Standard Template Library
Эпизод I

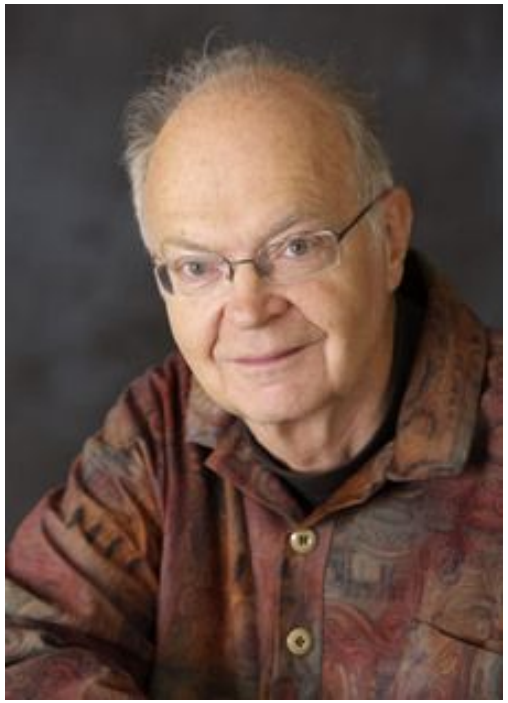
Антон Кухтичев

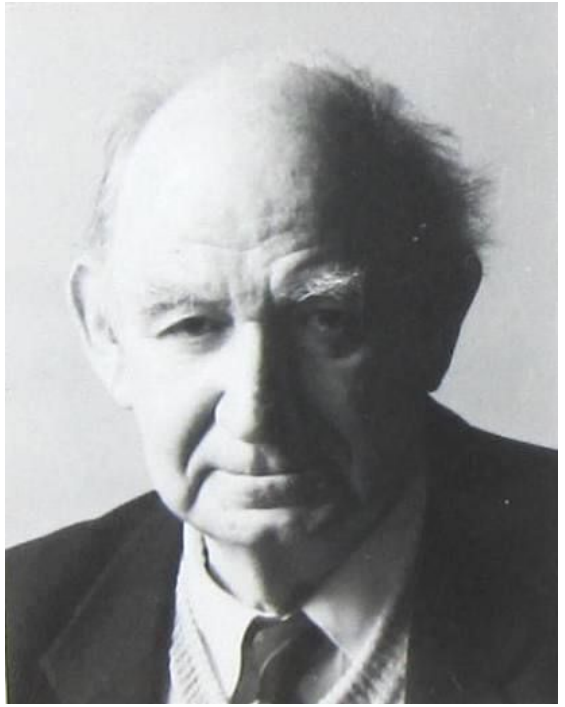
Содержание занятия

- Стандартная библиотека C++
- Библиотека шаблонов STL (Standard Template Library)
- Вспомогательные классы

Угадай программиста по фотографии

Кто это?





Стандартная библиотека C++

Стандартная библиотека C++

1. Ввод-вывод
2. Многопоточность
3. Регулярные выражения
4. Библиотека C
5. Библиотека шаблонов STL
6. Прочее (дата и время, обработка ошибок, поддержка локализации и т.д.)

Документация: https://en.cppreference.com/w/

std::ifstream

```
std::ifstream file("/tmp/file.txt");
if (!file)
{
    std::cout << "can't open file" ;
    return;
}

while (file.good())
{
    std::string s;
    file >> s;
}
```

std::ifstream

```
const int size = 1024;
char buf[size];

std::ifstream file("/tmp/file.data", std::ios::binary);
file.read(buf, size);
const auto read = file.gcount();
```

std::ofstream

Запись в файл.

```
std::ofstream file("/tmp/file.txt");
if (!file)
{
    std::cout << "can't open file" ;
    return;
}

file << "abc" << 123;
```

std::ofstream

```
const int size = 1024;
char buf[size];

std::ofstream file("/tmp/file.data", std::ios::binary);
file.write(buf, size);
```

Вспомогательные классы

std::optional

Шаблон класса, который содержит значение, которое может присутствовать или не присутствовать.

- `has_value()` — проверка, содержит ли переменная объект;
- `value()` — возвращает ссылку на объект, содержащийся в переменной;
- `operator=` — присвоить объект;
- `reset()` — уничтожает объект, содержащийся в переменной;
- `std::nullopt` — константа, обозначающая неинициализированное состояние.

std::optional

В инструкциях, где ожидается `bool` (`if`, `while`, `for`, `?:`) `std::optional` возвращает `true`, если в нём содержится объект, и `false` в противном случае.

- `if (opt)` эквивалентно `if (opt.has_value())`
- Объект может иметь значение `false`;

```
std::optional<bool> opt1 = false;
if (opt1) {
    // выполнится условие
}
std::optional<void*> opt2 = nullptr;
if (opt2) {
    // выполнится условие
}
```

tribool

Тип, подобный `bool`, который поддерживает три состояния, а не два: истина, ложь и неопределённость.

- Неявно преобразуется в `bool`. Если `tribool` имеет значение `true`, он преобразуется в `true`, в противном случае - в `false`;

```
#include <boost/logic/tribool.hpp>
using boost::logic::indeterminate;
boost::logic::tribool t = true, f = false, i = indeterminate;
```

std::pair

Тип позволяющий упаковать два значения в один объект.

```
#include <utility>

auto p1 = std::pair<int, double>(1, 2.0);
auto p2 = std::make_pair(1, 2.0);

auto x = p1.first; // int == 1
auto y = p1.second; // double == 2
```

`pair` имеет операторы сравнения позволяющие сделать лексикографическое сравнение элементов.

std::tuple

Тип позволяющий упаковать несколько значений в один объект.

```
#include <tuple>

auto t = std::make_tuple(1, 2.0, "abc");
int a = std::get<0>(t);
double b = std::get<1>(t);
std::string c = std::get<2>(t);
```

Соответствие типов проверяется на этапе компиляции.

Как и `pair` имеет лексикографические операторы сравнения.

std::tie

`tie`, как и `make_tuple` создает `tuple`, но не объектов, а ссылок на них.

Использование tie для написания операторов сравнения

```
struct MyClass
{
    int x_;
    std::string y_;
    double z_;

    bool operator<(const MyClass& o) const
    {
        return std::tie(x_, y_, z_) < std::tie(o.x_, o.y_, o.z_);
    }
};
```

Использование tie для написания операторов сравнения

```
bool operator<(const MyClass& o) const
{
    if (x_ != o.x_)
        return x_ < o.x_;
    if (y_ != o.y_)
        return y_ < o.y_;
    return z_ < o.z_;
}
```

Библиотека шаблонов STL (Standard Template Library)

Библиотека шаблонов STL (Standard Template Library)

1. Контейнеры (containers) – хранение набора объектов в памяти
2. Итераторы (iterators) – средства для доступа к источнику данных (контейнер, поток)
3. Алгоритмы (algorithms) – типовые операции с данными
4. Адаптеры (adaptors) – обеспечение требуемого интерфейса
5. Функциональные объекты (functors) – функция как объект для использования другими компонентами

O большое

«O» большое – математическое обозначение для сравнения асимптотического поведения алгоритма.

Фраза «сложность алгоритма есть `O(f(n))`» означает, что с ростом параметра `n` время работы алгоритма будет возрастать не быстрее, чем некоторая константа, умноженная на `f(n)`.

O большое

Типичные значения:

1. Время выполнения константно: $O(1)$
2. Линейное время: $O(n)$ $O(n+n) = O(2n) = O(cn) = O(n)$
3. Логарифмическое время: $O(\log n)$
4. Время выполнения «n логарифмов n»: $O(n \log n)$
5. Квадратичное время: $O(n^2)$

Контейнеры

1. Последовательные (Sequence containers)
2. Ассоциативные (Associative containers)
3. Неупорядоченные ассоциативные (Unordered associative containers)
4. Контейнеры-адаптеры (Container adaptors)

Последовательные контейнеры

std::array

```cpp
#include <array>

template <class T, size_t N>
class array
{
    T data_[N];
    size_t size_;
public:
    using size_type = size_t;
    using value_type = T;
    using reference = T&;
    using const_reference = const T&;
};
```

std::array

- `constexpr size_type size() const noexcept;`
- `constexpr bool empty() const noexcept;`
- `reference at(size_type pos);`
- `constexpr const_reference at(size_type pos) const;`
- `reference operator[](size_type pos);`
- `constexpr const_reference operator[](size_type pos) const;`
- `reference front();`
- `constexpr const_reference front() const;`
- `reference back();`
- `constexpr const_reference back() const;`
- `T* data() noexcept;`
- `const T* data() const noexcept;`

std::array

Пример:

```
std::array<int, 5> a = { 1, 2, 3, 4, 5 };
auto x = a[2];
a[2] = x * 2;
```

Вставка	Удаление	Поиск	Доступ
-	-	O(n)	O(1)

std::initializer_list

```
template <class T>
class initializer_list
{
public:
    size_type size() const noexcept;
    const T* begin() const noexcept;
    const T* end() const noexcept;
};
```

std::initializer_list

```
Array<int, 3> a = { 1, 2, 3 };

template <class T, size_t N>
class Array
{
public:
    Array(std::initializer_list<T> init)
    {
        size_t i = 0;
        auto current = init.begin();
        const auto end = init.end();
        while (current != end)
        {
            data_[i++] = *current++;
        }
    }
};
```

Аллокаторы

Назначение аллокатора - выделять и освобождать память.

`malloc` и `new` - аллокаторы

```
std::allocator<int> a1;
int* a = a1.allocate(1);
a1.construct(a, 7);
std::cout << a[0] << '\n';
a1.deallocate(a, 1);
```

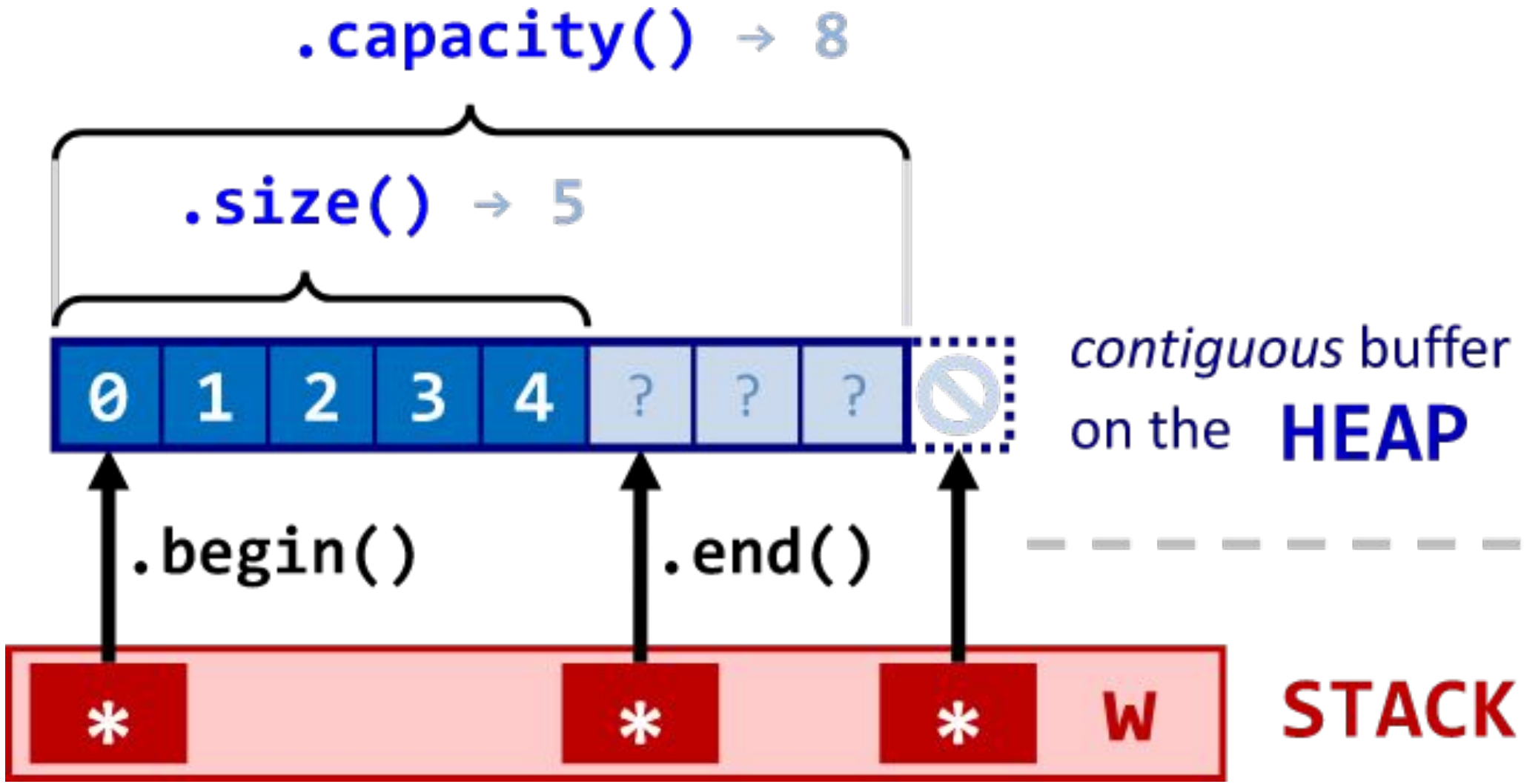
Аллокаторы

```
template<class T>
class allocator
{
public:
    using value_type = T;
    using pointer = T*;
    using size_type = size_t;

    pointer allocate(size_type count);
    void deallocate(pointer ptr, size_type count);

    // deprecated in C++17, removed in C++20
    size_t max_size() const noexcept;
};
```

std::vector


.capacity() → 8
.size() → 5
0
1
2
3
4
?
?
?
contiguous buffer
on the HEAP
.begin()
.end()
*
*
*
W
STACK

std::vector

```
template<class T, class Alloc = std::allocator<T>>
class vector
{
public:
    using size_type = size_t;
    using value_type = T;
    using reference = T&;
    using const_reference = const T&;
    using allocator_type = Alloc;
```

std::vector

```
explicit vector(size_type count);
vector(size_type count, const value_type& defaultValue);
vector(initializer_list<value_type> init);

iterator begin() noexcept; // аналогично end()
reverse_iterator rbegin() noexcept; // rend()
const_iterator cbegin() const noexcept; // cend()
const_reverse_iterator crbegin() const noexcept;// crend()
```

std::vector

```
void reserve(size_type count); // Выделяет память
size_type capacity() const noexcept;
void push_back(const value_type& value);

template<class... VT>
void emplace_back(VT&&... values);

iterator insert(const_iterator where, const T& value);
```

std::vector

Пример:

```
class A
{
    A(int, int) {}
    A(A&&) {}
};

A a(1, 2);

vec.push_back(std::move(a));
vec.emplace(1, 2);
```

Вставка	**Удаление**	**Поиск**	**Доступ**
В конце O(1) или O(n)	В конце O(1) или O(n)	O(n) В отсортированном O(log n)	O(1)

Трюки с вектором

1. Если порядок элементов не важен, то меняем удаляемый элемент с последним местами и удаляем последний (pop_back).
2. Изменение размера вектора перед вставкой

```
const auto size = file.size();
std::vector<char> data(size);
for (size_t i = 0; i < size; ++i)
    data[i] = file.read();
```

1. Очистка вектора

```
std::vector<int> v1;
std::vector<int> v2;
v1.push_back(1,2,3);
v1.clear();

v1.swap(std::vector<int>());
```

std::deque

Интерфейс повторяет интерфейс `std::vector`, отличие в размещении в памяти - `std::vector` хранит данные в одном непрерывном куске памяти, `std::deque` хранит данные в связанных блоках по n элементов.

Вставка	**Удаление**	**Поиск**	**Доступ**
В конце O(1) или O(n)	В конце O(1) или O(n)	O(n) В отсортированном O(log n)	O(1)

std::forward_list

Связный список, элементы которого хранятся в произвольных участках памяти.

Итератор списка не поддерживает произвольный доступ, следовательно алгоритмы STL, которые требуют random access iterator работать со списком не будут, например, `std::sort`.

Вставка	**Удаление**	**Поиск**	**Доступ**
O(1)	O(1)	O(n)	O(n)

std::list

Отличие от односвязного списка - возможность перемещаться в обратном направлении.

```
template <class T>
class Node
{
    T value_;
    Node<T>* prev_;
    Node<T>* next_;
};
```

Вставка	**Удаление**	**Поиск**	**Доступ**
O(1)	O(1)	O(n)	O(n)

Ассоциативные контейнеры

Контейнер позволяющий хранить пары вида (ключ, значение) и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

Элементы **отсортированы** по ключу:

1. `set<Key, Compare, Allocator>`
2. `map<Key, T, Compare, Allocator>`
3. `multiset<Key, Compare, Allocator>`
4. `multimap<Key, T, Compare, Allocator>`

Ассоциативные контейнеры

Элементы **не отсортированы** по ключу:

1. `unordered_set<Key, Hash, KeyEqual, Allocator>`
2. `unordered_map<Key, T, Hash, KeyEqual, Allocator>`
3. `unordered_multiset<Key, Hash, KeyEqual, Allocator>`
4. `unordered_multimap<Key, T, Hash, KeyEqual, Allocator>`

В `unordered_set` и `unordered_map` ключи уникальны, в multi версиях контейнеров допускаются наличие значений с одинаковым ключом.

Вставка	**Удаление**	**Поиск**	**Доступ**
O(1) или O(n)	O(1) или O(	O(1)	O(1)

Ассоциативные контейнеры

```
#include <unordered_map>

std::unordered_map<std::string, size_t> frequencyDictionary;
std::string word;
while (getWord(word))
{
    auto it = frequencyDictionary.find(word);
    if (it == frequencyDictionary.end())
        frequencyDictionary[word] = 1;
    else
        it->second++;
}
```

Контейнеры-адаптеры

Контейнеры-адаптеры

Являются обертками над другими контейнерами и предоставляют нужный интерфейс.

1. `stack<T, Container = std::deque<T>>`
2. `queue<T, Container>`
3. `priority_queue<T, Container, Compare>`

std::stack

Реализует интерфейс стека - положить значение в стек, извлечь значение из стека, последний пришел первый вышел (LIFO).

```
#include <stack>

std::stack<int> s;
s.push(3);
s.push(5);
int x = s.top(); // 5
s.pop();
int y = s.top(); // 3
```

std::queue

Реализует интерфейс очереди - положить значение в стек, извлечь первое значение из стека, первый пришел первый вышел (FIFO).

```
template<class T, class Container = std::deque<T>>
class queue;

void push(const value_type& value);
void push(value_type&& value);
reference front();
const_reference front() const;
void pop();
```

std::priority_queue

Отличие от queue - за O(1) можно извлечь элемент наиболее полно удовлетворяющий условию.

```
#include <queue>

template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>>
class priority_queue;
```

Code time

- Хотим получить TOP-k наибольших элементов в массиве!

Standard Template Library
Эпизод II

Антон Кухтичев

Содержание занятия

- Итераторы
- Красно-чёрное дерево
- Библиотека алгоритмов STL

Итераторы (iterators)

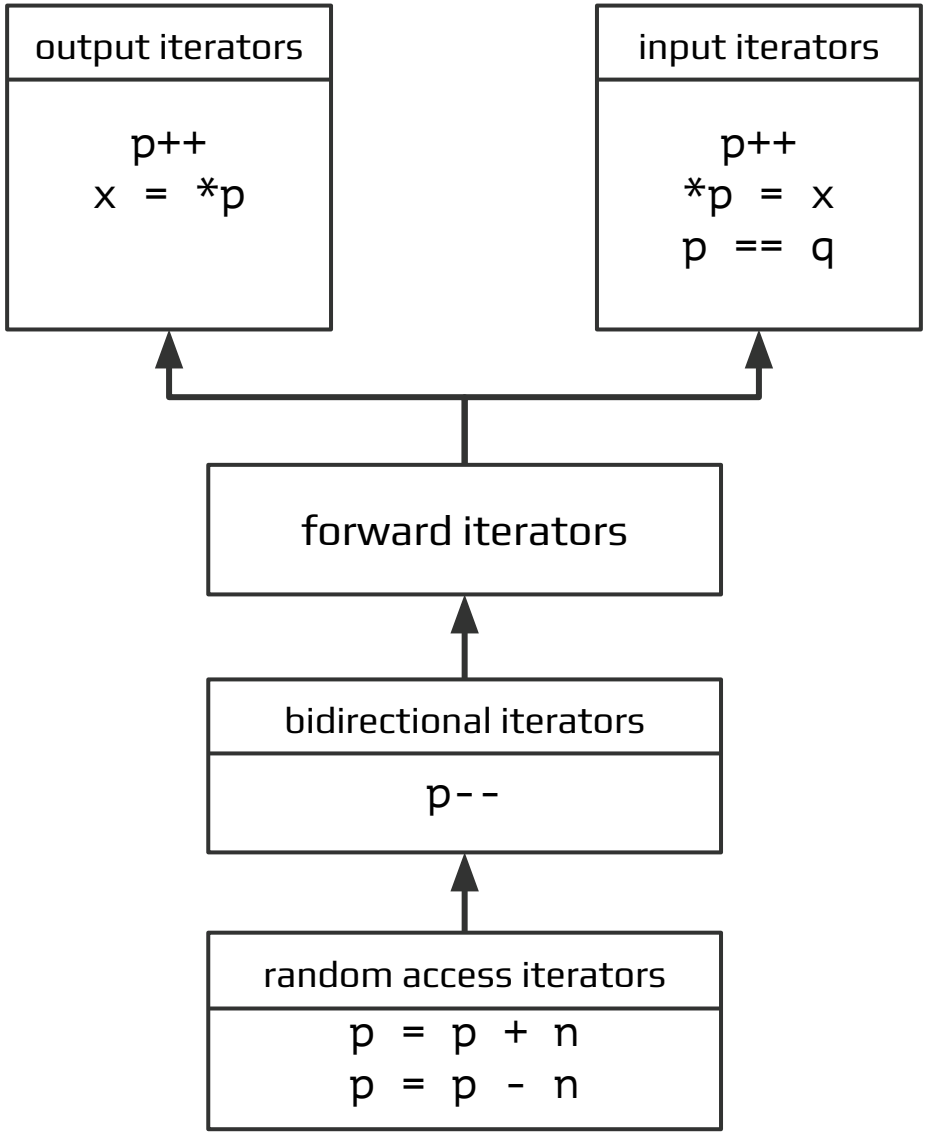
Объект предоставляющий доступ к элементам коллекции и осуществляющий навигацию по ним.

Позволяет реализовать универсальные алгоритмы работы с контейнерами.

Классификация итераторов:

1. Ввода (Input Iterator)
2. Однонаправленные (Forward Iterator)
3. Двунаправленные (Bidirectional Iterator)
4. Произвольного доступа (Random Access Iterator)
5. Вывода (Output Iterator)

Итераторы (iterators)



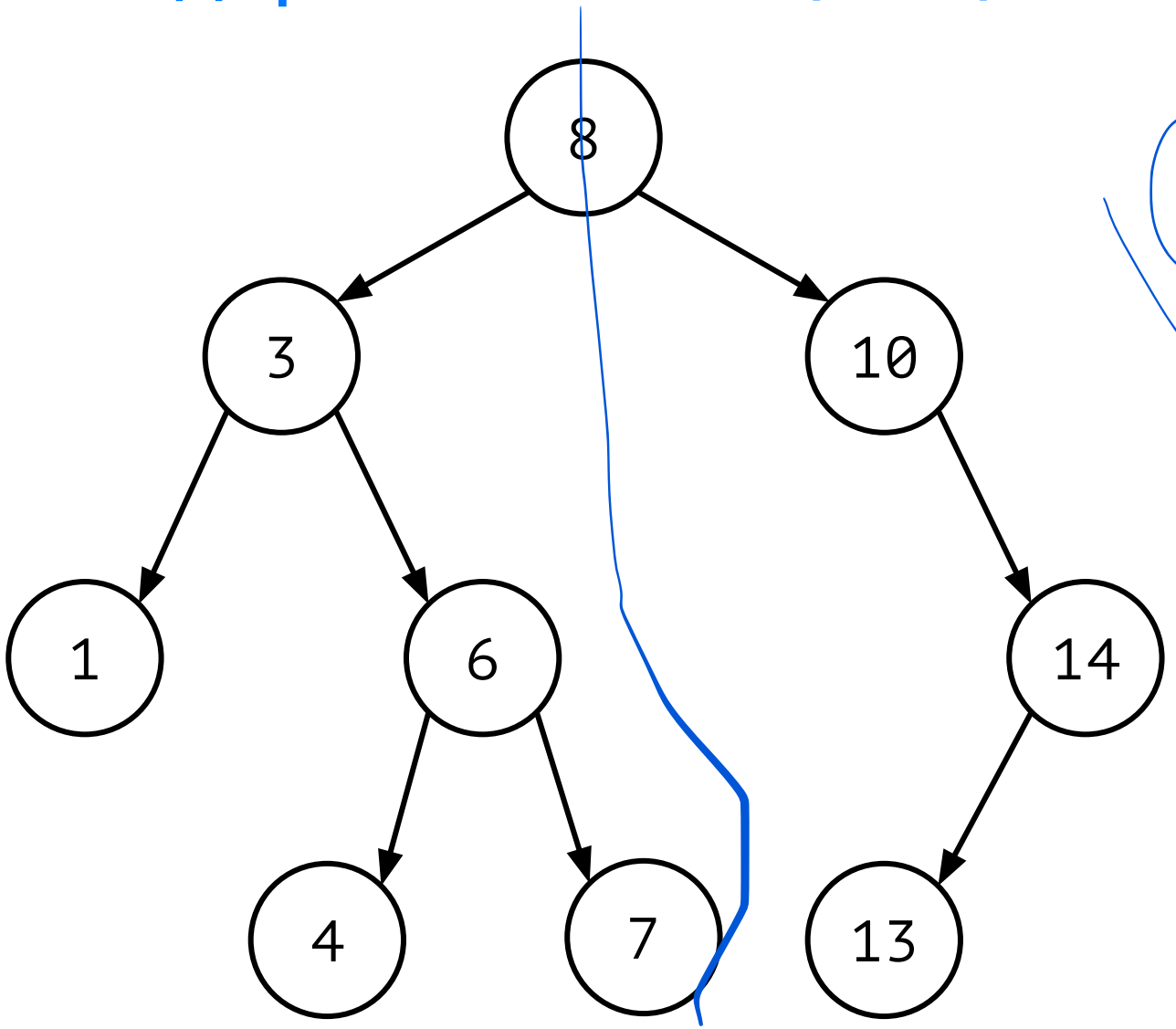
Итераторы (iterators)

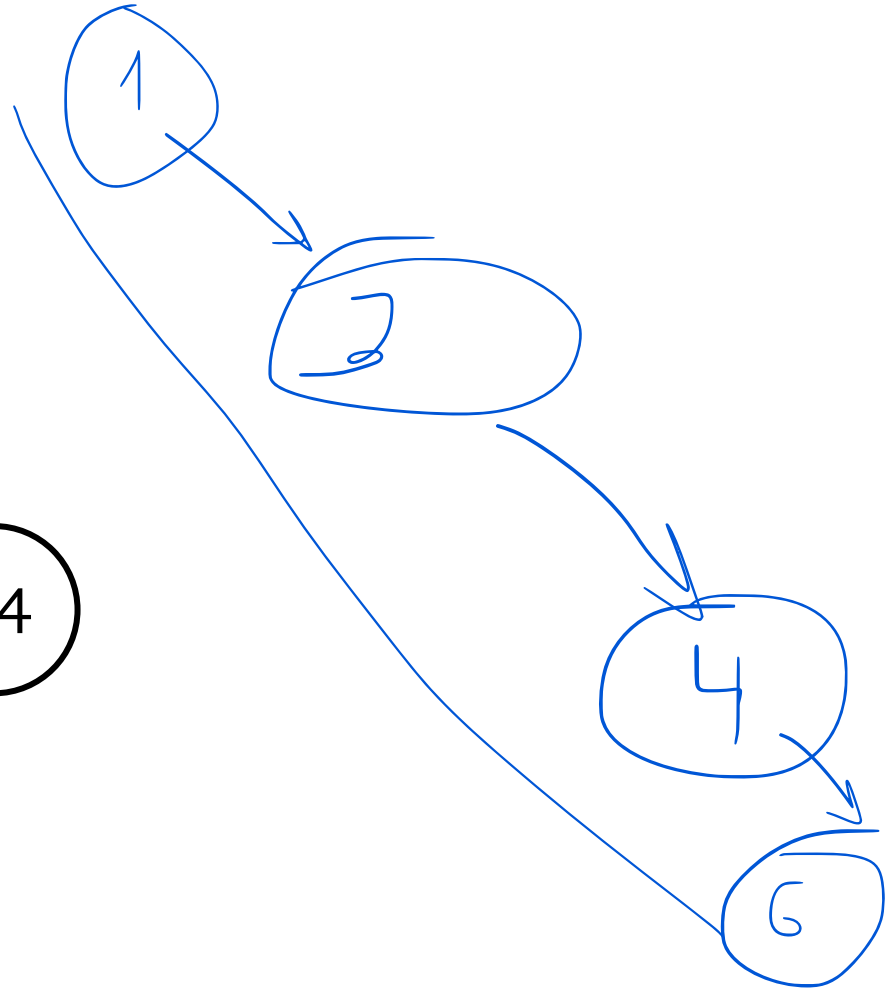
```
template< typename _Category, typename _Tp,
          typename _Distance = ptrdiff_t,
          typename _Pointer = _Tp*,
          typename _Reference = _Tp&>
struct iterator
{
    typedef _Category  iterator_category;
    typedef _Tp        value_type;
    typedef _Distance  difference_type;
    typedef _Pointer   pointer;
    typedef _Reference reference;
};
```

Code time

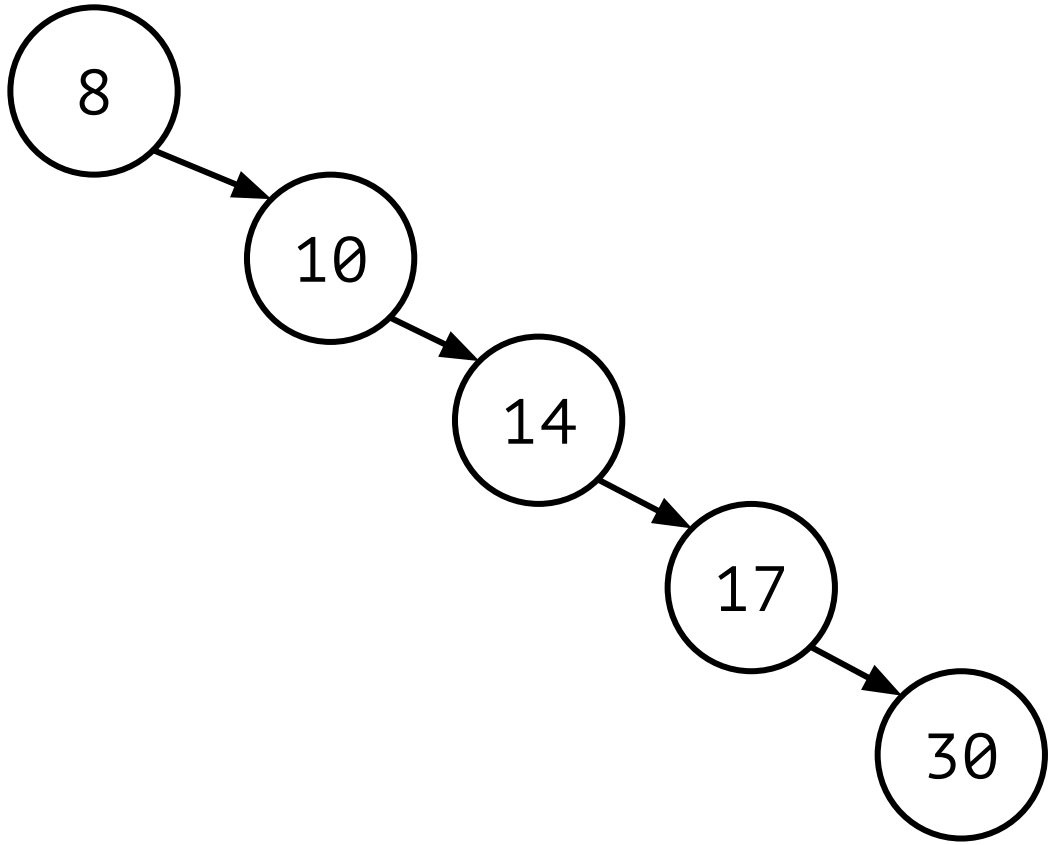
- Пишем свой итератор!

Бинарное дерево поиска (BST)





Бинарное дерево поиска



Красно-чёрные деревья

Красно-чёрное дерево (Red-black tree)

Бинарное дерево поиска содним дополнительным битом цвета в каждом узле.

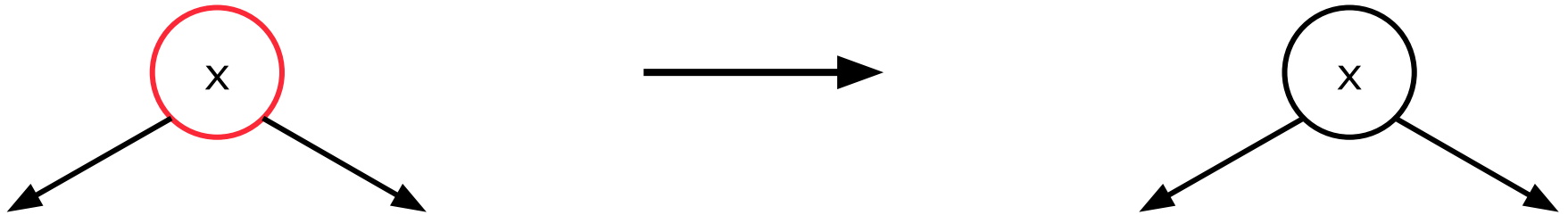
Свойства:

1. Каждый узел является красным или чёрным
2. Корень дерева является чёрным
3. Каждый лист дерева является чёрным
4. Если узел – красный, то оба его дочерних узла – чёрные
5. Для каждого узла все пути от него до листьев, являющихся потомками данного узла, содержал одно и то же количество чёрных узлов (чёрная высота)

1. Томас Кормен. Алгоритмы: построение и анализ. Глава 13.

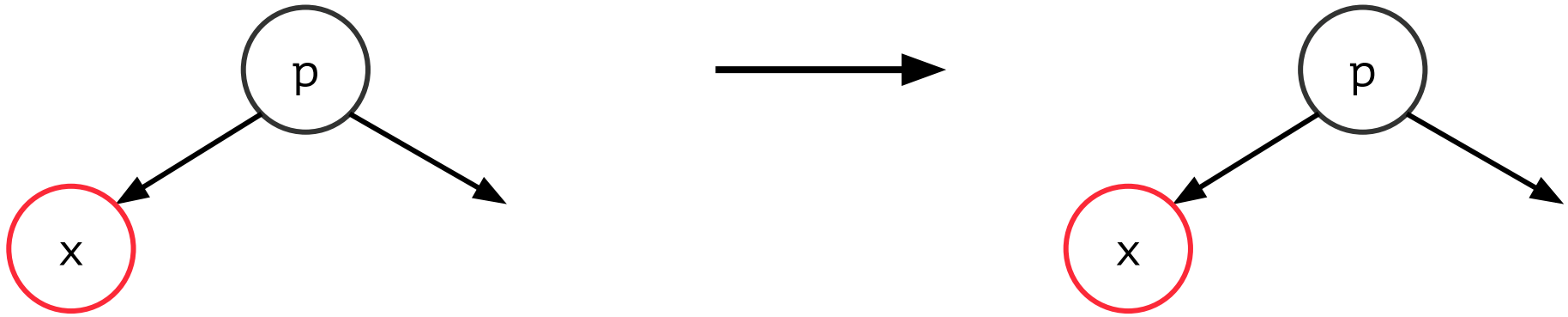
Вставка

Случай 1. Вставляемый узёл – корень. Просто перекрашиваем его в чёрный.



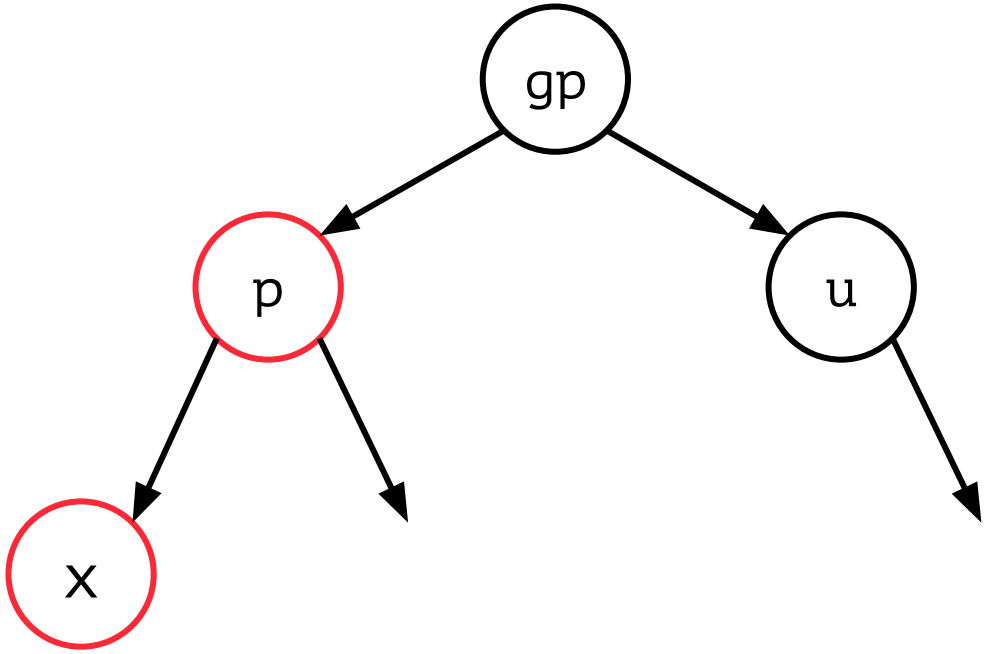
Вставка

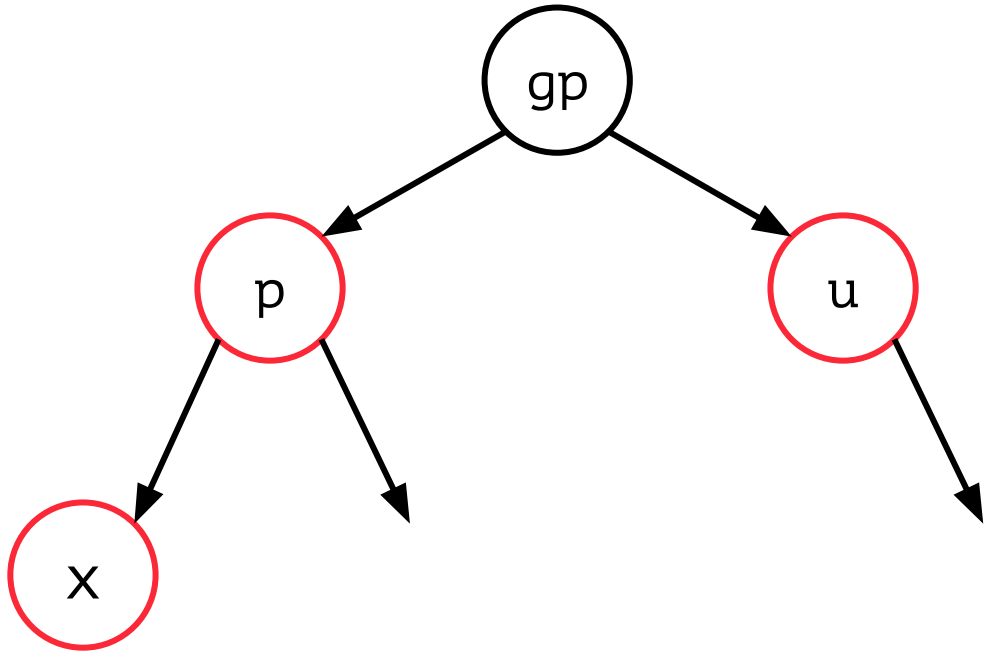
Случай 2. Родитель вставляемого узла – чёрный. Ничего не делаем



Вставка

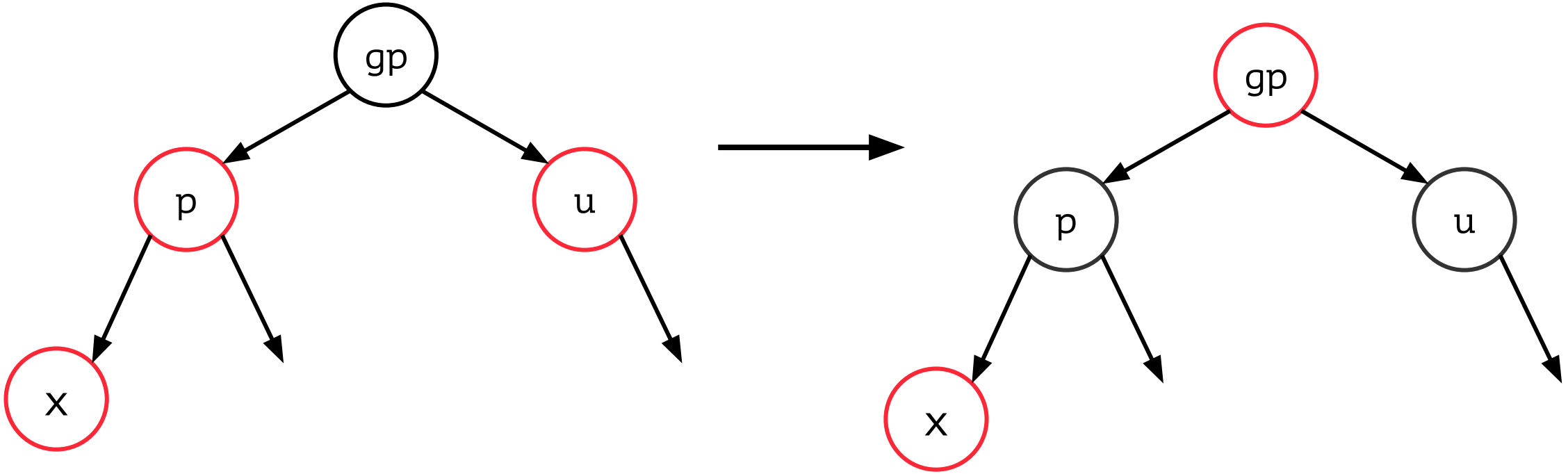
Случай 3 и 4. Родитель вставляемого узла – красный. Смотрим на цвет дяди.



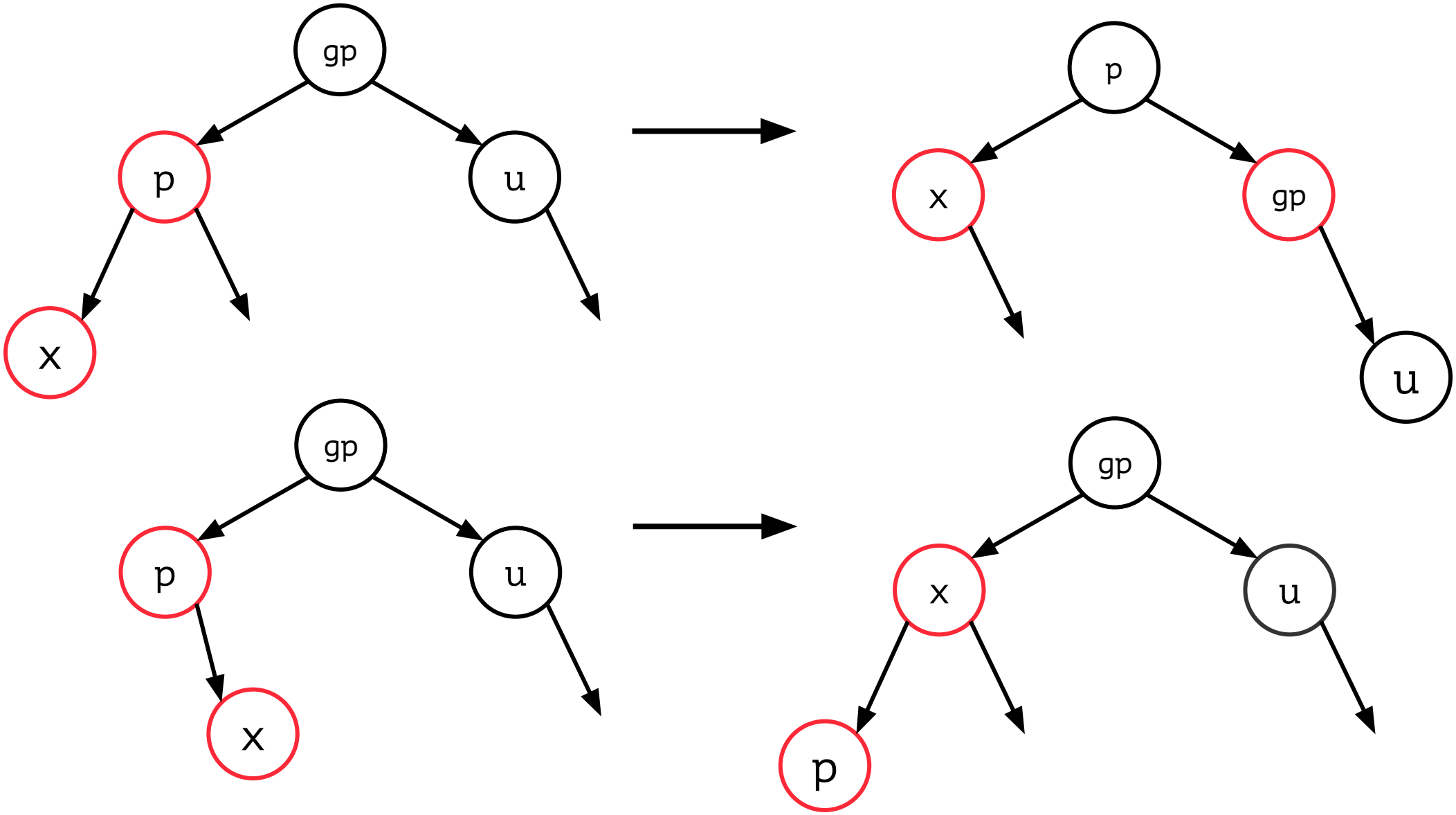


Вставка

Случай 3. Родитель вставляемого узла – красный, а цвет дяди - красный. Меняем цвета у дедушки на красный, а папу и дядю – на чёрный.

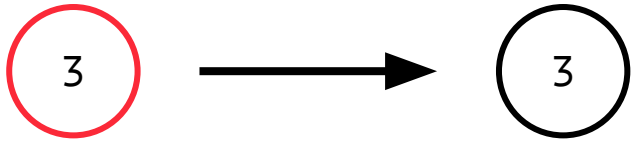


Вставка



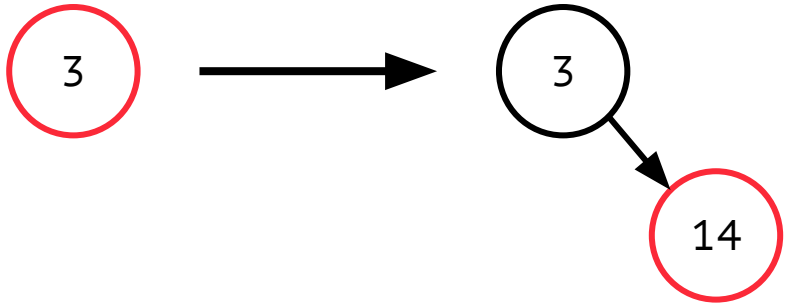
Вставка

Попробуем построить КЧ-дерево для последовательности {3, 14, 15, 9, 2, 6, 5, 35}



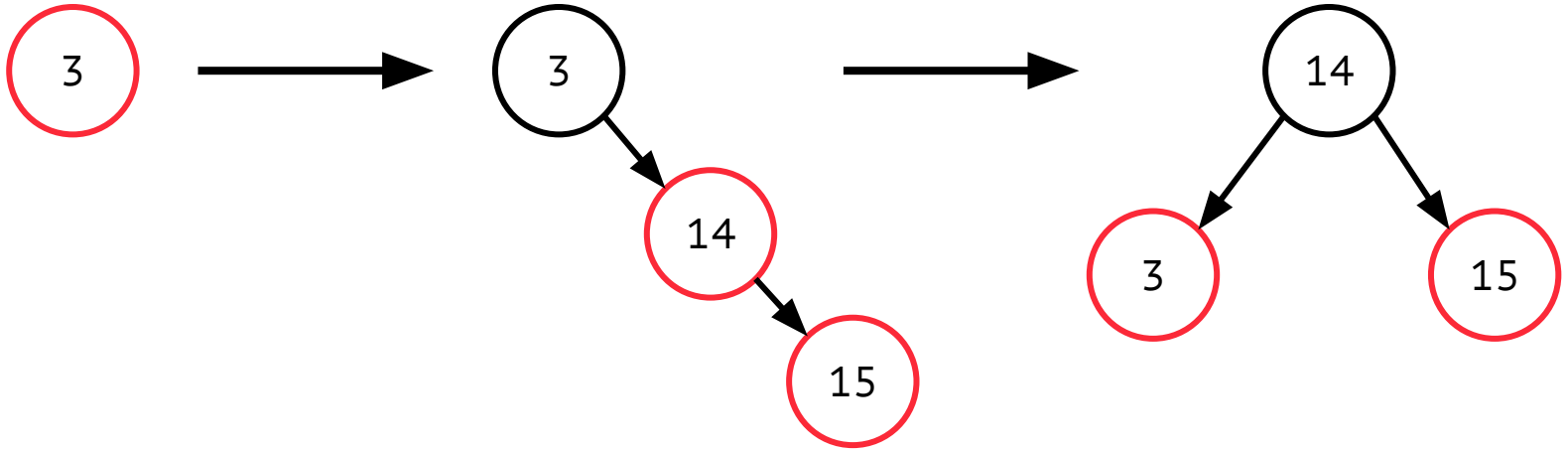
Вставка

Попробуем построить КЧ-дерево для последовательности {3, 14, 15, 9, 2, 6, 5, 35}



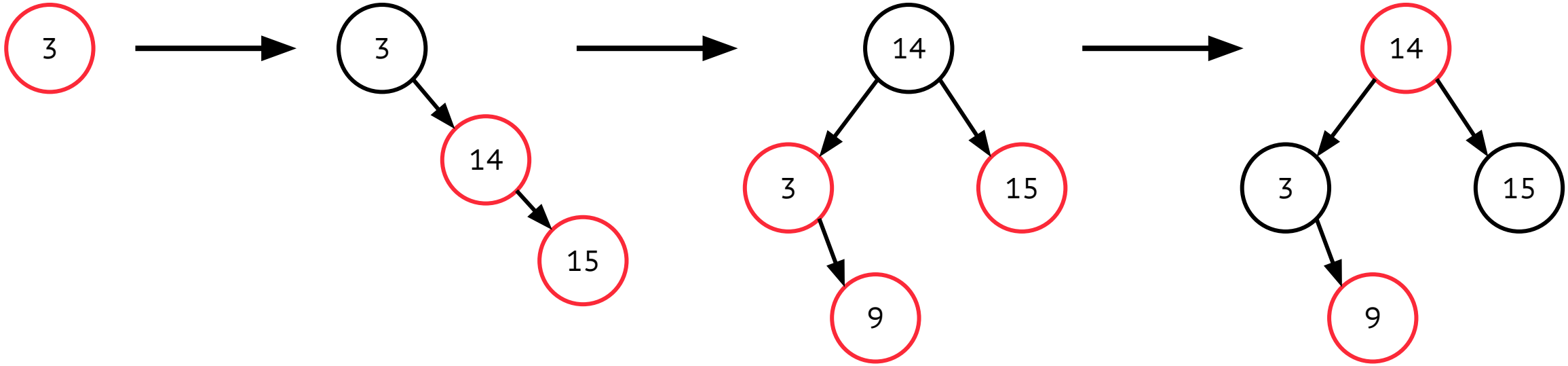
Вставка

Попробуем построить КЧ-дерево для последовательности {3, 14, 15, 9, 2, 6, 5, 35}



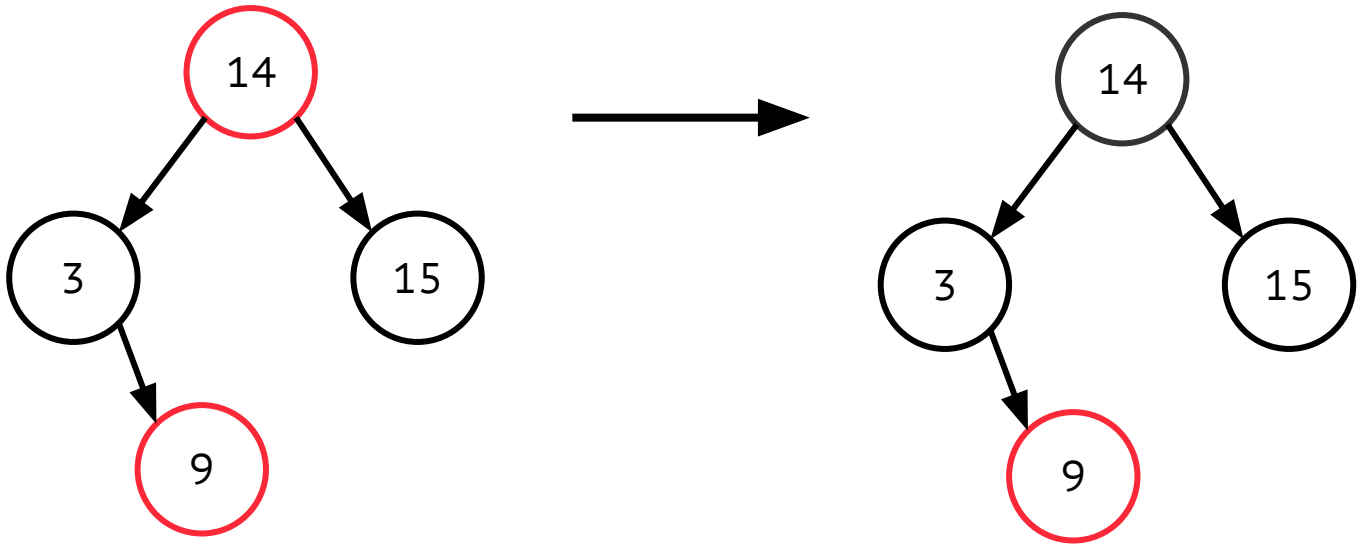
Вставка

Попробуем построить КЧ-дерево для последовательности {3, 14, 15, 9, 2, 6, 5, 35}



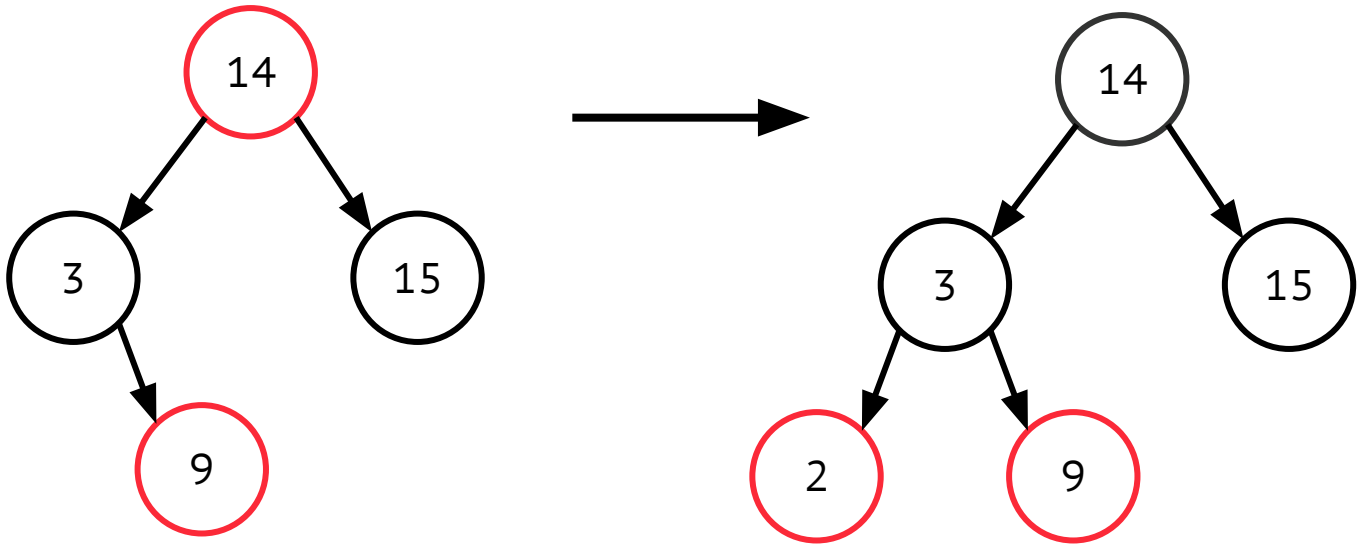
Вставка

Попробуем построить КЧ-дерево для последовательности {3, 14, 15, 9, 2, 6, 5, 35}



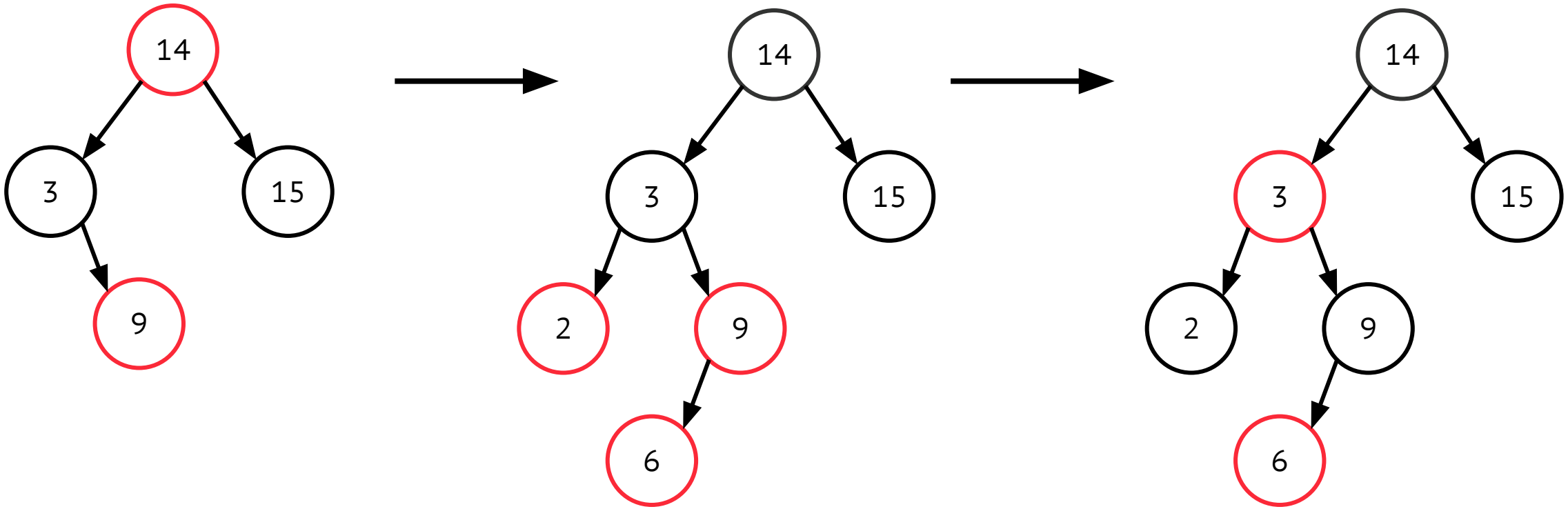
Вставка

Попробуем построить КЧ-дерево для последовательности {3, 14, 15, 9, 2, 6, 5, 35}



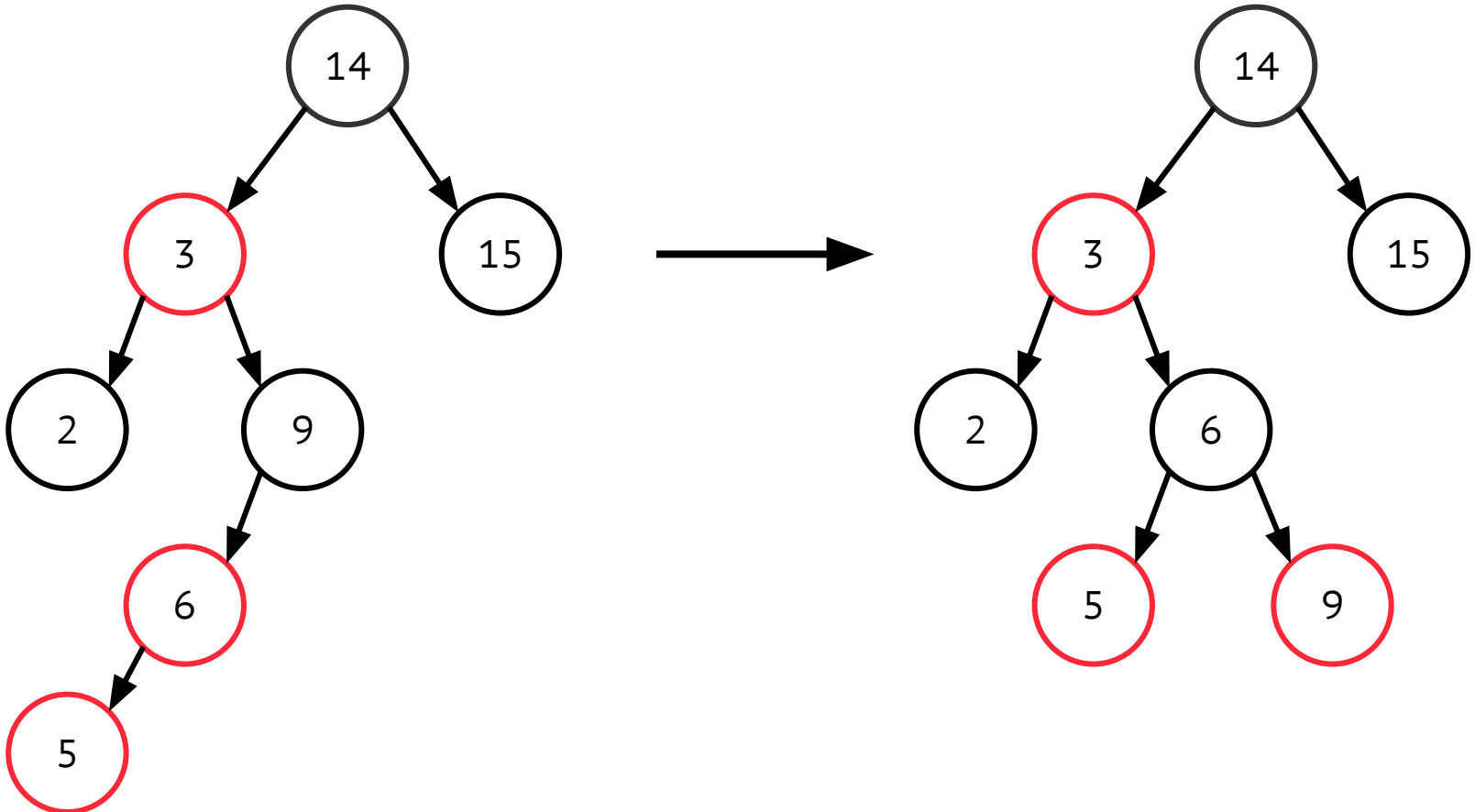
Вставка

Попробуем построить КЧ-дерево для последовательности {3, 14, 15, 9, 2, 6, 5, 35}



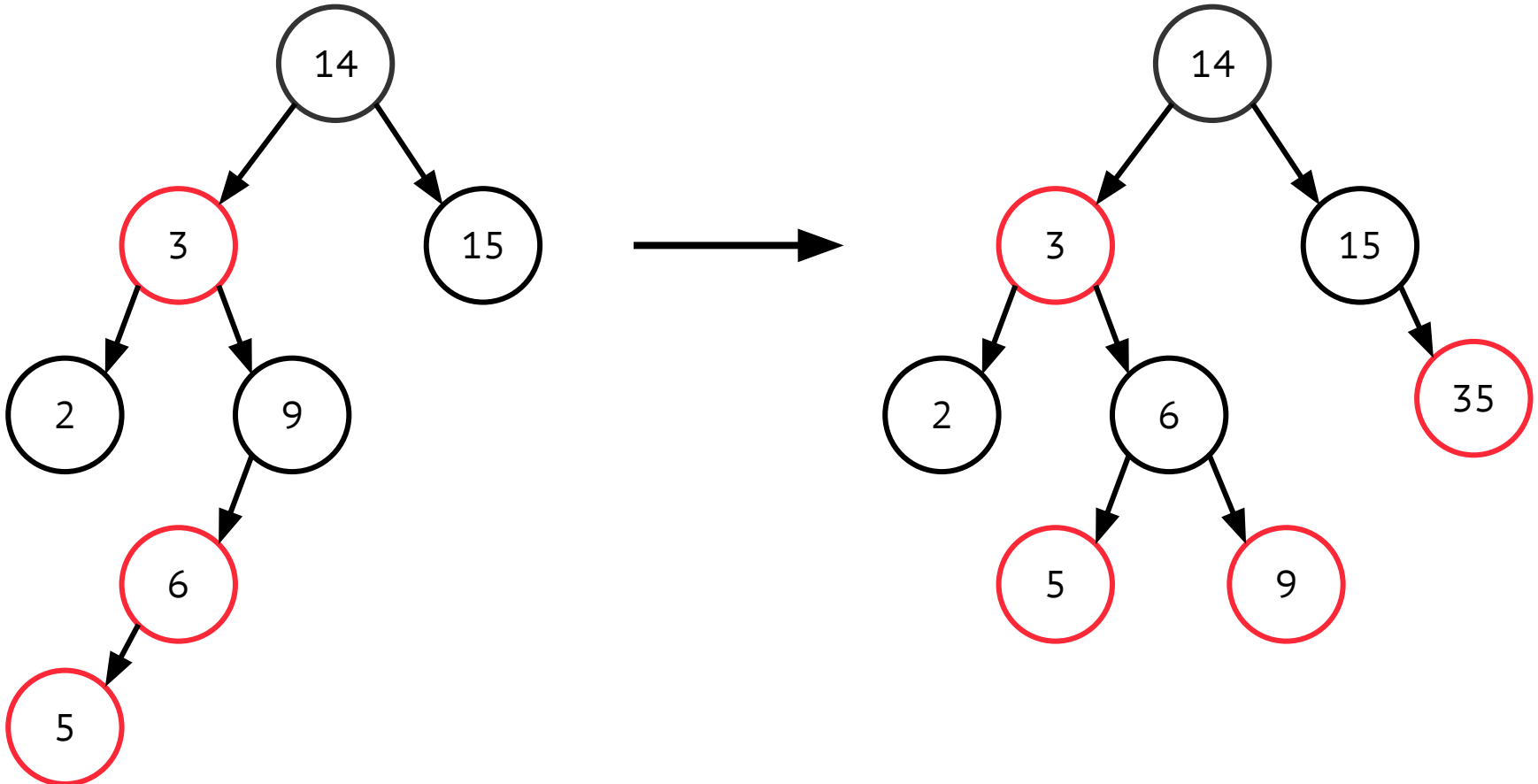
Вставка

Попробуем построить КЧ-дерево для последовательности {3, 14, 15, 9, 2, 6, 5, 35}



Вставка

Попробуем построить КЧ-дерево для последовательности {3, 14, 15, 9, 2, 6, 5, 35}



Библиотека алгоритмов STL

Библиотека алгоритмов STL

1. Не изменяющие последовательные алгоритмы
2. Изменяющие последовательные алгоритмы
3. Алгоритмы сортировки
4. Бинарные алгоритмы поиска
5. Алгоритмы слияния
6. Кучи
7. Операции отношений

Не изменяющие последовательные алгоритмы

Не изменяют содержимое последовательности и решают задачи поиска, подсчета элементов, установления равенства последовательностей.

adjacent_find

`ForwardIterator adjacent_find([ep], fwd_begin, fwd_end, [pred])`

Возвращает итератор, указывающий на первую пару одинаковых объектов, если такой пары нет, то итератор - `end`.

```
std::vector<int> v { 1, 2, 3, 3, 4 };
auto i = std::adjacent_find(v.begin(), v.end());
// *i == 3
```

Не изменяющие последовательные алгоритмы

all_of

```
bool all_of([ep], ipt_begin, ipt_end, pred)
```

Проверяет, что все элементы последовательности удовлетворяют предикату.

```
std::vector<int> v { 1, 2, 3, 4 };
if (std::all_of(v.begin(), v.end(), [](int x) { return x < 5; }))
    std::cout << "all elements are less than 5";
```

any_of

Проверяет, что хоть один элемент последовательности удовлетворяет предикату.

Не изменяющие последовательные алгоритмы

none_of

Проверяет, что все элементы последовательности не удовлетворяют предикату.

```
bool none_of([ep], ipt_begin, ipt_end, pred);
```

count, count_if

Возвращает количество элементов, значение которых равно value или удовлетворяет предикату.

```
std::vector<int> v { 3, 2, 3, 4 };
auto n = std::count(v.begin(), v.end(), 3);
// n == 2
```

Не изменяющие последовательные алгоритмы

equal

Проверяет, что две последовательности идентичны.

```
bool equal([ep], ipt_begin1, ipt_end1, ipt_begin2, [ipt_end2], [pred])
```

```
bool isPalindrome(const std::string& s)
{
    auto middle = s.begin() + s.size() / 2;
    return std::equal(s.begin(), mid, s.rbegin());
}

isPalindrome("level"); // true
```

Есть версия принимающая предикат.

Не изменяющие последовательные алгоритмы

find, find_if, find_if_not

Находит первый элемент последовательности удовлетворяющий условию.

```
InputIterator find([ep], ipt_begin, ipt_end, value);
InputIterator find_if([ep], ipt_begin, ipt_end, pred);
InputIterator find_if_not([ep], ipt_begin, ipt_end, pred);
```

find_end

Находит последний элемент последовательности удовлетворяющий условию.

Не изменяющие последовательные алгоритмы

find_first_of

Ищет в первой последовательности первое вхождение любого элемента из второй последовательности.

```
std::vector<int> v { 0, 2, 3, 25, 5 };
std::vector<int> t { 3, 19, 10, 2 };
auto result = std::find_first_of(v.begin(), v.end(), t.begin(),
t.end());
if (result == v.end())
    std::cout << "no matches found\n";
else
    std::cout << "found a match at "
        << std::distance(v.begin(), result) << "\n";
}
```

Не изменяющие последовательные алгоритмы

for_each

Вызывает функцию с каждым элементом последовательности.

```
for_each([ep], ipt_begin, ipt_end, fn);
```

```
std::vector<int> v { 3, 2, 3, 4 };
auto print = [](int x) { std::cout << x; };
std::for_each(v.begin(), v.end(), print);
```

Не изменяющие последовательные алгоритмы

search

Ищет вхождение одной последовательности в другую последовательность.

```
ForwardIterator search([ep], fwd_begin1, fwd_end1, fwd_begin2, fwd_end2, [pred])
```

search_n

Возвращает итератор на начало последовательности из n одинаковых элементов или end.

```
auto it = search_n(data.begin(), data.end(), howMany, value);
```

Не изменяющие последовательные алгоритмы

mismatch

Возвращает пару итераторов на первое несовпадение элементов двух последовательностей.

```
std::pair<Itr, Itr> mismatch([ep], ipt_begin1, ipt_end1, ipt_begin2, [ipt_end2], [pred]);
```

```
std::vector<int> x { 1, 2 };
std::vector<int> y { 1, 2, 3, 4 };
auto pair = std::mismatch(x.begin(), x.end(), y.begin());
// pair.first == x.end()
// pair.second = y.begin() + 2
```

Модифицирующие последовательные алгоритмы

Изменяют содержимое последовательности, решают задачи копирования, замены, удаления, перестановки значений и т.д.

copy, copy_if, copy_n

Копируют диапазон последовательности в новое место.

```
std::vector<int> data { 1, 2, 3, 4 };
std::copy(data.begin(), data.end(),
          std::ostream_iterator<int>(std::cout, " "));
std::vector<int> data { 1, 2, 3, 4 };
std::vector<int> out;
std::copy(data.begin(), data.end(), std::back_inserter(out));
```

Модифицирующие последовательные алгоритмы

copy_backward

Аналогично copy, но в обратном порядке.

move, move_backward

Аналогично copy, но вместо копирования диапазона используется перемещение.

fill, fill_n

Заполнение диапазона значениями.

```
std::vector<int> data { 1, 2, 3, 4 };
std::fill(data.begin(), data.end(), 0);
```

Модифицирующие последовательные алгоритмы

generate, generate_n

Заполнение сгенерированными значениями.

```
std::vector<int> randomNumbers;
auto iter = std::back_inserter(randomNumbers);
std::generate_n(iter, 100, std::rand);
```

Модифицирующие последовательные алгоритмы

remove, remove_if

Удаляет элементы удовлетворяющие критерию. Если быть точным данные алгоритмы ничего не удаляют, просто изменяют последовательность так, чтобы удаляемые элементы были в конце и возвращают итератор на первый элемент.

```
std::string str = "Text\t with\t \ttabs";
auto from = std::remove_if(str.begin(), str.end(),
                           [](char x) { return x == '\t'; })
// Text with tabs\t\t\t
str.erase(from, str.end());
// Text with tabs
```

Модифицирующие последовательные алгоритмы

remove_copy, remove_copy_if

То же, что и remove, но то, что не должно удаляться копируется в новое место.

```
std::string str = "Text with   spaces";
std::remove_copy(str.begin(), str.end(),
                 std::ostream_iterator<char>(std::cout), ' ');
// Textwithspaces
```

Модифицирующие последовательные алгоритмы

replace, replace_if

Заменяет элементы удовлетворяющие условию в последовательности.

```
std::string str = "Text\twith\ttabs";
std::replace_if(str.begin(), str.end(),
    [](char x) { return x == '\t'; }, ' ');
```

reverse

Поворачивает элементы последовательности задом наперед.

Модифицирующие последовательные алгоритмы

swap

Меняет два элемента местами.

```
int x = 3;
int y = 5;
std::swap(x, y);
```

iter_swap

Меняет два элемента на которые указывают итераторы местами.

Модифицирующие последовательные алгоритмы

shuffle

Перемешивает диапазон последовательности.

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::random_device rd;
std::mt19937 gen(rd());

std::shuffle(v.begin(), v.end(), gen);
```

Модифицирующие последовательные алгоритмы

unique

Удаляет (аналогично remove) дубликаты в последовательности, последовательность должна быть отсортирована.

```
std::vector<int> v { 1, 1, 2, 3, 3 };
const auto from = std::unique(v.begin(), v.end());
// 1 2 3 1 3
v.erase(from, v.end());
// 1 2 3
```

Алгоритмы сортировки

is_sorted

Проверяет упорядочена ли последовательность.

```
std::vector<int> v = { 1, 2, 3 };
const bool isSorted =
    std::is_sorted(v.begin(), v.end());
// true
```

Алгоритмы сортировки

sort

Сортирует последовательность.

```
std::vector<int> v = { 2, 3, 1 };
std::sort(v.begin(), v.end(),
          [](int x, int y) { return x > y; });
// 3 2 1
// Сложность O(n * log n)
```

Алгоритмы сортировки

partial_sort

Сортирует часть последовательности (TOP-N).

```
std::array<int, 10> s { 5, 7, 4, 2, 8, 6, 1, 9, 0, 3 };
std::partial_sort(s.begin(), s.begin() + 3, s.end());
// 0 1 2 7 8 6 5 9 4 3
```

Сложность O((last-first) * log (middle-first))

Алгоритмы сортировки

stable_sort

Сортирует последовательность, если два объекта равны, их порядок не изменится.

Сложность $O(n * \log_2 n)$

Алгоритмы сортировки

nth_element

Помещает элемент в позицию n, которую он занимал бы после сортировки всего диапазона.

```
std::vector<int> v { 3, 1, 4, 5, 2 };
const auto medianIndex = v.size() / 2;
std::nth_element(v.begin(), v.begin() + medianIndex, v.end());
const auto median = v[medianIndex];
// 3
```

Алгоритмы бинарного поиска

Последовательности к которым применяются алгоритмы должны быть отсортированы.

binary_search

Поиск по отсортированной последовательности.

```
std::vector<int> v { 1, 2, 3, 4, 5 };
bool has2 = std::binary_search(v.begin(), v.end(), 2);
// true
```

Алгоритмы бинарного поиска

lower_bound

Возвращает итератор, указывающий на первый элемент, который не меньше, чем value.

```
std::vector<int> v { 1, 2, 3, 4, 5 };
//                      ^
auto it = std::lower_bound(v.begin(), v.end(), 2);
```

Алгоритмы бинарного поиска

upper_bound

Возвращает итератор, указывающий на первый элемент, который больше, чем value.

```
std::vector<int> v { 1, 2, 3, 4, 5 };
//                        ^
auto it = std::upper_bound(v.begin(), v.end(), 2);
```

Алгоритмы бинарного поиска

equal_range

Возвращает такую пару итераторов, что элемент на который указывает первый итератор не меньше value, а элемент на который указывает второй итератор больше value.

```
std::vector<int> v { 1, 2, 3, 4, 5 };
//                      ^  ^
auto pair = std::equal_range(v.begin(), v.end(), 2);
```

Домашнее задание

Домашнее задание #7 (1)

Написать свой контейнер AVL аналогичный `std::map`, аллокатор и итератор произвольного доступа для него. Из поддерживаемых методов достаточно `operator[]`, `at`, `insert`, `erase`, `find`, `contains`, `empty`, `size`, `clear`, `begin`, `rbegin`, `end`, `rend`.

Интерфейс аллокатора и итератора смотрите в документации.

Полезная литература в помощь

- Документация стандартной библиотеки

Спасибо за внимание!

Вопросы?