Санкт-Петербургский Национальный Исследовательский Университет
Информационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий

Домашнее задание №2
Вариант №1

Выполнил:
Бацанова Е. А.
Проверил
Мусаев А.А.

Санкт-Петербург,
2024

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Хэширование – метод адресации данных для быстрого поиска по ключевым выражениям.

Целью данной работы является изучение алгоритмов хэширования текстовой информации, их реализация и тестирование на языке питон. Основными задачами работы являются:

1) реализация метода деления для хэширования строки;
2) реализация метода CRC-32 для хэширования строки;
3) тестирование программы в формате ввода пользователем строки для хэширования.

Задание 1

Задание: Пользователь вводит текст. Программа осуществляет хэширование по указанному методу.

Решение:

Взят вариант 1:

➔ Алгоритм 1 – Деление

Реализуем хэширование методом деления с помощью функции *division_hash(data, size)* (рис. 1). Для этого необходимо выбрать размер таблицы хэширования (в данном случае *11*). После этого преобразуем текст в ключ – целочисленное значение, например, путем сложения значений кодов символов текста, используя метод *ord()*. Функция возвращает целое число, которое и будет являться хэшем – остаток от деления полученной суммы на размер таблицы хэширования.

```
def division_hash(data, size):
    s = 0
    for i in data:
        s += ord(i)
    return s % size
```

Рисунок 1 – Реализация хэширования методом деления

➔ Алгоритм 2 – CRC-32.

Далее реализуем алгоритм хэширования CRC-32. Всего это можно сделать тремя способами: с помощью встроенных методов python, стандартного расчета без использования библиотек и табличного (быстрого) метода расчета, который также не использует встроенные библиотеки для расчета хэша методом CRC-32.

Рассмотрим все эти методы и в конце сравним полученные значения хэша.

- Использование встроенных библиотек (рис. 2)

В python существует модуль *zlib* для вычисления CRC-32 для переданных данных. Импортируем ее и создадим функцию *crc(data)*, которая будет принимать строку данных *data* в качестве входного аргумента, затем кодировать ее в байтовый формат с использованием метода *encode()* и применять к ней метод *crc32* из модуля *zlib* для вычисления хэша.

```
def crc(data):
    return zlib.crc32(data.encode())
```

Рисунок 2 – Реализация хэширования CRC-32 с помощью встроенных библиотек python

- Расчет стандартным методом (рис. 3)

Реализуем первый способ самостоятельного расчета хэша. Для этого создадим функцию *crc32(data)*, которая также будет принимать на вход один аргумент *data*, представляющий строку данных, для которой необходимо рассчитать хэш, а затем преобразует ее в байтовую строку text. После этого инициализируются переменные crc (начальное значение CRC) и poly (порождающий полином для CRC-32, стандартно берется полином *0xEDB88320*). Далее происходит итерация по каждому байту в строке данных. Для каждого байта выполняется операция *XOR* с текущим значением *crc*. После этого во вложенном цикле происходит проверка младшего бита – если он равен 1, то выполняется сдвиг вправо на 1 бит и применяется *XOR* с порождающим полиномом. В противном случае выполняется только сдвиг вправо на 1 бит. После обработки всех байтов входных данных выполняется окончательное смещение CRC на 32 бита (используя *XOR* с 0xFFFFFFFF) и возвращается полученное значение хэша.

```
def crc32(data):
    text = data.encode()
    crc = 0xFFFFFFFF
    poly = 0xEDB88320
    for byte in text:
        crc ^= byte
        for _ in range(8):
            if crc & 1:
                crc = (crc >> 1) ^ poly
            else:
                crc >>= 1
    return crc ^ 0xFFFFFFFF
```

Рисунок 3 – Реализация хэширования CRC-32 стандартным методом

- Расчет табличным методом (рис. 4)

Реализуем второй способ самостоятельного расчета хэша. В данном случае таблица масок CRC-32 рассчитывается и заполняется единожды, а затем просто используется для расчета контрольной суммы (хэша). Данный способ считается более быстрым и удобным при расчете хэша для множества строк, но по сравнению со стандартным методом занимает больше памяти.

Для начала сгенерируем таблицу значений CRC-32 для всех возможных байтов (256 значений) с помощью функции *generate_crc_32_table()*. Внутри цикла производим операции стандартное вычисление хэша CRC-32. Саму таблицу сохраняем в переменную *crc32_table*.

Далее реализуем функцию, непосредственно расчитывающую хэш строки *crc32_table_method(data)*. Это функция снова же принимает строку данных, кодирует ее в байты, и считает CRC-32 значения для каждого байта, используя ранее сгенерированную таблицу. Результат находится в *crc* и возвращается после применения финального *XOR* оператора с *0xFFFFFFFF*.

```python
def generate_crc32_table():
    table = []
    poly = 0xEDB88320
    for i in range(256):
        crc = i
        for _ in range(8):
            if crc & 1:
                crc = (crc >> 1) ^ poly
            else:
                crc >>= 1
        table.append(crc)
    return table


def crc32_table_method(data):
    buf = data.encode()
    crc = 0xFFFFFFFF
    for byte in buf:
        crc = (crc >> 8) ^ crc32_table[(crc ^ byte) & 0xFF]
    return crc ^ 0xFFFFFFFF


crc32_table = generate_crc32_table()
```

Рисунок 4 – Реализация хэширования CRC-32 табличным методом

Проверим работу программы. Видим, что хэш CRC-32, рассчитанный всеми тремя способами, совпал.

```
Введите текст для хэширования:
abc
Хэш методом деления:  8
Хэш CRC-32 встроенным методом:  891568578
Хэш CRC-32 стандартным методом:  891568578
Хэш CRC-32 табличным методом:  891568578
```

Рисунок 5 – Вывод программы расчета хэша методом деления и CRC-32

ЗАКЛЮЧЕНИЕ

В результате выполнения работы были реализованы алгоритмы хэширования текстовой информации методом деления и CRC-32.

Сначала был реализован метод деления. Данный метод является простым в реализации, но способен создавать большое количества коллизий, из-за чего на входной набор строк необходимо накладывать ограничения.

После этого тремя методами было реализо хэширование CRC-32: с помощью встроенного модуля *zlib*, стандартным и табличным методом. Вывод для всех трех вариантов оказался одинаков, что говорит о том, что самостоятельный расчет хэша был реализован верно.

В итоге мы получили два метода, которые способны успешно хэшировать входные данные. Таким образом, можно сделать вывод, что алгоритмы хэширования методом деления и CRC-32 можно успешно применять для обработки текстовой информации в различных прикладных задачах.

СПИСОК ЛИТЕРАТУРЫ

1) Викиучебник. Реализации алгоритмов/Циклический избыточный код. [Электронный ресурс] – (Дата последнего обращения 19.05.2024);
2) Wikipedia. Хеш-функция. [Электронный ресурс] – (Дата последнего обращения 19.05.2024).

ПРИЛОЖЕНИЕ

Для удобства все файлы выгружены на GitHub: https://github.com/kathykkKk/Algorithms-and-Data-Structures-ICT.git