
}
```

```
void Hash::resize()
{
    list <string> *temp = new list <string>[BUCKET];

    for (int i = 0; i < BUCKET; i++)
        for (auto x : table[i])
        {
            int index = hashFunction(x);
            temp[index].push_back(x);
        }
    max_it = 0;
    count_iterations = 0;
    ITEMS = 0;
    for(int i = 0 ; i < BUCKET; i++)
        table[i].clear();
    BUCKET *= 2;
    table = new list <string>[BUCKET];

    for (int i = 0; i < BUCKET/2; i++)
        for (auto x : temp[i])
        {
            int index = hashFunction(x);
            table[index].push_back(x);
            count_iterations += table[index].size();
            ITEMS++;
        }
}

void Hash::clearHashTable()
{
    max_it = 0;
    count_iterations = 0;
    ITEMS = 0;
    for(int i = 0 ; i < BUCKET; i++)
        table[i].clear();
    BUCKET = SIZE;
}
int Hash::hashFunction(string x) // adaptive method
{
    long int s = 0;
    for(int i = 0 ; i < x.length(); i++)
        s += x[i];
    return s % BUCKET;
}
int Hash::hashFunction2(string x) // multiplicative method
{
    const int p = 31;
```

```
    const int m = pow(10, 9)+9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (int i = 0 ; i < x.length(); i++)
    {
        hash_value = (hash_value + (x[i] - '0' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value % BUCKET;
}
void Hash::displayInfo(QTextEdit *&info)
{
    ofstream f("C:/Users/Adm/Desktop/testdata26.txt", ios::app);
    string out = "";
    int counter_coll = 0;
    for(int i = 0; i < BUCKET; i++)
        if(table[i].size() > 1)
            counter_coll += (table[i].size() - 1);

    info->setPlainText("Amount of Collisions: " + QString::number(counter_coll)
+
                       "\nNumber of Stored Items: " + QString::number(ITEMS) +
                       "\nNumber of Buckets: " + QString::number(BUCKET) +
                       "\nLoad Factor: " + QString::number((float)ITEMS/BUCKET)
+
                       "\nAmount    of    iterations    of    inserts:    "    +
QString::number(count_iterations) +
                       "\nMaximum   of   iterations   in   single   chain:   "   +
QString::number(max_it));

    /*out = to_string(counter_coll) + " " + to_string(max_it);
    f << out << endl;*/
    f.close();
}

void Hash::displayHash(QTextEdit *&output) //print hash table
{
    QString outputstr;
    for (int i = 0; i < BUCKET; i++)
    {
        outputstr += "[" + QString::number(i) + "]  ";
        for (auto x : table[i])
            outputstr += QString::fromStdString(x) + " ----- ";
        outputstr += "NULL\n";
    }
    outputstr += "\n";
    output->setPlainText(outputstr);
}
```

ПРИЛОЖЕНИЕ Д

ИСХОДНЫЙ КОД ПРОГРАММЫ. HASH.H

```
#ifndef HASH_H
#define HASH_H
#include <iostream>
#include <fstream>
#include <string>
#include <cmath>
#include <list>
#include <cstdlib>
#include <ctime>
#include <QMainWindow>
#include <QGraphicsItem>
#include <QGraphicsView>
#include <QGraphicsEffect>
#include <QFileDialog>
#include <QStandardPaths>
#include <QtGui>
#include <QLabel>
#include <QDateTime>
#include <QTime>
#include <QColorDialog>
#include <QInputDialog>
#include <QMainWindow>
#include <QPushButton>
#include <QMessageBox>
#include <QStringList>
#include <QTextEdit>
#include <stack>
#define SIZE 15 // 601, 211, 89, 34996 175939

using namespace std;

class Hash
{
    int BUCKET; // No. of buckets

    int ITEMS; // No. of stored items

    list <string> *table; // Pointer to an array containing buckets

    public:
        Hash(int n);

        void insertItem(string value);

        void deleteItem(string key);
```

```
        int hashFunction(string x);

        int hashFunction2(string x);

        int hashFunction3(string x);

        void displayHash(QTextEdit *&output);

        void displayInfo(QTextEdit *&info);

        void clearHashTable();

        void resize();
};

#endif // HASH_H
```

ПРИЛОЖЕНИЕ Е

ИСХОДНЫЙ КОД ПРОГРАММЫ. PROCESSING.CPP

```
#include <processing.h>

Hash h(SIZE);

void createHashTable(QTextEdit *&output, QTextEdit *&in, QTextEdit *&info)
{
    h.clearHashTable();
    ifstream f(qPrintable(in->toPlainText()), ios::in);
    string out;
    getline(f, out);
    f.close();

    QStringList array = QString::fromStdString(out).split(" ");
    string* a = new string[array.length()];
    for (int i = 0; i < array.length(); ++i)
        a[i] = array[i].toStdString();

    for (int i = 0; i < array.length(); i++)
        h.insertItem(a[i]);
    h.displayHash(output);
    h.displayInfo(info);
}

void insertItemToHashTable(QTextEdit *&output, QLineEdit *&input, QTextEdit
*&info)
{
    string x = input->text().toStdString();
    h.insertItem(x);
    h.displayHash(output);
    h.displayInfo(info);
}

void deleteItemFromHashTable(QTextEdit *&output, QLineEdit *&input, QTextEdit
*&info)
{
    string x = input->text().toStdString();
    h.deleteItem(x);
    h.displayHash(output);
    h.displayInfo(info);
}

void generateWorstCase(QTextEdit *&in, QLineEdit *&sizeac, QLineEdit *&len)
{
    int size = sizeac->text().toInt();
    if(size < 0)
```

```
        size -= size - 1;
    srand(time(NULL));
    string arr[size];
    for(int i = 0 ; i < size; i++)
        arr[i] = "";

    int newLen = len->text().toInt();
    if(newLen < 0)
        newLen -= newLen - 1;
    string newRandomValue;
    string a[newLen];
    for (int i = 0; i < newLen; i++)
    {
        newRandomValue = "";
        newRandomValue = generateRandomString(1);
        a[i] = newRandomValue;
    }

    for (int i = 0; i < size;)
    {
        for(int j = 0; j < newLen; j++)
            arr[i] += a[j];
        for (int k = 0; k < newLen; k++)
            swap(a[k], a[rand() % newLen]);
        i++;
    }

    string out = "";
    for (int i = 0; i < size; i++)
        out += arr[i] + " ";
    out.erase(out.size()-1, 1);

    ofstream f(qPrintable(in->toPlainText()), ios::out);
    f << out;
    f.close();
}

void generateWorstCase2(QTextEdit *&in, QLineEdit *&sizeac, QLineEdit *&len)
{
    int size = sizeac->text().toInt();
    if(size < 0)
        size -= size - 1;
    string arr[size];
    string newRandomValue = "";
    for(int i = 0 ; i < size; i++)
        arr[i] = "";
    int newLen = len->text().toInt();
    if(newLen < 0)
```

```
        newLen -= newLen - 1;
    newRandomValue = generateRandomString(newLen);
    for (int i = 0; i < size;)
    {
        arr[i] = newRandomValue;
        i++;
    }
    string out = "";
    for (int i = 0; i < size; i++)
        out += arr[i] + " ";
    out.erase(out.size()-1, 1);
    ofstream f(qPrintable(in->toPlainText()), ios::out);
    f << out;
    f.close();
}

void generateAverageCase(QTextEdit *&in, QLineEdit *&sizeac, QLineEdit *&len)
{
    int size = sizeac->text().toInt();
    if(size < 0)
        size -= size - 1;
    srand(time(NULL));
    string arr[size];
    for(int i = 0 ; i < size; i++)
        arr[i] = "";

    int newLen = len->text().toInt();
    if(newLen < 0)
        newLen -= newLen - 1;
    string newRandomValue;
    for (int i = 0; i < size;)
    {
        newRandomValue = "";
        newRandomValue = generateRandomString(newLen);
        arr[i] = newRandomValue;
        i++;
    }

    string out = "";
    for (int i = 0; i < size; i++)
        out += arr[i] + " ";
    out.erase(out.size()-1, 1);

    ofstream f(qPrintable(in->toPlainText()), ios::out);
    f << out;
    f.close();
}
```

```
string generateRandomString(int length)
{
    auto randchar = []() -> char
    {
        const char alphanum[] =
        "0123456789"
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        "abcdefghijklmnopqrstuvwxyz";
        return alphanum[ rand() % (sizeof(alphanum) - 1) ];
    };

    string str(length, 0);
    generate_n(str.begin(), length, randchar);
    return str;
}
```

ПРИЛОЖЕНИЕ Ж

ИСХОДНЫЙ КОД ПРОГРАММЫ. PROCESSING.H

```
#ifndef PROCESSING_H
#define PROCESSING_H
#include <hash.h>

void deleteItemFromHashTable(QTextEdit *&output, QLineEdit *&input, QTextEdit
*&info);

void insertItemToHashTable(QTextEdit *&output, QLineEdit *&input, QTextEdit
*&info);

void createHashTable(QTextEdit *&output, QTextEdit *&in, QTextEdit *&info);

void generateAverageCase(QTextEdit *&output, QLineEdit *&sizeac, QLineEdit
*&len);

void generateWorstCase(QTextEdit *&in, QLineEdit *&sizeac, QLineEdit *&len);

void generateWorstCase2(QTextEdit *&in, QLineEdit *&sizeac, QLineEdit *&len);

string generateRandomString(int len);

#endif // PROCESSING_H
```