МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

БРЯНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра «Информатика и программное обеспечение»

Курсовая работа

по дисциплине «Структуры и алгоритмы обработки данных»

Тема: «Расширяемое хеширование»

Всего листов:15

Студент гр. О-22-ПРИ-1-рпс-Б

_____Семин И. Д.

№ зач. книжки 22.0356

Преподаватель

_____ Трубаков А. О.

БРЯНСК 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

При частом добавлении новых элементов в хеш-таблицу может возникнуть проблема полного заполнения таблицы, что требует перехеширования. Хотя при малых размерах таблицы это не вызывает трудностей, при больших размерах таблицы полное перехеширование может потребовать значительного времени. Кроме того, если новые значения поступают часто, то может потребоваться частое перехеширование таблицы или выделение больших объемов памяти, которые могут не понадобиться и будут зарезервированы впустую. Дополнительной проблемой является риск коллизий, когда два разных значения попадают в одну ячейку. Расширяемое хеширование предлагает решение этих проблем, позволяя эффективно управлять памятью и избегать коллизий, не требуя при этом выделения дополнительных ресурсов.

Цель данной работы – реализация алгоритма расширяемого хеширования и анализ его работы на разных наборах значений для оценки эффективности.

АНАЛИТИЧЕСКАЯ ЧАСТЬ

Описание расширяемого хеширования

Расширяемое хеширование (Extendible Hashing) — это алгоритм хеширования, разработанный для решения проблемы неравномерного распределения хеш-значений, которое может привести к коллизиям и снижению эффективности работы. Этот алгоритм использует список разделов, которое позволяет расширять таблицу хеширования в процессе добавления новых данных, обеспечивая равномерное распределение хеш-значений и снижая вероятность коллизий. Расширяемое хеширование нашло широкое применение в различных областях, таких как базы данных, сети и криптография, благодаря своей способности масштабироваться и поддерживать высокую производительность при работе с большими объемами данных.

Общее описание алгоритма

Метод расширяемого хеширования (англ. *extendible hashing*) заключается в том, что хеш-таблица представлена как каталог (англ. *directory*), а каждая ячейка будет указывать на блок (англ. *bucket*) который имеет определенную вместимость (англ. capacity). Сама хеш-таблица будет иметь глобальную глубину (англ. *global depth*), а каждый из блоков имеет локальную глубину (англ. *local depth*). Глобальная глубина показывает сколько последних бит будут использоваться для того чтобы определить в какой блок следует заносить значения. А из разницы локальной глубины и глобальной глубины можно понять сколько ячеек каталога ссылаются на блок. Это можно показать формулой $K{=}2^{G-L}$ где G — глобальная глубина, L — локальная глубина, а K количество ссылающихся ячеек.

Хеш-функция преобразует ключи в произвольную битовую строку (псевдоключ). Элементы адресуются по некоторому количеству битов псевдоключа. При поиске берется некоторое количество битов псевдоключа равное глобальной глубине и через каталог находится адрес искомого блока.

Добавление элементов происходит по следующему алгоритму:

- Если блок неполон запись добавляется в него
- Если блок заполнен он разбивается на 2 блока и записи перераспределяются между ними.
- Если же у переполненного блока локальная глубина совпадает с глобальной, то в этом случае размер списка удваивается и каждому вновь созданному элементу присваивается указатель, который содержит его родитель. Таким образом несколько элементов могут указывать на один блок.

Удаление производится по обратному алгоритму.

Преимущества алгоритма

Основным достоинством расширяемого хеширования является высокая скорость доступа к данным, которая не падает с увеличением объема данных. Также можно отметить оптимальный расход места на устройстве хранения данных, так как блоки выделяются только для существующих данных, а список указателей имеет минимально необходимые размеры для адресации необходимого числа блоков.

Время доступа к данным - всегда 1

Использование памяти - ln2

Недостатки алгоритма

К недостаткам можно отнести быстрый рост таблицы в оперативной памяти при неравномерном распределении значений функции хеширования.

РЕАЛИЗАЦИЯ АЛГОРИТМА

Описание класса блока

```
class Bucket {
public:
    int localDepth;
    int size;
    vector<int> keys;
    vector<int> values;

    Bucket(int size, int localDepth) {
        this->localDepth = localDepth;
        this->size = 0;
        keys.resize(size);
        values.resize(size);
    }
    bool isFull() {
        return size == keys.size();
    }

    bool isEmpty() {
        return size == 0;
    }

    void insert(int key, int value) {
        for (int i = 0; i < size; i++) {
            if (keys[i] == key) {
                values[i] = value;
                return;
            }
        }
        if (isFull()) {
            throw runtime_error("Bucket is full");
        }
        keys[size] = key;
        values[size] = value;
        size++;
    }

    ~Bucket() {
        keys.clear();
        values.clear();
    }
};
```

Листинг 1. Класс блока хеш-таблицы

Данный класс описывает блок, в который записываются данные. Блок содержит поля для хранения массивов данных заданного размера в формате ключ-значение, поля для хранения локальной глубины и текущего размера

блока. Конструктор блока позволяет инициализировать его, задавая параметры максимального размера блока и его локальной глубины. Также в блоке описаны вспомогательные методы для проверки блока на пустоту или заполненность. Описан метод insert реализующий вставку данных в блок и деструктор для очистки памяти.

Описание класса хэш-таблицы

```
     class ExtendibleHash {
     private:
         vector<Bucket*> directory;
         int global_depth;
         ExtendibleHash(int bucketSize, int global_depth)
         :global_depth(global_depth), directory(1 <<
global_depth){
             for (size_t i = 0; i < directory.size(); ++i) {
                 directory[i] = new Bucket(bucketSize,
global_depth);
             }
         }

         int get_index(int key) const{…}
         void expand_directory(){…}
         void rehash(Bucket* bucket, int local_depth){…}
         void insert(int key, int value){…}
     };
```

Листинг 2. Класс хеш-таблицы

Класс хеш-таблицы хранит массив указателей на блоки и глобальную глубину таблицы. Таблица инициализируется значениями максимального размера каждого блока и глобальной глубины. В зависимости от глобальной глубины выделяется память под 2^G ячеек таблицы и для каждой ячейки создается блок с локальной глубиной, равной глобальной. Также в классе описана хеш-функция для получения псевдоключа, по которому можно найти нужный элемент, и методы для удвоения размера таблицы, разделения блока и вставки элемента.

Описание хеш-функции

```
int get_index(int key) const {
    int index = 0;
    int mask = 1 << (global_depth - 1);
    for (int i = global_depth - 1; i >= 0; --i) {
        if (key & mask) {
            index |= (1 << i);
        }
        mask >>= 1;
    }
    return index;
}
```

Листинг 3. Хеш-функция

Хеш функция возвращает определенное количество последних бит числа, зависящее от глобальной глубины. Для этого используются логические операции и операции побитового сдвига.

Описание алгоритма вставки

```
void insert(int key, int value) {
    int index = get_index(key);
    Bucket* bucket = directory[index];

    if (!bucket->isFull()) {
        bucket->insert(key, value);
    }
    else {
        if (bucket->localDepth == global_depth) {
            expand_directory();
            global_depth++;
        }
        rehash(bucket, bucket->localDepth);
        insert(key, value);
    }
}
```

Листинг 4. Метод вставки нового набора данных

Вставка данных происходит по следующему алгоритму: Сначала формируется псевдоключ при помощи хеш-функции, затем, если в блоке по адресу ячейки с данным псевдоключом есть место, то происходит вставка данных в этот блок, иначе проверяется совпадение значений локальной глубины блока с глобальной глубиной таблицы и если они совпадают, то размер таблицы

удваивается по соответствующему алгоритму, и затем происходит разделение блока и рекурсивный вызов функции.

Описание алгоритма разделения блока

```
void rehash(Bucket* bucket, int local_depth) {
    int new_local_depth = local_depth + 1;

    vector<Bucket*> new_buckets(2);
    for (size_t i = 0; i < 2; ++i) {
        new_buckets[i] = new Bucket(bucket->keys.size(),
new_local_depth);
    }

    for (size_t i = 0; i < bucket->size; ++i) {
        int key = bucket->keys[i];
        int value = bucket->values[i];
        int index = get_index(key);
        new_buckets[(key >> local_depth) & 1]->insert(key, value);
    }

    int directory_size = 1 << global_depth;
    for (int i = 0; i < directory_size; ++i) {
        if (directory[i] == bucket) {
            directory[i] = new_buckets[(i >> local_depth) & 1];
        }
    }

    delete bucket;
}
```

Листинг 5. Метод разделения блока

Разделение блока происходит по следующему алгоритму: сначала создаются 2 новых блока с локальной глубиной на 1 больше чем у предыдущего, затем данные из старого блока распределяются между двумя новыми блоками в зависимости от ключа с учетом новой локальной глубины, после чего указатели на новые блоки присваиваются соответствующим ячейкам таблицы, а указатель на старый блок удаляется.

Описание алгоритма удвоения размера каталога

```
void expand_directory() {
    vector<Bucket*> new_directory(directory.size() * 2);
    for (size_t i = 0; i < directory.size(); ++i) {
        new_directory[i] = directory[i];
        new_directory[i + directory.size()] = directory[i];
```

```
    }
    directory = new_directory;
}
```

Листинг 6. Метод удвоения размера каталога

Данный метод удваивает размер каталога и присваивает каждой новой ячейке адрес блока, на который ссылались ячейки с таким же набором последних G бит ключа.

ОЦЕНКА ЭФФЕКТИВНОСТИ

Для оценки эффективности алгоритма мы протестируем и оценим работу алгоритма на разных наборах данных: упорядоченный, с равномерным и неравномерным распределением. Для наглядности будем использовать разное количество данных: 1000, 10000 и 100000.

Для измерения времени работы алгоритма с упорядоченным набором данных используем следующий алгоритм:

```
ExtendibleHash hash(2, 1);
clock_t start = clock();
for (int i = 0; i < 1000; i++)
{
    hash.insert(i, i);
}
clock_t end = clock();
double time = (double)(end - start);
printf("The time: %f miliseconds\n", time);
```

Листинг 7. Вставка упорядоченного набора данных

Для измерения времени работы алгоритма используется функция clock(), с помощью которой можно найти разницу между временем начала и окончания работы алгоритма.

Для измерения времени работы алгоритма с неупорядоченным набором данных с равномерным распределением изменим цикл:

```
    for (int i = 0; i < 100000; i++)
    {
        int key = rand() % 100000;
        hash.insert(key, i);
    }
```

Листинг 8. Вставка набора данных с равномерным распределением

Для измерения времени работы алгоритма с неупорядоченным набором данных с неравномерным распределением изменим цикл:

```
    for (int i = 0; i < 100000; i++)
    {
        int key;
        if (rand() % 2)
            key = rand() % 100000;
        else key = rand() % 1000;
        hash.insert(key,i);
    }
```

Листинг 9. Вставка набора данных с неравномерным распределением

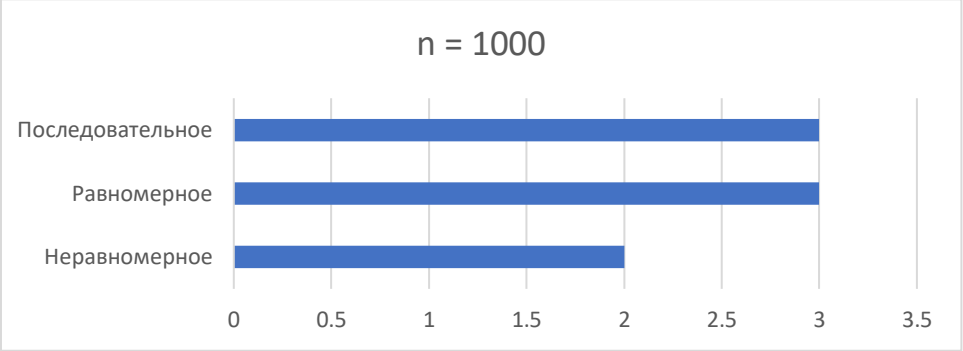
В результате работы получили следующие данные:

	n	1000	10000	100000
Неравномерное	t	2	218	2826
Равномерное	t	3	383	3843
Последовательное	t	3	130	10488

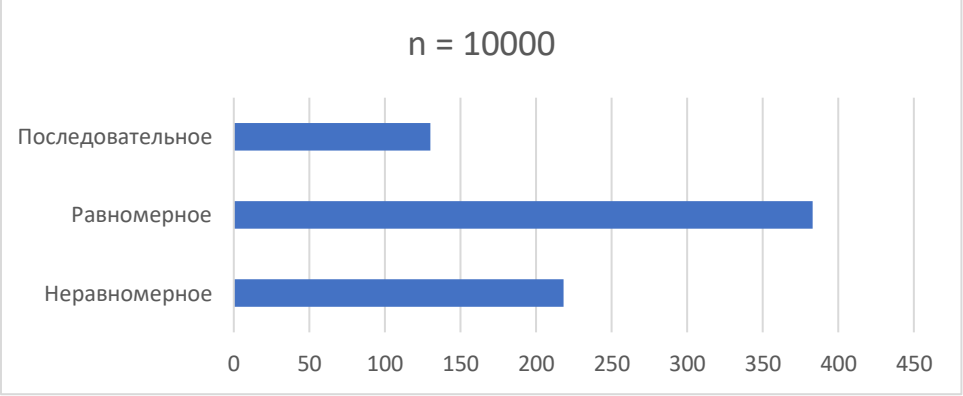
Таб 1. Эффективность алгоритма

Где n – количество вставок, а t – время, затраченное на заполнение хеш-таблицы в милисекундах.

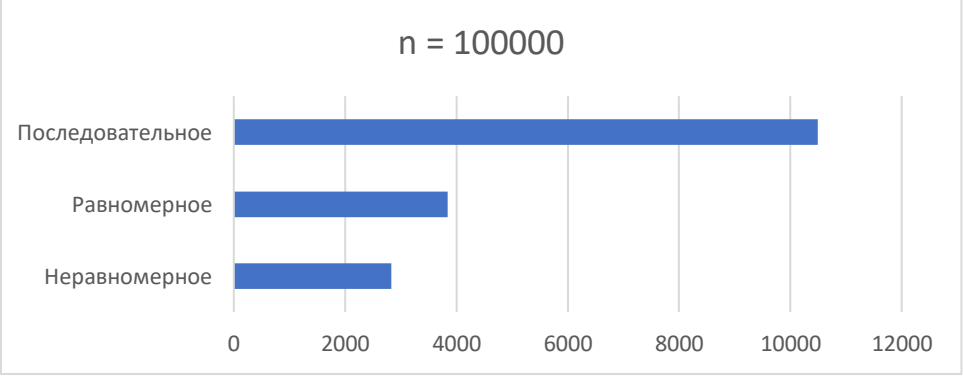
Для наглядности построим гистограммы с использованием полученных данных.



При малом количестве данных разница во времени работы несущественна, алгоритм работает достаточно быстро.



При увеличении количества данных становится заметно, что равномерное распределение данных по таблице занимает больше времени.



На большом наборе данных алгоритм начинает заметно медленней работать при вводе упорядоченных данных, за счет большого количества схожих битовых окончаний.

ЗАКЛЮЧЕНИЕ

В данной работе мы продемонстрировали пример реализации алгоритма расширяемого хеширования и проанализировали скорость работы алгоритма на разных наборах данных.

Данный алгоритм является вариантом решения наиболее существенной проблемы хеширования – статичный размер хеш-таблицы. Расширяемое хеширование не только позволяет подстраивать размер таблицы под необходимое количество данных, но и делать это с минимальными затратами ресурсов, так как изменению обычно подвергается только один блок, а в худшем случае таблица увеличивается вдвое, что делает хеш-таблицу гибкой и масштабируемой.

СПИСОК ЛИТЕРАТУРЫ

1. Гулаков В.К., Гулаков К.В. Введение в хеширование: учеб. пособие. М.: БГТУ, 2011.-131 с.
2. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible Hashing-A Fast Access Method for Dynamic Files: научная статья. IBM Research Laboratory, 1981.-30с.
3. Расширяемое хеширование: статья. ИТМО (ссылка)