HASH TABLE

HARD SKILL

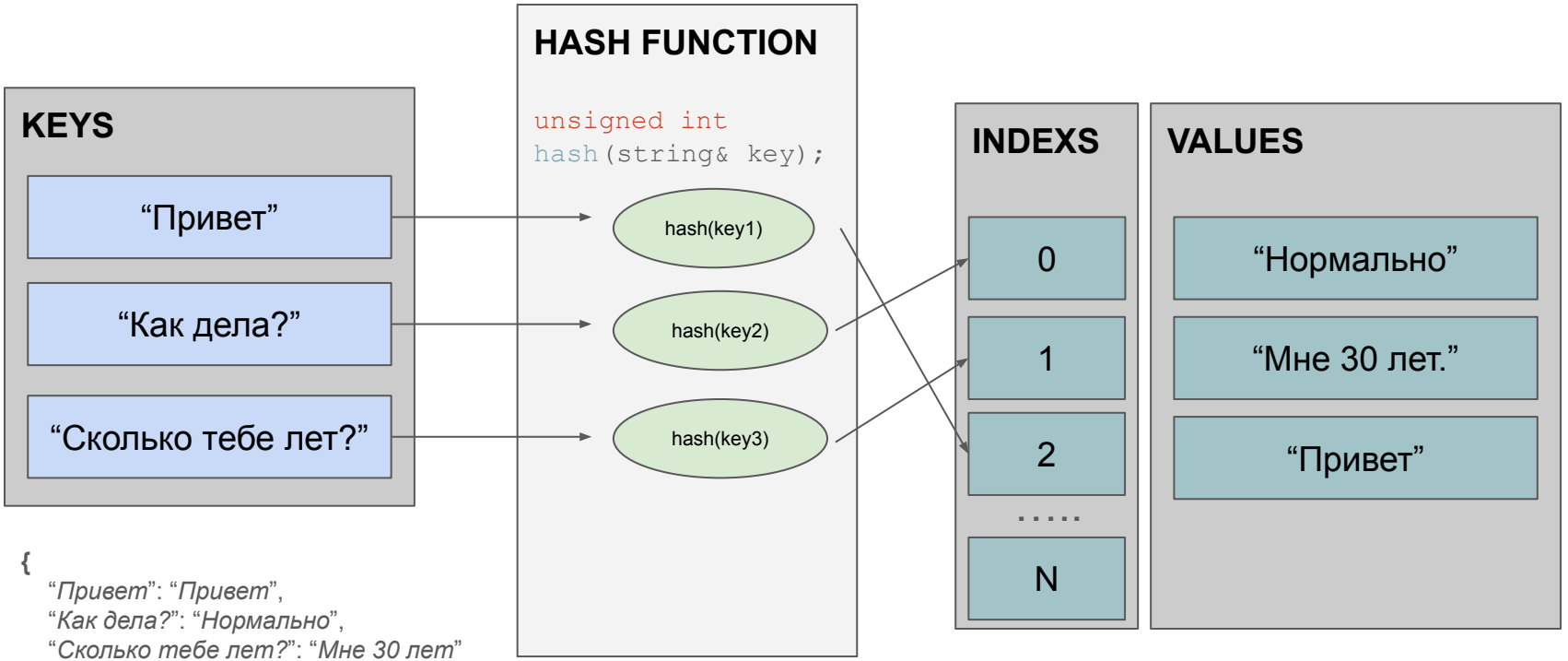
Основные понятия.

Хэш-таблица (hash table) - это структура данных, предназначенная для хранения пар "ключ-значение" (key-value pairs), где ключ используется для быстрого доступа к соответствующему значению. Она основана на идее хэширования, которая позволяет быстро находить информацию в таблице по ключу.

Ключ (key) - это уникальный идентификатор, который используется для доступа к соответствующему значению в структуре данных, такой как ассоциативный массив или хэш-таблица.

Значение (value) - это данные, связанные с определенным ключом. Ключи обычно используются для быстрого поиска, доступа, вставки или удаления значений в структурах данных, где они представляют собой связи между данными и их идентификаторами.

Схема работы.



{
“*Привет*”: “*Привет*”,
“*Как дела?*”: “*Нормально*”,
“*Сколько тебе лет?*”: “*Мне 30 лет*”
}

базовый класс

```
class HashTable {
private:
    Std::vector<Std::pair<std::string,  int>> table;
    int size; // размер таблицы

    // хеш-функция
    unsigned int hash(const std::string& key) {
        unsigned int hash = 0;
        for (char c : key) {
            hash = hash *  31 + c;
        }
        return hash % size;
    }

public:
    // Конструктор
    HashTable(int size) : size(size) {table.resize(size);}
```

основные методы.

- Метод вставки элемента - void insert(const std::string& key, int value)
- Метод поиска элемента по ключу - int find(const std::string& key) const;
- Метод удаления элемента по ключу - void remove(const std::string& key)
- Конструктор класса.
- Деструктор при необходимости.
- hash функция - в private секции;
- Метод возврата размера таблицы - int getSize() const;
- Метод вывода таблицы в консоль - void print() const;

КОЛЛИЗИЯ

Когда два ключа хешируются в одно и то же значение, это называется **коллизией**. Есть несколько методов решения коллизий в хеш-таблицах, вот некоторые из них:

Открытая адресация: При этом методе элементы вставляются непосредственно в саму таблицу. Когда возникает коллизия, новый элемент ищет следующую доступную ячейку. Этот процесс может быть реализован различными способами, такими как линейное пробирование, квадратичное пробирование или двойное хеширование.

Метод цепочек: При этом методе каждая ячейка хеш-таблицы является связным списком элементов, которые имеют одинаковый хеш-код. Когда происходит коллизия, новый элемент добавляется в конец списка в соответствующей ячейке. Этот метод обычно требует дополнительной памяти для хранения указателей на списки.

Коэффициент заполнения и перехеширование: Важно следить за коэффициентом заполнения хеш-таблицы, который представляет собой отношение числа элементов к размеру таблицы. Если коэффициент заполнения становится слишком высоким, производительность может ухудшиться из-за увеличения коллизий. Перехеширование позволяет увеличивать размер таблицы и перераспределять элементы, чтобы уменьшить коэффициент заполнения и сохранить быструю производительность.

ОТКРЫТАЯ АДРЕСАЦИЯ;

```
void insert(int key, int value) {
    if (size == capacity)
        throw std::overflow_error("Hash table is full");

    int index = hash(key);
    int attempt = 0;

    while (table[index].first != -1) { // Пока не найдем пустую ячейку
        attempt++;
        index = linearProbe(index, attempt);
    }

    table[index] = std::make_pair(key, value);
    size++;
}
```

```
int linearProbe(int index, int attempt)
{
    return (index + attempt) % capacity;
// Линейное пробирование
}
```

МЕТОД ЦЕПОЧЕК;

```
void insert(int key, int value) {
    int index = hash(key);
    table[index].push_back(std::make_pair(key, value));
}
```

КОЭФ. ПЕРЕЗАПОЛНЕНИЯ/ПЕРЕХЕШИРОВАНИЕ

Коэффициент заполнения (load factor) в хеш-таблице - это отношение числа элементов к размеру таблицы. Он играет важную роль в производительности хеш-таблицы. При увеличении коэффициента заполнения вероятность коллизий увеличивается, что может привести к ухудшению производительности операций вставки, поиска и удаления.

Чтобы поддерживать хорошую производительность, коэффициент заполнения обычно поддерживают на относительно низком уровне, например, менее **0.7**. Когда коэффициент заполнения становится слишком высоким, рекомендуется увеличить размер таблицы и перехешировать все элементы.

Перехеширование (rehashing) - это процесс изменения размера таблицы и перераспределения элементов по новым индексам после изменения размера. Обычно размер таблицы увеличивают в два раза или более.

Пример кода. PRIVATE SECTION

```
private:
    std::vector<std::list<std::pair<int, int>>> table;
    int size;
    int capacity;
    float loadFactorThreshold;

    int hash(int key) {
        return key % capacity;
    }
    void rehash() {...
```

Пример кода. rehash

```
void rehash() {
    int newCapacity = capacity * 2;
    std::vector<std::list<std::pair< int, int>>> newTable(newCapacity);

    // Перехеширование всех элементов в новую таблицу
    for (auto& list : table) {
        for (const auto& pair : list) {
            int newIndex = pair.first % newCapacity;
            newTable[newIndex].push_back(pair);
        }
    }

    table = std::move(newTable);
    capacity = newCapacity;
}
```

Пример кода. insert

```
void insert(int key, int value) {
    if ((float)(size + 1) / capacity > loadFactorThreshold) {
        rehash();
    }

    int index = hash(key);
    table[index].push_back(std::make_pair(key, value));
    size++;
}
```

<hash table with Dynamic Hash Function>

Хеш-таблицы с динамическими хеш-функциями - это специальный тип хеш-таблиц, где хеш-функции могут изменяться в процессе работы в зависимости от образцов запросов или изменений в данных. Вместо того чтобы использовать одну фиксированную хеш-функцию для всех ключей, динамические хеш-функции могут адаптироваться к изменяющимся условиям и данным, что позволяет уменьшить количество коллизий и повысить производительность.

Основная идея динамических хеш-таблиц заключается в том, что хеш-функция может изменяться или выбираться из некоторого набора хеш-функций в зависимости от текущих обстоятельств. Это может быть особенно полезно, когда статическая хеш-функция оказывается неэффективной из-за изменения распределения ключей или образцов запросов.

Преимущества динамических хеш-таблиц:

1. **Адаптивность к изменениям:** Позволяет адаптироваться к изменениям в распределении ключей или образцов запросов, что может привести к улучшению производительности.
2. **Уменьшение коллизий:** Выбор или изменение хеш-функции может помочь уменьшить количество коллизий, так как новая функция может лучше распределять ключи по корзинам.
3. **Большая гибкость:** Динамические хеш-таблицы предоставляют большую гибкость в выборе стратегии хеширования, что может быть полезно в различных сценариях.

1.1 Пример

```
class DynamicHashTable {
private:
    std::vector<std::list<int>> table;
    int size;
    int capacity;
    float loadFactorThreshold;
    int numHashFunctions; // Количество хеш-функций

    int hash(int key, int index) {
        return (key + index) % capacity; // Простое линейное пробирование
    }

    void rehash() {
        ...
```

1.2 Пример

```
void rehash() {
        int newCapacity = capacity * 2;
        std::vector<std::list< int>> newTable(newCapacity);
        // Перехеширование всех элементов в новую таблицу
        for (auto& list : table) {
            for (int key : list) {
                for (int i = 0; i < numHashFunctions; ++i) {
                    int newIndex = hash(key, i);
                    newTable[newIndex].push_back(key);
                }
            }
        }
        table = std::move(newTable);
        capacity = newCapacity;
    }
    // Обновление количества хеш-функций на основе статистики использования ключей
    void updateNumHashFunctions() {
        float avgBucketSize = (float)size / capacity;
        numHashFunctions = std::ceil(-std::log(1 - avgBucketSize) / std::log(0.5));
        // Используем формулу numHashFunctions = -log(1 - alpha), где alpha - коэффициент заполнения
    }
```

1.3 Пример

```
public:
    DynamicHashTable(int initialCapacity, float loadFactor = 0.7)
        : size(0), capacity(initialCapacity), loadFactorThreshold(loadFactor), numHashFunctions(2) {
        table.resize(capacity);
    }

    void insert(int key) {
        if ((float)(size + 1) / capacity > loadFactorThreshold) {
            rehash();
            updateNumHashFunctions();
        }
        for (int i = 0; i < numHashFunctions; ++i) {
            int index = hash(key, i);
            table[index].push_back(key);
        }
        size++;
    }
```

Применение хеш-таблиц

Хеш-таблицы находят широкое применение в различных областях из-за своей высокой эффективности при операциях вставки, поиска и удаления элементов. Вот некоторые интересные примеры применения хеш-таблиц:

1. **Хранилище данных:** Хеш-таблицы широко используются в базах данных и кеш-хранилищах для быстрого доступа к данным по ключу. Например, хеш-таблицы могут использоваться для кеширования результатов запросов к базе данных или для хранения метаданных файловой системы.
2. **Сетевое программирование:** В компьютерных сетях хеш-таблицы могут использоваться для реализации таблиц маршрутизации, фильтров пакетов или хранения информации о сетевых соединениях.
3. **Ассоциативные массивы:** Хеш-таблицы часто используются для реализации ассоциативных массивов, где данные хранятся в форме пар "ключ-значение". Это позволяет эффективно выполнять операции поиска, вставки и удаления элементов.

Применение хеш-таблиц

4. Криптография: В криптографии хеш-таблицы могут использоваться для реализации хеш-функций, которые преобразуют произвольные входные данные в фиксированные выходные значения (хеш-коды). Хеш-таблицы также могут использоваться для хранения хеш-таблиц словарей или списков подозрительных хеш-значений для обнаружения атак на криптографические алгоритмы.

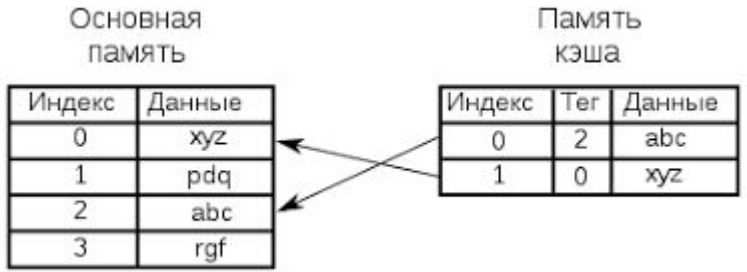
5. Индексирование и поиск: Хеш-таблицы широко применяются для индексирования и поиска текстовых данных, таких как слова в словарях или частота встречаемости слов в текстах. Это позволяет быстро находить и извлекать информацию из больших объемов текстовых данных.

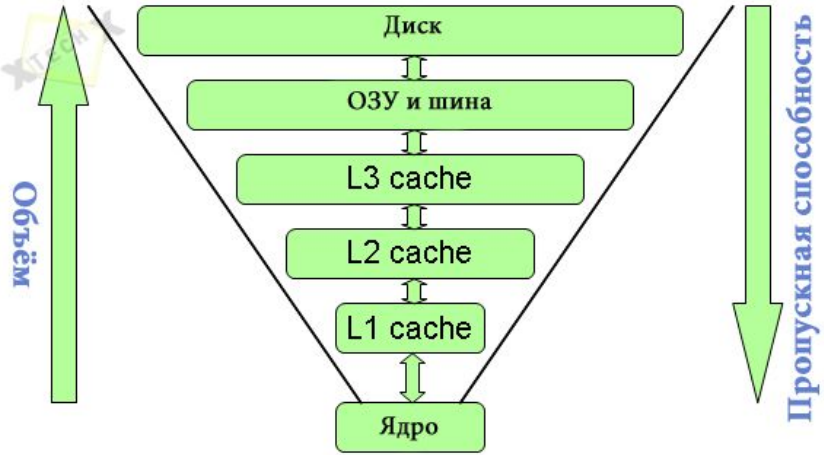
6. Управление памятью: Хеш-таблицы могут использоваться для реализации структур данных, таких как мемоизация результатов функций или управление памятью в языках программирования. Например, виртуальная таблица методов в объектно-ориентированных языках программирования может быть реализована с использованием хеш-таблиц.

7. Решение задач оптимизации: Хеш-таблицы могут быть использованы для решения различных задач оптимизации, таких как нахождение ближайших соседей в пространствах больших размерностей или оптимизация работы с кэш-памятью компьютера.

Понятие КЭШ

Кэш (cache) - это небольшое, но очень быстрое и доступное хранилище данных, которое используется для временного хранения информации, к которой процессор имеет быстрый доступ. Кэш ускоряет доступ к данным, которые часто запрашиваются процессором, за счет сокращения времени, необходимого для обращения к основной памяти (RAM). Кэш состоит из нескольких уровней (обычно три), каждый из которых имеет разную емкость и скорость доступа. Чем ближе кэш к процессору, тем он быстрее и меньше по объему.





КЭШ уровни

Уровень 1 (L1):

- **Расположение:** L1 кэш находится непосредственно внутри процессора (интегрированный в ядро процессора).
- **Емкость:** Обычно имеет небольшой объем, например, от нескольких десятков килобайт до нескольких сотен килобайт.
- **Скорость доступа:** Очень быстрый доступ, обычно от 1 до 3 тактов процессора.
- **Назначение:** Обычно разделяется на два подкэша: L1I (инструкционный) и L1D (данных), хранящие копии инструкций и данных соответственно.

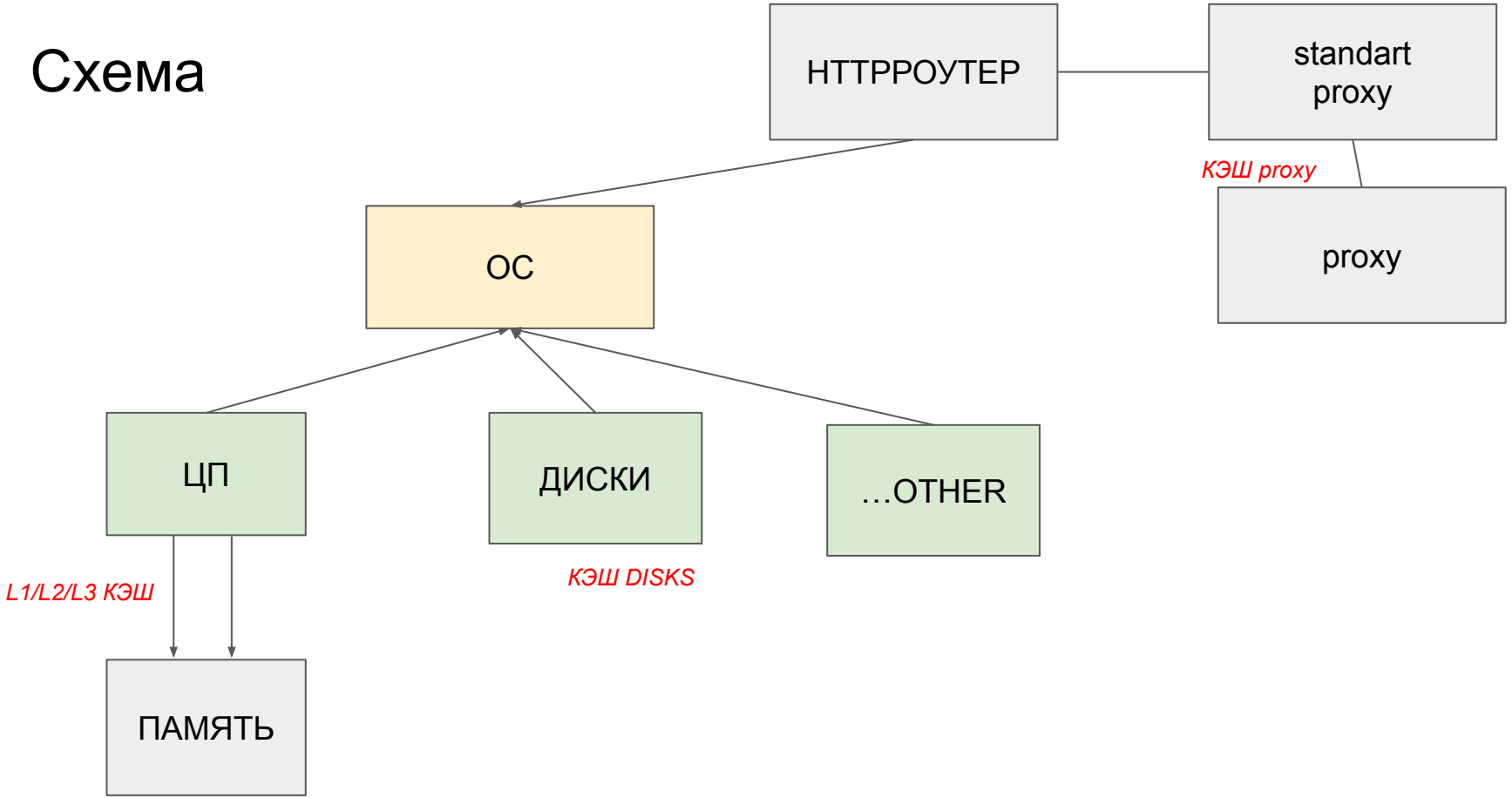
Уровень 2 (L2):

- **Расположение:** L2 кэш обычно находится внутри процессора, но уже не так близко к ядру, как L1.
- **Емкость:** Обычно имеет больший объем, чем L1, например, от нескольких сотен килобайт до нескольких мегабайт.
- **Скорость доступа:** Скорость доступа к L2 кэшу может быть несколько выше, чем к L1, но обычно составляет несколько тактов процессора.
- **Назначение:** Обычно используется для хранения копий данных и инструкций, которые были извлечены из памяти L1 кэша или напрямую из оперативной памяти.

Уровень 3 (L3):

- **Расположение:** L3 кэш находится на более высоком уровне кэш-иерархии и обычно обслуживает все ядра процессора.
- **Емкость:** Обычно имеет еще больший объем, чем L2, может достигать нескольких мегабайт или даже десятков мегабайт в случае многоядерных процессоров.
- **Скорость доступа:** Скорость доступа к L3 кэшу может быть немного выше, чем к оперативной памяти, но обычно медленнее, чем к L1 и L2.
- **Назначение:** Используется для обмена данными между разными ядрами процессора, а также для временного хранения копий данных и инструкций, общих для всех ядер.

Схема


HTTPPOУТЕР
standart proxy
КЭШ proxy
proxy
ОС
ЦП
ДИСКИ
…OTHER
L1/L2/L3 КЭШ
КЭШ DISKS
ПАМЯТЬ