Структуры и алгоритмы обработки данных. Часть 2

Файлы. Кэширование

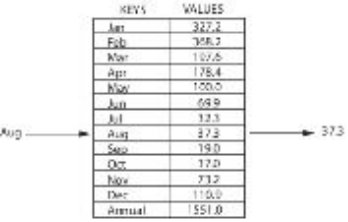
github.com/ivtipm/Data-structures-and-algorithms

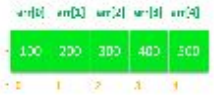
ИВТ ПМ

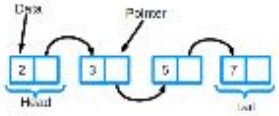
Содержание

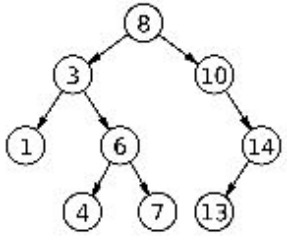
Структуры данных

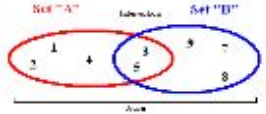
- Линейные
  - индексный доступ
    - Словарь
    - Хэш-таблица
  - прямой доступ
    - Массив
    - Запись
  - последовательный доступ
    - Файл
    - Список
    - Очередь
    - Стек
    - Очередь приоритетов
- Нелинейные
  - Иерархические
    - Дерево
    - Куча
  - Групповые
    - Множество
    - Граф

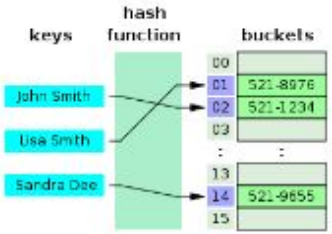


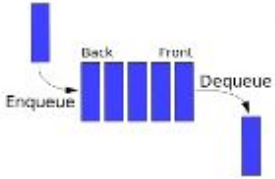


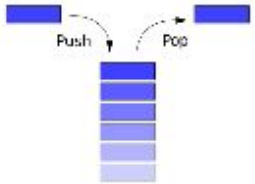












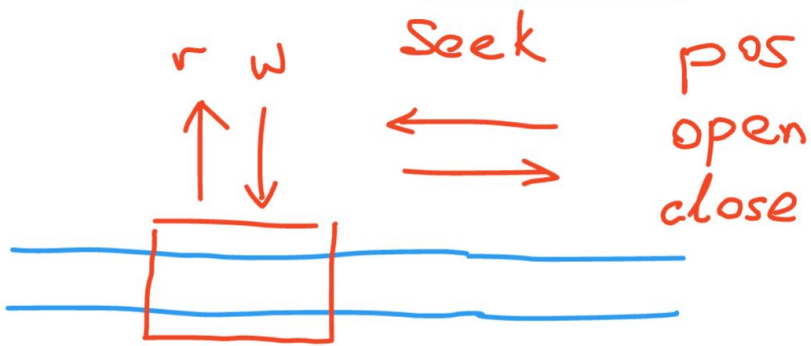
Файлы
Алгоритмы на файлах

Какие преимущества у файлов перед другими линейными структурами данных?

Модель файла

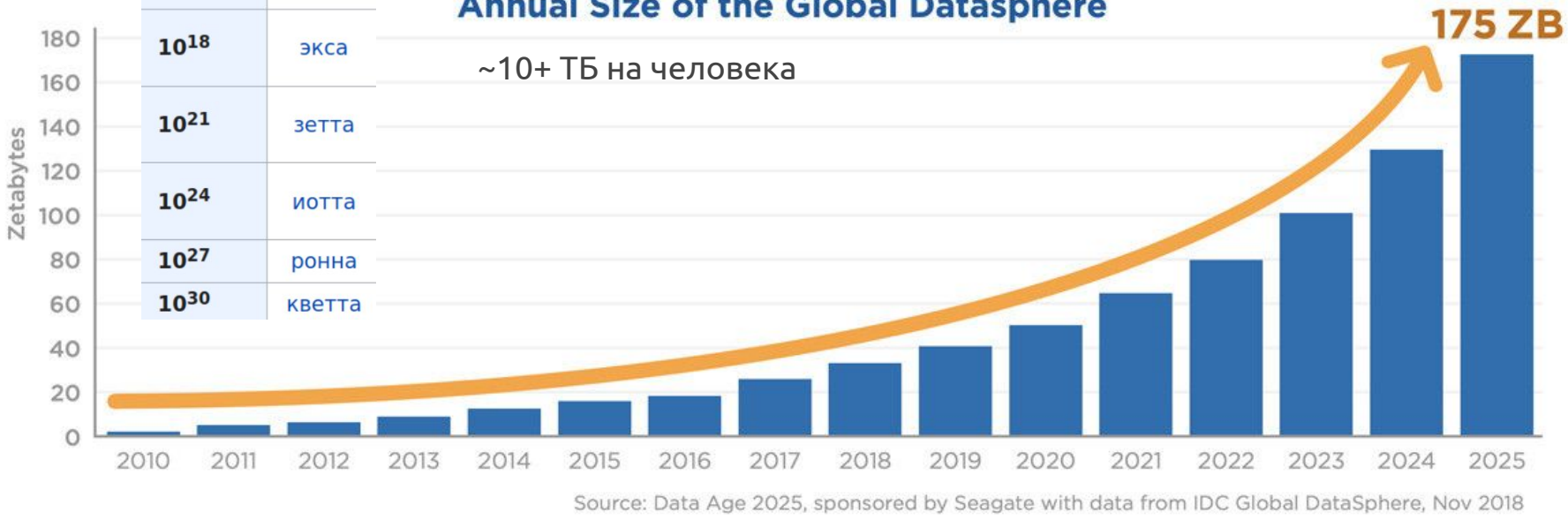
Операции

- open, close — открытие и закрытие (с записью ФБ)
- seek — перемещение курсора (считывающего устройства)
- pos — определение положения курсора
- r — чтение блока
- w — запись блока
- flush — форсированная запись файлового буфера



Big Data

10^{9}	гига
10^{12}	тера
10^{15}	пета
10^{18}	экса
10^{21}	зетта
10^{24}	иотта
10^{27}	ронна
10^{30}	кветта



raw.githubusercontent.com/SupaeroDataScience/OBD/master/readings/idc_data.pdf, upd 2020

Время доступа к разным видам памяти

Тип памяти	Время доступа	Относительная скорость
HDD	5–10 мс	1x
SSD	~0.1 мс	50–100x быстрее HDD
NVMe SSD	~10 мкс	500–1000x быстрее HDD
RAM	10–100 нс	50,000–500,000x быстрее HDD
Кэш CPU	0.5–15 нс	333,000–10,000,000x быстрее HDD

Текстовые файлы	Характеристика	Бинарные файлы
Человекочитаемое представление, данные хранятся в виде текста.	**Формат данных**	Компактное представление данных, ориентировано на машину.
Занимают больше места из-за хранения символов как текста.	**Размер файла**	Меньший размер за счет хранения данных в их "сыром" виде.
Более низкая из-за необходимости конвертации данных.	**Скорость обработки**	Более высокая благодаря прямому доступу к данным.
Легко читаются и редактируются в текстовом редакторе.	**Читаемость**	Не читаемы человеком без декодирования.
Логи, настройки, данные для обмена между системами.	**Применение**	Хранение больших массивов чисел, объектов, мультимедиа.

Чтение файла C++

```
ifstream file("f.txt");
string line;
while (getline(file, line)){
   cout << line << endl;}
file.close()
```

endl выполняет сброс буфера (flush) — данные из буфера выводятся в файл. Это полезно для немедленного обновления, но замедляет запись при работе с большими объемами данных. Альтернатива — использовать символ '\n'

```
ifstream f("data.bin", binary);
int num;
while( f.read(reinterpret_cast<char*>(&num),sizeof(num)) )
    cout << num << endl;
f.close();
```

reinterpret_cast используется для преобразования указателей одного типа в другой, не меняя битовое представление данных

sizeof(num) —> количество байт, которое нужно записать

Запись файла C++

```
ofstream file("f.txt");
file<<"Hello, World!"<< endl;
file.close();
```

```
ofstream file("data.bin", ios::binary)
int num = 42;
file.write(reinterpret_cast<char*>(&num), sizeof(num))
file.close();
```

Текстовые файлы. Зачем

- **Журналы и логи**
- Конфигурационные файлы
- Текстовые данные (документы, исходные коды, …)
- Табличные данные (CSV)

CSV

CSV (Comma-Separated Values) — это простой текстовый формат для хранения табличных данных, где каждая строка представляет запись, а поля отделяются запятыми (или другим разделителем, например, точкой с запятой).

1. **Обмен данными**: CSV используется для передачи структурированных данных между системами (например, из базы данных в Excel).

2. **Простота**: Читается и редактируется текстовыми редакторами.

3. **Совместимость**: Поддерживается большинством языков программирования, библиотек и инструментов анализа данных.

4. **Малый размер**: CSV файлы занимают меньше места, чем форматы вроде XML или JSON.

Пример

```
id,name,age,grade

1,John Doe,20,A

2,Jane Smith,21,B+
```

Первая строка содержит заголовки (id, name, age, grade).

Остальные строки — это данные.

Разделитель полей — запятая.

Поиск в файлах

- Простой последовательный поиск
- Бинарный поиск
- Поиск на основе индексов

Пример: Поиск строки в текстовом файле

```
ifstream file("log.txt");
string line, target = "ERROR";
int line_number = 0;
while (getline(file, line)) {
   line_number++;
   if (line.find(target) != string::npos)
       cout << "Line " << line_number << ": " << line << '\n';
}
file.close();
```

Как зависит количество необходимой для алгоритма оперативной памяти от объёма файла?

Поиск с использованием индексов

Index

Поиск в файлах с использованием индекса

1. **Индекс в оперативной памяти**:
    - Храните компактный индекс (например, хэш-таблицу с *позициями строк*) в памяти.
    - Поиск выполняется через индекс, а сами данные читаются из файла по указанным позициям.
2. **Индекс на диске (внешняя память)**:
    - Если индекс тоже велик, его можно разбить на части и хранить на диске.
    - Например, индексация слов может быть выполнена через B-деревья, которые оптимизированы для дисковой памяти

В этих случаях нет необходимости просматривать весь файл последовательно, чтобы найти искомую строку.

Поиск в файлах с использованием индекса

Использование индексов оправдано для поиска в файлах, которые читаются часто и не помещаются в память

Индексы не подходят если:

- данные в файле слишком разрознены, а операции поиска редки, создание индекса может оказаться избыточным.
- файл читается или обрабатывается последовательно, индексы не дают преимуществ.

Поиск в файлах с использованием индекса

```
ifstream file("large_text.txt");
unordered_map<string, vector<streampos>> index;
string line, word;
streampos position;

// Построение индекса
while (getline(file, line)) {
    position = file.tellg();  // Позиция в файле после чтения строки
    stringstream ss(line);
    while (ss >> word)
        index[word].push_back(position);
}
```

Поиск в файлах с использованием индекса

```
// Поиск слова
string target = "example";
if (index.count(target)) {
    cout << "Word '" << target << "' found at positions:\n";
    for (streampos pos : index[target]) {
        file.seekg(pos);
        getline(file, line);
        cout << "At position " << pos << ": " << line << '\n';
    }
} else {
    cout << "Word not found.\n";
}
```

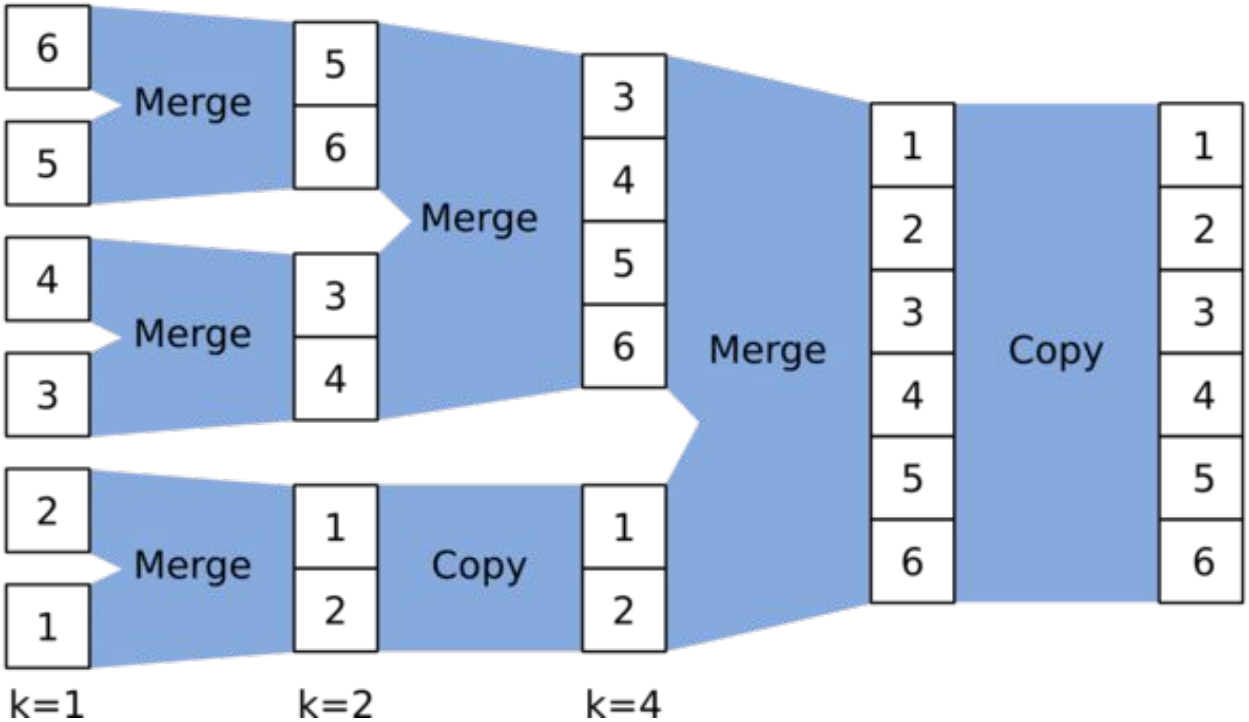
Частичный индекс

Вместо того чтобы индексировать все слова в файле, можно ограничить индекс только ключевыми или важными словами, как это делается в предметных указателях.

Например, можно индексировать только:

- Слова, которые часто встречаются в контексте поиска.
- Термины, связанные с конкретной темой (например, если файл содержит технические или юридические данные, можно индексировать только технические термины или юридические термины).
- Слова, которые имеют особое значение в контексте документа (например, заголовки, ключевые фразы, имена людей и т.д.).

Сортировка


6
5
Merge
5
6
4
3
Merge
3
4
2
1
Merge
1
2
Copy
3
4
5
6
Merge
1
2
3
4
5
6
Copy
1
2
3
4
5
6
k=1
k=2
k=4

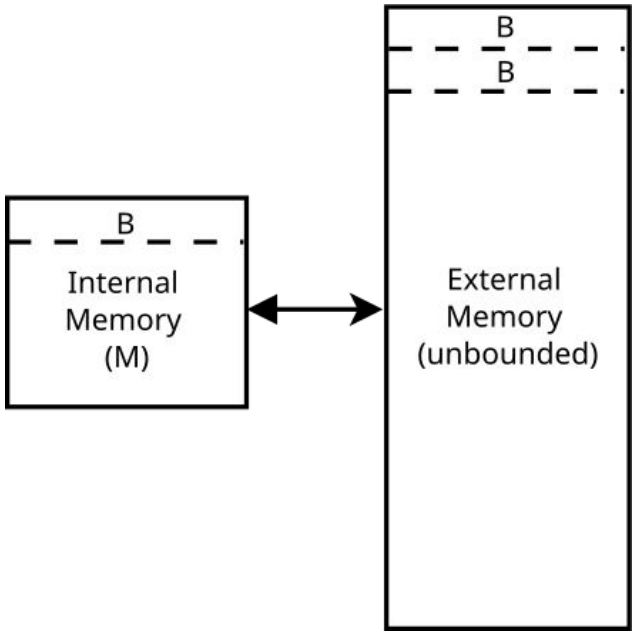
Внешняя сортировка

Для сортировки на внешней памяти (файле) подойдёт любой алгоритм сортировки, который не предполагает хранения сразу всего объёма данных в оперативной памяти.

B — объём одного блока

M — объём внутренней памяти (например RAM).

M может включать в себя несколько (M/B) блоков размера B



Внешняя сортировка. Сложность

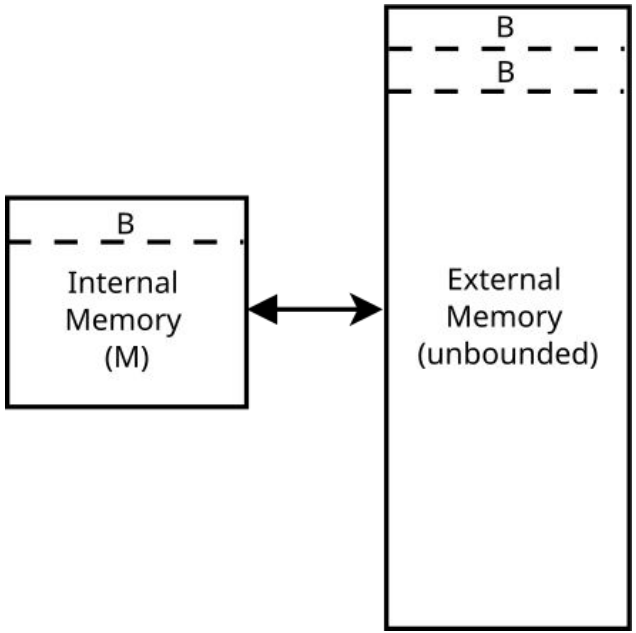
$$O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$$

B — объём одного блока

M — объём внутренней памяти (например RAM).

M может включать в себя несколько (M/B) блоков размера B

N — количество элементов



Внешняя сортировка (External Sort)

1. Разделение на отсортированные части:
   a. Читать данные из файла блоками (которые помещаются в оперативную память)
   b. Сортировать каждый блок в памяти.
   c. Записать отсортированные блоки во временные файлы.

2. Слияние отсортированных частей:
   a. Открыть все временные файлы для чтения.
   b. Создать выходной файл для записи результата.
   c. Сравнить первые элементы всех файлов, выбрать наименьший (наибольший) элемент и записать его в выходной файл.
   d. Повторять процесс, пока все элементы не будут отсортированы.

3. Итеративное слияние:
   a. Если после слияния осталось более одного файла, повторяем процесс слияния.
   b. Каждая итерация уменьшает количество файлов вдвое.
   c. Продолжаем, пока не останется один полностью отсортированный файл

Разделение входного файла на отсортированные блоки

```
void createInitialRuns(const string& input_file, size_t run_size, size_t num_ways) {
   ifstream input(input_file);
   vector<int> buffer(run_size);

   for (size_t i = 0; i < num_ways; i++) {
       if (input.eof()) break;
       // Чтение блока данных
       size_t j;
       for (j = 0; j < run_size && !input.eof(); j++)  input >> buffer[j];
       // Сортировка блока
       sort(buffer.begin(), buffer.begin() + j);
       // Запись отсортированного блока во временный файл
       string output_file = "temp_" + to_string(i);
       ofstream output(output_file);
       for (size_t k = 0; k < j; k++)   output << buffer[k] << " ";
       output.close();
   }
   input.close();   }
```

Слияние отсортированных блоков

```
void mergeFiles(const string& output_file, size_t num_ways) {
// Открытие файлов
   vector<ifstream> input_files(num_ways);
   for (int i = 0; i < num_ways; i++)   input_files[i].open("temp_" + to_string(i));

   ofstream output(output_file);
   priority_queue< pair<int, int>, vector<pair<int, int>>, greater<>> min_q;

// Инициализация очереди приоритетов
   for (size_t i = 0; i < num_ways; i++) {
       int el; if (input_files[i] >> el) min_q.push({el, i});    }

// Слияние
   while (!min_q.empty()) {
       auto [el, file_index] = min_q.top();      min_q.pop();
       output << el << " ";
       int next_el;
       if (input_files[file_index] >> next_el) min_q.push({next_el, file_index}); }
// Закрытие файлов
// Удаление временных файлов
```

Слияние отсортированных блоков

```
void mergeFiles(const string& output_file, size_t num_ways) {
    // Открытие файлов
    /* ... */

    // Инициализация очереди приоритетов
    /* ... */

    // Слияние
    /* ... */

    // Закрытие файлов
    for (auto& file : input_files)  file.close();   }
    output.close();

    // Удаление временных файлов
    for (int i = 0; i < num_ways; i++)   remove(("temp_" + to_string(i)).c_str());
}
```

Основная функция внешней сортировки

```
void externalSort(const string& input_file, const string& output_file,
                                                        size_t num_ways){
    createInitialRuns(input_file, BLOCK_SIZE, num_ways);
    mergeFiles(output_file, num_ways);
}



int main() {
    string input_file = "input.txt";
    string output_file = "output.txt";
    int num_ways = 10; // Количество временных файлов

    externalSort(input_file, output_file, num_ways);

    cout << "Сортировка завершена. Результат в файле " << output_file << endl;
    return 0;
}
```

Кэширование

Кэширование

Кэширование — процесс хранения временных данных в быстрой памяти (кэше) для ускорения доступа к ним при повторных запросах.

Зачем?
Сократить время доступа к часто используемым данным, увеличить скорость выполнения программ и снизить нагрузку.

Например

Веб-кэширование (Web Caching)

Кэширование в базе данных (Database Caching)

Кэширование в системах управления памятью (Memory Management Systems)

Кэширование в компьютерных сетях (Network Caching)

Кэширование вычислений

Кэш в браузере хранит изображения и страницы, чтобы при повторном визите не загружать их с сервера заново.

Процессорный кэш хранит копии часто используемых инструкций и данных из оперативной памяти.

Аппаратный и программный кэш

Аппаратный

- Процессорный кэш (L1, L2, L3)
  Ускоряет доступ к данным и инструкциям, которые находятся в оперативной памяти.
  Пример: Процессор сохраняет текущие инструкции и данные в L1, самый быстрый, но небольшой кэш.

- Дисковый кэш
  Ускоряет запись и чтение с жёсткого диска.
  Пример: Буфер на SSD, который записывает данные в память быстрее, чем на сам накопитель.

Программный кэш.

- Кэш приложений
  Пример: Веб-сервер Nginx использует кэширование запросов для ускорения работы сайта.

- Кэш баз данных
  СУБД, например PostgreSQL, кэшируют результаты запросов.

- Кэш результатов вычислений:
  Пример: Функция с мемоизацией сохраняет результаты для избежания повторных вычислений.

Кэширование

Промах кэша (Cache Miss) — ситуация, когда данные отсутствуют в кэше, требуется обращение к более медленной памяти или выполнение программы (подпрограммы).

Попадание в кэш (Cache Hit) — данные присутствуют в кэше, доступ ускорен.

Политика кэширования (Caching Policy) — алгоритм, определяющий, какие данные сохранять в кэше и когда их удалять.

Время доступа к кэшу (Cache Latency) — Время, необходимое для доступа к данным в кэше.

Кэш-ключ — Уникальный идентификатор данных в кэше.
Для кэширования результата API-запроса ключом может быть URL и параметры запроса.

Хэш-функции для кэширования — используются для преобразования данных в компактные ключи.
Например MD5 или SHA для создания уникального идентификатора запроса.

Срок жизни данных (TTL, Time-to-Live) — время, в течение которого данные считаются актуальными.

Кэширование цен на товары в интернет-магазине с TTL = 10 минут.

Обновление кэша (cache invalidation) — механизм удаления или замены устаревших данных в кэше.

При изменении профиля пользователя в базе данных кэш профиля должен быть обновлён.

Основные алгоритмы кэширования

- LRU (Least Recently Used)
- LFU (Least Frequently Used)
- FIFO (First In, First Out)
- Алгоритмы адаптивного кэширования

LRU (Least Recently Used) — вытеснение давно неиспользуемых

Удаляет из кэша те данные, к которым не обращались дольше всего.

Преимущества:

- Простой и эффективный способ управления кэшом.
- Хорошо подходит для приложений с предсказуемыми паттернами доступа.

Недостатки:

- Требует дополнительной памяти для хранения меток времени или порядка доступа.

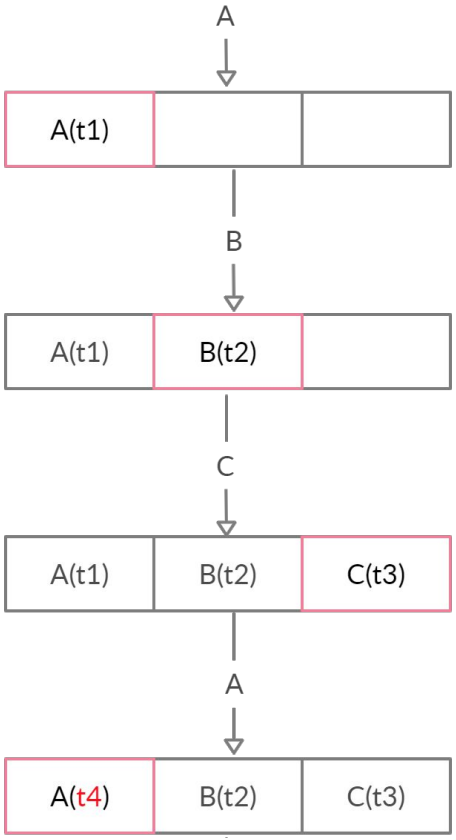
Ситуации и примеры применения:

- Часто используется в операционных системах для управления страницами памяти.

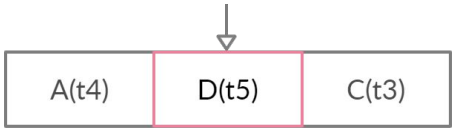
LRU (Least Recently Used)

Размер кэша — 3 элемента

tx — метка времени добавления.
На рис время считается с 1
Чем меньше x, тем старее элемент.



Куда вставить новый элемент?



Кэширование вычислений в Python

```python
from functools import lru_cache

def fibonacci(n):
    if n < 2: return n
    return fibonacci(n-1) + fibonacci(n-2)

@lru_cache(maxsize=None)
def fibonacci_cached(n):
    if n < 2:
        return n
    return fibonacci_cached(n-1) + fibonacci_cached(n-2)

# Измерение времени без кэширования
start = time.time()
fibonacci(30)
end = time.time()
print(f"Время без кэширования: {end - start:.5f} секунд")
```

Кэширование вычислений в Python

```python
from functools import lru_cache

def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)


@lru_cache(maxsize=None)
def fibonacci_cached(n):
    if n < 2:
        return n
    return fibonacci_cached(n-1) + fibonacci_cached(n-2)
```

LFU (Least Frequently Used)

Удаляет из кэша те данные, к которым обращаются реже всего.

Преимущества:

- Оптимален для приложений, где важна частота доступа к данным.

Недостатки:

- Может занимать много времени для обновления частоты доступа.
- Может сохранять устаревшие данные, если новые данные временно востребованы чаще.

Ситуации и примеры применения:

- Применяется в системах, где часто используются одни и те же данные, например, в веб-кэшах.

FIFO (First In, First Out)

Удаляет из кэша самые старые данные, вне зависимости от частоты их использования.

Преимущества:

- Простая и быстрая реализация.
- Не требует отслеживания меток времени или частоты доступа.

Недостатки:

- Может быть менее эффективным в случаях, когда старые данные все еще часто используются.

Ситуации и примеры применения:

- Подходит для систем с постоянным потоком новых данных, например, сетевые буферы.

Алгоритмы адаптивного кэширования

Комбинация различных подходов (например, LRU и LFU) для адаптации к изменяющимся паттернам доступа.

Преимущества:

- Более гибкие и эффективные в сложных сценариях.
- Улучшают производительность в различных условиях.

Недостатки:

- Более сложная реализация.
- Требует тщательной настройки и мониторинга.

Ситуации и примеры применения:

- Используются в высокопроизводительных системах, требующих адаптации к изменяющимся условиям, таких как базы данных и распределенные системы.

Могут быть адаптированы под конкретные задачи и условия, что позволяет выбрать наиболее подходящий вариант для оптимизации производительности системы

См. также – Redis – популярная программа для кэширования