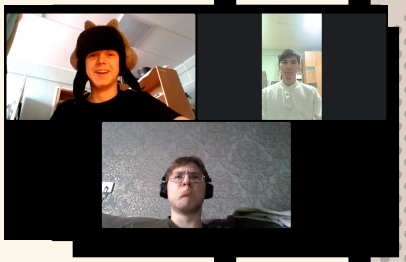
Алгоритм вытеснения кэша: LFU



Проект выполнили: Ниф Насыров, Наф Фарафонтов и Нуф Ултургашев

Содержимое zashity

“У нас было 2 блока данных, 75 записей с амплитудой LFU, 5 пакетов с данными на динамическое выделение памяти и солонка, наполовину заполненная информацией о кеш-попаданиях. А еще целое море разнообразных алгоритмов замещения, барьеров и стратегий вытеснения”

Алгоритм БАЗАвого LRU	Как работает алгоритм LRU, на котором основан наш метод LFU
НАШ слон (алгоритм LFuuuu)	Ну что ж, пришло время узнать, на что нам осталось три дня
Недостатки метода LRU	Раскопаем белье LRU и поймаем его за выброшенный элемент
Реализация	Что мы там напечатали и как решили делить
Сравнение LRU и LFU	Наконец-то! Достойный противник!
Итоги	В конце они оба умрут

01

Алгоритм БАЗАвого LRU

Как работает алгоритм LRU, на котором основан наш метод LFU

Least Recently Used (LRU) – это алгоритм замещения. Основная идея LRU заключается в том, что элементы, которые дольше всего не использовались, будут выбраны для удаления в случае необходимости освобождения места для новых элементов.

Обычно он применяется в базах данных для кэширования часто выполняемых запросов, чтобы ускорить доступ к часто запрашиваемым данным и снизить нагрузку на сервер. Также LFU может использоваться для кэширования рекомендаций или профилей пользователей, которые часто взаимодействуют с системой.

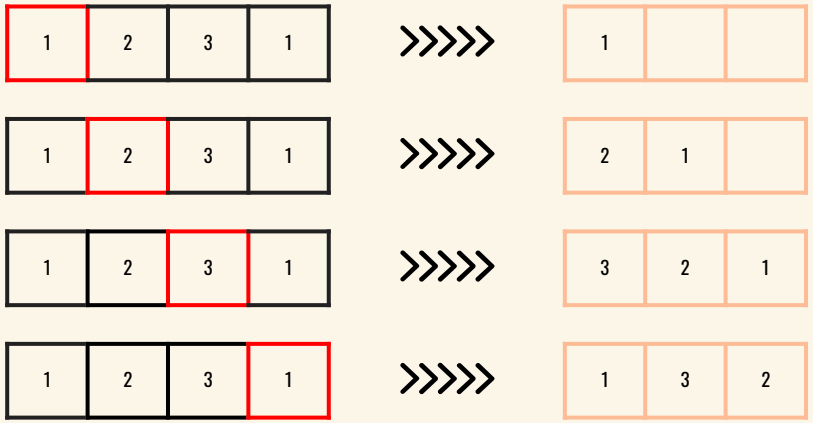



Как работает LRU?

- Инициализация: кэш инициализируется с заданным размером.
- Доступ к данным: при каждом доступе к элементу, элемент помечается как недавно использованный.
- Добавление нового элемента: если кэш заполнен, удаляется элемент, к которому не обращались дольше всего.
- Обновление порядка: для отслеживания порядка доступа используется двусвязный список и хэш-таблица.

Рассмотрим схематичный вид алгоритма:

- Если запрошенный элемент найден, он перемещается вперёд
- Если нет, он помещается вперёд, а последний вытесняется



02

НАШ слон (алгоритм LFuuuu)

Ну что ж, пришло время узнать, на что нам осталось три дня

LFU (Least Frequently Used) — это алгоритм кэширования, который удаляет наименее часто используемый элемент, чтобы освободить место для нового элемента.

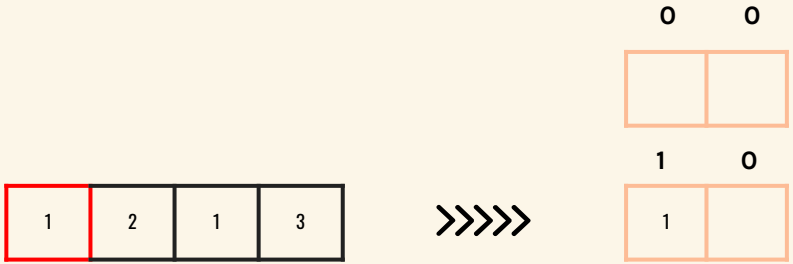
Обычно он применяется в кэшах (например, кэш веб-страницы, кэш запросов к базе данных и т.д.), где имеется ограниченное количество места для хранения данных.

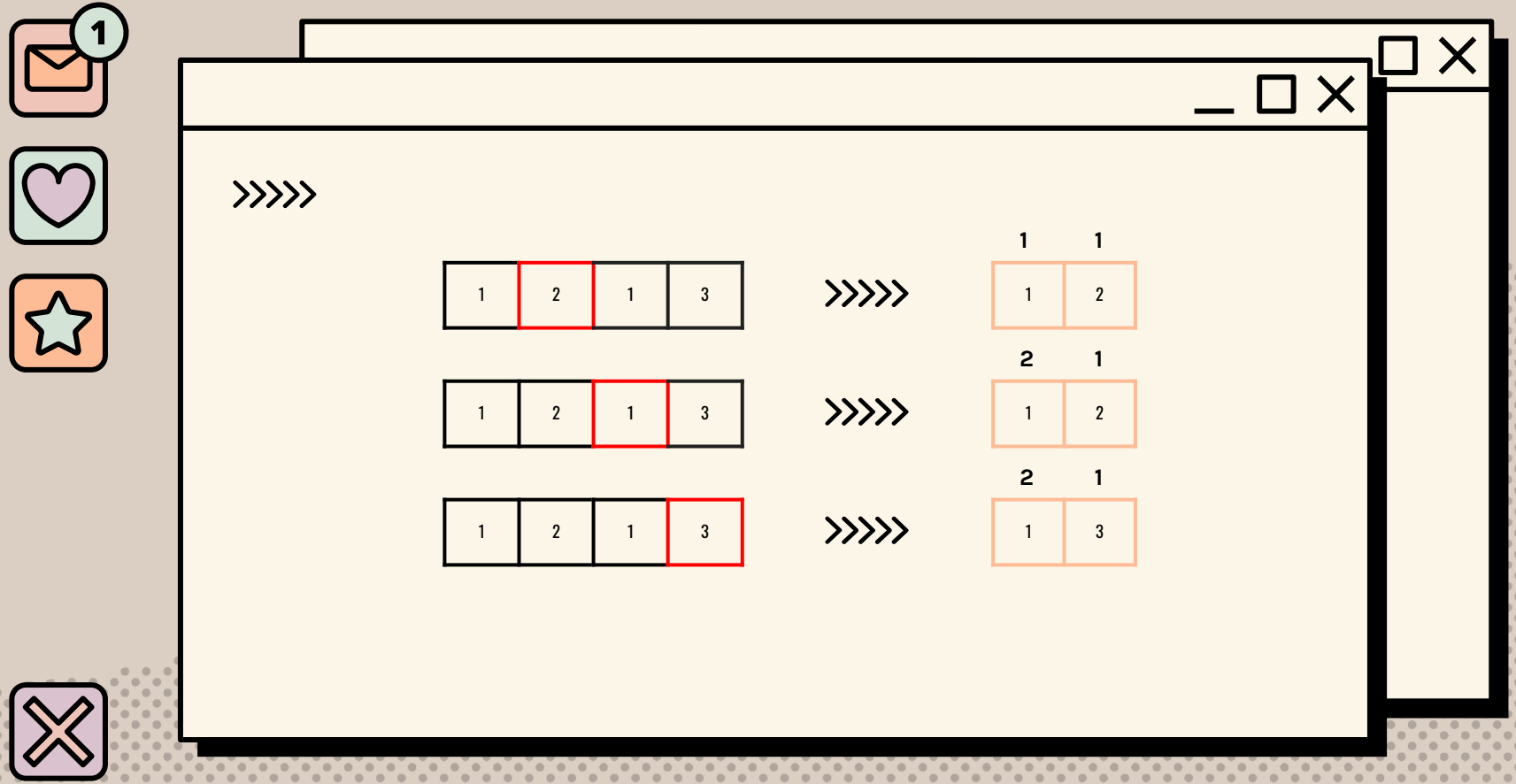
Как работает LFU?

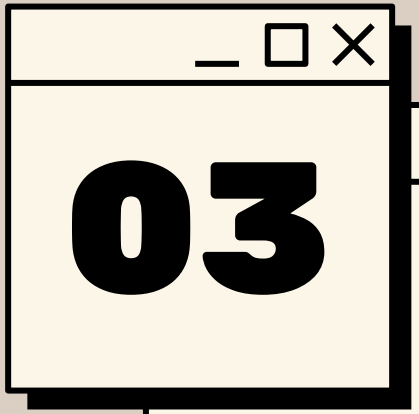
- Инициализация: кэш инициализируется с заданным размером.
- Доступ к данным: при каждом доступе к элементу увеличивается счётчик его использования.
- Добавление нового элемента: если кэш заполнен, удаляется элемент с наименьшим счётчиком использования.
- Обновление счётчиков: элементы с увеличенным счётчиком перемещаются в соответствующую позицию для поддержания порядка.

Рассмотрим схематичный вид алгоритма:

- Проверка кэша: ищем элемент в кэше.
- Обновление счётчика: если элемент найден, увеличиваем его счётчик использования.
- Добавление: если элемент не найден, добавляем его с начальным счётчиком. Если кэш заполнен, удаляем элемент с наименьшим счётчиком.




1
1 2 1 3
1 1
1 2
1 2 1 3
2 1
1 2
1 2 1 3
2 1
1 3



Недостатки алгоритма LRU

Раскопаем белье LRU и поймаем его за выброшенный элемент

Какие-то проблемы?

Частоты

Игнорирование частоты:
LRU не учитывает, как часто используются элементы

Циклы

Проблемы с циклическими паттернами:
Может быть неэффективен при циклическом доступе к данным

04

Реализация алгоритма LFU

Что мы там напечатали и как решили делить

Структуры:

Наш метод LFU реализован через использование двусвязного списка и частотного счетчика.

```
typedef struct Cache
{
    int capacity;
    int size;
    CacheNode *head;
    CacheNode *tail;
} Cache;
```

Структура кэша содержит:

- capacity: максимальная емкость кэша.
- size: текущий размер кэша.
- head и tail: указатели на первый и последний элементы двусвязного списка.

Структуры:

```
typedef struct
CacheNode
{
    int key;
    int value;
    int freq;
    struct CacheNode
next;
    struct CacheNode*
prev;
} CacheNode;
```

Каждый узел в кэше содержит:

- key: ключ, идентифицирующий элемент.
- value: значение элемента.
- freq: частота доступа к этому элементу.
- next и prev: указатели на следующий и предыдущий элементы в двусвязном списке.

Функции

```
CacheNode* createNode(int key, int
value)
{
    CacheNode* node =
(CacheNode*)malloc(sizeof(CacheNode));
    node->key = key;
    node->value = value;
    node->freq = 1;
    node->prev = NULL;
    node->next = NULL;
    return node;
}
```

Создает новый узел с заданным ключом, значением и частотой, равной 1.

Функции

```
Cache* createCache int capacity)
{
    Cache* cache =
(Cache*)malloc(sizeof(Cache));
    cache->capacity = capacity;
    cache->size = 0;
    cache->head = NULL;
    cache->tail = NULL;
    return cache;
}
}
```

Создает новый кэш с заданной емкостью.

Функции

```
void removeNode(Cache* cache, CacheNode*
node)
{
    if (node->prev)
    {
        node->prev->next = node->next;
    }
    else
    {
        cache->head = node->next;
    }

    if (node->next)
    {
        node->next->prev = node->prev;
    }
    else
    {
        cache->tail = node->prev;
    }

    free(node);
    cache->size--;
}
```

Удаляет узел из кэша и освобождает его память.

Функции

```
void insertHead(Cache* cache,
CacheNode* node)
{
    node->next = cache->head;
    node->prev = NULL;

    if (cache->head)
    {
        cache->head->prev = node;
    }
    else
    {
        cache->tail = node;
    }

    cache->head = node;
    cache->size++;
}
}
```

Вставляет узел в начало кэша.

Функции

Код функции updateFrequency
Не влез, поэтому вот qr-котик на наш репозиторий

Обновляет частоту доступа к узлу и перемещает его в соответствующую позицию в кэше на основе новой частоты.

Функции

```
void insertHead(Cache* cache,
CacheNode* node)
{
    node->next = cache->head;
    node->prev = NULL;

    if (cache->head)
    {
        cache->head->prev = node;
    }
    else
    {
        cache->tail = node;
    }

    cache->head = node;
    cache->size++;
}
}
```

Вставляет узел в начало кэша.

Функции

```
int get(Cache* cache, int key) {
    CacheNode* current =
cache->head;

    while (current) {
        if (current->key == key) {
            updateFrequency(cache,
current);
            return current->value;
        }
        current = current->next;
    }

    return -1;
}
```

Получение значения из кэша по заданному ключу.

Функции

```
void put(Cache* cache, int key, int value) {
    CacheNode* current = cache->head;

    while (current) {
        if (current->key == key) {
            current->value = value;
            updateFrequency(cache, current);
            return;
        }
        current = current->next;
    }

    if (cache->size == cache->capacity) {
        removeNode(cache, cache->tail);
    }

    CacheNode* node = createNode(key, value);
    insertHead(cache, node);
}
```

Вставка нового значения в кэш.

Функции

```
int LFUCacheHits(int capacity, int n, int* requests) {
    Cache* cache = createCache(capacity);
    int hits = 0;

    for (int i = 0; i < n; i++) {
        int page = requests[i];

        if (get(cache, page) != -1) {
            hits++;
        } else {
            put(cache, page, page);
        }
    }

    CacheNode* current = cache->head;

    while (current) {
        CacheNode* next = current->next;
        free(current);
        current = next;
    }

    free(cache);

    return hits;
}
```

Обработка массива запросов и подсчет количества попаданий (хитов) в кэш.

05

Сравнение результатов работы LFU и LRU

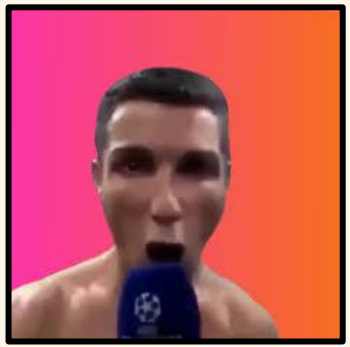
Что мы там напечатали и как решили делить

Результаты:

- В 13 тестах из 27 LRU показал лучшее количество попаданий, а LFU - в 14 тестах. Это свидетельствует о том, что в плане количества попаданий оба метода показывают почти равные результаты, с небольшим преимуществом LFU
- Время выполнения алгоритма LFU было значительно меньше во всех тестах по сравнению с LRU. В среднем, время выполнения для LRU колебалось от 0.001046 до 0.001343 секунд, тогда как для LFU - от 0.000443 до 0.000770 секунд. Это означает, что LFU гораздо более эффективен по времени выполнения.

	Количество попаданий		Время выполнения, с	
	LRU	LFU	LRU	LFU
Тест 1	2523	2513	0.001185	0.000706
Тест 2	2462	2469	0.001170	0.000653
Тест 3	2460	2456	0.001159	0.000686
Тест 4	2400	2410	0.001172	0.000660
Тест 5	2461	2462	0.001160	0.000653
Тест 6	2523	2518	0.001197	0.000651
Тест 7	2549	2549	0.001212	0.000648
Тест 8	2488	2487	0.001172	0.000650
Тест 9	2426	2422	0.001169	0.000659
Тест 10	2478	2473	0.001165	0.000660
Тест 11	2509	2500	0.001166	0.000650
Тест 12	2488	2485	0.001213	0.000653
Тест 13	2498	2501	0.001285	0.000661
Тест 14	2517	2520	0.001172	0.000654
Тест 15	2494	2498	0.001146	0.000944
Тест 16	2507	2504	0.001160	0.000779
Тест 17	2463	2464	0.001169	0.000661
Тест 18	2470	2467	0.001204	0.000662
Тест 19	2549	2550	0.001352	0.000653
Тест 20	2529	2523	0.001175	0.000655
Тест 21	2563	2567	0.001168	0.000645
Тест 22	2544	2554	0.001171	0.000647
Тест 23	2484	2491	0.001159	0.000659
Тест 24	2509	2517	0.001215	0.000647
Тест 25	2475	2479	0.001319	0.000672
Тест 26	5968	5964	0.001343	0.000458
Тест 27	5629	5627	0.001170	0.000444
Итог	**13**	**14**	**0**	**27**

Выводы



LFiiiuuu

Название говорит
само за себя

Выбрать

LRU

Чудной, нелепый,
но зато родной

Выбрать

Где быстрее, Лебовски?

Эффективность

Оба алгоритма могут быть использованы в зависимости от требований кэширования, так как по количеству попаданий они близки друг к другу

Производительность

Если важна скорость выполнения, то LFU является более предпочтительным выбором, так как он показывает меньшее время выполнения во всех тестах

Баланс

LFU лучше подходит для минимального время отклика, тогда как LRU может быть выбран, если его характер кэширования более подходит под конкретные требования приложения