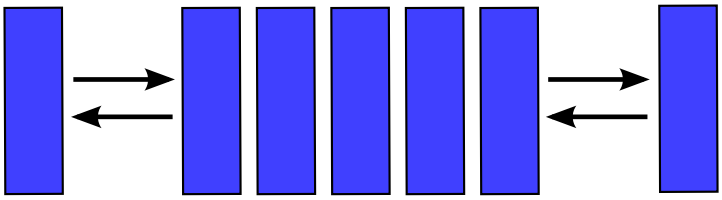
# Алгоритмы и структура данных

## std::deque и его представление

**Двусторонняя очередь** (англ. **deque**) — структура данных, в которой элементы можно добавлять и удалять как в начало, так и в конец, то есть дисциплинами обслуживания являются одновременно FIFO (First In, First Out — «первым пришёл — первым ушёл») и LIFO (Last In, First Out, «последним пришёл — первым ушёл»).



**Пример**

```
deque<int> Q;
Q.push_back(3);
Q.push_front(1);
Q.insert(Q.begin() + 1, 2);
Q[2] = 0;
copy(Q.begin(), Q.end(), ostream_iterator<int>(cout, " "));
// Результатом будет - 1 2 0
```

Ссылка на документацию

Ссылка на Википедию

---

## Stack как адаптер стандартного контейнера std::deque

Что же такое адаптер-контейнера? Это, по существу, устоявшийся паттерн, который приходит к нам из реального мира. Мы берём одну сущность и адаптируем её к новым условиям через другую сущность.

Почему они адапторы? Потому что внутри себя используют полноценный контейнеры, такие как `std::vector` и `std::deque`, и адаптируют их интерфейс под другой интерфейс. К примеру, `std::vector` имеет методы `push_back`, `insert` и `pop_back`, но такие операции для стека не нужны, стеку нужны 3 операции (основные) `top`, `pop` и `push`. Но все эти операции реализованы через соответсвующие операции `std::vector`, который скрывается в недрах `std::stack`. Поэтому stack не выделяют как самостоятельный контейнер — он есть адаптер для `std::vector`.

Итак, любая последовательность, поддерживающая операции `back`, `push_back` и `pop_back`, может использоваться для модификации stack. В частности, могут использоваться `vector`, `list` и `deque`.

```
template <class Container>
class stack {
friend bool operator==(const stack<Container>& x, const stack<Container>&
y);
friend bool operator<(const stack<Container>& x, const stack<Container>& y);
public:
    typedef Container::value_type value_type;
    typedef Container::size_type size_type;
protected:
    Container c;
public:
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    value_type& top() { return c.back(); }
    const value_type& top() const { return c.back(); }
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_back(); }
 };

template <class Container>
bool operator==(const stack <Container>& x, const stack<Container>& y)
    { return x.c == y.c;}

template <class Container>
bool operator<(const stack<Container>& x, const stack<Container>& y)
    { return x.c < y.c; }
```

Например, `stack<vector<int>>` - целочисленный стек, сделанный из vector, а `stack<deque<char>>` - символьный стек, сделанный из `deque`.

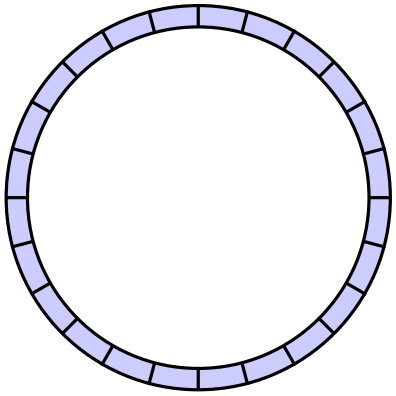
Ссылка на руководство

Ссылка на StackOverflow

---

# Циклический буфер (кольцо). Стандартная библиотека boost

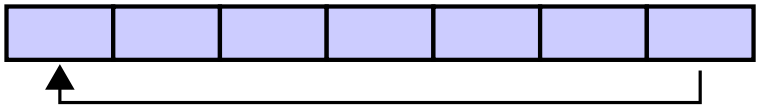
**Кольцевой буфер**, или **циклический буфер** — это структура данных, использующая единственный буфер фиксированного размера, как будто бы после последнего элемента сразу же снова идет первый. Такая структура легко предоставляет возможность буферизации потоков данных.



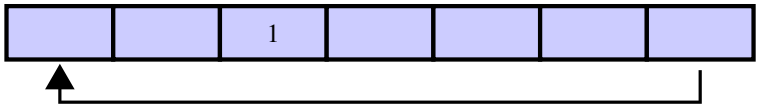
Кольцевой буфер. Иллюстрация визуально показывает, что у буфера нет настоящего конца. Тем не менее, поскольку физическая память никогда не делается закольцованной, обычно используется линейное представление, как показано ниже.

## Как он устроен

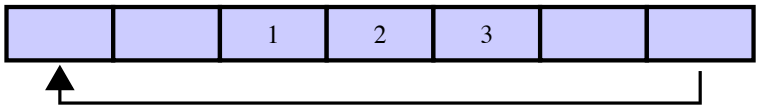
Кольцевой буфер создается пустым, с некоторой заранее определенной длиной. Например, это семиэлементный буфер:



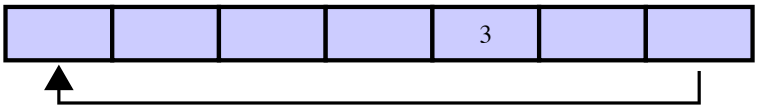
Предположим, что в середину буфера записывается 1 (в кольцевом буфере точная начальная ячейка не имеет значения):



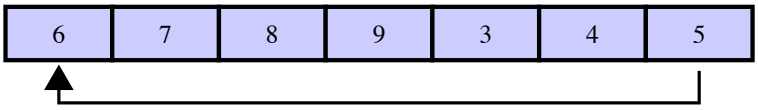

Затем предположим, что после единицы были добавлены ещё два элемента — 2 и 3:



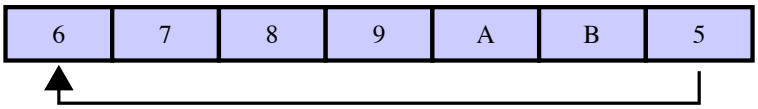
Если после этого два элемента должны быть удалены из буфера, то выбираются два наиболее старых элемента. В нашем случае удаляются элементы 1 и 2, в буфере остается только 3:




Если в буфере находится 7 элементов, то он заполнен:

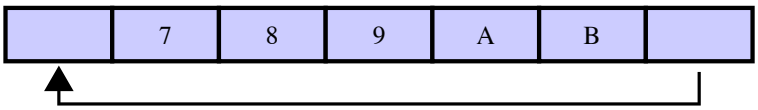



Если продолжить запись в буфер, не принимая во внимание его заполненность, то новые данные начнут перезаписывать старые данные. В нашем случае, добавляя элементы A и B, мы перезапишем 3 и 4:




В другом варианте реализации процедуры, обслуживающие буфер, могут предотвратить перезапись данных и вернуть ошибку или выбросить исключение. Перезапись или её отсутствие оставляется на усмотрение обслуживающих процедур буфера или приложения, использующего кольцевой буфер.

Наконец, если теперь удалить из буфера два элемента, то удалены будут не 3 и 4, а 5 и 6, потому что A и B перезаписали элементы 3 и 4; буфер придет в состояние:




## Библиотека Boost

circular_buffer специально разработана, чтобы обеспечить фиксированную емкость.

```
#include <boost/circular_buffer.hpp>

// Создать кольцевой буфер с емкостью для 3-х int'ов.
boost::circular_buffer<int> cb(3);

// Положим 3 элемента в буфер.
cb.push_back(1);
cb.push_back(2);
cb.push_back(3);

int a = cb[0];  // a == 1
int b = cb[1];  // b == 2
int c = cb[2];  // c == 3

// Сейчас буфер заполнен, при последующем добавленнии
// элементов более старые перезаписываются.

cb.push_back(4);  // Перезаписываем 1 добавляя 4.
cb.push_back(5);  // Перезаписываем 2 добавляя 5.

// Сейчас буфер содержит 3, 4 и 5.
a = cb[0];  // a == 3
b = cb[1];  // b == 4
c = cb[2];  // c == 5

// Элементы могут быть удалены с конца.
cb.pop_back();  // 5 удален.
cb.pop_front(); // 3 удален.

// Оставив только один элемент со значением = 4.
int d = cb[0];  // d == 4
```

Ссылка на Википедию

Ссылка на документацию кольцевого буфера в Boost

---

## Шаблон структуры std::pair

```
template<
    class T1,
    class T2
> struct pair;
```

*std::pair* является шаблоном структуры, который предоставляет возможность хранить два разнородных объекта, как единое целое.

**Структура pair определяется следующим образом:**

```
namespace std
{
   template <class T1, class T2>
   struct pair
   {
      // ...
      // здесь опущены определения типов,
      // реализации различных версий конструкторов
      // и метода swap
      // ...

      T1 first;
      T2 second;
   }
}

//Реализуем метод создания pair
template<class T1, class T2>
pair<T1, T2> make_pair(const T1&, const T2&) {
    return pair<T1, T2>(x, y);
}

std::pair<int, string> p = make_pair(22, "Moscow");

//Или

//Хранит пары "имя - средний балл аттестата"
std::map<string, double> pupils;
pupils.insert(make_pair("Ivanov", 4.5));
pupils.insert(make_pair("Petrov", 5.0));
```

**Более рабочий пример**

```cpp
#include <utility>
#include <string>
#include <complex>
#include <tuple>
#include <iostream>

int main()
{
    std::pair<int, float> p1;
    std::cout << "Value-initialized: "
              << p1.first << ", " << p1.second << '\n';

    std::pair<int, double> p2(42, 0.123);
    std::cout << "Initialized with two values: "
              << p2.first << ", " << p2.second << '\n';

    std::pair<char, int> p4(p2);
    std::cout << "Implicitly converted: "
              << p4.first << ", " << p4.second << '\n';

    std::pair<std::complex<double>, std::string> p6(
                    std::piecewise_construct,
                    std::forward_as_tuple(0.123, 7.7),
                    std::forward_as_tuple(10, 'a'));
    std::cout << "Piecewise constructed: "
              << p6.first << ", " << p6.second << '\n';
}
```

Ссылка на документацию

Ссылка на примеры

---

## Шаблон std::unordered_map. Bucket и chaining

```cpp
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```

*std::unordered_map* является ассоциативным контейнером, который содержит пары ключ-значение с уникальными ключами. Поиск, вставка и удаление выполняются за константное время.

Внутри, элементы не отсортированы в каком-либо определенном порядке, но организованы в блоки (buckets). Каждый блок элемента находится в полной зависимости от хэш-ключа. Это дает быстрый доступ к отдельным элементам, так как вычисляется хэш-код.

**Пример**

```
#include <unordered_map>
#include <vector>
#include <bitset>
#include <string>
#include <utility>

struct Key {
    std::string first;
    std::string second;
};

struct KeyHash {
 std::size_t operator()(const Key& k) const
 {
     return std::hash<std::string>()(k.first) ^
            (std::hash<std::string>()(k.second) << 1);
 }
};

struct KeyEqual {
 bool operator()(const Key& lhs, const Key& rhs) const
 {
    return lhs.first == rhs.first && lhs.second == rhs.second;
 }
};

int main()
{
    // default constructor: empty map
    std::unordered_map<std::string, std::string> m1;

    // list constructor
    std::unordered_map<int, std::string> m2 =
    {
        {1, "foo"},
        {3, "bar"},
        {2, "baz"},
    };

    // copy constructor
    std::unordered_map<int, std::string> m3 = m2;
```

```
    // move constructor
    std::unordered_map<int, std::string> m4 = std::move(m2);

    // range constructor
    std::vector<std::pair<std::bitset<8>, int>> v = { {0x12, 1}, {0x01,-1}
};
    std::unordered_map<std::bitset<8>, double> m5(v.begin(), v.end());

    // constructor for a custom type
    std::unordered_map<Key, std::string, KeyHash, KeyEqual> m6 = {
            { {"John", "Doe"}, "example"},
            { {"Mary", "Sue"}, "another"}
    };
}
```

**Коллизией хеш-функции** $H$ называется два различных входных блока данных $x$ и $y$ таких, что $H(x) = H(y)$.

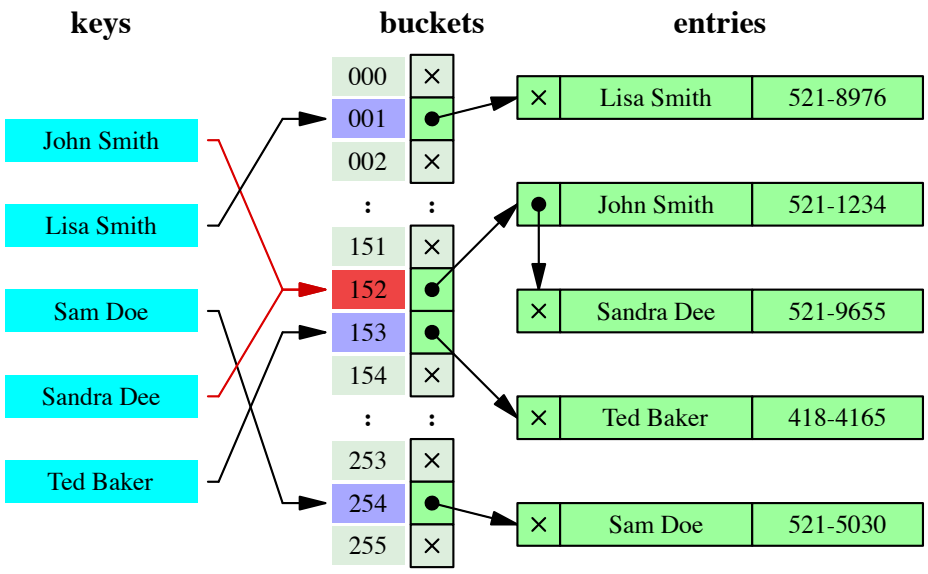
То есть, говоря своими словами, возможна такая ситуация, что хеш-фукнция может сгенерировать одинаковые хэш-ключи.

## Метод цепочек

Каждая ячейка массива $H$ является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента.

Операции поиска или удаления элемента требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом. Для добавления элемента нужно добавить элемент в конец или начало соответствующего списка, и, в случае, если коэффициент заполнения станет слишком велик, увеличить размер массива H и перестроить таблицу.

При предположении, что каждый элемент может попасть в любую позицию таблицы H с равной вероятностью и независимо от того, куда попал любой другой элемент, среднее время работы операции поиска элемента составляет $O(1 + \alpha)$, где $\alpha$ — коэффициент заполнения таблицы.




Ссылка на документацию

Ссылка на статью «Решение коллизий»

Ссылка на Википедию

---

# Фактор нагрузки, рехеширование и использование функции reserve

## Фактор нагрузки

Критическая статистика для хэш-таблицы - коэффициент нагрузки (**фактор нагрузки**), определяемый как

$$\text{load factor} = \frac{n}{k},$$

где

- $n$ это число entries (записей);
- $k$ это число buckets (блоков).

По мере того как коэффициент нагрузки становится больше, хэш-таблица становится медленнее, и это может привести к остановке работы (в зависимости от используемого метода). Ожидаемое константное время свойство хэш-таблицы предполагает, что коэффициент нагрузки поддерживается ниже некоторой границы. При фиксированном числе блоков (buckets), время для поиска растет с числом записей (entries) и, следовательно, желаемое котное время не достигается.

Второе, что можно исследовать, дисперсию числа записей в каждый блок. Например, две таблицы, обе имеют 1000 записей и 1000 блоков; ровно один вход в каждый блок, а другой имеет все записи в один и тот же блок. Очевидно, что хеширование не работает во втором блоке.

Низкий коэффициент нагрузки не очень полезен. Когда коэффициент нагрузки приближается к 0, доля неиспользуемых площадей в хэш-таблице увеличивается, но это не обязательно любое снижение «стоимости» поиска. Это приводит к потере памяти.

## Метод повторного хеширования (рехеширование)

Пусть результатом вычисления хеш-функции для некоторого имени $S$ является число $h$.
Пусть элемент с этим инд-м $h$ уже занят, т.е. является не пустым, возникает коллизия, которую надо устранить путём выбора другой ячейки таблицы для имени $S$.
Выбор такой ячейки производится:
$h_1 = (h + p_1) \mod N$, $p_1$ – некоторое приращение

Если элемент таблицы $h_1$ тоже не пустой, то рассматривается новый элемент:
$h_2 = (h + p_2) \mod N$

---

$h_i = (h + p_i) \mod N$, до тех пор пока не будет найден элемент таблицы, что:

1. элемент пустой, тогда имя $S$ в таблице отсутствует и записывается в таблице под инд. $h_i$.
2. элемент с индексом $h_i$ является не пустым и содержит имя, совпадающее с именем $h_S$ , в этом случае хешир-е имя уже есть в таблице и не должно быть записано в таблице.
3. $h_i = h$, т.е. таблица имен заполнена, т.е. нет пустых ячеек.
   В этом случае $S$ нельзя записать в таблицу и процесс трансляции прекращён.

Операция деления по $\mod N$ при вычислении $h_i$ обеспечивает циклический просмотр адресов. С точки зрения эффективности организации хеш-табл. следует отметить, что время поиска элемента определяется количеством возникших коллизий. Время вычисления хеш-функции является постоянной величиной, а количество коллизий должно быть уменьшено путем выбора соответствующего способа **рехеширования**.
Чтобы уменьшить количество возникших коллизий приращение $p_i$ должно вычисляться с учётом 2-х условий:

- ожидаемое число сравнения элемента $S$ др. элементами таблицы должно быть минимальным.
- значение адресов $h_i$ должно равномерно распределяться по таблице.
  В идеал. случае значение $p_i$ должно охватывать целые числа из $[1, N-1]$ строго 1 раз.
  Различ. варианты повторного хеширования, отличается способом вычисления $p_i$.

## Функции reserve

```
void reserve( size_type size );
```

Задает емкость контейнера, по крайней мере `size`. Новая память выделяется при необходимости.

В описании контейнера `vector` можно встретить такие функции как `size` и `resize`, `capacity` и `reserve`. С первого взгляда кажется, что половина явно лишняя, ведь размер у вектора может быть только один.
Почему так? Потому что память для вектора выделяется "про запас", и надо уметь управлять и количеством элементов вектора, и количеством памяти, которое он занимает. `size` и `resize` - нужны для работы с реальным числом элементов вектора, `capacity` и `reserve` - для работы с памятью.

`size` - выдает количество элементов в векторе

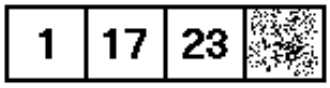
`resize` - изменяет количество элементов в векторе

`capacity` - выдает под сколько элементов выделена память

`reserve` - резервиует память

```
vector <int> v;
v.push_back(1); //size==1, capacity==1
v.push_back(17); //size==2, capacity==2
v.push_back(23); //size==3, capacity==4
```

`vector` будет выглядеть вот так:




Зачем нужен этот запас?
Допустим, у нас есть вектор из $n$ элементов. Что происходит, когда программист добавляет еще один? Когда есть запасная память, элемент пишется туда. Когда ее нет, выделяется непрерывный кусок памяти достаточный для $n * K$ элементов, где $K$ - коэффицент. В него копируются предыдущие $n$, добавляется наш новый элемент, старый кусок размером $n$ освобождается. Если бы запаса не было, то память бы выделялась каждый раз при добавлении нового элемента, что страшно неэффективно.
Зачем нужен `reserve`, если все и без участия программиста так хорошо работает? Это может быть полезно в некоторых ситуациях. Например, знание того, как выделяется память под вектор, можно использовать и в начале, при его задании. Допустим, я точно знаю, что в векторе будет где-то 100-110 элементов. Я сразу же создам вектор размером 110, это поможет избежать перевыделений памяти, что положительно скажется на производительности.

```
vector <int> v;
v.reserve(110);
```

Это вовсе не означает, что зарезервировано место для 110-ти элементов ровно. Зарезервировано место как минимум для 110 элементов, возможно больше.

Link on Wikipedia

Хорошее объяснение reserve

Ссылка на документацию по reserve

Ссылка про рехеширование

---

## Хеширование: линейное, двойное, квадратичное пробирование, folding

Ниже приведены некоторые распространенные типы последовательностей проб. Сразу оговорим, что нумерация элементов последовательности проб и ячеек хеш-таблицы ведётся от нуля, а $N$ — размер хеш-таблицы (и, как замечено выше, также и длина последовательности проб).

- **Линейное пробирование**: ячейки хеш-таблицы последовательно просматриваются с некоторым фиксированным интервалом k между ячейками (обычно $k = 1$), то есть $i$-й элемент последовательности проб — это ячейка с номером $(hash(x) + ik) \mod N$. Для того, чтобы все ячейки оказались просмотренными по одному разу, необходимо, чтобы k было взаимно-простым с размером хеш-таблицы.
- **Квадратичное пробирование**: интервал между ячейками с каждым шагом увеличивается на константу. Если размер хеш-таблицы равен степени двойки ($N = 2p$), то одним из примеров последовательности, при которой каждый элемент будет просмотрен по одному разу, является:
$hash(x) \mod N, (hash(x) + 1) \mod N, (hash(x) + 3) \mod N, (hash(x) + 6) \mod N$, ...
- **Двойное хеширование**: интервал между ячейками фиксирован, как при линейном пробировании, но, в отличие от него, размер интервала вычисляется второй, вспомогательной хеш-функцией, а значит, может быть различным для разных ключей. Значения этой хеш-функции должны быть ненулевыми и взаимно-простыми с размером хеш-таблицы, что проще всего достичь, взяв простое число в качестве размера, и потребовав, чтобы вспомогательная хеш-функция принимала значения от $1$ до $N - 1$.

Другим методом создания хеш-функции называется сверткой (**folding**) и основан на выполнении некоторых арифметических действий над различными частями поля хеширования. При этом символьные строки преобразуются в целые числа с использованием некоторой кодировки (на основе расположения букв в алфавите или кодов символов ASCII). Например, можно преобразовать в целое число первые два символа поля табельного номера сотрудника (атрибут staffNo), а затем сложить полученное значение с остальными цифрами этого номера. Вычисленная сумма используется в качестве адреса дисковой страницы, на которой будет храниться данная запись. Более популярный альтернативный метод основан на хешировании с

применением остатка от деления. В этом методе используется функция MOD, которой передается значение поля. Функция делит полученное значение на некоторое заранее заданное целое число, после чего остаток от деления используется в качестве адреса на диске.

Ссылка на статью «Решение коллизий»

Ссылка на неплохое объяснение

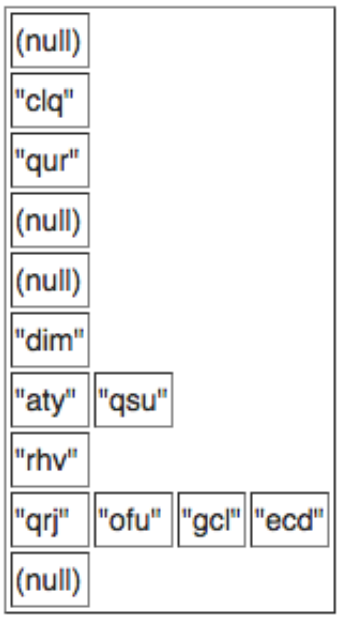
Ссылка на Википедию

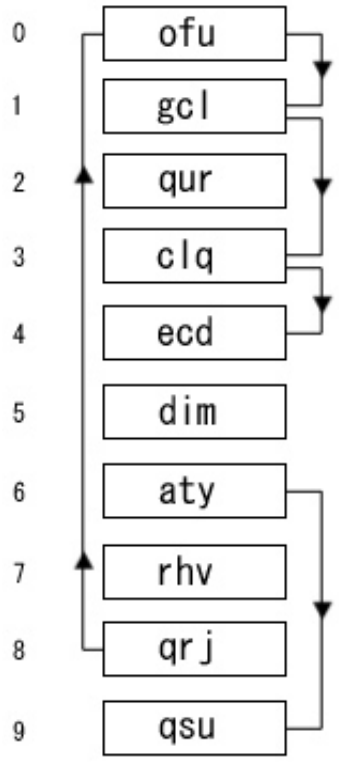
---

# Coalesced hashing

**Coalesced hashing**, также называемая **coalesced chaining**, это стратегия решения коллизий в хеш-таблицах, что образует смесь *раздельных цепочек* и *открытой адресации*. В хеш-таблице *раздельных цепочек*, элементы этого хеша размещены по тому же адресу в списке или "цепочке" по этому адресу. Этот метод может привести к большой потери памяти, потому что сама хеш-таблица должна быть достаточно большой чтобы поддерживать коэффициент нагрузки (фактор нагрузки), который также выполняется (обычно в два раза ожидаемого числа элементов), и дополнительная память должна быть использована для всех, но с первого пункта в цепочке (за исключением списка используемых заголовков, в случае выделения дополнительной памяти должны применяться для всех элементов в цепи).

Дана случайно сгенерированные трех-буквенные последовательности строк "qrj," "aty," "qur," "dim," "ofu," "gcl," "rhv," "clq," "ecd," "qsu", в следую бщей таудут генерироваться (используя алгоритм Боба Дженкинса) таблица с размером 10:






*Пример Coalesced Hashing. Для целей этого примера, коллизии блоков распределяются в порядке возрастания, начиная с блока 0.*

Данная стратегия эффективна, и очень легко реализовывается. Однако, иногда использование дополнительной памяти может быть запрещено, а самая распространенная альтернатива, открытая адресация, имеет неудобные недостатки, снижающие производительность. Основной недостаток открытой адресации является первичная и вторичная кластеризация, в которых поиски могут открыть длинные последовательности используемых сегментов, содержащих элементы с различными хеш-адресами; таким образом, элементы с хеш-адресами могут увеличить время поисков предметов с другими хеш-адресоми.

Одно такое решение и есть coalesced hashing. Coalesced hashing использует похожую технику с *раздельными цепочками*, but instead of allocating new nodes for the linked list, buckets in the actual table are used. The first empty bucket in the table at the time of a collision is considered the collision bucket. When a collision occurs anywhere in the table, the item is placed in the collision bucket and a link is made between the chain and the collision bucket. It is possible for a newly inserted item to collide with items with a different hash address, such as the case in the example above when item "clq" is inserted. The chain for "clq" is said to "coalesce" with the chain of "qrj," hence the name of the algorithm. However, the extent of coalescing is minor compared with the clustering exhibited by open addressing. For example, when coalescing occurs, the length of the chain grows by only 1, whereas in open addressing, search sequences of arbitrary length may combine.

Важная оптимизация, чтобы уменьшить влияние объединения, нужно ограничить адресное пространство хэш-функции только подмножества хеш-таблицы. Для примера, если таблица имеет размер $M$ with buckets numbered from 0 to M − 1, we can restrict the address space so that the hash function only assigns addresses to the first N locations in the table. The remaining M − N buckets, called the cellar, are used exclusively for storing items that

collide during insertion. No coalescing can occur until the cellar is exhausted.

Оптимальный выбор $N$ относительно $M$ depends upon the load factor (or fullness) of the table. A careful analysis shows that the value $N = 0.86 \times M$ yields near-optimum performance for most load factors. Other variants for insertion are also possible that have improved search time. Deletion algorithms have been developed that preserve randomness, and thus the average search time analysis still holds after deletions.
Вставки в C:

```
/* htab это хеш-таблица,
   N это размер адресного пространства хеш-функции, и
   M is the size of the entire table including the cellar.
   Collision buckets are allocated in decreasing order, starting with bucket
M-1. */

int insert ( char key[] )
{
  unsigned h = hash ( key, strlen ( key ) ) % N;

  if ( htab[h] == NULL ) {
    /* Создает новую цепь */
    htab[h] = make_node ( key, NULL );
  } else {
    struct node *it;
    int cursor = M-1;

    /* Находит первый пустой блок */
    while ( cursor >= 0 && htab[cursor] != NULL )
      --cursor;

    /* Таблица заполнена, неудачное завершение */
    if ( cursor == -1 )
      return -1;

    htab[cursor] = make_node ( key, NULL );

    /* Нахождение последнего узла в цепочке и указатель на него */
    it = htab[h];

    while ( it->next != NULL )
      it = it->next;

    it->next = htab[cursor];
  }

  return 0;
}
```

Одно из преимуществ этой стратегии является то, что алгоритм поиска для разделеных цепочек может использоваться без изменений в объединенной хэш-таблице.

```
char *find ( char key[] )
{
  unsigned h = hash ( key, strlen ( key ) ) % N;

  if ( htab[h] != NULL ) {
    struct node *it;

    /* Поиск цепочки с индексом h */
    for ( it = htab[h]; it != NULL; it = it->next ) {
      if ( strcmp ( key, it->data ) == 0 )
        return it->data;
    }
  }

  return NULL;
}
```

#### Производительность

Coalesced chaining avoids the effects of primary and secondary clustering, and as a result can take advantage of the efficient search algorithm for separate chaining. If the chains are short, this strategy is very efficient and can be highly condensed, memory-wise. As in open addressing, deletion from a coalesced hash table is awkward and potentially expensive, and resizing the table is terribly expensive and should be done rarely, if ever.

Link on Wikipedia

---

# Понимание алгоритмов сортировки

Визуализатор алгоритмов

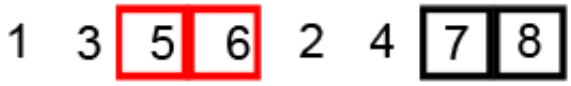
## Пузырьковая сортировка (Bubble sort)

**Пузырьковая сортировка** — простой алгоритм сортировки. Для понимания и реализации этот алгоритм — простейший, но эффективен он лишь для небольших массивов. Сложность алгоритма: $O(n^2)$.

#### Алгоритм

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются $N-1$ раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на

одну позицию к началу массива («всплывает» до нужной позиции, как пузырёк в воде, отсюда и название алгоритма).




**Реализация на C++**

```
void bubble_sort(int *a, int length)
{
     for (int i = 0; i < length-1; i++) {
        bool swapped = false;
         for (int j = 0; j < length-i-1; j++) {
             if (a[j] > a[j + 1]) {
                 int b = a[j];
                 a[j] = a[j + 1];
                 a[j + 1] = b;
                 swapped = true;
             }
         }

         if(!swapped)
            break;
     }
 }
```

Ссылка на Википедию

## Сортировка расчёской (Comb sort)

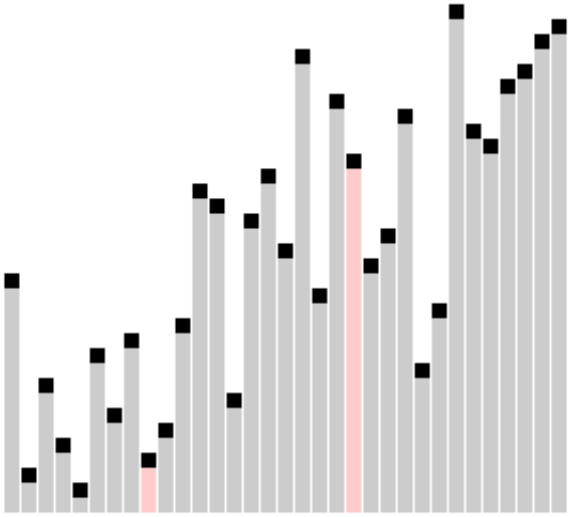
**Сортировка расчёской** улучшает сортировку пузырьком, и конкурирует с алгоритмами, подобными быстрой сортировке. Основная идея — устранить *черепах*, или маленькие значения в конце списка, которые крайне замедляют сортировку пузырьком (*кролики*, большие значения в начале списка, не представляют проблемы для сортировки пузырьком).

В сортировке пузырьком, когда сравниваются два элемента, промежуток (расстояние друг от друга) равен 1. Основная идея сортировки расчёской в том, что этот промежуток может быть гораздо больше, чем единица (сортировка Шелла также основана на этой идее, но она является модификацией сортировки вставками, а не сортировки пузырьком).

**Алгоритм**

В «пузырьке», «шейкере» и «чёт-нечете» при переборе массива сравниваются соседние элементы. Основная идея «расчёски» в том, чтобы первоначально брать достаточно большое расстояние между сравниваемыми элементами и по мере упорядочивания массива сужать это расстояние вплоть до минимального. Таким образом, мы как бы причёсываем массив, постепенно разглаживая на всё более аккуратные пряди. Первоначальный разрыв между сравниваемыми элементами лучше брать с учётом специальной величины, называемой фактором уменьшения, оптимальное значение которой равно примерно $1,247$. Сначала расстояние между элементами равно размеру массива, разделённого на фактор уменьшения (результат округляется до ближайшего целого). Затем, пройдя массив с этим шагом, необходимо поделить шаг на фактор уменьшения и пройти по списку вновь. Так продолжается до тех пор, пока разность индексов не достигнет единицы. В этом случае массив досортировывается обычным пузырьком.

Оптимальное значение фактора уменьшения $1,247...$ можно представить формулой в следующем виде $\approx \frac{1}{1-e^{(-\phi)}}$, где $e$ - экспонента; $\phi$ - «золотое» число.



**Реализация на C++**

```
int comb(vector<double> sort)
{
    int n = 0; // количество перестановок
    double fakt = 1.2473309; // фактор уменьшения
    int step = sort.size() - 1;
    bool swapped = true;
    while (step > 1)
    {
        swapped = false;
        for (int i = 0; i + step < sort.size(); ++i)
        {
            if (sort[i] > sort[i + step])
            {
                swap(sort[i], sort[i + step]);
                swapped = true;
                n++;
            }
        }
        step /= fakt;
    }

    // сортировка пузырьком
    for (int i = 0; i < sort.size() - 1; i++)
    {
        bool swapped = false;
        for (int j = 0; j < sort.size() - i - 1; j++)
        {
            if (sort[j] > sort[j + 1]) {
                swap(sort[j], sort[j + 1]);
                swapped = true;
                ++n;
            }
        }

        if (!swapped)
            break;

        }
        return n;
    }
```

Ссылка на Википедию

### Сортировка подсчётом (Counting sort)

Ссылка на Википедию

### Сортировка кучей (Heapsort)

**Сортировка кучей** (или «Пирамидальная сортировка») — алгоритм сортировки, работающий в худшем, в среднем и в лучшем случае (то есть гарантированно) за $O(n \log n)$ операций при сортировке n элементов. Количество применяемой служебной памяти не зависит от размера массива (то есть, $O(1)$).

Может рассматриваться как усовершенствованная сортировка пузырьком, в которой элемент всплывает (min-heap) / тонет (max-heap) по многим путям.

**Алгоритм**

Сортировка пирамидой использует бинарное сортирующее дерево. Сортирующее дерево — это такое дерево, у которого выполнены условия:

1. Каждый лист имеет глубину либо $d$, либо $d - 1$, $d$ — максимальная глубина дерева. Значение в любой вершине не меньше (другой вариант — не больше) значения её потомков.
2. Удобная структура данных для сортирующего дерева — такой массив `Array`, что `Array[1]` — элемент в корне, а потомки элемента `Array[i]` являются `Array[2i+1]` и `Array[2i+2]`.

Алгоритм сортировки будет состоять из двух основных шагов:

- Выстраиваем элементы массива в виде сортирующего дерева:

  $$\mathbf{Array}[i] \geq \mathbf{Array}[2i + 1]$$

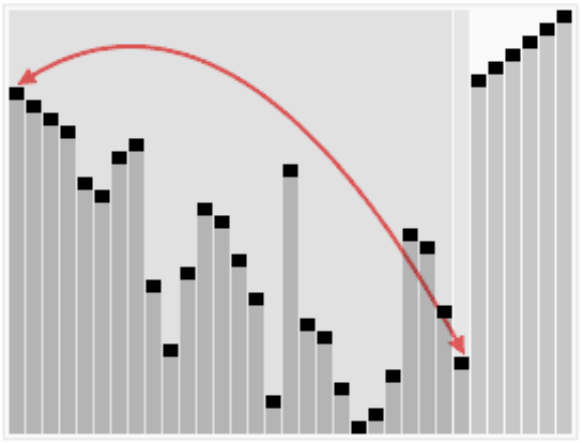
  $$\mathbf{Array}[i] \geq \mathbf{Array}[2i + 2]$$

  при $1 \leq i < n/2$.

Этот шаг требует $O(n)$ операций.

- Будем удалять элементы из корня по одному за раз и перестраивать дерево. То есть на первом шаге обмениваем Array[1] и Array[n], преобразовываем `Array[1]`, `Array[2]`, … , `Array[n-1]` в сортирующее дерево. Затем переставляем `Array[1]` и `Array[n-1]`, преобразовываем `Array[1]`, `Array[2]`, … , `Array[n-2]` в сортирующее дерево. Процесс продолжается до тех пор, пока в сортирующем дереве не останется один элемент. Тогда `Array[1]`, `Array[2]`, … , `Array[n]` — упорядоченная последовательность.

Этот шаг требует $O(n \log n)$ операций.



**Реализация на C++**

```
template <typename T>
void ShiftDown(T a[], int i, int j)
{
    int MaxNodeId;
    bool ShiftDone = false;
    while (((i * 2 + 1) < j) && !ShiftDone)
    {
        if (i * 2 + 1 == j - 1 || a[i * 2 + 1] > a[i * 2 + 2])
        {
            MaxNodeId = i * 2 + 1;
        }
        else
        {
            MaxNodeId = i * 2 + 2;
        }

        if (a[i] < a[MaxNodeId])
        {
            std::swap(a[i], a[MaxNodeId]);
            i = MaxNodeId;
        }
        else
        {
            ShiftDone = true;
        }
    }
}

template <typename T>
void HeapSort(T a[], int l)
{
    int i;
    //Строим дерево поиска
    for (i = (l / 2) - 1; i > -1; i--)
    {
        ShiftDown(a, i, l);
    }

    //Забираем максимальный (0) элемент дерева в i-ю позицию
    //Перемещаем новый 0 элемент на правильную позицию в дереве
    for (i = l - 1; i > 0; i--)
    {
        std::swap(a[0], a[i]);
        ShiftDown(a, 0, i);
    }
}
```

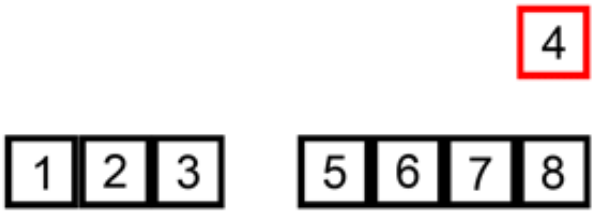
Ссылка на Википедию

## Сортировка вставками (Insertion sort)

**Сортировка вставками** — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. Основным преимуществом алгоритма сортировки вставками является возможность сортировать массив по мере его получения.То есть имея часть массива, можно начинать его сортировать. В параллельном программирование такая особенность играет не маловажную роль. Вычислительная сложность — $O(n^2)$.

### Алгоритм

Сортируемый массив можно разделить на две части — отсортированная часть и неотсортированная. В начале сортировки первый элемент массива считается отсортированным, все остальные — не отсортированные. Начиная со второго элемента массива и заканчивая последним, алгоритм вставляет неотсортированный элемент массива в нужную позицию в отсортированной части массива. Таким образом, за один шаг сортировки отсортированная часть массива увеличивается на один элемент, а неотсортированная часть массива уменьшается на один элемент. Например, когда мы тянем карту из колоды, смотрим на наши разложенные по возрастанию карты и в зависимости от достоинства вытянутой карты помещаем карту в соответствующее место.




### Реализация на C++

```
void Sort(int* arr, int n)
{
    int counter = 0;
    for(int i = 1; i < n; i++)
    {
        for(int j = i; j > 0 && arr[j - 1] > arr[j]; j--)
        {
            counter++;
            int tmp = arr[j - 1];
            arr[j - 1] = arr[j];
            arr[j] = tmp;
        }
    }
    cout << counter << endl;
}
```

Ссылка на Википедию

Ссылка на Habrahabr

## Сортировка слиянием (Merge sort)

**Сортировка слиянием** — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

**Алгоритм**

Для решения задачи сортировки эти три этапа выглядят так:

1. Сортируемый массив разбивается на две части примерно одинакового размера;
2. Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
3. Два упорядоченных массива половинного размера соединяются в один.

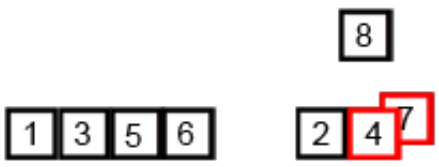
1.1. — 2.1. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).

3.1. Соединение двух упорядоченных массивов в один.

Основную идею слияния двух отсортированных массивов можно объяснить на следующем примере. Пусть мы имеем два уже отсортированных по возрастанию подмассива. Тогда:

3.2. Слияние двух подмассивов в третий результирующий массив.
На каждом шаге мы берём меньший из двух первых элементов подмассивов и записываем его в результирующий массив. Счётчики номеров элементов результирующего массива и подмассива, из которого был взят элемент, увеличиваем на 1.

3.3. «Прицепление» остатка.
Когда один из подмассивов закончился, мы добавляем все оставшиеся элементы второго подмассива в результирующий массив.




**Реализация на C++**

```
template <typename Item>
void Merge(Item Mas[], int left, int right, int medium)
{
    int j = left;
    int k = medium + 1;
    int count = right - left + 1;

    if (count <= 1) return;

    Item *TmpMas = new Item[count];

    for (int i = 0; i < count; ++i) {
        if (j <= medium && k <= right) {
            if (Mas[j] < Mas[k])
                TmpMas[i] = Mas[j++];
            else
                TmpMas[i] = Mas[k++];
        } else {
            if (j <= medium)
                TmpMas[i] = Mas[j++];
            else
                TmpMas[i] = Mas[k++];
        }
    }

    j = 0;
    for (int i = left; i <= right; ++i) {
        Mas[i] = TmpMas[j++];
    }
    delete[] TmpMas;
}

template <typename Item>
void MergeSort(Item a[], int l, int r)
{
    int m;

    // Условие выхода из рекурсии
    if(l >= r) return;

    m = (l + r) / 2;

    // Рекурсивная сортировка полученных массивов
    MergeSort(a, l, m);
    MergeSort(a, m + 1, r);
    Merge(a, l, r, m);
}
```

Ссылка на Википедию

## Быстрая сортировка (Quicksort)

**Быстрая сортировка** — широко известный алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром во время его работы в МГУ в 1960 году.

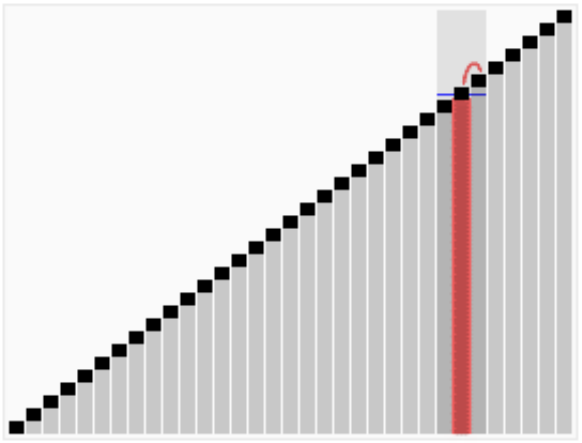
Один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем $O(n \log n)$ обменов при упорядочении $n$ элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

### Алгоритм

Быстрая сортировка использует стратегию «разделяй и властвуй». Шаги алгоритма таковы:

1. Выбираем в массиве некоторый элемент, который будем называть опорным элементом. Для корректности алгоритма значение этого элемента должно быть между максимальным и минимальным значениями в массиве (включительно). С точки зрения повышения эффективности алгоритма выгоднее всего выбирать медиану; но без дополнительных сведений о сортируемых данных её обычно невозможно получить. Известные стратегии: выбирать постоянно один и тот же элемент, например, средний или последний по положению; выбирать элемент со случайно выбранным индексом. Часто хороший результат даёт выбор в качестве опорного элемента среднего арифметического между минимальным и максимальным элементами массива, особенно для целых чисел (в этом случае опорный элемент не обязан быть элементом сортируемого массива).
2. Операция разделения массива: реорганизуем массив таким образом, чтобы все элементы со значением меньшим или равным опорному элементу, оказались слева от него, а все элементы, превышающие по значению опорный — справа от него. Обычный алгоритм операции:
3. Два индекса — l и r, приравниваются к минимальному и максимальному индексу разделяемого массива, соответственно.
4. Вычисляется значение опорного элемента m по одной из стратегий.
5. Индекс l последовательно увеличивается до тех пор, пока l-й элемент не окажется больше или равен опорному.
6. Индекс r последовательно уменьшается до тех пор, пока r-й элемент не окажется меньше или равен опорному.
7. Если r = l — найдена середина массива — операция разделения закончена, оба индекса указывают на опорный элемент.
8. Если l < r — найденную пару элементов нужно обменять местами и продолжить операцию разделения с тех значений l и r, которые были достигнуты. Следует учесть, что если какая-либо граница (l или r) дошла до опорного элемента, то при обмене значение m изменяется на r-й или l-й элемент соответственно, изменяется именно индекс опорного элемента и алгоритм продолжает своё выполнение.
9. Рекурсивно упорядочиваем подмассивы, лежащие слева и справа от опорного элемента.
10. Базой рекурсии являются наборы: пустой или состоящий из одного элемента, которые возвращаются в исходном виде. Все такие отрезки уже упорядочены в процессе разделения.

Поскольку в каждой итерации (на каждом следующем уровне рекурсии) длина обрабатываемого отрезка массива уменьшается, по меньшей мере, на единицу, терминальная ветвь рекурсии будет достигнута обязательно, и обработка гарантированно завершится.



**Реализация на C++**

```
void SortAlgo::quickSort(int* data, int const len)
{
  int const lenD = len;
  int pivot = 0;
  int ind = lenD/2;
  int i,j = 0,k = 0;
  if(lenD > 1){
    int* L = new int[lenD];
    int* R = new int[lenD];
    pivot = data[ind];
    for(i = 0; i < lenD; i++){
      if(i != ind){
        if(data[i] < pivot){
          L[j] = data[i];
          j++;
        }
        else {
          R[k] = data[i];
          k++;
        }
      }
    }
    quickSort(L, j);
    quickSort(R, k);
    for(int cnt = 0; cnt < lenD; cnt++){
      if(cnt < j){
        data[cnt] = L[cnt];;
      }
      else if(cnt == j){
        data[cnt] = pivot;
      }
      else{
        data[cnt] = R[cnt-(j+1)];
      }
    }
  }
}
```

Ссылка на Википедию

## Поразрядная сортировка (Radix sort)

Ссылка на Википедию

## Сортировка выбором (Selection sort)

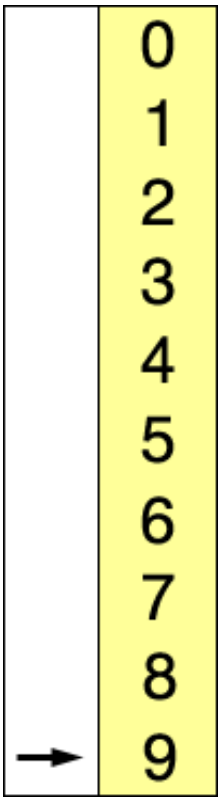
**Сортировка выбором** — алгоритм сортировки. Может быть как устойчивый, так и неустойчивый. На массиве из $n$ элементов имеет время выполнения в худшем, среднем и лучшем случае $O(n^2)$, предполагая что сравнения делаются за постоянное время

**Алгоритм**

Шаги алгоритма:

1. находим номер минимального значения в текущем списке
2. производим обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции)
3. теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы

Для реализации устойчивости алгоритма необходимо в пункте 2 минимальный элемент непосредственно вставлять в первую неотсортированную позицию, не меняя порядок остальных элементов.




**Реализация на C++**

```cpp
#include <cstddef>
#include <utility>

template<typename T>
void selection_sort(T array[], std::size_t size)
{
    for (std::size_t idx_i = 0; idx_i < size - 1; idx_i++)
    {
        std::size_t min_idx = idx_i;
        for (std::size_t idx_j = idx_i + 1; idx_j < size; idx_j++)
        {
            if (array[idx_j] < array[min_idx])
            {
                min_idx = idx_j;
            }
        }

        if (min_idx != idx_i)
        {
            std::swap(array[idx_i], array[min_idx]);
        }
    }
}
```

Ссылка на Википедию

## Сортировка Шелла (Shell sort)

**Сортировка Шелла** — алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Дональда Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга; иными словами — это сортировка вставками, но с предварительными «грубыми» проходами.

Невзирая на то, что сортировка Шелла во многих случаях медленнее, чем быстрая сортировка — она имеет ряд преимуществ:

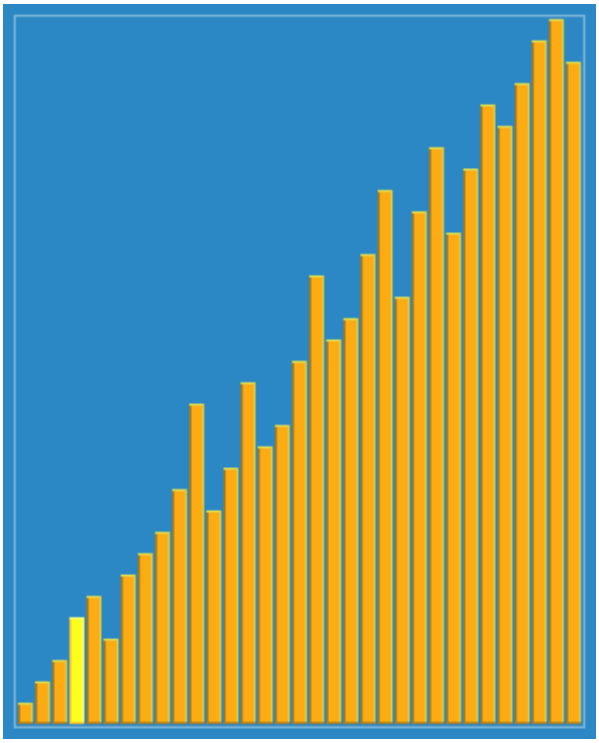
- Отсутствие потребности в памяти под стек;
- Отсутствие деградации при неудачных наборах данных — быстрая сортировка легко деградирует до $O(n^2)$, что хуже, чем худшее гарантированное время для сортировки Шелла.

### Алгоритм

Этапы алгоритма:

1. **Инициализация:** задание начального значения $d$ — которое впоследствии будет уменьшаться;
2. **Основная часть:**
3. Упорядочение массива:

    1. Выбор пары элементов, индексы которых отличаются на $d$;
    2. Определение разности элементов выбранной пары;
    3. Упорядочение элементов в рамках пары согласно разности значений;
    4. Переход к следующей паре;
4. Уменьшение значения $d$;
5. **Проверка:** если $d > 0$ — вернуться к пункту "1" основной части (и если $d = 1$ — для проведения обычной сортировки вставками);
6. **Завершение работы** (например, вывод результатов на экран).
7. Эффективность сортировки Шелла, — в определённых случаях, — обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, — например, пузырьковой, — каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, а при сортировке Шелла — это число может быть больше).



Сортировка с шагами 23, 10, 4, 1.

**Пример**

| Исходный массив | 32 | 95 | 16 | 82 | 24 | 66 | 35 | 19 | 75 | 54 | 40 | 43 | 93 | 68 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| После сортировки с шагом 5 | 32 | 35 | 16 | 68 | 24 | 40 | 43 | 19 | 75 | 54 | 66 | 95 | 93 | 82 | 6 обменов |
| После сортировки с шагом 3 | 32 | 19 | 16 | 43 | 24 | 40 | 54 | 35 | 75 | 68 | 66 | 95 | 93 | 82 | 5 обменов |
| После сортировки с шагом 1 | 16 | 19 | 24 | 32 | 35 | 40 | 43 | 54 | 66 | 68 | 75 | 82 | 93 | 95 | 15 обменов |

Пусть дан список $A = (32, 95, 16, 82, 24, 66, 35, 19, 75, 54, 40, 43, 93, 68)$ и выполняется его сортировка методом Шелла, а в качестве значений $d$ выбраны $5, 3, 1$.

На первом шаге сортируются подсписки $A$, составленные из всех элементов $A$, различающихся на 5 позиций, то есть подсписки $A_{5,1} = (32, 66, 40)$, $A_{5,2} = (95, 35, 43)$, $A_{5,3} = (16, 19, 93)$, $A_{5,4} = (82, 75, 68)$, $A_{5,5} = (24, 54)$.

В полученном списке на втором шаге вновь сортируются подсписки из отстоящих на 3 позиции элементов.

Процесс завершается обычной сортировкой вставками получившегося списка.

**Реализация на C++**

```
int increment(long inc[], long size) {
// inc[] массив, в который заносятся инкременты
// size размерность этого массива
 int p1, p2, p3, s;

  p1 = p2 = p3 = 1;
  s = -1;
  do {// заполняем массив элементов по формуле Роберта Седжвика
    if (++s % 2) {
      inc[s] = 8*p1 - 6*p2 + 1;
    } else {
      inc[s] = 9*p1 - 9*p3 + 1;
      p2 *= 2;
      p3 *= 2;
    }
    p1 *= 2;
// заполняем массив, пока текущая инкремента хотя бы в 3 раза меньше
количества элементов в массиве
  } while(3*inc[s] < size);

  return s > 0 ? --s : 0;// возвращаем количество элементов в массиве
}

template<class T>
void shellSort(T a[], long size) {
```

```
// inc инкремент, расстояние между элементами сравнения
// i и j стандартные переменные цикла
// seq[40] массив, в котором хранятся инкременты
  long inc, i, j, seq[40];
  int s;//количество элементов в массиве seq[40]

  // вычисление последовательности приращений
  s = increment(seq, size);
  while (s >= 0) {
    //извлекаем из массива очередную инкременту
    inc = seq[s--];
// сортировка вставками с инкрементами inc
    for (i = inc; i < size; i++) {
      T temp = a[i];
// сдвигаем элементы до тех пор, пока не дойдем до конца или не упорядочим в нужном порядке
      for (j = i-inc; (j >= 0) && (a[j] > temp); j -= inc)
        a[j+inc] = a[j];
// после всех сдвигов ставим на место j+inc элемент, который находился на i месте
      a[j+inc] = temp;
    }
  }
}
```

Ссылка на Википедию

---

## Хеширование. Метод Брента

Один класс методов перемещает элементы во время вставки при двойном хешировании, делая успешный поиск более эффективным. Брент (Brent) разработал метод, при использовании которого даже в заполненной таблице среднее время успешного поиска ограничено константой. Такой метод может быть удобным в приложениях, в которых основной операцией является успешный поиск.

Link on Wikipedia

> Метод Брента применяется для нахождения корня $f(x)$ при $f(x) = 0$

# Ответы по прошлым темам

## 1. Как бы вы определили понятие «структура данных»

**Структура данных** (англ. **data structure**) — программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных в вычислительной технике. Для добавления, поиска, изменения и удаления данных структура данных предоставляет некоторый набор функций, составляющих её интерфейс.

## 6. Объясните отличие класса (class) от структуры (struct)

В Си++ основная разница между структурой и классом - это модификатор доступа, который используется по умолчанию для их членов. Для классов, по умолчанию используется модификатор private, а для структур - public. Конечно, принципы инкапсуляции структуры таким образом подрывают, но классы, в свою очередь тормозят стадию проектирования, которая затрагивает структурную эволюцию проекта.

Т.е., к примеру: выделить класс из структуры проще, чем из класса, т.к. для класса придется пересматривать логику взаимодействия свойств, которые ранее были на одном уровне доступа. Этот процеесс выливается в дописывание/переписывание методов, обеспечивающих инкапсуляцию.

Все было бы хорошо, если бы на какой-то очередной стадии проектирования Вы вдруг не осознаете, что порой ходите кругами, делая пустую работу, прикрывая тылы инкапсуляции.

С одной стороны, можно конечно занять позицию рецензора Си++ и следовать "букве закона ООП", т.е. смириться с этой неизбежной бюрократией. Но с другой стороны - это ведь Ваш проект, и Вы вправе строить его по своим законам, давая волю свободному проектированию какого-то сложного класса на структурах, а его финальные версии закрепить на классах по всем правилам ООП.

Ссылка