

Алгоритмы и структуры данных

Курс «Технология программирования»

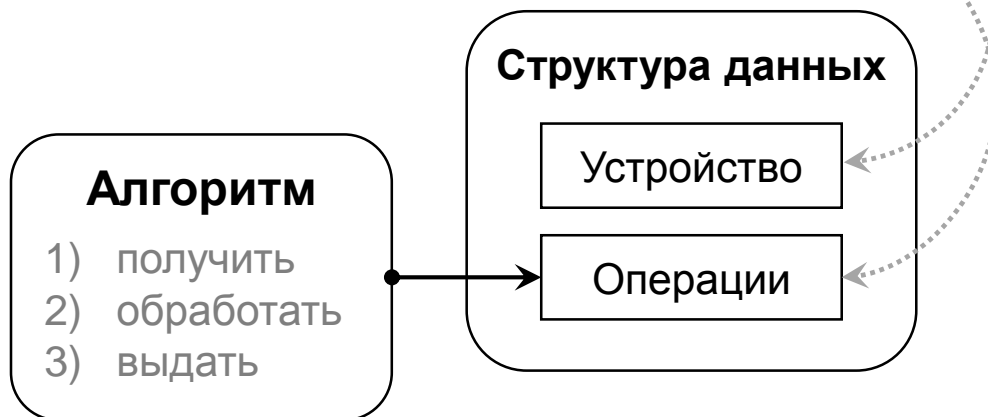
Кафедра управления и информатики НИУ «МЭИ»

Осень 2015 г.

Алгоритмы и структуры данных

- Алгоритм —
набор инструкций вычислителя, описывающих
порядок выполнения действий для достижения результата
при любых входных данных за конечное число действий.
- Структура данных —
способ представления данных в памяти
для их эффективного использования в программе.

Как это понимать,
измерять,
сравнивать?



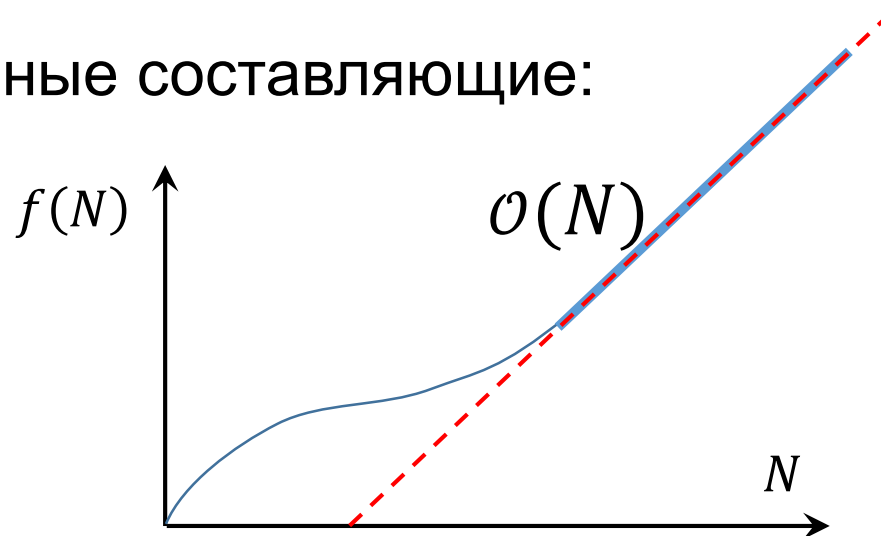
Алгоритмическая сложность

- Асимптотическая оценка сложности алгоритма.
- Сложность — число элементарных операций f при объеме данных N .
 - Элементарная операция едина для класса алгоритмов.
 - Обычно самая вычислительно сложная.
 - Пример: для сортировок — сравнение элементов.
 - Объем данных:
 - Обычно количество элементов N .
 - Иногда характеризуется несколькими числами.
 - *Пример:* для умножения матриц $A^{N \times M} \cdot B^{M \times K}$ — три размера N , M и K .

Асимптотическая оценка

* с практической точки зрения

- Показывает характер $f(N)$ при $N \rightarrow \infty$.
 - При малых N «не работает»!
- Учитывает самую быстрорастущую составляющую:
 - $5 + 3N + 3N^2 + \frac{1}{2}N^3 \rightarrow \mathcal{O}(N^3)$
- Не учитывает постоянные составляющие:
 - $100500 \rightarrow \mathcal{O}(1)$
 - $300N \rightarrow \mathcal{O}(N)$
 - $\log_5 N \rightarrow \mathcal{O}(\log N)$,
 $\log_{200} N^2 \rightarrow \mathcal{O}(\log N)$
 - $\log_a N^b = b \cdot \log_a N$



Линейный поиск

- Алгоритм:

для каждого элемента коллекции:

если это искомый элемент, **то**
завершить поиск.

- Оценка сложности:

- Что есть элементарная операция?
 - Сравнение элементов.
- Сколько нужно сравнений?
 - В контейнере N элементов, искомый может быть последним — $\mathcal{O}(N)$.

- Использует только доступ к очередному элементу.
 - Не предъявляет особых требований к структуре данных.

Двоичный поиск (binary search)

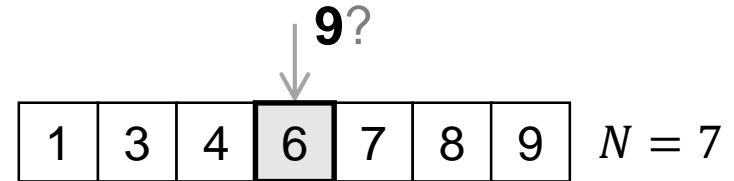
- Искомый элемент (9) меньше среднего (6).

- В упорядоченном массиве
большие элементы правее меньших.

➤ Искомый элемент в правой половине.

- На каждом шаге область поиска
(часть массива) сокращается вдвое.

- Случай (этап), когда $N = 1$:
 - ✓ единственный элемент — искомый;
 - × искомый элемент отсутствует.



- Применим к упорядоченным массивам.
- Использует доступ к элементам по индексу.

Быстрее линейного?

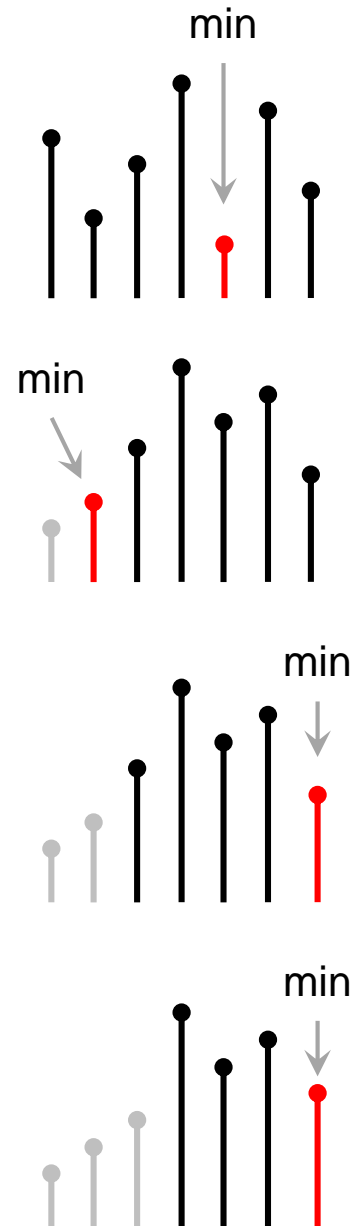
«Анализ» асимптотики двоичного поиска

- Что есть элементарная операция?
 - Сравнение элемента с искомым.
- Сколько будет сравнений?
 - Столько же, сколько шагов.
- Сколько будет шагов?
 - Столько же, сколько раз можно (нацело) поделить N на 2, пока не останется 1.
 - В какую степень возвести 2, чтобы получилось N .
 - Логарифм! $\mathcal{O}(\log N)$
 - Или меньше, если искомый элемент встретится раньше.

Сортировка вставками (insertion sort)

```
void insertion_sort(vector<double>& data)
{
    if (data.empty())
        return;
    for (unsigned int i = 0; i < data.size() - 1; ++i) {
        // Найти наименьший из оставшихся:
        unsigned int min_index = i;
        for (unsigned int j = i + 1; j < data.size(); ++j) {
            if (data[j] < data[min_index])
                min_index = j;
        }
        // Поместить минимальный элемент на место текущего.
        swap(data[i], data[min_index]);
    }
}
```

Обмен значений *a* и *b*.



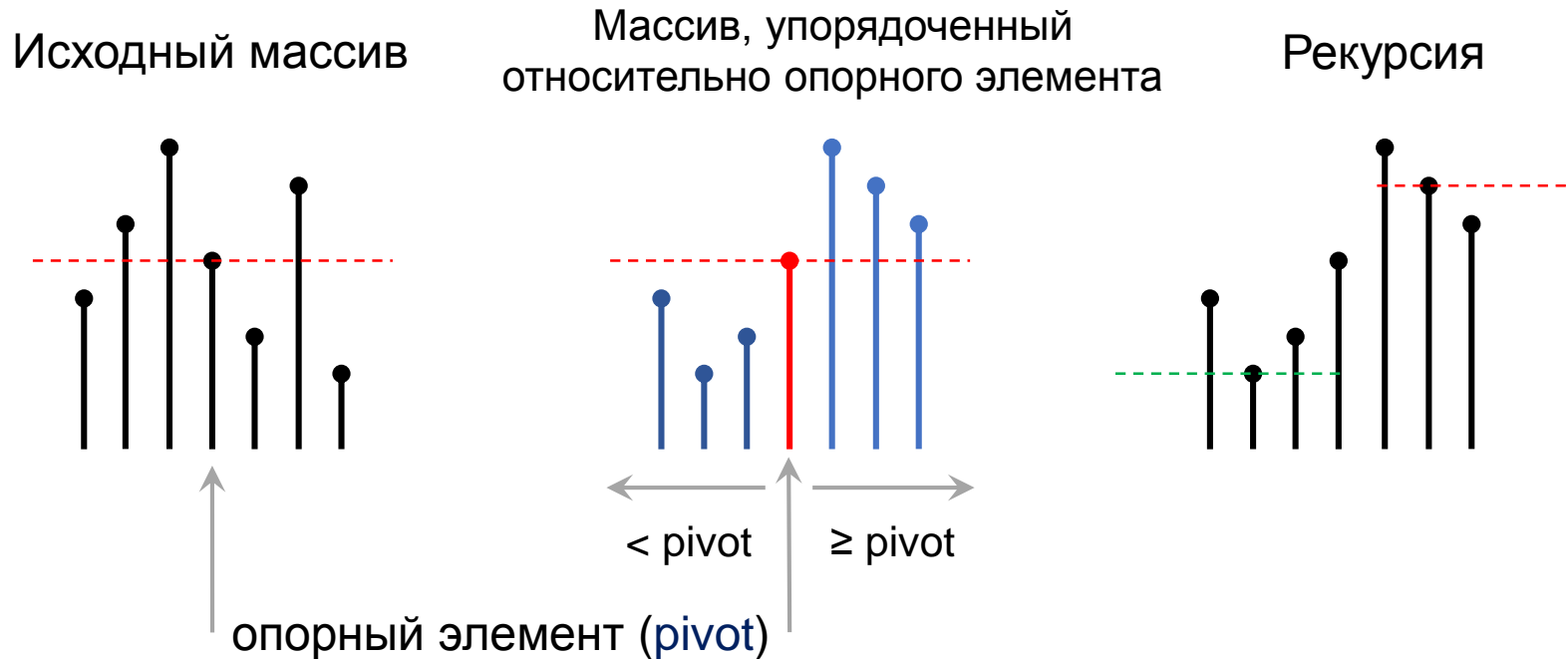
Сортировка вставками

- Что есть элементарная операция?
 - Сравнение элементов (при поиске наименьшего).
- Сколько будет сравнений?
 - В 1-м проходе — $(N - 1)$, во 2-м проходе — $(N - 2)$, и т. д.
 - Проходов $(N - 1)$.
 - Арифметическая прогрессия:

$$f(n) = (N - 1) \frac{(1 + (N - 1))}{2} = \frac{1}{2}N^2 - \frac{1}{2}N$$

- Сложность: $\mathcal{O}\left(\frac{1}{2}N^2 - \frac{1}{2}N\right) = \mathcal{O}\left(\frac{1}{2}N^2\right) = \mathcal{O}(N^2)$
 - В любом случае.
 - Оптимизация: если в проходе не было обменов, закончить.
 - Если массив отсортирован, сложность снижается до $\mathcal{O}(N)$.

«Быстрая» сортировка



- Деление может быть не на равные части.
- Может потребоваться переместить опорный элемент.

Реализация «быстрой» сортировки

Алгоритм обрабатывает часть массива `data`: 

```
void quick_sort(  
    vector<double> & data, unsigned int from, unsigned int to)  
{
```

1) Распределить элементы относительно опорного (`pivot`).

```
    const unsigned int pivot = partition(data, from, to);
```

2) Применить алгоритм к элементам, меньшим опорного.

```
    quick_sort(data, from, pivot - 1);
```

3) Применить алгоритм к элементам, большим или равным опорному.

```
    quick_sort(data, pivot + 1, to);
```

```
}
```

Распределение элементов относительно опорного

Перед этой ячейкой
помещаются элементы,
меньшие опорного.



Меньше 5?

Новый индекс опорного элемента. По нему делится массив.

Реализация «быстрой» сортировки (продолжение)

```
unsigned int partition(vector<double>& data, unsigned int from, unsigned int to)
{
    unsigned int const pivot_index = from + (to - from) / 2;
    double const pivot_value = data[pivot_index];
    swap(data[pivot_index], data[to]);
    unsigned int new_pivot_index = from;
    for (unsigned int i = from; i < to; ++i)
        if (data[i] < pivot_value) {
            swap(data[i], data[new_pivot_index]);
            ++new_pivot_index;
        }
    swap(data[new_pivot_index], data[to]);
    return new_pivot_index;
}
```

Выбор опорного элемента.

- Подходит любой.
- Производительность?
- Варианты:
 - посередине;
 - случайный;
 - медиана из начального, конечного и посередине.

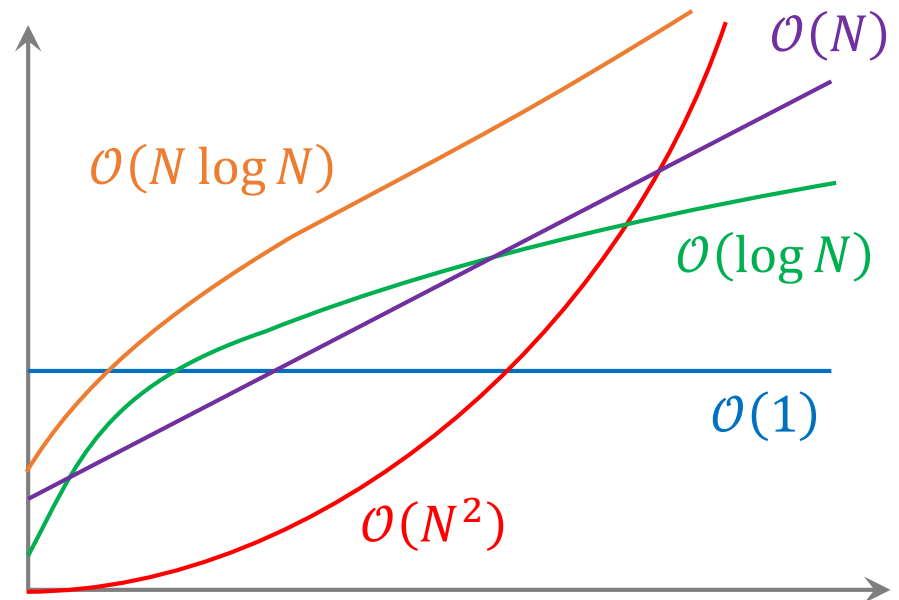
Анализ асимптотики «быстрой» сортировки

- Элементарная операция — сравнение элементов.
- На каждом шаге обрабатываются все N элементов.
- Сколько шагов-разбиений?
 - Сколько раз можно разбить N на две (неравные) части?
 - Отделяя по одному элементу, N раз.
 - Худший случай: $(N^{\text{сравнений}} / \text{разбиение}) \times (N \text{ разбиений}) \rightarrow \mathcal{O}(N^2)$.
 - Массив, отсортированный в обратном порядке.
 - Когда число разбиений минимально?
 - При делении пополам — $\log_2 N$ шагов.
 - Лучший случай: $(N^{\text{сравнений}} / \text{разбиение}) \times (\log_2 N \text{ разбиений}) \rightarrow \mathcal{O}(N \log N)$.
- Лучший случай статистически — средний:

$$\mathcal{O}(N \log N)$$

Классы сложности

- Постоянная $O(1)$
- Линейная $O(N)$:
 - цикл.
- Логарифмическая $O(\log N)$:
 - при дроблении задачи на каждом шаге.
- «Лог-линейная» $O(N \log N)$
- Полиномиальная $O(N^{const})$:
 - вложенные циклы.



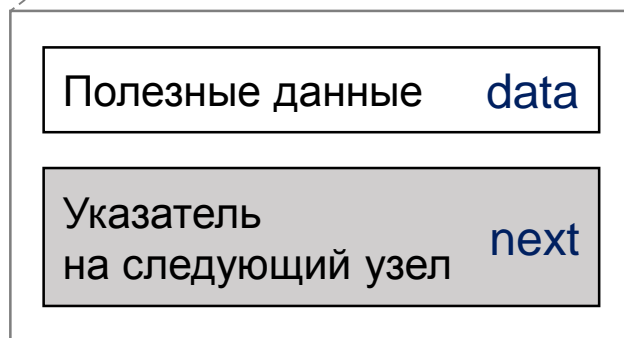
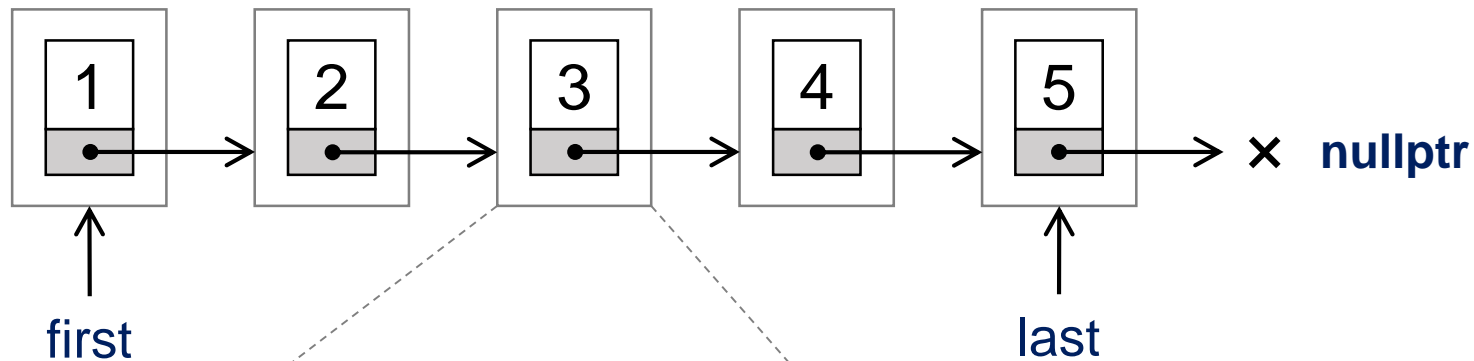
➤ Сложности вложенных действий перемножаются.

➤ Из сложностей последовательных действий остается наивысшая.

Массив как структура данных

- **Организация:**
непрерывный блок памяти с элементами друг за другом.
- **Операции:**
 - доступ по индексу — $O(1)$;
 - вставка и удаление элементов — $O(N)$:
 - сдвинуть каждый элемент в сторону за $O(N)$;
 - при вставке: поместить элемент по индексу за $O(1)$;
 - если блок памяти заполнен, требуется скопировать все элементы в новый за $O(N)$, а старый освободить;
 - поиск и сортировка только алгоритмами.

Односвязный список (singly-linked list)



Элемент (узел, node)

```
struct Node
{
    double data;
    Node* next;
};
```

Класс связанного списка

- Класс связывает данные и способы их обработки:
 - структура (**struct**)
 - и функции над ней.
- Данные называются полями, или атрибутами.
- Функции называются методами.
- Зачем?
 - Оперировать списком как единым целым.
 - Инициализация и очистка автоматически.

```
struct LinkedList
{
    Node* first_;
    Node* last_;
    size_t size_;

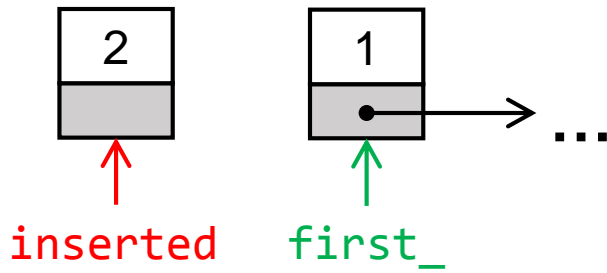
    LinkedList();
    ~LinkedList();

    void push_front(double item);
    void clear();
    void get_by_index ( size_t index );
    // ...
};
```

Вставка элемента: а) в начало списка

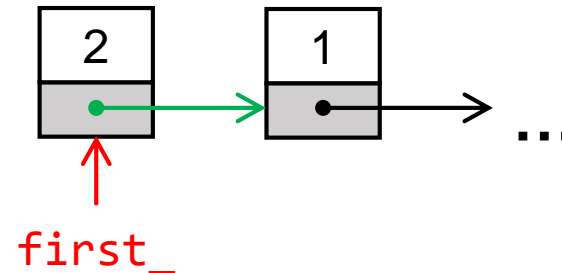
Было:

СПИСОК И НОВЫЙ УЗЕЛ



Стало:

СПИСОК С НОВЫМ УЗЛОМ В НАЧАЛЕ



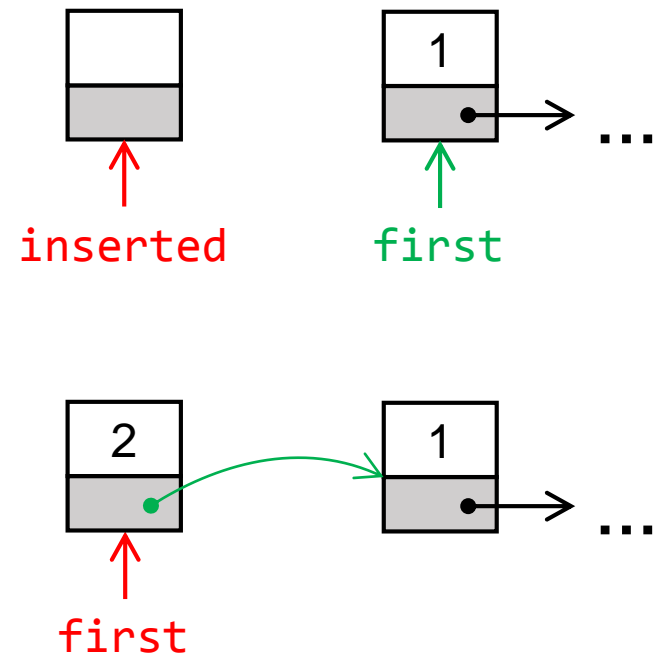
```
inserted->next = first_;  
first_ = inserted;
```

`pointer->field` —
это `(*pointer).field`

- ❖ Один цвет — один адрес (значение указателя), кроме черных и серых (которые не важны).

Реализация вставки в список

```
void LinkedList::push_front(double value)
{
    Node* inserted = new Node;
    inserted->value = value;
    inserted->next = first_;
    first_ = inserted;
    if (last_ == nullptr) {
        last_ = inserted;
    }
    ++size_;
}
```

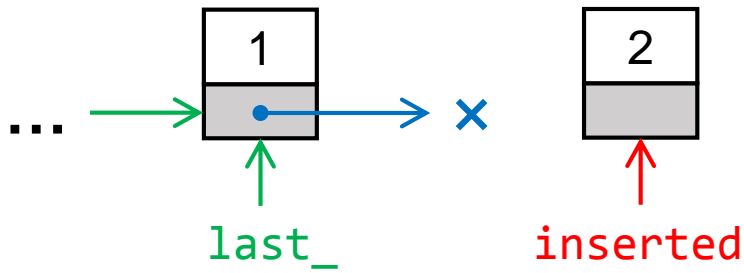


Вставка элемента:

б) в конец списка

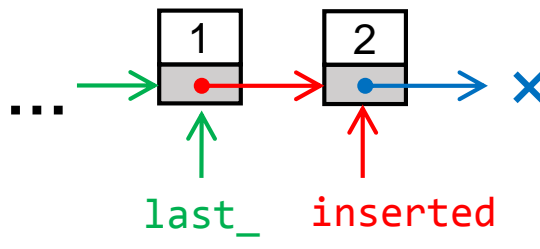
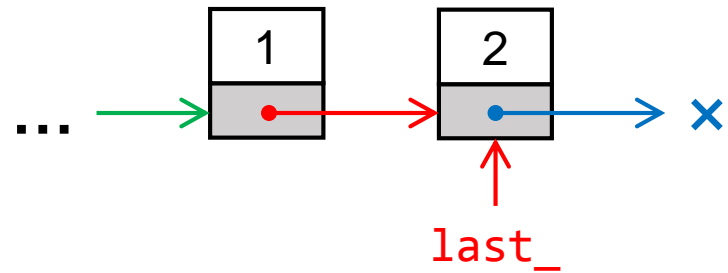
Было:

СПИСОК И НОВЫЙ узел



Стало:

СПИСОК С НОВЫМ УЗЛОМ В КОНЦЕ



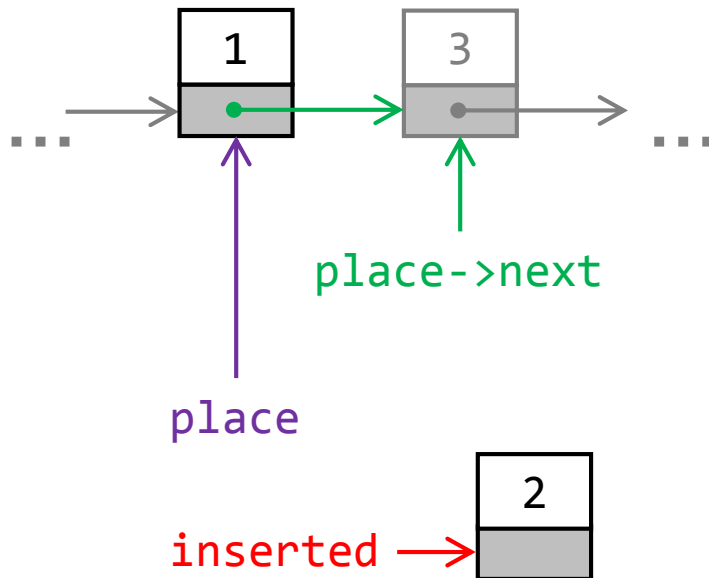
```
inserted->next = nullptr;  
last_->next = inserted;  
last_ = inserted;
```

Вставка элемента:

в) в произвольное место

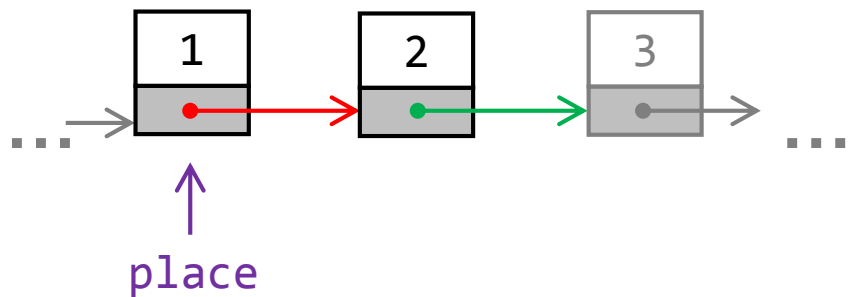
Было:

СПИСОК, НОВЫЙ УЗЕЛ
и указатель на узел в списке



Стало:

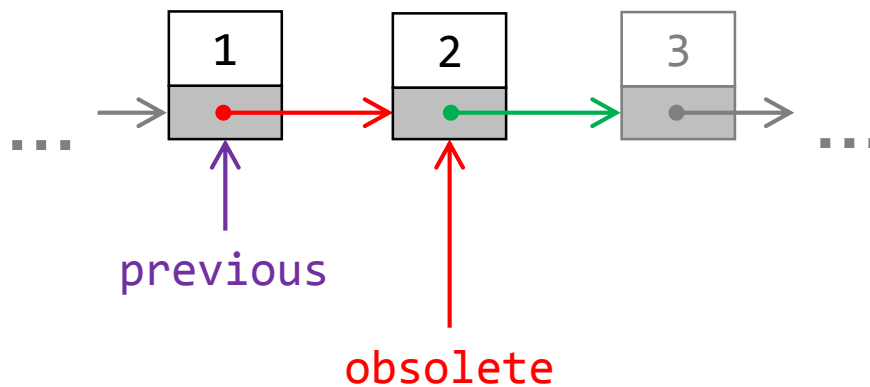
СПИСОК С НОВЫМ УЗЛОМ В КОНЦЕ



```
inserted->next = place->next;  
place->next = inserted;
```

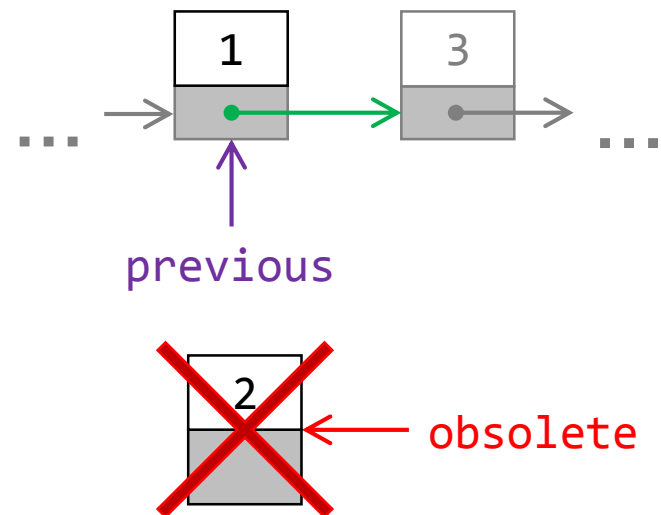
Удаление элемента

Было:
список и узел к удалению



```
previos->next = obsolete->next;  
delete obsolete;
```

Стало:
список без указанного узла



Проход по списку (enumeration)

- Размер списка неизвестен:

```
Node* current = first_;  
while (current)  
    current = current->next;
```

- До элемента № index:

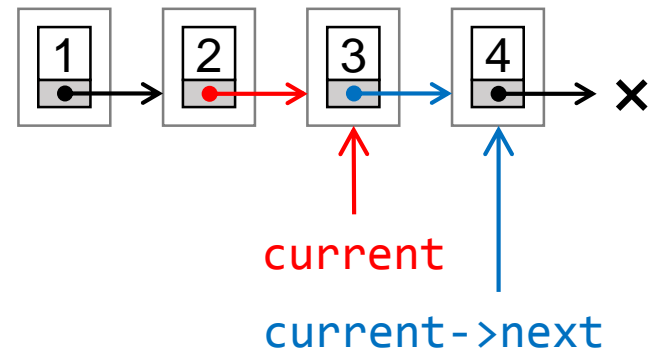
```
Node* current = first_;  
for (i = 0; i < index; ++i)  
    current = current->next;
```

- Интервал [first; last) или любой другой:

```
Node* current = first_;  
while (current != last_)  
    current = current->next;
```

- Можно проверять current:

```
while (current && current != last_) {  
    ...  
}
```



Реализация очистки списка

```
void LinkedList::clear()
{
    Node* current = first_;
    while ( current ) {
        Node* victim = current;
        current = current->next;
        delete victim;
    }
    first_ = last_ = nullptr;
    size_ = 0;
}
```

Одни методы могут
вызывать другие.

Изначально список пуст.

```
LinkedList::LinkedList()
    : first_ { nullptr }
    , last_ { nullptr }
    , size { 0 }
{
    // first_ = last_ = nullptr;
    // size = 0;
}

LinkedList::~~LinkedList() {
    clear();
}
```

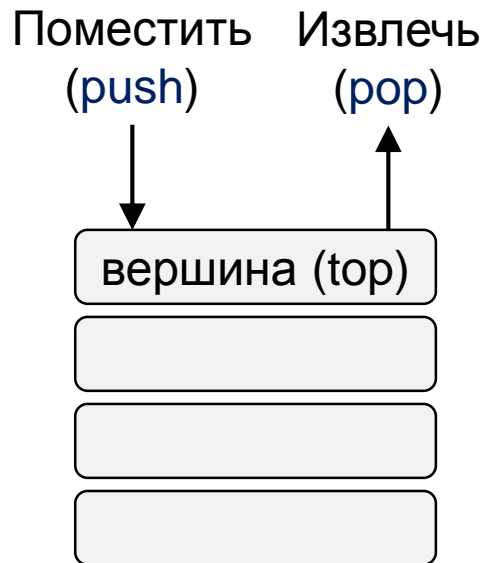
Производительность связанных списков

- Доступ по индексу: $O(N)$.
 - В односвязном списке — доступ только к следующему элементу.
- Вставка и удаление элементов: $O(1)$.
 - Соединение списков за $O(1)$.
- При изменении списка адреса узлов сохраняются.
 - Доступ по адресу за $O(1)$.
- Расход памяти строго под число элементов,
 - но есть накладные расходы на каждый узел,
 - особенно в двусвязных списках. —————→
- Никогда не требуется копировать данные.

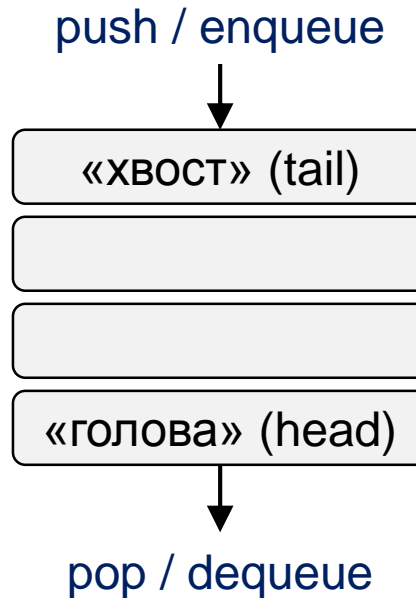
```
struct Node
{
    double data;
    Node * next;
    Node * previous;
};
```

Стек, очередь и дек

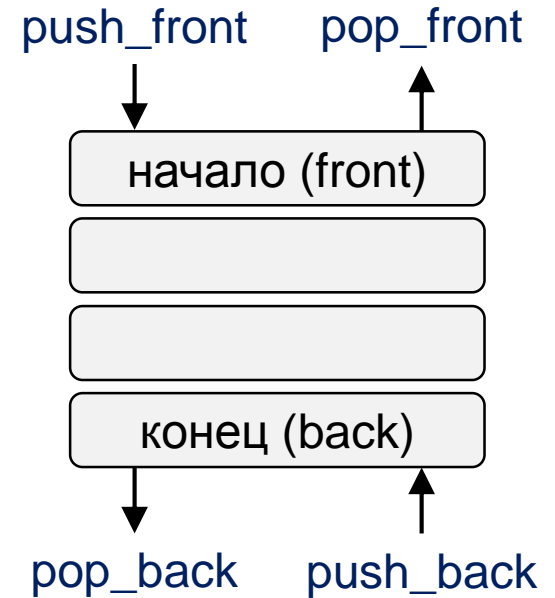
Стек (stack)



Очередь (queue)



Дек (deque)

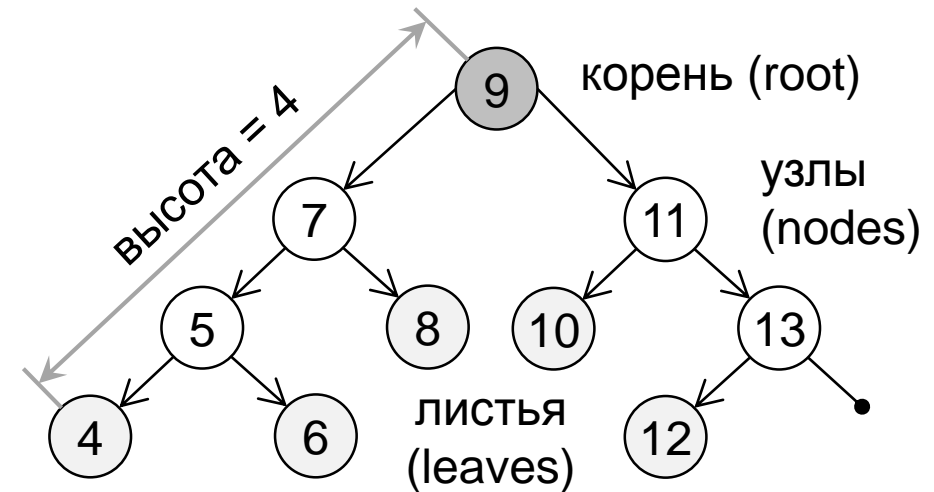


Массив или связанный список с ограниченным набором операций.

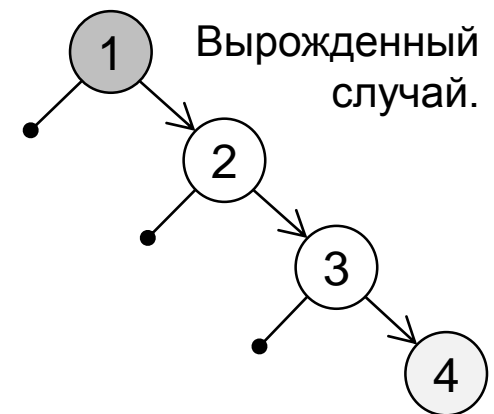
Деревья (trees)

- У каждого узла, кроме **корня**, один родитель.
- Узлы без потомков называются **листьями** (`left == right == nullptr`).
- **Высота** дерева считается по самой длинной ветви.
- Каждая ветвь — полноценное дерево.
 - Алгоритмы обработки часто рекурсивны.

Необязательно.

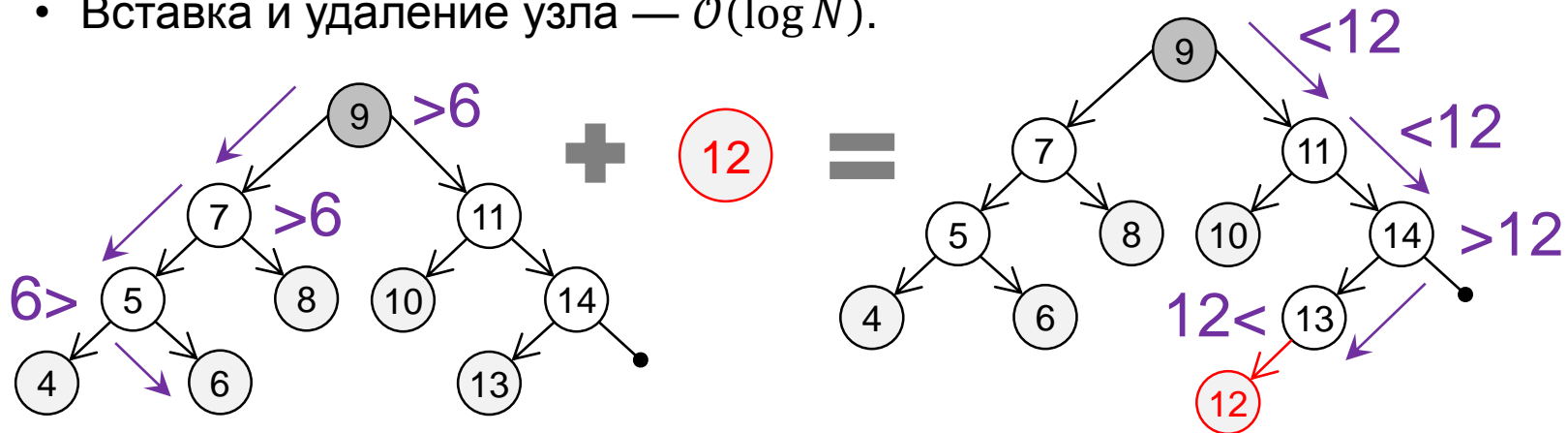


```
struct Node
{
    double data;
    Node* left;
    Node* right;
    Node* parent;
};
```



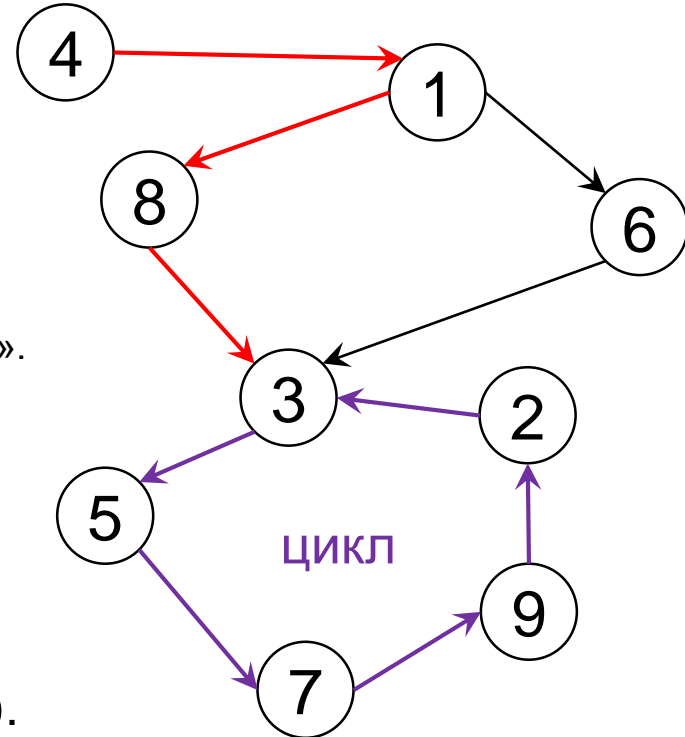
Двоичное дерево (binary tree)

- Не больше двух дочерних узлов.
 - Значения узлов слева меньше родительского, значения узлов справа **НЕ** меньше родительского.
- **Сбалансированное** двоичное дерево
 - содержит в левой и правой ветви равное число узлов,
- Иначе: узел делит дочерние на меньшие и большие его,
 - как любой элемент упорядоченного массива.
 - Поиск в двоичном дереве — двоичный поиск за $O(\log N)$.
- Вставка и удаление узла — $O(\log N)$.



Графы (graphs)

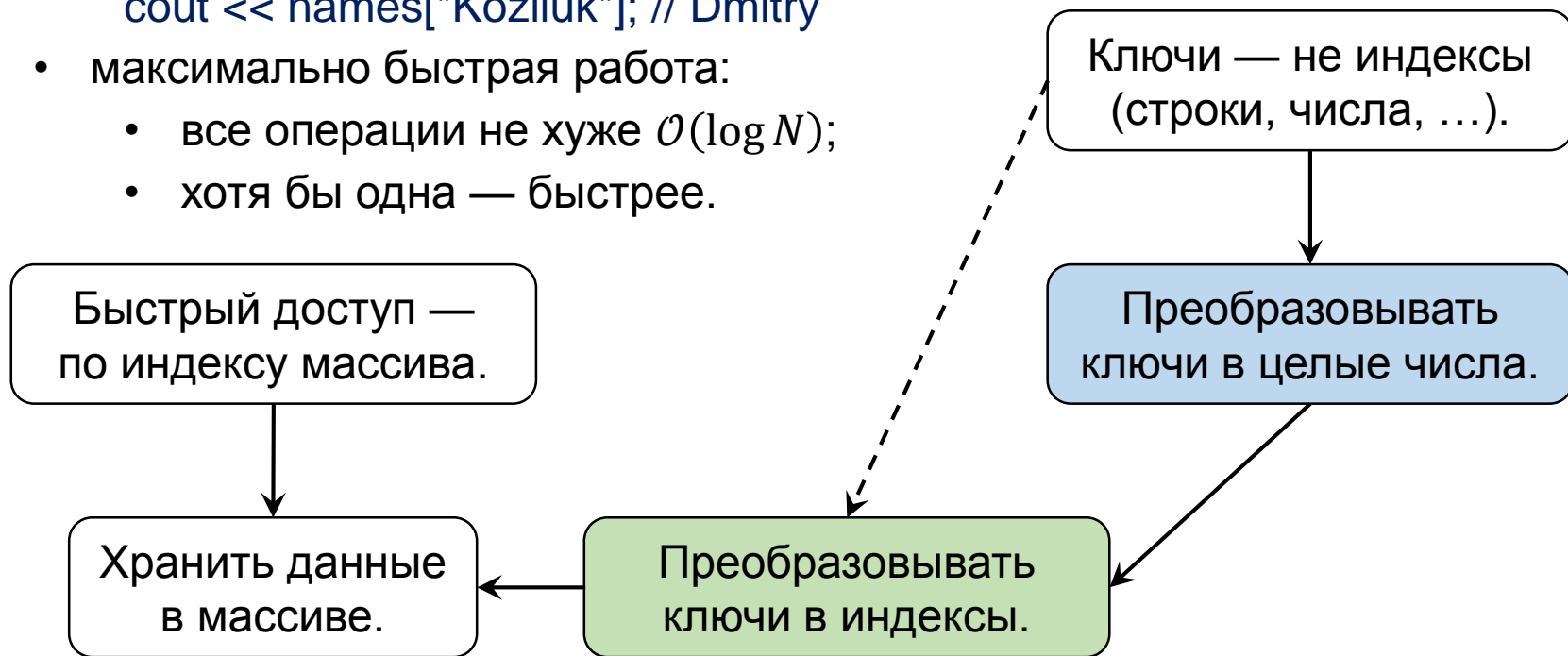
- **Граф** (*в программировании*) — структура данных, в которой N узлов связаны произвольно.
 - По K связей в среднем.
- Типовые задачи:
 - поиск **пути между узлами**:
 - кратчайшего;
 - с посещением заданных узлов;
 - всех — «задача коммивояжера».
 - поиск **циклов** в графе.
- Алгоритмы на графах:
 - рекурсивные,
 - без упрощения задачи.
 - Встречаются $\mathcal{O}(e^N)$, $\mathcal{O}(N!)$, $\mathcal{O}(K^N)$.



Хэш-таблица (hash table)

Задача:

- доступ по ключу любого типа (ассоциативный массив, «словарь»):
`names["Kozliuk"] = "Dmitry";`
`cout << names["Kozliuk"]; // Dmitry`
- максимально быстрая работа:
 - все операции не хуже $O(\log N)$;
 - хотя бы одна — быстрее.



Хэширование

- Хэш-функция

```
hash < string > hash_function;  
hash_function ( "Dmitry" ) == 3512122310  
hash_function ( "Vadim" ) == 4155307891
```

- Коллизии —

совпадения значений хэш-функции для различных аргументов.

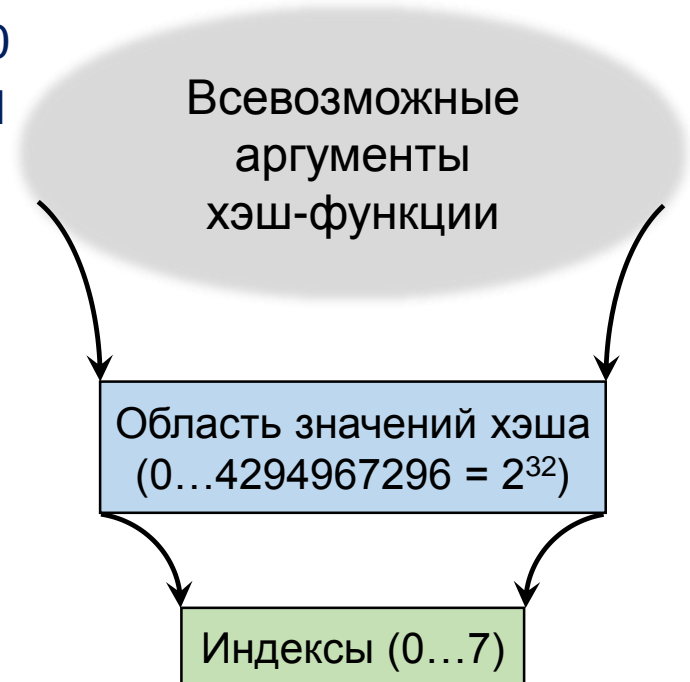
- Получение индекса из хэша

```
dmitry_index = 3512122310 % 8; // 6  
vadim_index = 4155307891 % 8; // 3
```

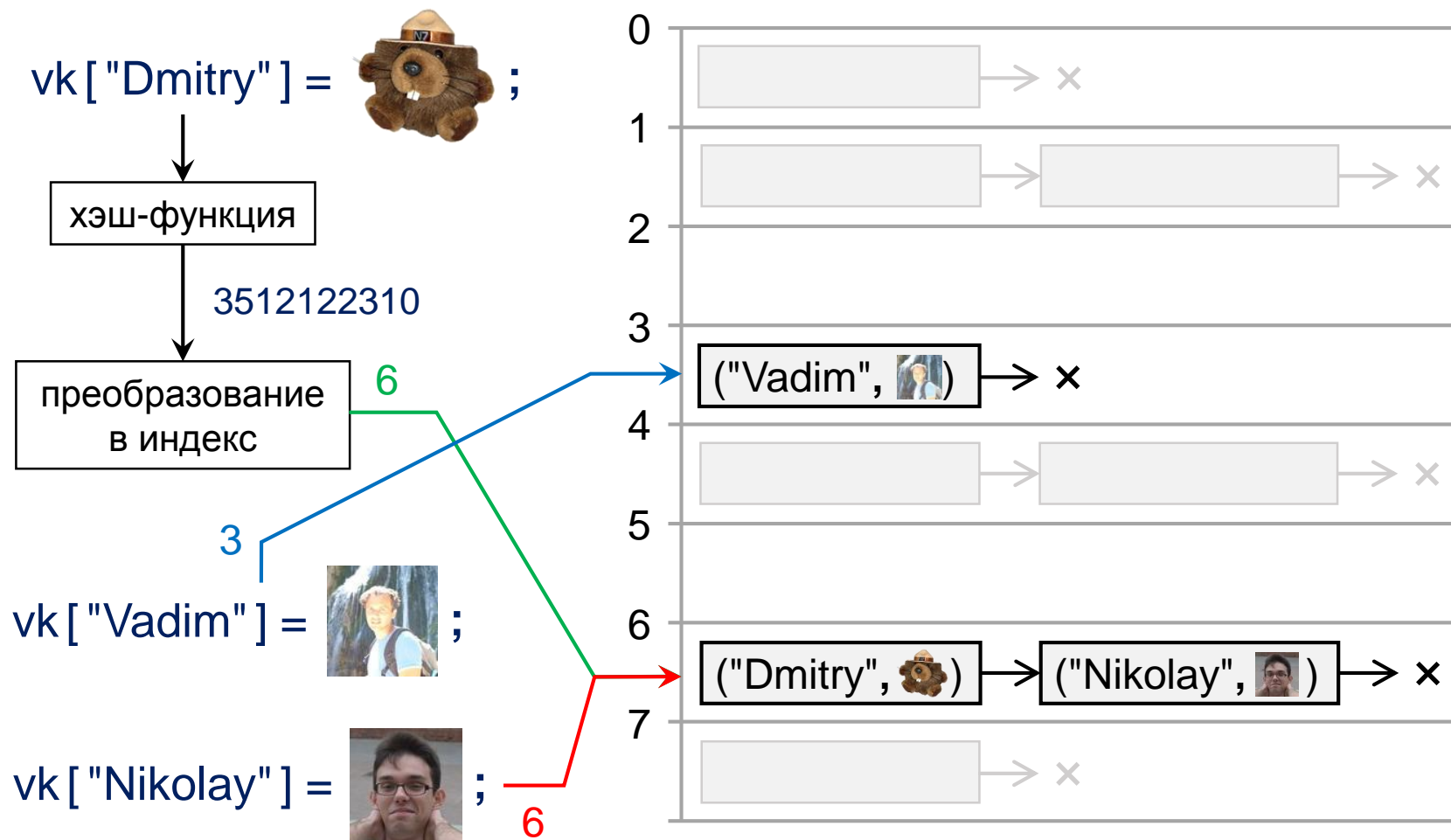
- Дополнительные коллизии

```
hash_function ( "Nikolay" ) % 8 == 6
```

```
int result = 7;  
for (unsigned int i = 0; i < input . size(); ++i)  
    result = 31 * result + input [ i ];
```




Устройство хэш-таблицы (прямая адресация)



Устройство хэш-таблицы (открытая адресация)

Наполнение таблицы

vk ["Vadim"] =  ;

3

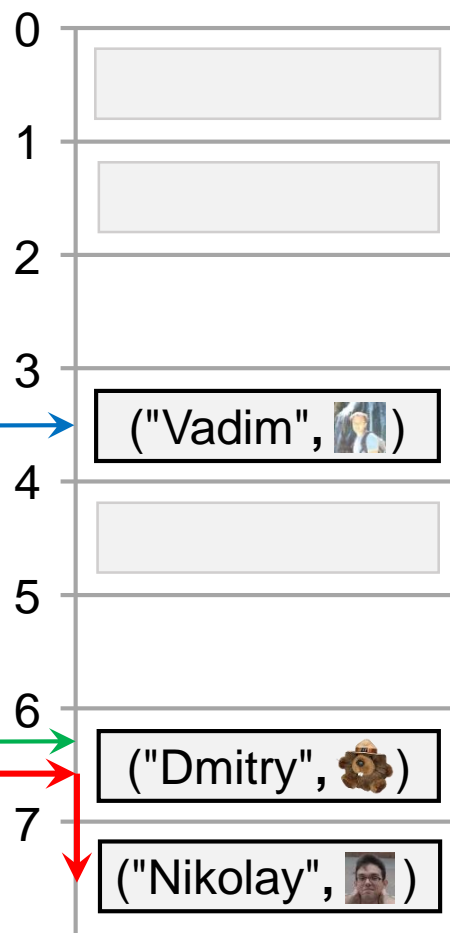
vk ["Dmitry"] =  ;

6

vk ["Nikolay"] =  ;

6

Занято!



Поиск

vk ["Nikolay"]

3682486214

6

Nikolay?

Nikolay!

Производительность хэш-таблиц

- Доступ по ключу: $\mathcal{O}(1)$
- Вставка и удаление элементов: $\mathcal{O}(1)$
- Скорость деградирует до $\mathcal{O}(N)$ из-за коллизий;
 - выбор хэш-функции критически важен!
 - При больших N скорость всегда немного снижается:
время доступа $\mathcal{O}(1) = \text{поиск ячейки } \mathcal{O}(1) + \text{поиск в коротком списке } \mathcal{O}(N)$.
- Существенный перерасход памяти под пустые ячейки $\mathcal{M}(N)$.
 - Больше ячеек \Rightarrow меньше коллизий \Rightarrow выше скорость.
 - Нужен баланс между расходом памяти и скоростью.

Литература к лекции

- Роберт Седжвик. *Фундаментальные алгоритмы на C++*.
- Томас Х. Кормен. *Алгоритмы. Вводный курс*.
- Томас Х. Кормен. *Алгоритмы: построение и анализ*.
- Стивен Скиена. *Алгоритмы. Руководство по разработке*.
- <http://sorting-algorithms.com>
- Programming: Principles and Practices Using C++:
 - раздел 20.4 — связанный список, алгоритмическая сложность;
 - раздел 20.6 — двоичное дерево, хэш-таблица.
- Авторский конспект 2013 г.
 - Лекция 5 «Алгоритмы сортировки»
 - Есть демонстрационная программа.
 - Лекция 6 «Динамические структуры данных»
 - Несколько опережает программу 2014 г. в отношении C++.

❖ Точные выходные сведения книг — на странице курса.