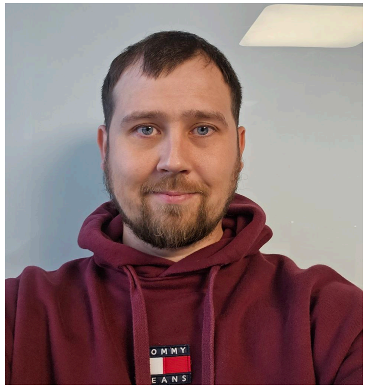
АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Лекция 1

Введение в курс. Массивы.

Наша команда



Алексей Крымов
VK

Дмитрий Корепанов
ABBYY

Дмитрий Глушенков
VK

Позвольте представиться

Крымов Алексей

Руководитель группы разработки

Окончил МГТУ им. Баумана в 2009 г.

Работаю в VK с 2010 г.

- Поиск
- Маруся
- Заметки https://notes.mail.ru/
- Биллинг в Почте/Облаке

Зачем нужно знать алгоритмы?

Алгоритмы нужно знать, чтобы:

- правильно использовать библиотечные АиСД
- писать свои эффективные реализации АиСД
- понимать ограничения текущей реализации программы
- уметь планировать рост парка железа при росте кол-ва данных/нагрузки
- работать в хорошей компании

Структура курса

3 модуля

- 2 лекции
- 3 семинара
- 1 рубежный контроль

Максимум можно набрать 100 баллов.

За РК можно получить 0-8 баллов.

За задачи 25-26 баллов.

В каждом модуле есть обязательные задачи.

Оценки:

Отлично – 84-100 баллов

Хорошо – 67-83 балла

Удовлетворительно – 50-66 баллов

Чему научимся?

- Программировать и понимать основные АиСД
- Анализировать алгоритмы
- Тестировать алгоритмы
- Выбирать правильные библиотечные реализации в зависимости от задачи
- Основные АиСД из STL перестанут быть чёрным ящиком

План занятий

1. Базовые алгоритмы и структуры данных.

- *Лекция 1. Массивы. Базовые структуры данных. Динамическое программирование.*
- Семинар 1. Бинарный поиск. Списки, стек, очередь, дек. Тестирование.
- *Лекция 2. Сортировки. Порядковые статистики.*
- Семинар 2. Динамическая куча. Сортировки.
- Семинар 3. Сортировки, порядковые статистики. Числовые сортировки.
- Рубежный контроль 1.

План занятий

2. Хеш-таблицы. Деревья.

- *Лекция 3. Хеш-таблицы.*
- Семинар 4. Хеш-таблицы.
- *Лекция 4. Деревья.*
- Семинар 5. Двоичные деревья поиска. B-деревья. АВЛ деревья.
- Семинар 6. Коды Хаффмана.
- Рубежный контроль 2.

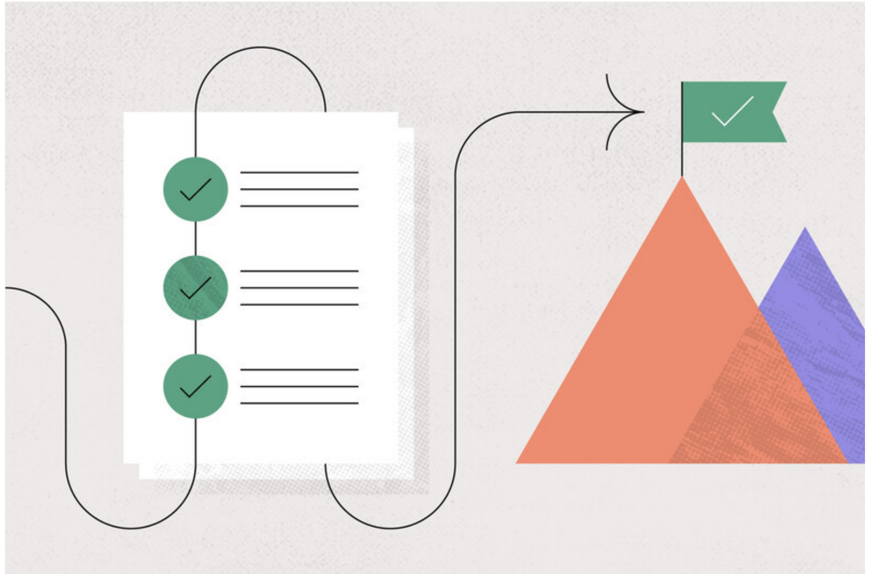
План занятий

3. Алгоритмы на графах.

- *Лекция 5. Обходы графов. Поиск циклов, поиск компонент связности, топологическая сортировка.*
- Семинар 7. Реализация графа. Обходы графов.
- *Лекция 6. Кратчайшие пути. Остовные деревья.*
- Семинар 8. Алгоритм Дейкстры.
- Семинар 9. Остовные деревья.
- Рубежный контроль 3.

План лекции 1 «Введение в курс. Элементарные алгоритмы»

- Понятие алгоритма и структуры данных.
- Понятие вычислительной сложности. O-нотация.
- Проверка числа на простоту.
- Быстрое возведение числа в целую степень (за log(n)).
- Массивы. Однопроходные алгоритмы. Бинарный поиск.
- Динамический массив.
- Списки.
- Стек, очередь, дек.
- Динамическое программирование.
- Кеш процессора



Определения

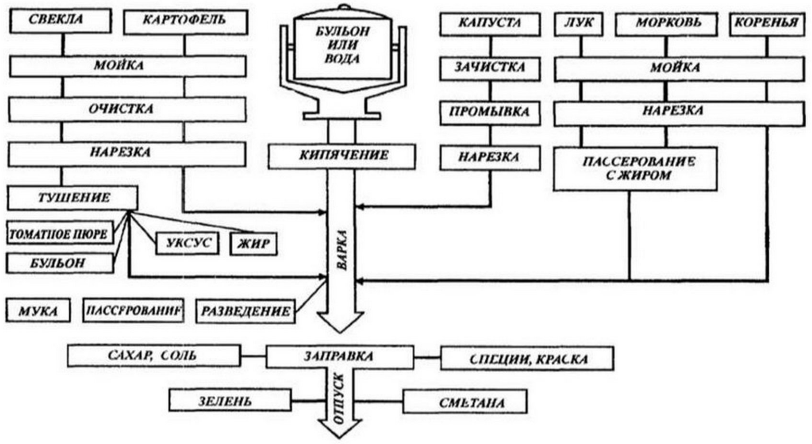
Алгоритм — это формально описанная вычислительная процедура, получающая исходные данные (input), называемые также входом алгоритма или его аргументом, и выдающая результат вычисления на выход (output).

```
output Функция( input )
{
    процедура;
}
```

Алгоритм определяет функцию (отображение) $F : X \to Y$.

X – множество исходных данных, Y – множество значений.

Технологическая схема приготовления борща

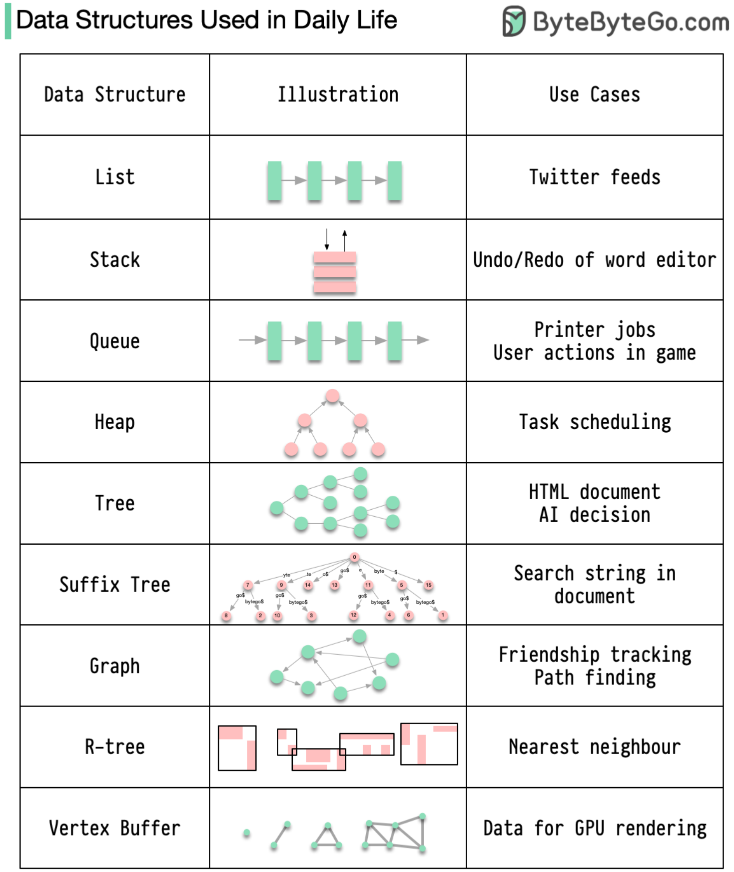


Определения

Структура данных — программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных.

Типичные операции:

- добавление данных,
- изменение данных,
- удаление данных,
- поиск данных.


Data Structures Used in Daily Life

ByteByteGo.com

Data Structure	Illustration	Use Cases
List		Twitter feeds
Stack		Undo/Redo of word editor
Queue		Printer jobs User actions in game
Heap		Task scheduling
Tree		HTML document AI decision
Suffix Tree		Search string in document
Graph		Friendship tracking Path finding
R-tree		Nearest neighbour
Vertex Buffer		Data for GPU rendering

Анализ алгоритмов

Анализ алгоритмов делается для:

- Сравнения алгоритмов между собой
- Для приблизительной оценки производительности программы в новой среде
- Для выявления практической применимости алгоритма

Анализ алгоритмов

Эффективность алгоритма определяется:

- Временем работы,
- Объемом дополнительно используемой памяти,
- Другими характеристики.
  Например, количеством операций сравнения или количеством обращений к диску.

Анализ алгоритмов

Часто исходные данные характеризуются натуральным числом **n**.

Тогда время работы алгоритма – **T(n)**.

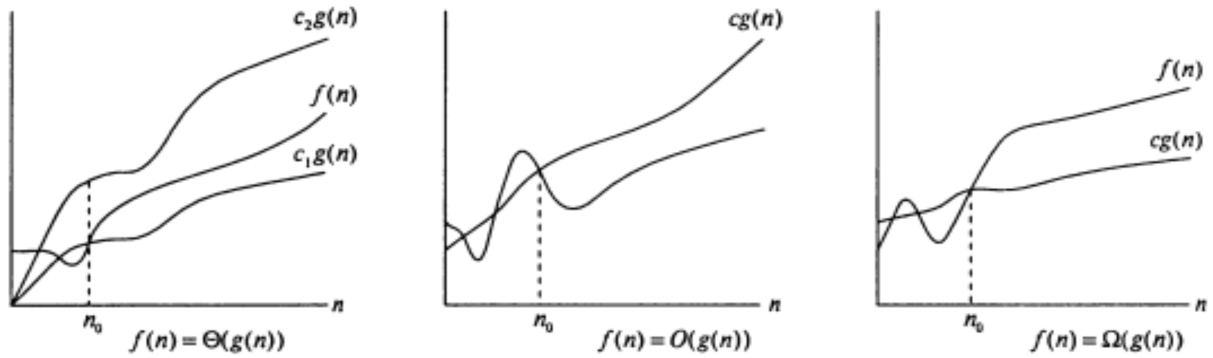
Объем доп. памяти – **M(n)**.

Пример 1. Сортировка массива. Важен размер исходного массива – **n**.

Исходные данные могут характеризоваться несколькими числами.

Пример 2. Поиск всех вхождений строки-шаблона T длины **k** в строку S длины n. В этой задаче время работы алгоритма **T(n, k)** может зависеть от двух чисел **n** и **k**.

Асимптотические обозначения



Θ, O и Ω – обозначения асимптотического поведения времени работы алгоритма (или объема памяти)

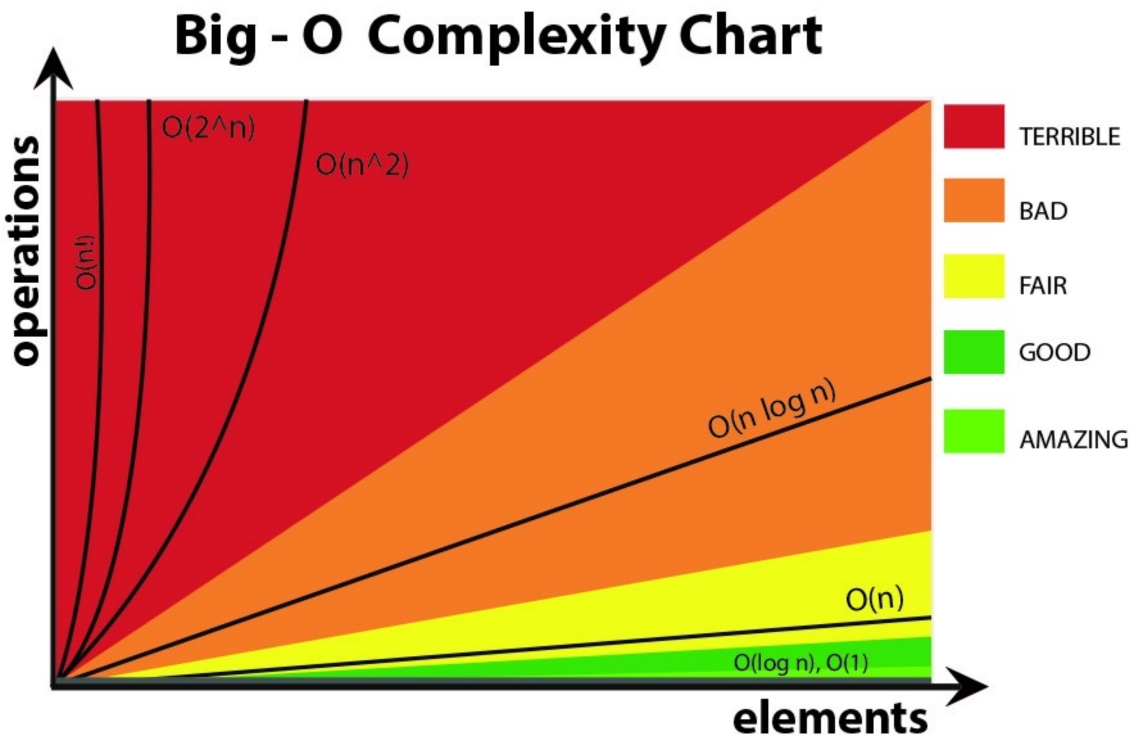
$$f(n) \in \Omega\big(g(n)\big) = \{\exists c > 0, n_0 > 0 : 0 \le cg(n) \le f(n), \forall n > n_0\}$$
$$f(n) \in \Theta\big(g(n)\big) = \{\exists c_1 > 0, c_2 > 0, n_0 > 0 : 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall n > n_0\}$$
$$f(n) \in \mathrm{O}\big(g(n)\big) = \{\exists c > 0, n_0 > 0 : 0 \le f(n) \le cg(n), \forall n > n_0\}$$

Вместо записи «$T(n) \in \Theta(g(n))$» часто используют запись «$T(n) = \Theta(g(n))$».

Асимптотические обозначения

Примеры

- $T(n) = O(n)$. Линейное время. Разг. «за линию».
- $T(n) = O(n^2)$. Квадратичное время. Разг. «за квадрат».
- $T(n) = O(\log n)$. Логарифм.
- $T(n) = O(n \log n)$.
- $T(n) = O(e^n)$.



Проверка числа на простоту

Задача

Проверить, является ли заданное натуральное число n простым.

Можем быстро определять, делится ли одно натуральное число (n) на другое (k), проверив остаток от деления:

```
n % k == 0
```

Будем перебирать все числа от 1 до $\sqrt{\boldsymbol{n}}$, проверяя, делит ли какое-нибудь из них *n*.

Проверка числа на простоту

```
bool is_prime( int n ) {
    if( n == 1 ) {
        return false;
    }
    for( int i = 2; i * i <= n; ++i ) {
        if( n % i == 0 ) {
            return false;
        }
    }
    return true;
}
```

Проверка числа на простоту

Время работы $\boldsymbol{T(n) = O(\sqrt{n})}$.

Объем доп. памяти $\boldsymbol{M(n) = O(1)}$.

- Существует алгоритм, проверяющий число на простоту за полиномиальное время (от log) – Тест Агравала – Каяла – Саксены (2002 год).
  Время работы его улучшенной версии (2011год) $\boldsymbol{T(n) = O(\log^3 n)}$.

Быстрое возведение в степень

Задача.

Дано число a и неотрицательное целое число n.
Найти $\boldsymbol{a^n}$.

Тривиальный алгоритм: перемножить n-1 раз число a: $\boldsymbol{a \cdot a \cdot \dots \cdot a}$

Время работы тривиального алгоритма $\boldsymbol{T(n) = O(n)}$.

- Воспользуемся тем, что $\boldsymbol{a^{2^k} = \left(\left(a^2\right)^2\right)^{\dots 2}}$ $(\boldsymbol{k}\ \text{раз})$.

  Если $\boldsymbol{n = 2^{k_1} + 2^{k_2} + \dots + 2^{k_s}, k_1, k_2, \dots, k_s}$ – различны, то $\boldsymbol{a^n = a^{2^{k_1}} a^{2^{k_2}} \cdots a^{2^{k_s}}}$.

Быстрое возведение в степень

- Как получить разложение $n = 2^{k_1} + 2^{k_2} + \dots + 2^{k_s}$, где $k_1, k_2, \dots, k_s$ – различны?

  `n = 10011010` – в двоичной системе счисления.

  `     7  43 1` – степени двоек в разложении n.

Начнем с младших степеней.

`result = 1, aInPowerOf2 = a.`

`n = 10011`~~`010`~~ – пусть пройдено 3 шага алгоритма.

Если следующий бит == 1, то домножим `result` на `aInPowerOf2` = a^{2^3}.

Вне зависимости от бита возводим `aInPowerOf2` в квадрат.

Быстрое возведение в степень

Как извлекать очередной бит из n?

Будем на каждом шаге сдвигать n вправо на 1 бит.

Т.е. делить n пополам.

```
n = n >> 1;
```

Тогда интересующий бит будет всегда располагаться последним.

Последний бит соответствует четности числа,

достаточно проверить остаток от деления на 2:

```
n % 2 == 1
```

Или с помощью битовых операций:

```
n & 1 == 1;
```

Быстрое возведение в степень

```
double power(double a, int n) {
    double result = 1;
    // Для хранения результата.
    double aInPowerOf2 = a;
    // Текущее значение ((a^2)^2…)^2
    while (n > 0) {
        // Добавляем нужную степень двойки к результату,
        // если она есть в разложении n.
        if ((n & 1) == 1) {
                result *= aInPowerOf2;
        }
        aInPowerOf2 *= aInPowerOf2;
        // Не стоит использовать битовый сдвиг со знаковыми числами.
        n = n >> 1; // Можно писать n /= 2.
    }
    return result;
}
```

Быстрое возведение в степень

Количество итераций цикла = степень двойки, не превышающая n, т.е. $\boldsymbol{log(n)}$.

Каждая итерация цикла требует ограниченное количество операций, $\boldsymbol{O(1)}$.

Доп. память не используется.

$$\boldsymbol{T(n) = O(log\, n)}$$
$$\boldsymbol{M(n) = O(1)}$$

Массивы

Определение 1

Массив – набор однотипных элементов, расположенных в памяти непосредственно друг за другом, доступ к которым осуществляется по индексам.

Традиционно индексирование элементов массивов начинают с 0.

Определение 2

Размерность массива – количество индексов, необходимое для однозначного доступа к элементу массива.

Одномерный массив целых чисел

20	34	11	563	23	-1	2	0	-33	7
0	1	2	3	4	5	6	7	8	9

Массивы. Линейный поиск

Задача 1. Проверить, есть ли заданный элемент в массиве.

Решение. Последовательно проверяем все элементы массива, пока не найдем заданный элемент, либо пока не закончится массив.

Время работы в худшем случае $T(n) = O(n)$, где n – количество элементов в массиве.

```
bool has_element(const double *arr, int count, double element) {
    for (int i = 0; i < count; ++i) {
        if (arr[i] == element) {
            // Нашли.
            return true;
        }
    }
    return false;
}
```

Массивы. Бинарный поиск.

Определение

Упорядоченный по возрастанию массив – массив A, элементы которого сравнимы, и для любых индексов k и l, $k < l$:

$$A[k] \leq A[l].$$

Упорядоченный по убыванию массив определяется аналогично.

-40	-12	0	1	2	6	22	54	343	711
0	1	2	3	4	5	6	7	8	9

Массивы. Бинарный поиск

Задача (Бинарный поиск = Двоичный поиск)

Проверить, есть ли заданный элемент в упорядоченном массиве. Если он есть, вернуть позицию его первого вхождения. Если его нет, вернуть -1.

Решение

Шаг. Сравниваем элемент в середине массива (медиану) с заданным элементом. Выбираем нужную половинку массива в зависимости результата сравнения.
Повторяем этот шаг до тех пор, пока размер массива не уменьшится до 1.

-40	**-12**	**0**	**1**	**2**	**6**	**22**	**54**	**343**	**711**
0	1	2	3	4	5	6	7	8	9

Массивы. Бинарный поиск

```
// Бинарный поиск без рекурсии. Ищем первое вхождение.
int binary_search(const double *arr, int count, double element) {
   int first = 0;
   int last = count; // Элемент в last не учитывается.
   while (first < last) {
      int mid = (first + last) / 2;
      if (arr[mid] < element) {
            first = mid + 1;
      } else { // В случае равенства arr[mid] останется справа.
            last = mid;
      }
   }
   // Все элементы слева от first строго меньше искомого.
   return (first == count || arr[first] != element) ? -1 : first;
}
```

Массивы. Бинарный поиск

-40	-12	0	1	2	6	22	54	343	711
0	1	2	3	4	5	6	7	8	9

Ищем element = -12

mid = (0 + 10) / 2 = 5

```
int binary_search(const double *arr, int
count, double element) {
    int first = 0;
    int last = count;
    // first = 0; last = 10;
    while (first < last) {
        int mid = (first + last) / 2;
        if (arr[mid] < element) {
                first = mid + 1;
        } else {// last = 5;

                last = mid;
        }
    }
    return (first == count || arr[first] !=
    element) ? -1 : first;
}
```

Массивы. Бинарный поиск

-40	-12	0	1	2	6	22	54	343	711
0	1	2	3	4	5	6	7	8	9

-40	-12	0	1	2
0	1	2	3	4

mid = (0 + 5) / 2 = 2

```
int binary_search(const double *arr, int
count, double element) {
    int first = 0;
    int last = count;
    // first = 0; last = 5;
    while (first < last) {
        int mid = (first + last) / 2;
        if (arr[mid] < element) {
            first = mid + 1;
        } else {// last = 2;

            last = mid;
        }
    }
    return (first == count || arr[first] !=
    element) ? -1 : first;
}
```

Массивы. Бинарный поиск

-40	-12	0	1	2	6	22	54	343	711
0	1	2	3	4	5	6	7	8	9

-40	-12	0	1	2
0	1	2	3	4

-40	-12
0	1

mid = (0 + 2) / 2 = 1

```
int binary_search(const double *arr, int
count, double element) {
    int first = 0;
    int last = count;
    // first = 0; last = 2;
    while (first < last) {
        int mid = (first + last) / 2;
        if (arr[mid] < element) {
                first = mid + 1;
        } else {// last = 1;

                last = mid;
        }
    }
    return (first == count || arr[first] !=
    element) ? -1 : first;
}
```

Массивы. Бинарный поиск

-40	**-12**	**0**	**1**	**2**	**6**	**22**	**54**	**343**	**711**
0	1	2	3	4	5	6	7	8	9

-40	**-12**	**0**	**1**	**2**
0	1	2	3	4

-40	**-12**
0	1

mid = (0 + 1) / 2 = 0

```
int binary_search(const double *arr, int
count, double element) {
    int first = 0;
    int last = count;
    // first = 0; last = 1;
    while (first < last) {
        int mid = (first + last) / 2;
        if (arr[mid] < element) {
            first = mid + 1; //first = 1;
        } else {

            last = mid;
        }
    }
    return (first == count || arr[first] !=
    element) ? -1 : first;
}
```

Массивы. Бинарный поиск

-40	-12	0	1	2	6	22	54	343	711
0	1	2	3	4	5	6	7	8	9

-40	-12	0	1	2
0	1	2	3	4

-40	-12
0	1

```
int binary_search(const double *arr, int count, double element) {
    int first = 0;
    int last = count;
    // first = 0; last = 1;
    while (first < last) {
        int mid = (first + last) / 2;
        if (arr[mid] < element) {
                first = mid + 1;
        } else {
                last = mid;
        }
    }
    // first = 1; last = 1;
    return (first == count || arr[first] != element) ? -1 : first;
}
```

После выхода из цикла мы либо ушли за границу массива (элемент не найден), либо на позиции first искомый элемент, либо первый элемент больше искомого.

Массивы. Бинарный поиск

```
// Возвращает позицию вставки элемента на отрезке [first, last).
// Равные элементы располагаются после. (std::lower_bound)
int lower_bound(const double *arr, int count, double element) {
    int first = 0;
    int last = count; // Элемент в last не учитывается.
    while (first < last) {
        int mid = (first + last) / 2;
        if (arr[mid] < element) {
                first = mid + 1;
        } else { // В случае равенства arr[mid] останется справа.
                last = mid;
        }
    }
    return first;
}
```

Массивы. Бинарный поиск

```
// Возвращает позицию вставки элемента на отрезке [first, last).
// Равные элементы располагаются после. (std::lower_bound)
int lower_bound(const double *arr, int count, double element) {
    int first = 0;
    int last = count; // Элемент в last не учитывается.
    while (first < last) {
        int mid = (first + last) / 2;
        if (arr[mid] < element) {
                first = mid + 1;
        } else { // В случае равенства arr[mid] останется справа.
                last = mid;
        }
    }
    return first;
}
```

Что вернет **lower_bound** для массива ниже, если element = 20?

10 10 10 20 20 20 30 30

Массивы. Бинарный поиск

Время работы

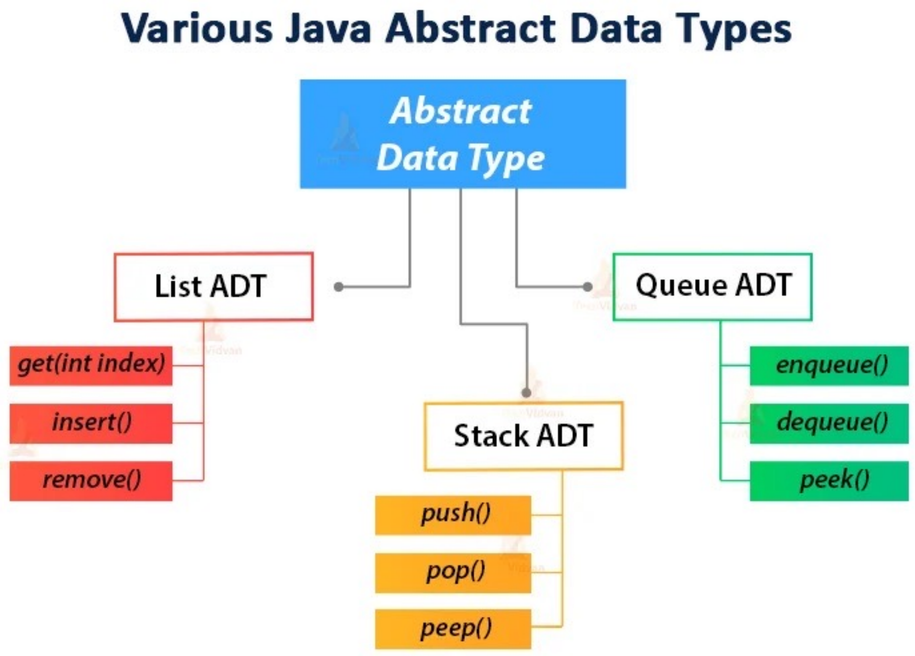
$T(n) = O(\log n)$, где n – количество элементов в массиве.

Объем дополнительной памяти

$M(n) = O(1)$.

Абстрактные типы данных и структуры данных

Абстрактный тип данных (АТД) — это тип данных, который предоставляет для работы с элементами этого типа определённый набор функций, а также возможность создавать элементы этого типа при помощи специальных функций.



Вся внутренняя структура такого типа спрятана – в этом и заключается суть абстракции.

АТД = Интерфейс

СД «Динамический массив»

АТД «Динамический массив» — интерфейс с операциями

- Добавление элемента в конец массива «Add» (или PushBack),
- Доступ к элементу массива по индексу за $O(1)$ «GetAt» (или оператор []).

Структура данных Динамический массив содержит внутренний массив фиксированной длины для хранения элементов. Внутренний массив называется **буфером**.

Помнит текущее количество добавленных элементов.

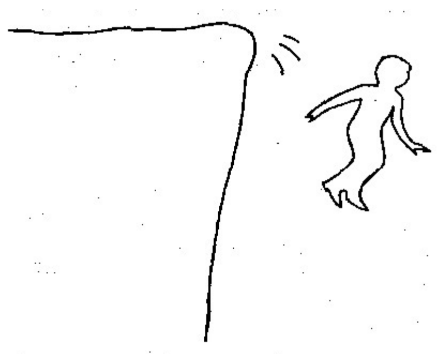
Размер буфера имеет некоторый запас для возможности добавления новых элементов.

Пример. Буфер размера 14 заполнен 10 элементами.

Т	**е**	**х**	**н**	**о**	**п**	**а**	**р**	**к**	**!**				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

СД «Динамический массив»

Буфер может закончиться…



Если буфер закончился, то
при добавлении нового элемента:

- выделим новый буфер,
  больший исходного;
- скопируем содержимое старого буфера в новый;
- добавим новый элемент.

Т	е	х	н	о	п	а
0	1	2	3	4	5	6

Т	е	х	н	о	п	а	р						
0	1	2	3	4	5	6	7	8	9	10	11	12	13

СД «Динамический массив»

```
const int DefaultInitialSize = 2;

// Класс «Динамический массив».
class Array {
public:
    Array() : buffer(0), buffer_size(0), real_size(0) {}
    ~Array() { delete[] buffer; }

    // Размер.
    int size() const { return real_size; }
    // Доступ по индексу.
    double get_at(int index) const;
    double operator[](int index) const { return get_at(index); }
    double &operator[](int index);

    // Добавление нового элемента.
    void push_back(double element);

private:
    double *buffer; // Буфер.
    int buffer_size; // Размер буфера.
    int real_size; // Количество элементов в массиве.

    void grow();
};
```

СД «Динамический массив»

```
// Доступ к элементу.
double Array::get_at(int index) const {
    assert(index >= 0 && index < real_size && buffer != 0);
    return buffer[index];
}
// Увеличение буфера.
void Array::grow() {
    int new_buffer_size = std::max(buffer_size * 2, DefaultInitialSize);
    double *new_buffer = new double[new_buffer_size];
    for (int i = 0; i < real_size; ++i) // Лучше использовать std::copy
        new_buffer[i] = buffer[i];
    delete[] buffer;
    buffer = new_buffer;
    buffer_size = new_buffer_size;
}
// Добавление элемента.
void Array::push_back(double element) {
    if (real_size == buffer_size)
        grow();
    assert(real_size < buffer_size && buffer != 0);
    buffer[real_size++] = element;
}
```

СД «Динамический массив»

Как долго работает функция Add добавления элемента?

- В лучшем случае = $O(1)$
- В худшем случае = $O(n)$
- В среднем?

Имеет смысл рассматривать несколько операций добавления и оценить среднее время в контексте последовательности операций.

Подобный анализ называется **амортизационным**.

Амортизационный анализ

Определение (по Кормену)

При **амортизационном анализе** время, требуемое для выполнения последовательности операций над структурой данных, усредняется по всем выполняемым операциям.

Этот анализ можно использовать, например, чтобы показать, что даже если одна из операций последовательности является дорогостоящей, то при усреднении по всей последовательности средняя стоимость операций будет небольшой.

При амортизационном анализе гарантируется **средняя производительность операций в наихудшем случае**.

Амортизационный анализ

Пусть S(n) – время выполнения последовательности всех n операций в наихудшем случае.

Амортизированной стоимостью (временем) AC(n) называется среднее время, приходящееся на одну операцию $S(n)/n$.

Оценим амортизированную стоимость операций Add динамического массива.

СД «Динамический массив»

<u>**Утверждение.**</u> Пусть в реализации функции *grow()* буфер удваивается. Тогда амортизированная стоимость функции *Add* составляет $O(1)$.

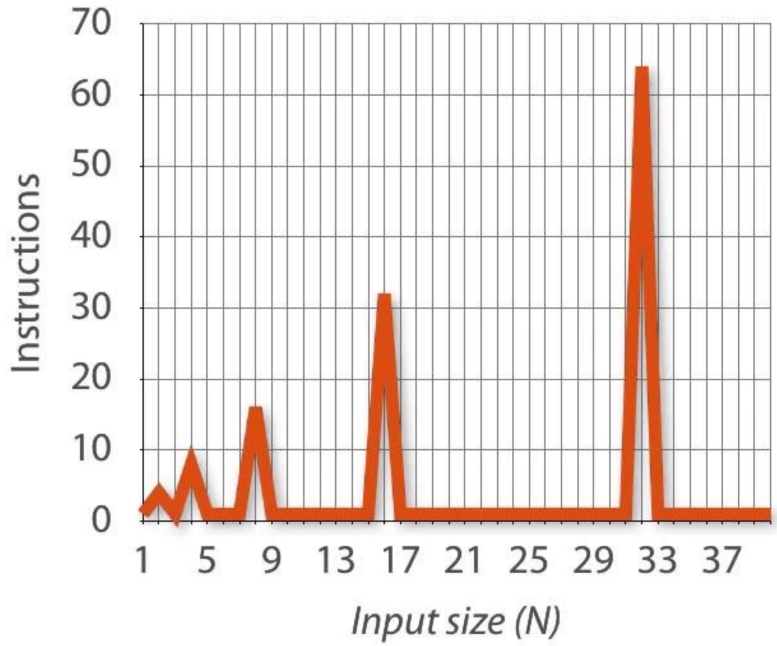
<u>**Доказательство.**</u> Рассмотрим последовательность из n операций *Add*. Обозначим $P(k)$ - время выполнения *Add* в случае, когда *RealSize = k*.

- $P(k) \le c_1 k$, если $k = 2^m$.
- $P(k) \le c_2$, если $k \ne 2^m$.

$S(n) = \sum_{k=0}^{n-1} P(k) \le c_1 \sum_{m:2^m<n} 2^m + c_2 \sum_{k:k\ne 2^m} 1 \le$

$2c_1 n + c_2 n = (2c_1 + c_2)n.$

Амортизированное время $AC(n) = {}^{S(n)}/_{n} \le 2c_1 + c_2 = O(1)$.



Связные списки

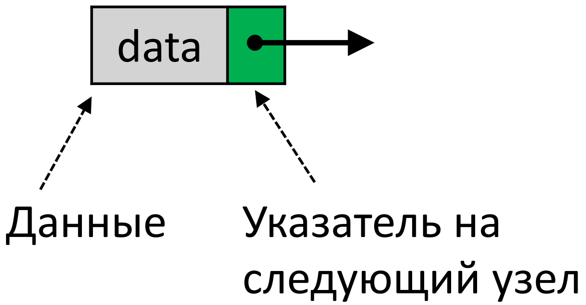
Связный список — динамическая структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка.

Преимущество перед массивом:

- Порядок элементов списка может не совпадать с порядком расположения элементов данных в памяти, а порядок обхода списка всегда явно задаётся его внутренними связями.

Односвязный список

Узел односвязного списка

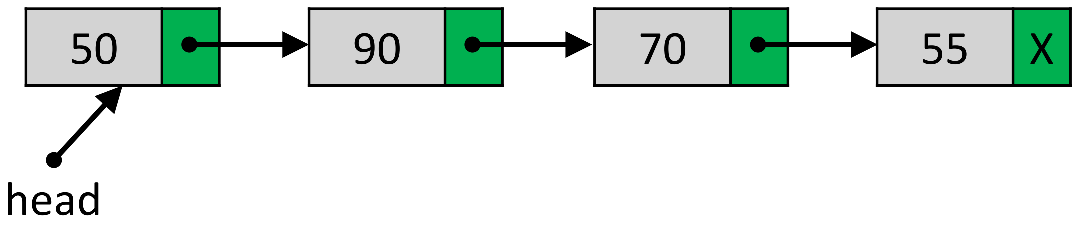



```
struct Node {
   int data;
   Node* next;

   Node() : data( 0 ),
            next( nullptr ) {}
};
```

Ссылка в каждом узле одна.
Указывает на следующий узел в списке.
Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

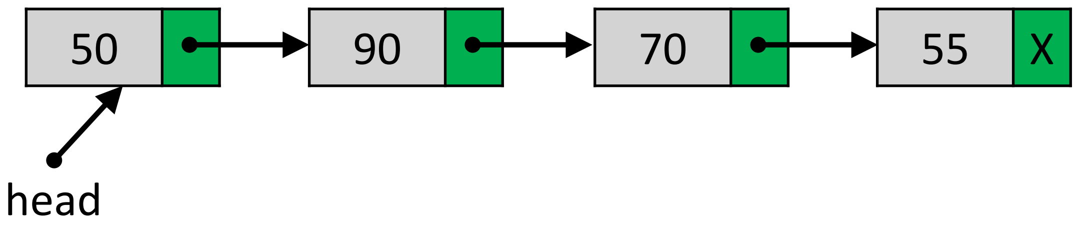
Односвязный список

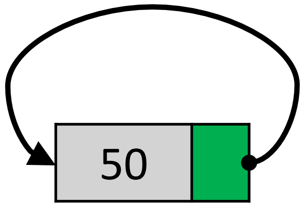
Примеры односвязных списков



Односвязный список

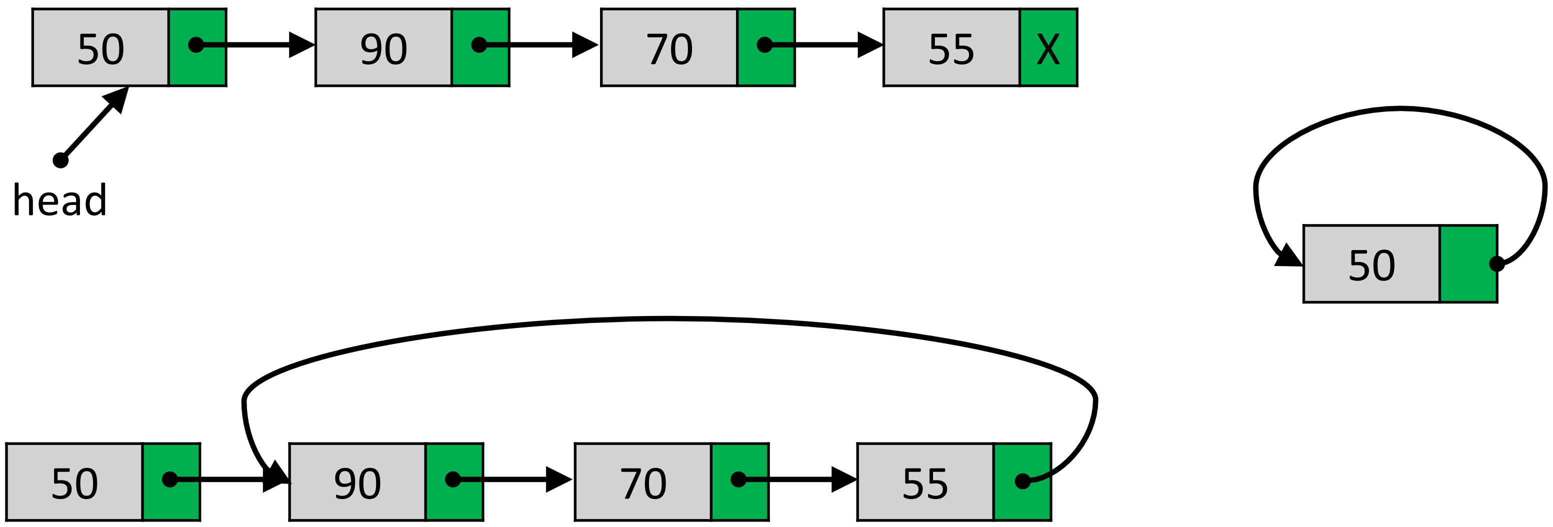
Примеры односвязных списков





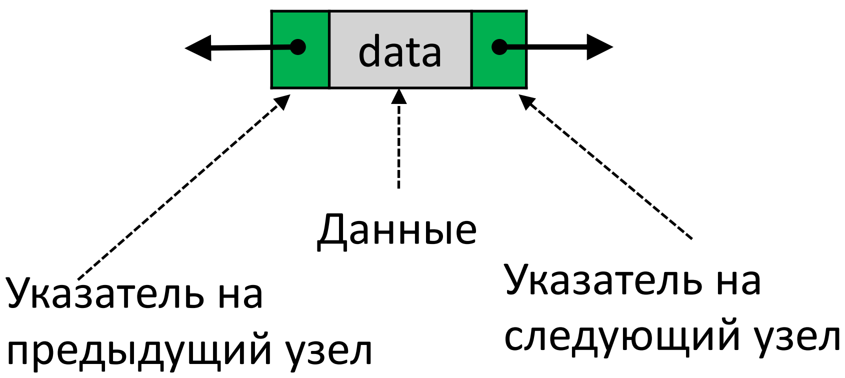
Односвязный список

Примеры односвязных списков

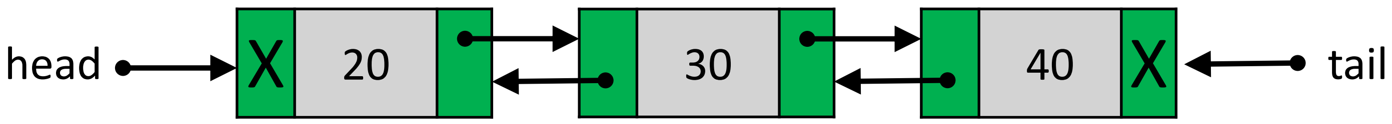


Двусвязный список

Узел двусвязного списка




```
struct Node {
    int data;
    Node* next;
    Node* prev;

    Node() : data( 0 ),
             next( nullptr ),
             prev( nullptr ) {}
};
```



Указатель в каждом узле указывают на предыдущий и на последующий узел в списке.

Связные списки

Операции со списками

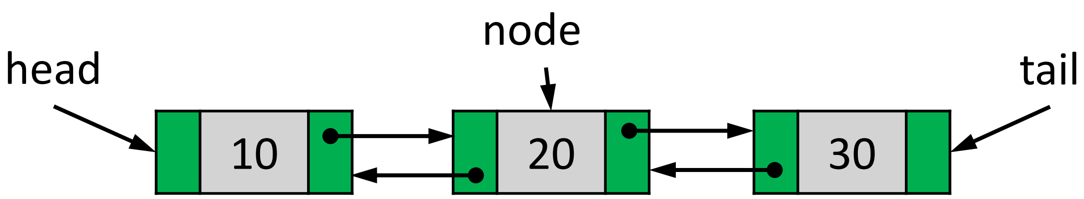
- Поиск элемента,
- Вставка элемента,
- Удаление элемента,
- Объединение списков,

Связные списки. Поиск.

```
// Линейный поиск элемента «a» в списке.
// Возвращает 0, если элемент не найден.
Node* Search( Node* head, int a )
{
   Node* current = head;
   while( current != nullptr ) {
      if( current->data == a )
         return current;
      current = current->next;
   }
   return 0;
}
```

Время работы в худшем случае = **O(n)**, где n – длина списка.

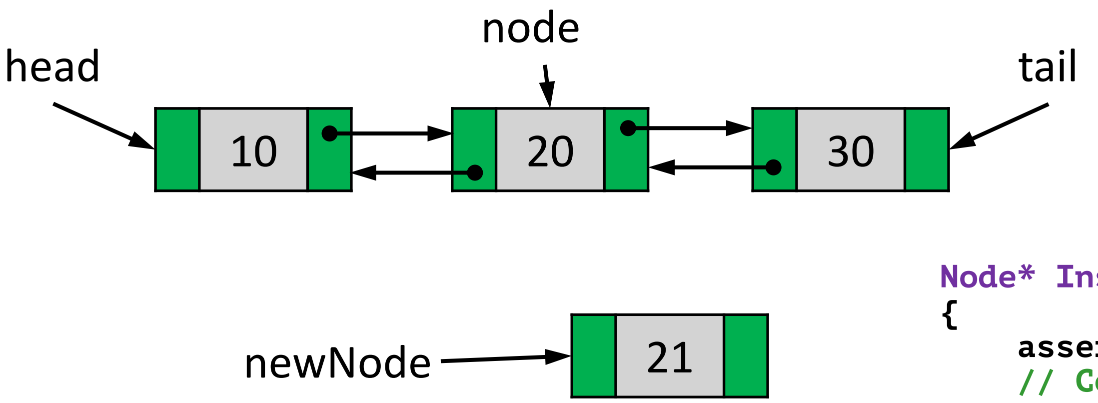
Связные списки. Вставка. Шаг 1.




```
Node* InsertAfter( Node* node, int a )
{
    assert( node != nullptr );
    // Создаем новый элемент.
    Node* newNode = new Node();
    newNode->data = a;
    newNode->next = node->next;
    newNode->prev = node;
    // Обновляем prev следующего
    // элемента, если он есть.
    if( node->next != nullptr ) {
        node->next->prev = newNode;
    }
    // Обновляем Next текущего элемента.
    node->next = newNode;
    return newNode;
}
```

Шаг 1. Проходим по списку и ищем позицию для вставки

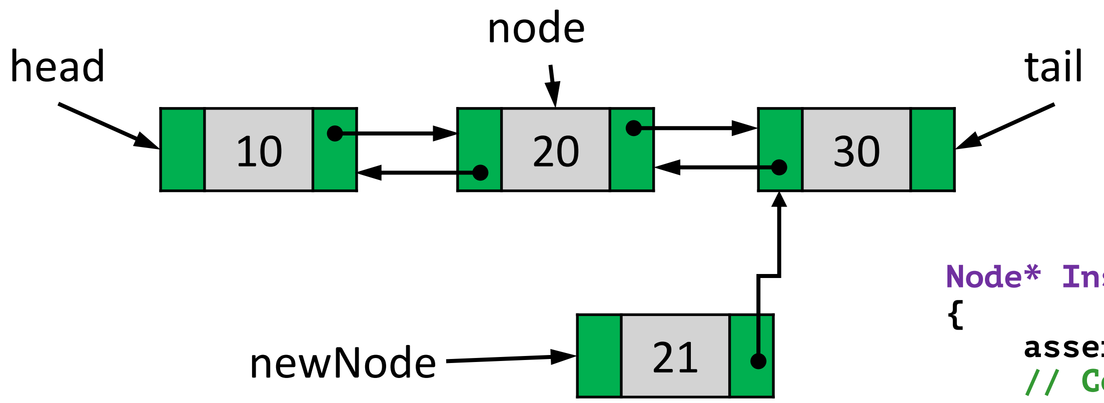
Связные списки. Вставка. Шаг 2.




```
Node* InsertAfter( Node* node, int a )
{
    assert( node != nullptr );
    // Создаем новый элемент.
    Node* newNode = new Node();
    newNode->data = a;
    newNode->next = node->next;
    newNode->prev = node;
    // Обновляем prev следующего
    // элемента, если он есть.
    if( node->next != nullptr ) {
        node->next->prev = newNode;
    }
    // Обновляем Next текущего элемента.
    node->next = newNode;
    return newNode;
}
```

Шаг 2. Создаем новый узел и заполняем у него поле с данными

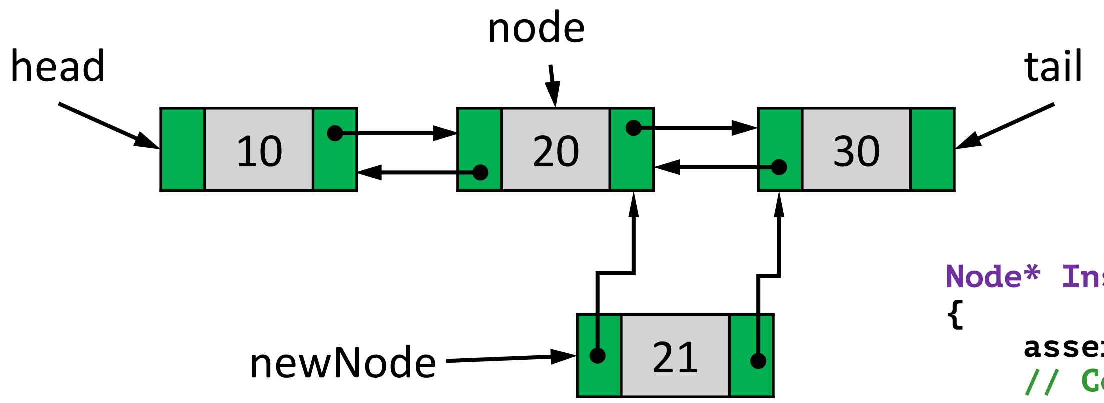
Связные списки. Вставка. Шаг 3.




```
Node* InsertAfter( Node* node, int a )
{
    assert( node != nullptr );
    // Создаем новый элемент.
    Node* newNode = new Node();
    newNode->data = a;
    newNode->next = node->next;
    newNode->prev = node;
    // Обновляем prev следующего
    // элемента, если он есть.
    if( node->next != nullptr ) {
        node->next->prev = newNode;
    }
    // Обновляем Next текущего элемента.
    node->next = newNode;
    return newNode;
}
```

Шаг 3. Заполняем поле next у newNode

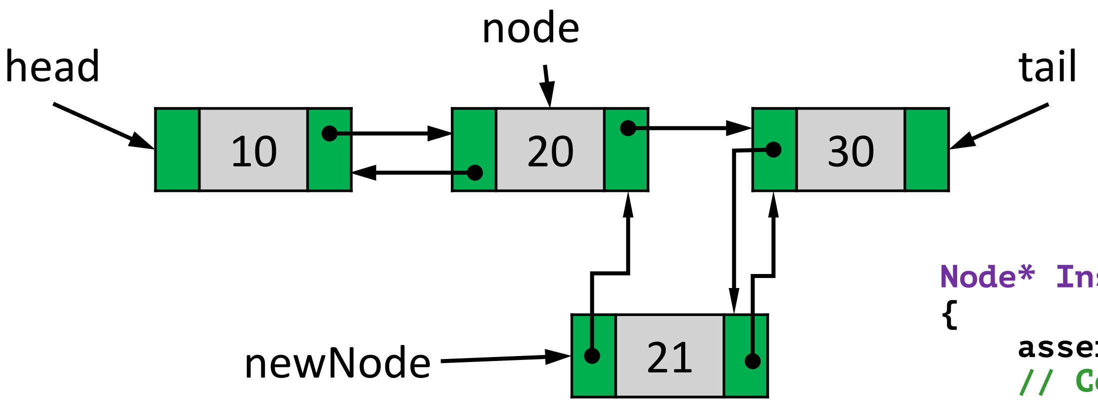
Связные списки. Вставка. Шаг 4.




```
Node* InsertAfter( Node* node, int a )
{
    assert( node != nullptr );
    // Создаем новый элемент.
    Node* newNode = new Node();
    newNode->data = a;
    newNode->next = node->next;
    newNode->prev = node;
    // Обновляем prev следующего
    // элемента, если он есть.
    if( node->next != nullptr ) {
        node->next->prev = newNode;
    }
    // Обновляем Next текущего элемента.
    node->next = newNode;
    return newNode;
}
```

Шаг 4. Заполняем поле prev у newNode

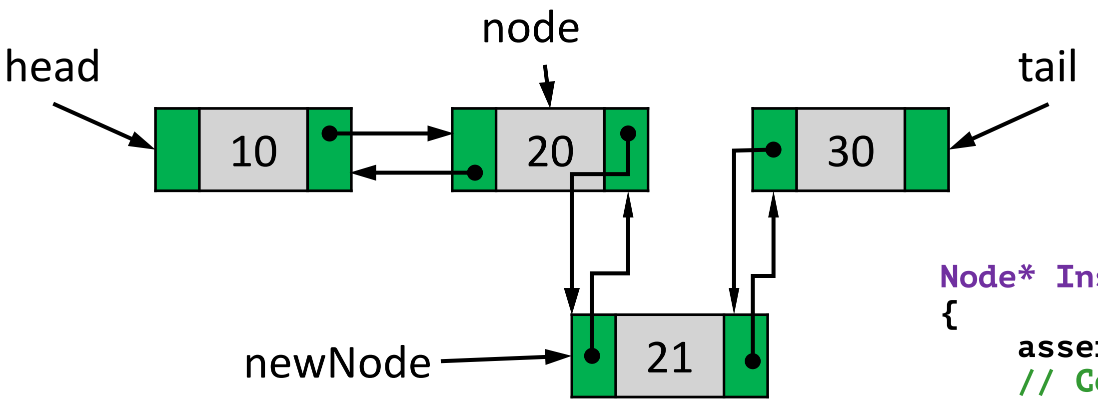
Связные списки. Вставка. Шаг 5.




```
Node* InsertAfter( Node* node, int a )
{
    assert( node != nullptr );
    // Создаем новый элемент.
    Node* newNode = new Node();
    newNode->data = a;
    newNode->next = node->next;
    newNode->prev = node;
    // Обновляем prev следующего
    // элемента, если он есть.
    if( node->next != nullptr ) {
        node->next->prev = newNode;
    }
    // Обновляем Next текущего элемента.
    node->next = newNode;
    return newNode;
}
```

Шаг 5. Заполняем поле prev у следующего узла

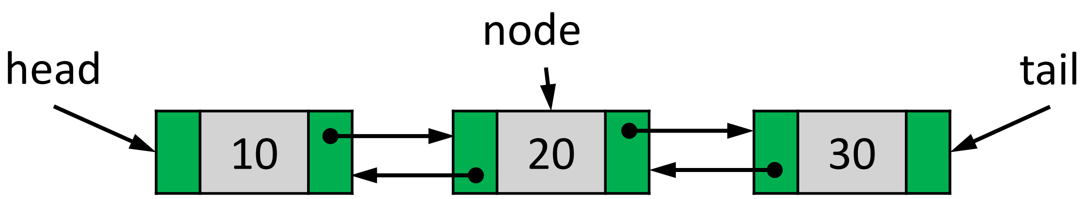
Связные списки. Вставка. Шаг 6.




```
Node* InsertAfter( Node* node, int a )
{
    assert( node != nullptr );
    // Создаем новый элемент.
    Node* newNode = new Node();
    newNode->data = a;
    newNode->next = node->next;
    newNode->prev = node;
    // Обновляем prev следующего
    // элемента, если он есть.
    if( node->next != nullptr ) {
        node->next->prev = newNode;
    }
    // Обновляем Next текущего элемента.
    node->next = newNode;
    return newNode;
}
```

Шаг 6. Заполняем поле Next у текущего узла

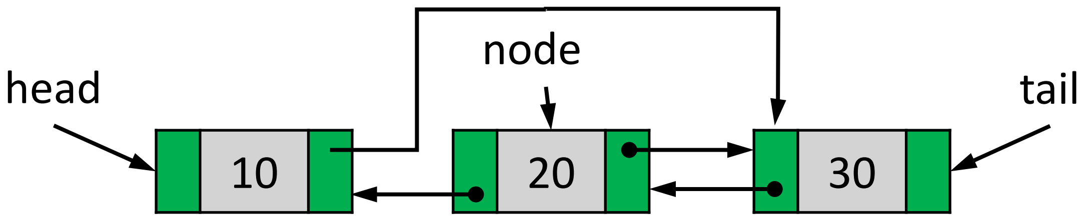
Связные списки. Удаление. Шаг 1.



Шаг 1. Проходим по списку и ищем удаляемый элемент

```
void DeleteAt( Node* node )
{
    assert( node != nullptr );

    // Обновляем next предыдущего
    // элемента, если он есть.
    if( node->prev != nullptr ) {
        node->prev->next = node->next;
    }
    // Обновляем prev следующего
    // элемента, если он есть.
    if( node->next != nullptr ) {
        node->next->prev = node->prev;
    }
    delete node;
}
```

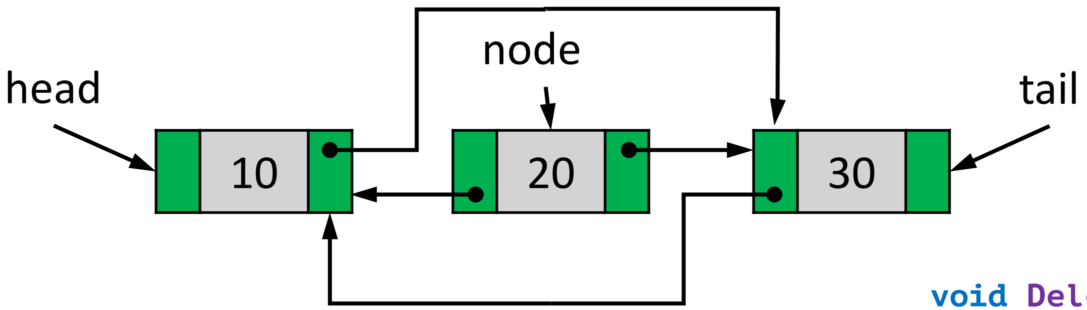
Связные списки. Удаление. Шаг 2.




```
void DeleteAt( Node* node )
{
    assert( node != nullptr );

    // Обновляем next предыдущего
    // элемента, если он есть.
    if( node->prev != nullptr ) {
        node->prev->next = node->next;
    }
    // Обновляем prev следующего
    // элемента, если он есть.
    if( node->next != nullptr ) {
        node->next->prev = node->prev;
    }
    delete node;
}
```

Шаг 2. Обновляем поле next у предыдущего узла

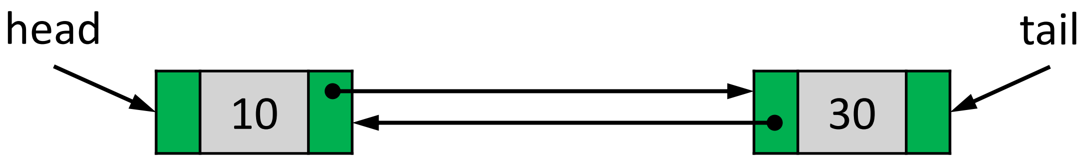
Связные списки. Удаление. Шаг 3.



Шаг 3. Обновляем поле Prev у следующего узла

```
void DeleteAt( Node* node )
{
    assert( node != nullptr );

    // Обновляем next предыдущего
    // элемента, если он есть.
    if( node->prev != nullptr ) {
        node->prev->next = node->next;
    }
    // Обновляем prev следующего
    // элемента, если он есть.
    if( node->next != nullptr ) {
        node->next->prev = node->prev;
    }
    delete node;
}
```

Связные списки. Удаление. Шаг 4.



Шаг 4. Удаляем узел

```
void DeleteAt( Node* node )
{
    assert( node != nullptr );

    // Обновляем next предыдущего
    // элемента, если он есть.
    if( node->prev != nullptr ) {
        node->prev->next = node->next;
    }
    // Обновляем prev следующего
    // элемента, если он есть.
    if( node->next != nullptr ) {
        node->next->prev = node->prev;
    }
    delete node;
}
```

Связные списки. Объединение.

```
// Объединение односвязных списков. К списку 1 подцепляем список 2.
// Возвращает указатель на начало объединенного списка.
Node* Union( Node* head1, Node* head2 ) {
   if( head1 == nullptr ) {
      return head2;
   }
   if( head2 == nullptr ) {
      return head1;
   }
   // Идем в хвост списка 1.
   Node* tail1 = head1;
   for( ; tail1->next != nullptr; tail1 = tail1->next );
   // Обновляем Next хвоста.
   tail1->next = head2;
   return head1;
}
```

Время работы = O(n), где n – длина первого списка.

Сравнение списков с массивами.

Сравнение сложности основных операций

	Массив	Односвязный список
Вставка	$O(n)$	$O(1)$
Добавление (вставка в конец)	$O(1)$	$O(n)/O(1)^*$
Удаление	$O(n)$	$O(1)$
Доступ по индексу	$O(1)$	$O(n)$

* - если храним tail

Сравнение списков с массивами.

Недостатки списков

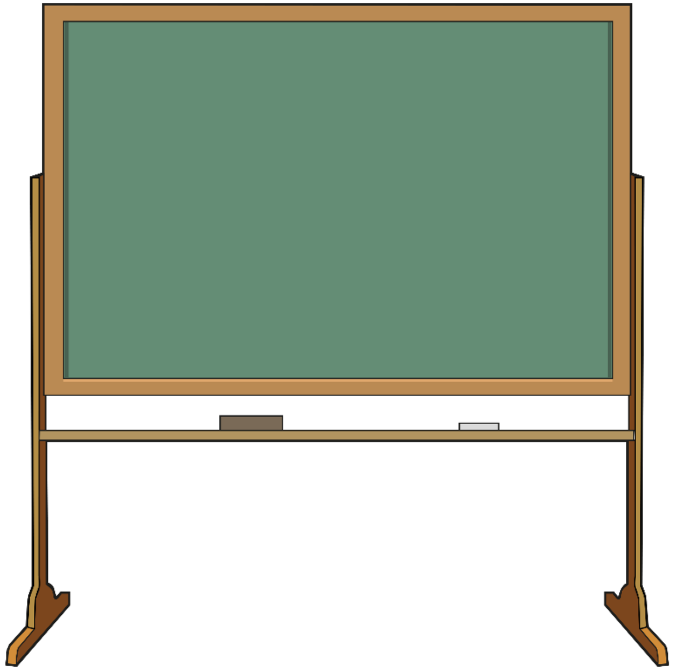
- Нет быстрого доступа по индексу.
- Расходуется память на указатели
- Узлы могут располагаться в памяти разреженно, что не позволяет использовать кэширование процессора.

Преимущества списков

- Быстрая вставка узла.
- Быстрое удаление узла.
- Не тратим доп. память по сравнению с динамическим массивом

Сравнение списков с массивами.

Порисуем?

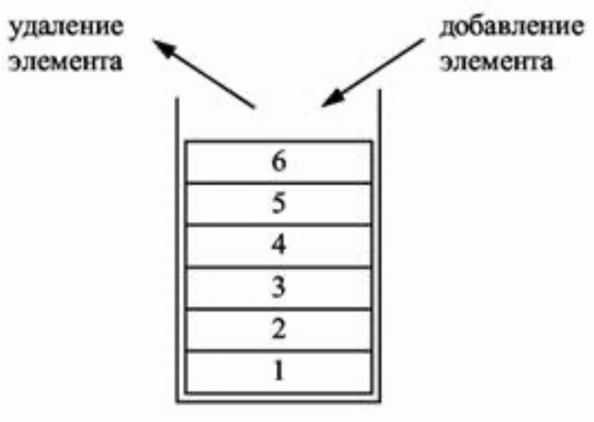


АТД «Стек»

<u>Определение.</u> Стек – абстрактный тип данных (или структура данных), представляющий из себя список элементов, организованный по принципу
LIFO = Last In First Out, «последним пришел, первым вышел».

Операции:

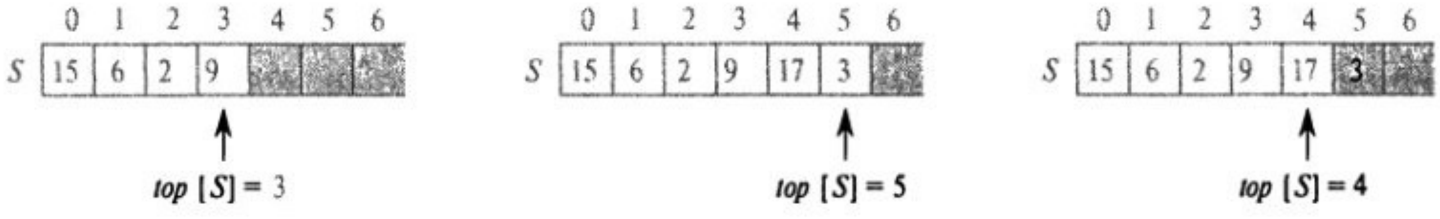
1. Вставка (Push)
2. Извлечение (Pop) – извлечение элемента, добавленного последним.



СД «Стек»

Стек можно реализовать с помощью массива или с помощью списка.

Реализация с помощью массива.



а) б) в)

Храним указатель на массив и текущее количество элементов в стеке.

Можно использовать динамический массив.

СД «Стек»

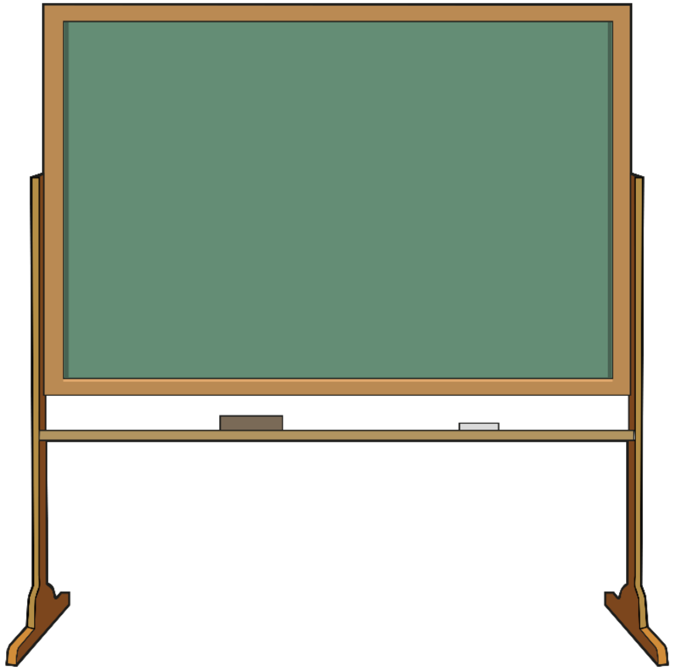
```
// Стек целых чисел, реализованный с помощью массива.
class Stack {
public:
    explicit Stack( int _bufferSize );
    ~Stack();

    // Добавление и извлечение элемента из стека.
    void Push( int a );
    int Pop();

    // Проверка на пустоту.
    bool IsEmpty() const { return top == -1; }

private:
    int* buffer;
    int bufferSize;
    int top; // Индекс верхнего элемента.
};
```

СД «Стек»

```
Stack::Stack( int _bufferSize ) :
    bufferSize( _bufferSize ),
    top( -1 )
{
    buffer = new int[bufferSize]; // Создаем буфер.
}
Stack::~Stack()
{
    delete[] buffer; // Удаляем буфер.
}
// Добавление элемента.
void Stack::Push( int a )
{
    assert( top + 1 < bufferSize );
    buffer[++top] = a;
}
// Извлечение элемента.
int Stack::Pop()
{
    assert( top != -1 );
    return buffer[top--];
}
```

СД «Стек»

Порисуем?

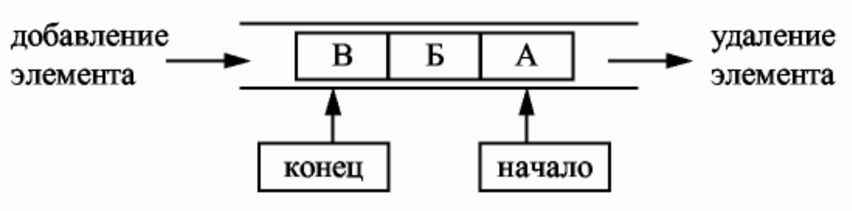


АТД «Очередь»

Определение. Очередь – абстрактный тип данных (или структура данных), представляющий из себя список элементов, организованный по принципу
FIFO = First In First Out, «первым пришел, первым вышел».

Операции:

1. Вставка (Enqueue)
2. Извлечение (Dequeue) – извлечение элемента, добавленного первым.

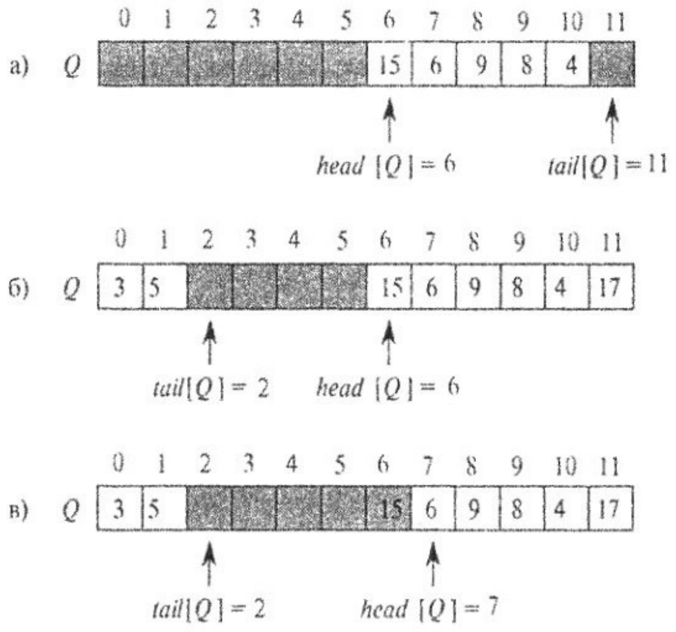


СД «Очередь»

Очередь также как и стек можно реализовать с помощью массива или с помощью списка.

Реализация с помощью массива.

Храним указатель на массив,
текущее начало и конец очереди.



Считаем массив зацикленным.

Можно использовать
динамически растущий буфер.

СД «Очередь»

```
// Очередь целых чисел, реализованная с помощью массива.
class Queue {
public:
    explicit Queue( int size );
    ~Queue() { delete[] buffer; }

    // Добавление и извлечение элемента из очереди.
    void Enqueue( int a );
    int Dequeue();

    // Проверка на пустоту.
    bool IsEmpty() const { return head == tail; }

private:
    int* buffer;
    int bufferSize;
    int head; // Указывает на первый элемент очереди.
    int tail; // Указывает на следующий после последнего.
};
```

СД «Очередь»

```
Queue::Queue( int size ) :
   bufferSize( size ),
   head( 0 ),
   tail( 0 )
{
   buffer = new int[bufferSize]; // Создаем буфер.
}

// Добавление элемента.
void Queue::Enqueue( int a )
{
   assert( ( tail + 1 ) % bufferSize != head );
   buffer[tail] = a;
   tail = ( tail + 1 ) % bufferSize;
}
// Извлечение элемента.
int Queue::Dequeue()
{
   assert( head != tail );
   int result = buffer[head];
   head = ( head + 1 ) % bufferSize;
   return result;
}
```

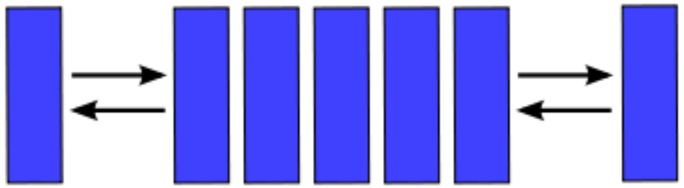
АТД «Дэк»

<u>Определение.</u> Двусвязная очередь – абстрактный тип данных (структура данных), в которой элементы можно добавлять и удалять как в начало, так и в конец, то есть принципами обслуживания являются одновременно FIFO и LIFO.

Операции:

1. Вставка в конец (PushBack),
2. Вставка в начало (PushFront),
3. Извлечение из конца (PopBack),
4. Извлечение из начала (PopFront).

Дек, также как стек или очередь, можно реализовать с помощью массива или с помощью списка.



Динамическое программирование

Динамическое программирование (ДП) – способ решения сложных задач путём разбиения их на более простые подзадачи.

Подход динамического программирования состоит в том, чтобы решить каждую подзадачу только один раз, сократив тем самым количество вычислений.

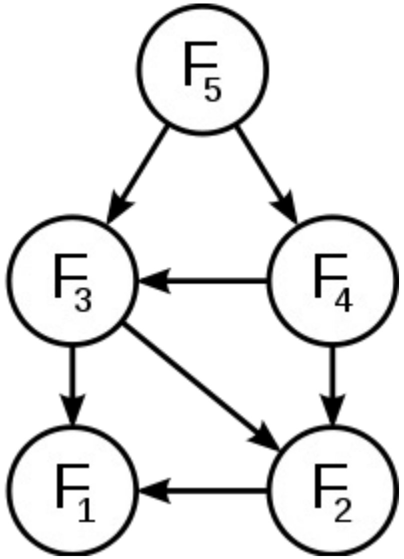
Термин «динамическое программирование» также встречается в теории управления в смысле «динамической оптимизации». Основным методом ДП является сведение общей задачи к ряду более простых экстремальных задач.

Лит.: Беллман Р., Динамическое программирование, пер. с англ., М., 1960.

Динамическое программирование

<u>Пример.</u> Числа Фибоначчи. $F_n = F_{n-1} + F_{n-2}, F_1 = 1, F_0 = 1.$

Можно вычислять рекурсивно:
$F_5 = F_4 + F_3,$
$F_4 = F_3 + F_2,$
$F_3 = F_2 + F_1,$



Многие значения могут вычисляться несколько раз.

Решение – сохранять результаты решения подзадач.
Этот подход называется кэшированием.

Динамическое программирование

<u>Пример.</u> Вычисление рекуррентных функций нескольких аргументов.

$$F(x, y) = 3 \cdot F(x-1, y) - 2 \cdot F^2(x, y-1),$$
$$F(x, 0) = x, F(0, y) = 0.$$

Вычисление $F(x, y)$ сводится к вычислению двух $F(\cdot,\cdot)$ от меньших аргументов.

Есть перекрывающиеся подзадачи.

$F(x-1, y-1)$ в рекурсивном решении вычисляется дважды.

$F(x-2, y-1)$ в рекурсивном решении вычисляется три раза.

$F(x-n, y-m)$ в рекурсивном решении вычисляется $C_{n+m}^{n} = \frac{(n+m)!}{m!}$ раз.

Снова будем использовать кэширование – сохранять результаты.

Вычисления будем выполнять от меньших аргументов к большим.

Динамическое программирование

Таблица значений рекуррентной функции

x

y

	0	**1**	**2**	**3**	**4**
0	0	1	2	3	4
1	0				
2	0				
3	0				
4	0				

$$F(x,y) = 3 \cdot F(x-1,y) - 2 \cdot F^2(x,y-1),$$
$$F(x,0) = x, F(0,y) = 0.$$

Динамическое программирование

Таблица значений рекуррентной функции

$\mathrm{F}(x, y-1)$

x

y

	0	1	2	3	4
0	0	1	2	3	4
1	0	-2			
2	0				
3	0				
4	0				

$\mathrm{F}(x-1, y)$

$$F(x, y) = 3 \cdot F(x-1, y) - 2 \cdot F^2(x, y-1),$$
$$F(x, 0) = x, F(0, y) = 0.$$

Динамическое программирование

Таблица значений рекуррентной функции

x

y

	0	**1**	**2**	**3**	**4**
0	0	1	2	3	4
1	0	-2			
2	0	-8			
3	0				
4	0				

$$F(x, y) = 3 \cdot F(x-1, y) - 2 \cdot F^2(x, y-1),$$
$$F(x, 0) = x, F(0, y) = 0.$$

Динамическое программирование

Таблица значений рекуррентной функции

x

y

	0	1	2	3	4
0	0	1	2	3	4
1	0	-2			
2	0	-8			
3	0	-128			
4	0				

$$F(x,y) = 3 \cdot F(x-1,y) - 2 \cdot F^2(x,y-1),$$
$$F(x,0) = x, F(0,y) = 0.$$

Динамическое программирование

Таблица значений рекуррентной функции

x

y

	0	1	2	3	4
0	0	1	2	3	4
1	0	-2			
2	0	-8			
3	0	-128			
4	0	-32768			

$$F(x, y) = 3 \cdot F(x-1, y) - 2 \cdot F^2(x, y-1),$$
$$F(x, 0) = x, F(0, y) = 0.$$

Динамическое программирование

Таблица значений рекуррентной функции

x

y

	0	1	2	3	4
0	0	1	2	3	4
1	0	-2			
2	0	-8			
3	0	-128			
4	0	-32768			

$$F(x, y) = 3 \cdot F(x-1, y) - 2 \cdot F^2(x, y-1),$$
$$F(x, 0) = x, F(0, y) = 0.$$

Динамическое программирование

Таблица значений рекуррентной функции

x

y

	0	1	2	3	4
0	0	1	2	3	4
1	0	-2	-14		
2	0	-8	-416		
3	0	-128			
4	0	-32768			

$$F(x,y) = 3 \cdot F(x-1,y) - 2 \cdot F^2(x,y-1),$$
$$F(x,0) = x, F(0,y) = 0.$$

Динамическое программирование

```
// Вычисление рекуррентного выражения от двух переменных.
int F( int x, int y )
{
    vector<vector<int> > values( x + 1 );
    for( int i = 0; i <= x; ++i ) {
        values[i].resize( y + 1 );
        values[i][0] = i; // F( x, 0 ) = x;
    }
    for( int i = 1; i <= y; ++i ) {
        values[0][i] = 0; // F( 0, y ) = 0;
    }
    // Вычисляем по столбцам для каждого x.
    for( int i = 1; i <= x; ++i ) {
        for( int j = 1; j <= y; ++j ) {
            values[i][j] = 3 * values[i - 1][j] -
                2 * values[i][j - 1] * values[i][j - 1];
        }
    }
    return values[x][y];
}
```

Динамическое программирование

При вычисление рекуррентной функции $F(x, y)$ можно было не хранить значения на всех рядах. Для вычисления очередного ряда достаточно иметь значения предыдущего ряда.

Важная оптимизация ДП: Запоминать только те значения, которые будут использоваться для последующих вычислений.

Для вычисления числа Фибоначчи F_i также достаточно хранить лишь два предыдущих значения: F_{i-1} и F_{i-2}.

Динамическое программирование

Принципы ДП

1. Разбить задачу на подзадачи.
2. Кэшировать результаты решения подзадач.
3. Удалять более неиспользуемые результаты решения подзадач (опционально).

Динамическое программирование

Два подхода динамического программирования:

1. *<u>Нисходящее динамическое программирование</u>*: задача разбивается на подзадачи меньшего размера, они решаются и затем комбинируются для решения исходной задачи. Используется запоминание для решений часто встречающихся подзадач.
2. *<u>Восходящее динамическое программирование</u>*: все подзадачи, которые впоследствии понадобятся для решения исходной задачи просчитываются заранее и затем используются для построения решения исходной задачи.

Второй способ лучше нисходящего программирования в смысле размера необходимого стека и количества вызова функций, но иногда бывает нелегко заранее выяснить, решение каких подзадач нам потребуется в дальнейшем.

Динамическое программирование

Иногда бывает нелегко заранее выяснить, решение каких подзадач нам потребуется в дальнейшем.

<u>Пример.</u> $F(n) = F\left(\frac{n}{2}\right) + F\left(\frac{2 \cdot n}{3}\right)$, $F(0) = F(1) = 1$, деление целочисленное.

Восходящее динамическое программирование применить можно, но будут вычислены все значения F от 1 до n. Сложно определить, для каких k требуется вычислять $F(k)$.

Нисходящее динамическое программирование позволит вычислить только те $F(k)$, которые требуются, но рекурсивно. Максимальная глубина рекурсии $= \log_{3/2}(n)$.

Динамическое программирование

```
// Вычисление рекуррентного выражения методом нисходящего ДП.
int F( int n )
{
    // Разумнее использовать std::map, а не std::vector.
    std::vector<int> values( n, -1 );
    values[0] = 1;
    values[1] = 1;
    return calcF( n, values );
}

// Рекурсивное вычисление с запоминанием.
int calcF( int n, std::vector<int>& values )
{
    int& cached = values[n];
    if( cached == -1 )
        cached = calcF( n / 2, values ) + calcF( 2 * n / 3, values );
    return cached;
}
```

Наибольшая общая подпоследовательность

Последовательность X является *подпоследовательностью* Y, если из Y можно удалить несколько элементов так, что получится последовательность X.

Задача

Наибольшая общая подпоследовательность (англ. longest common subsequence, LCS) – задача поиска последовательности, которая является подпоследовательностью нескольких последовательностей (обычно двух).

Элементами подпоследовательности могут быть числа, символы…

X = ABCAB,
Y = DCBA,
LCS(X, Y) = BA, CA, CB.

Наибольшая общая подпоследовательность

Будем решать задачу нахождения наибольшей общей подпоследовательности с помощью ДП.

Сведем задачу к подзадачам меньшего размера:

$f(n_1, n_2)$ – длина наибольшей общей подпоследовательности строк $s_1[0..n_1]$, $s_2[0..n_2]$.

$$f(n_1, n_2) = \begin{cases} 0, & n_1 = 0 \vee n_2 = 0 \\ f(n_1 - 1, n_2 - 1) + 1, & s[n_1] = s[n_2] \\ max(f(n_1 - 1, n_2), f(n_1, n_2 - 1)), & s[n_1] \neq s[n_2] \end{cases}$$

Время работы алгоритма O$(n_1 \cdot n_2)$

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0					
C	0					
B	0					
A	0					

В каждой ячейке – длина наибольшей общей подстроки соответствующих начал строк.

$$f(n_1, n_2) = \begin{cases} 0, & n_1 = 0 \vee n_2 = 0 \\ f(n_1 - 1, n_2 - 1) + 1, & s[n_1] = s[n_2] \\ max(f(n_1 - 1, n_2), f(n_1, n_2 - 1)), & s[n_1] \neq s[n_2] \end{cases}$$

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0	0				
C	0					
B	0					
A	0					

В каждой ячейке – длина наибольшей общей подстроки соответствующих начал строк.

$$f(n_1, n_2) = \begin{cases} 0, & n_1 = 0 \vee n_2 = 0 \\ f(n_1 - 1, n_2 - 1) + 1, & s[n_1] = s[n_2] \\ max(f(n_1 - 1, n_2), f(n_1, n_2 - 1)), & s[n_1] \neq s[n_2] \end{cases}$$

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0	0	0			
C	0					
B	0					
A	0					

В каждой ячейке – длина наибольшей общей подстроки соответствующих начал строк.

$$f(n_1, n_2) = \begin{cases} 0, & n_1 = 0 \vee n_2 = 0 \\ f(n_1 - 1, n_2 - 1) + 1, & s[n_1] = s[n_2] \\ max(f(n_1 - 1, n_2), f(n_1, n_2 - 1)), & s[n_1] \neq s[n_2] \end{cases}$$

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0	0	0	0	0	0
C	0					
B	0					
A	0					

В каждой ячейке – длина наибольшей общей подстроки соответствующих начал строк.

$$f(n_1, n_2) = \begin{cases} 0, & n_1 = 0 \vee n_2 = 0 \\ f(n_1 - 1, n_2 - 1) + 1, & s[n_1] = s[n_2] \\ max(f(n_1 - 1, n_2), f(n_1, n_2 - 1)), & s[n_1] \neq s[n_2] \end{cases}$$

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0	0	0	0	0	0
C	0	0	0	1		
B	0					
A	0					

Берем значение LCS(“AB”, “D”) + 1

$$f(n_1, n_2) = \begin{cases} 0, & n_1 = 0 \lor n_2 = 0 \\ f(n_1 - 1, n_2 - 1) + 1, & s[n_1] = s[n_2] \\ max(f(n_1 - 1, n_2), f(n_1, n_2 - 1)), & s[n_1] \neq s[n_2] \end{cases}$$

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0	0	0	0	0	0
C	0	0	0	1	1	
B	0					
A	0					

Берем max(LCS(“ABCA”, “D”), LCS(“ABC”, “DC”))

$$f(n_1, n_2) = \begin{cases} 0, & n_1 = 0 \lor n_2 = 0 \\ f(n_1 - 1, n_2 - 1) + 1, & s[n_1] = s[n_2] \\ max(f(n_1 - 1, n_2), f(n_1, n_2 - 1)), & s[n_1] \neq s[n_2] \end{cases}$$

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0	0	0	0	0	0
C	0	0	0	1	1	1
B	0	1	1	1	1	2
A	0	1	1	1	2	2

$$f(n_1, n_2) = \begin{cases} 0, & n_1 = 0 \lor n_2 = 0 \\ f(n_1 - 1, n_2 - 1) + 1, & s[n_1] = s[n_2] \\ max(f(n_1 - 1, n_2), f(n_1, n_2 - 1)), & s[n_1] \neq s[n_2] \end{cases}$$

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0	0	0	0	0	0
C	0	0	0	1	1	1
B	0	0	1	1	1	2
A	0	1	1	1	2	2

В каждой ячейке – длина наибольшей общей подстроки соответствующих начал строк.

Наибольшая общая подпоследовательность

$f(n_1, n_2)$ – длина НОП.

Как восстановить саму подпоследовательность?

Можно хранить в каждой ячейке таблицы «направление» перехода. Переход по диагонали означает, что очередной символ присутствует в обоих последовательностях.

Начинаем проход от правого нижнего угла.
Идем по стрелкам, на каждый переход по диагонали добавляем символ в начало строящейся подпоследовательности.

«Направления» можно не хранить, а вычислять по значениям в таблице.

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0	←↑ 0	←↑ 0	←↑ 0	←↑ 0	←↑ 0
C	0	←↑ 0	←↑ 0	↖ 1	← 1	← 1
B	0	←↑ 0	↖ 1	←↑ 1	←↑ 1	↖ 2
A	0	↖ 1	←↑ 1	←↑ 1	↖ 2	←↑ 2

Желтым выделены ячейки, лежащие на лучшем пути.
Переход по диагонали соответствует совпадению символа.
Возможно существованию несколько лучших путей – по одному на каждую Наибольшую Общую Подпоследовательность.

Наибольшая общая подпоследовательность

```
std::string LCS(const std::string &str1, const std::string &str2) {
    std::string result;

    if (str1.empty() || str2.empty())
        return result;
    // инициализируем таблицу
    std::vector<std::vector<int>> tbl;
    tbl.resize(str1.size() + 1);
    for (int i = 0; i < tbl.size(); ++i) {
        tbl[i].resize(str2.size() + 1, 0);
    }
    // заполняем таблицу
    for (int i = 1; i < tbl.size(); ++i) {
        for (int j = 1; j < tbl[i].size(); ++j) {
            if (str1[i - 1] == str2[j - 1]) {
                tbl[i][j] = tbl[i - 1][j - 1] + 1;
            } else {
                tbl[i][j] = std::max(tbl[i - 1][j], tbl[i][j - 1]);
            }
        }
    }
```

Наибольшая общая подпоследовательность

```
    for (int i = tbl.size() - 1, j = tbl[0].size() - 1; i > 0 && j > 0;) {
        if (str1[i - 1] == str2[j - 1]) {
            result.push_back(str1[i - 1]);
            i--;
            j--;
        } else if (tbl[i - 1][j] > tbl[i][j - 1]) {
                i--;
        } else {
                j--;
        }
    }
    std::reverse(result.begin(), result.end());
    return result;
}
```

Оптимизация алгоритмов

Алгоритмы, эффективные в теории, бывают не очень эффективными на практике из-за особенностей аппаратного обеспечения или ОС (например, пирамидальная сортировка и quick-sort).

Конкретная реализация алгоритма может быть не оптимальной и не учитывать особенностей аппаратного обеспечения или ОС.

Оптимизация алгоритмов

Какой цикл отработает быстрее?

```
int sum = 0;
// квадратная матрица
int *arr = new int[row_size * row_size];

// обходим по строкам
for (size_t i = 0; i < row_size; ++i) {
    for (size_t j = 0; j < row_size; ++j) {
        sum += arr[i * row_size + j];
    }
}

// обходим по колонкам
for (size_t j = 0; j < row_size; ++j) {
    for (size_t i = 0; i < row_size; ++i) {
        sum += arr[i * row_size + j];
    }
}
```

Оптимизация алгоритмов

Среднее время выполнения цикла (в микросекундах) при различных размерах массива

	По строкам	По колонкам
64x64 (16 кб)	22.6	22.6
1024x1024 (4 Мб)	5591	11112
5120x5120 (100 Мб)	$143 * 10^3$	$1160 * 10^3$

Среднее время одного чтения из памяти (в наносекундах) при различных размерах массива

	По строкам	По колонкам
64x64 (16 кб)	5.5	5.5
1024x1024 (4 Мб)	5.5	10.5
5120x5120 (100 Мб)	5.5	44.6

Иерархия памяти

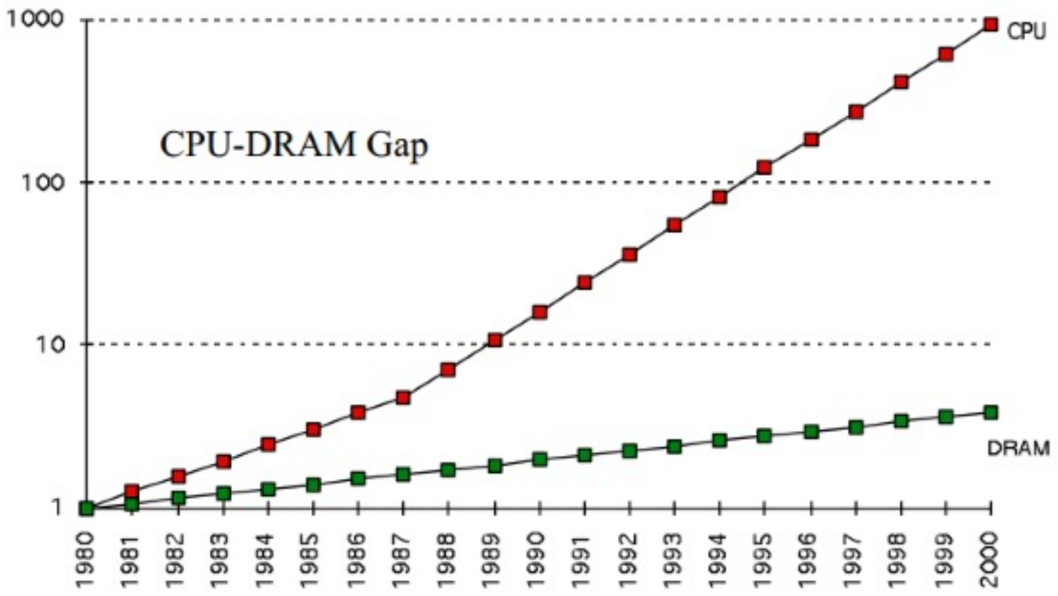
Производительность процессоров росла быстрее производительности памяти.

Использовать много быстрой памяти дорого.

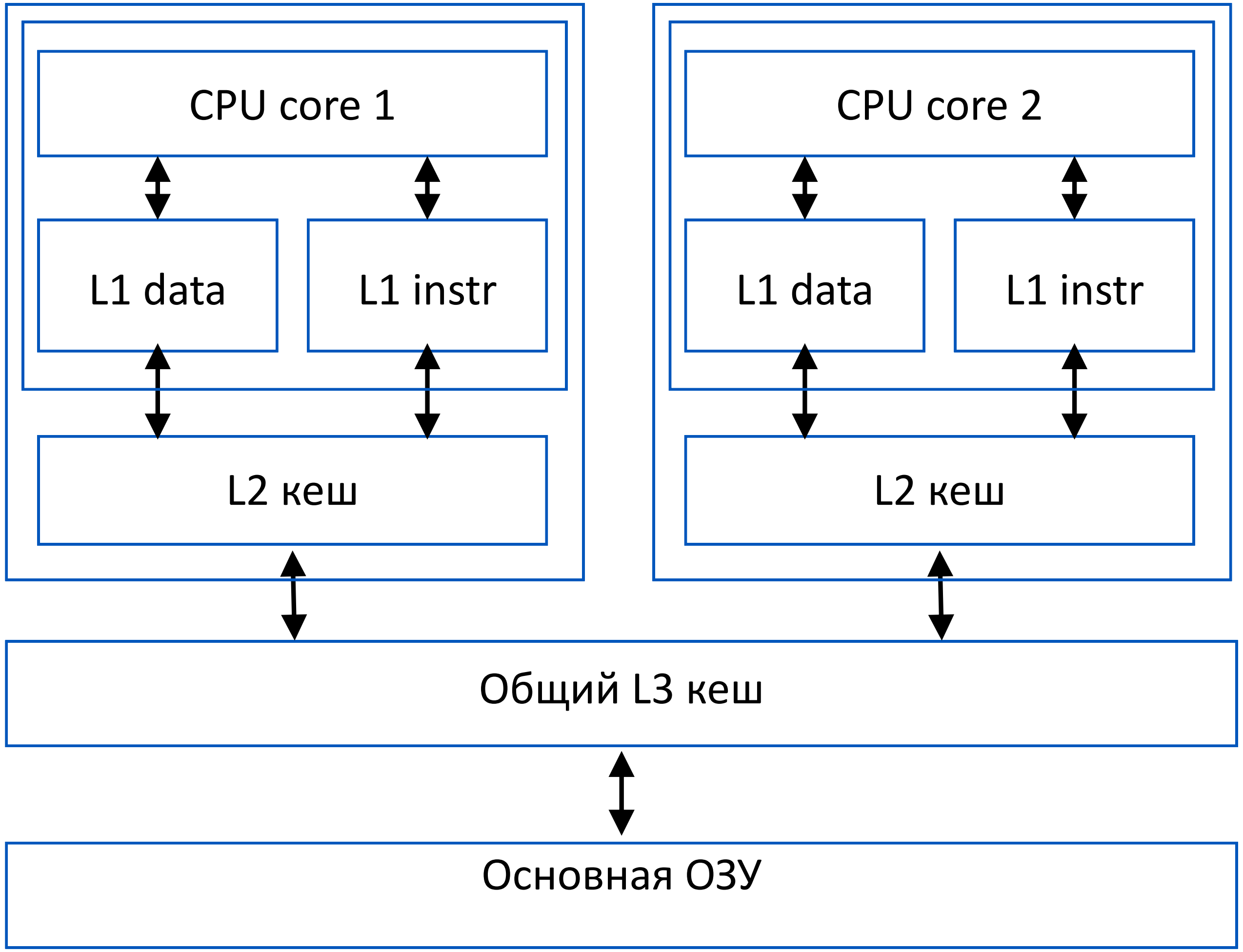
Решение – добавить небольшую промежуточную память между CPU и основной памятью.

Такая память называется
кеш процессора.

- Processor vs Memory Performance



Иерархия памяти



Время доступа к памяти

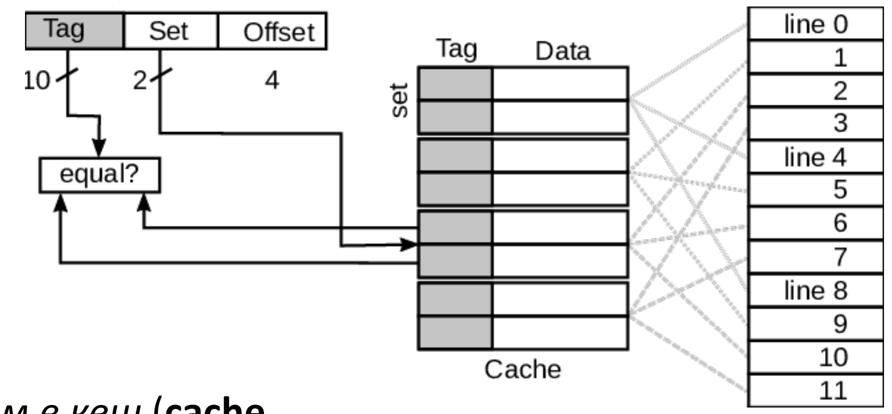
Реальное время очень сильно зависит от процессора, памяти, материнской платы, настроек BIOS, кол-ва плашек памяти и т.д.

	L1	L2	L3	ОЗУ	SSD	HDD
Время, нс	2	5.5	20	50	25×10^3	5×10^6
Время, циклов	4	11	40	100	–	–

Устройство кеша

Данные между кешем и памятью передаются блоками фиксированного размера, также называемые **линиями кеша** (*cache line*).

Типичный размер кеш-линии – 64 байта.



Случай совпадения с тегом какой-либо кеш-линии называется *попаданием в кеш* (**cache hit**), обратный же случай называется *кеш-промахом* (**cache miss**).

Пример:

2-way set associative cache

8 кеш-линий

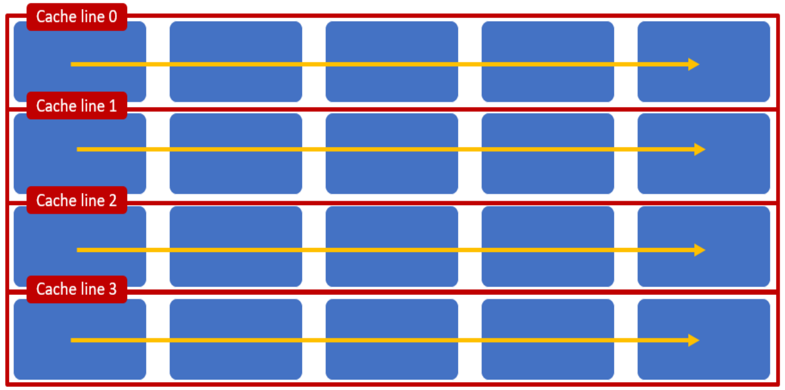
4 сета

Отношение количества попаданий в кеш к общему количеству запросов к памяти называют *рейтингом попаданий* (**hit rate**), оно является мерой эффективности кеша для выбранного алгоритма или программы.

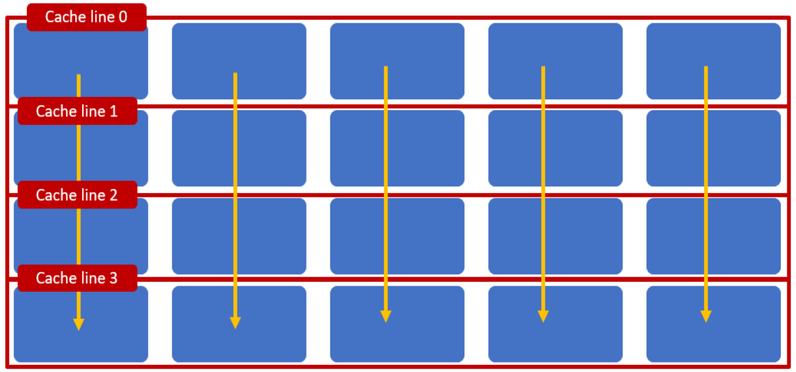
При cache miss кеш-линия считывается из ОЗУ и замещает одну из заполненных (**cache eviction**), как правило, используя политику LRU (least recently used, наиболее редко используемая).

Устройство кеша

При обходе двумерного массива по строкам каждый следующий элемент присутствовал к кеше, потому что кеш-линия хранит линейную последовательность байт.



При обходе двумерного массива по столбцам каждый следующий элемент попадает в другую кеш-линию, в которой может не оказаться нужных данных.



Кеш процессора

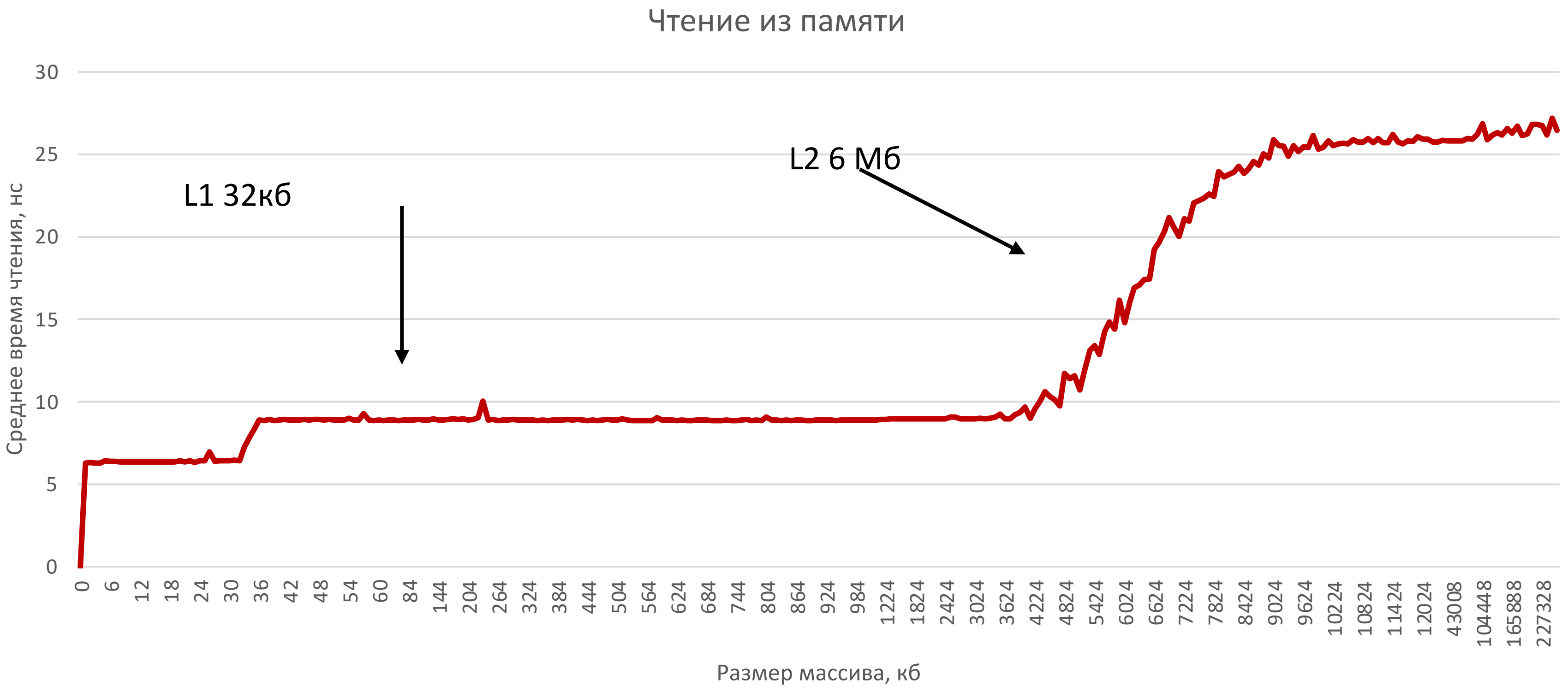
Тест для определения влияния кеша процессора на время доступа к данным.

```
int *arr = new int[arr_size]();
int step = 16; // sizeof(int) * 16 = 64 байта
for (int i = 0; i < arr_size; i += step) {
        arr[i] = i + step;
}

for (int count = 0; count < samples_count; ++count) {
    int idx = arr[0];
    while (idx < arr_size) {
        idx = arr[idx];
    }
}
delete[] arr;
```

Кеш процессора

Что можно сказать о кеше процессора по этому графику?

Кеш процессора



Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 8
On-line CPU(s) list: 0-7
Thread(s) per core: 1
Core(s) per socket: 4
Socket(s): 2
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 23
Model name: Intel(R) Xeon(R) CPU E5405 @ 2.00GHz
Stepping: 10
CPU MHz: 2000.071
BogoMIPS: 4000.14
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 6144K

Время доступа начинает увеличиваться еще до того, как размера массива превысит размер кеша, так как с ростом размера массива увеличивается вероятность вытеснения его частей из кеша, т.е. увеличивается количество cache miss.

Оптимизация алгоритмов

- По возможности нужно располагать данные в памяти последовательно и обращаться к ним тоже последовательно – увеличивать **локальность данных.**
- Переход по указателю может быть сравнительно дорогой операцией.
- Не нужно заниматься преждевременной оптимизацией, но и не нужно отказываться от «бесплатных» оптимизаций (пример – обход двумерного массива).

Литература

Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К, Алгоритмы. Построение и анализ., 2-е изд, М.: Издательский дом «Вильямс», 2005.

Седжвик Р., Фундаментальные алгоритмы на C++, части 1-4, анализ, структуры данных, сортировка, поиск, М.: ДиаСофт, 2001.

Шень А., Программирование: теоремы и задачи, 2-е изд., испр. и доп., М.: МЦНМО, 2004.

Викиконспекты,
http://neerc.ifmo.ru/wiki/index.php?title=Дискретная_математика,_алгоритмы_и_структуры_данных

What Every Programmer Should Know About Memory

Обратная связь

krymov@corp.mail.ru

https://t.me/alkrymov

Вопросы?