Углублённое программирование на C++

Standard Template Library
Эпизод II
Атака алгоритмов

Кухтичев Антон

education

27 ноября 2025 года

Содержание занятия

- Small String Optimization
- Итераторы
- Красно-чёрное дерево
- Библиотека алгоритмов STL

Small String Optimization (SSO)

Small String Optimization

Оптимизация, применяемая в шаблоне класса `std::basic_string` и его аналогах.

Позволяет избегать дополнительных аллокаций динамической памяти для строк небольшого размера и размещать их внутри самого объекта.

Итераторы

Итераторы (iterators)

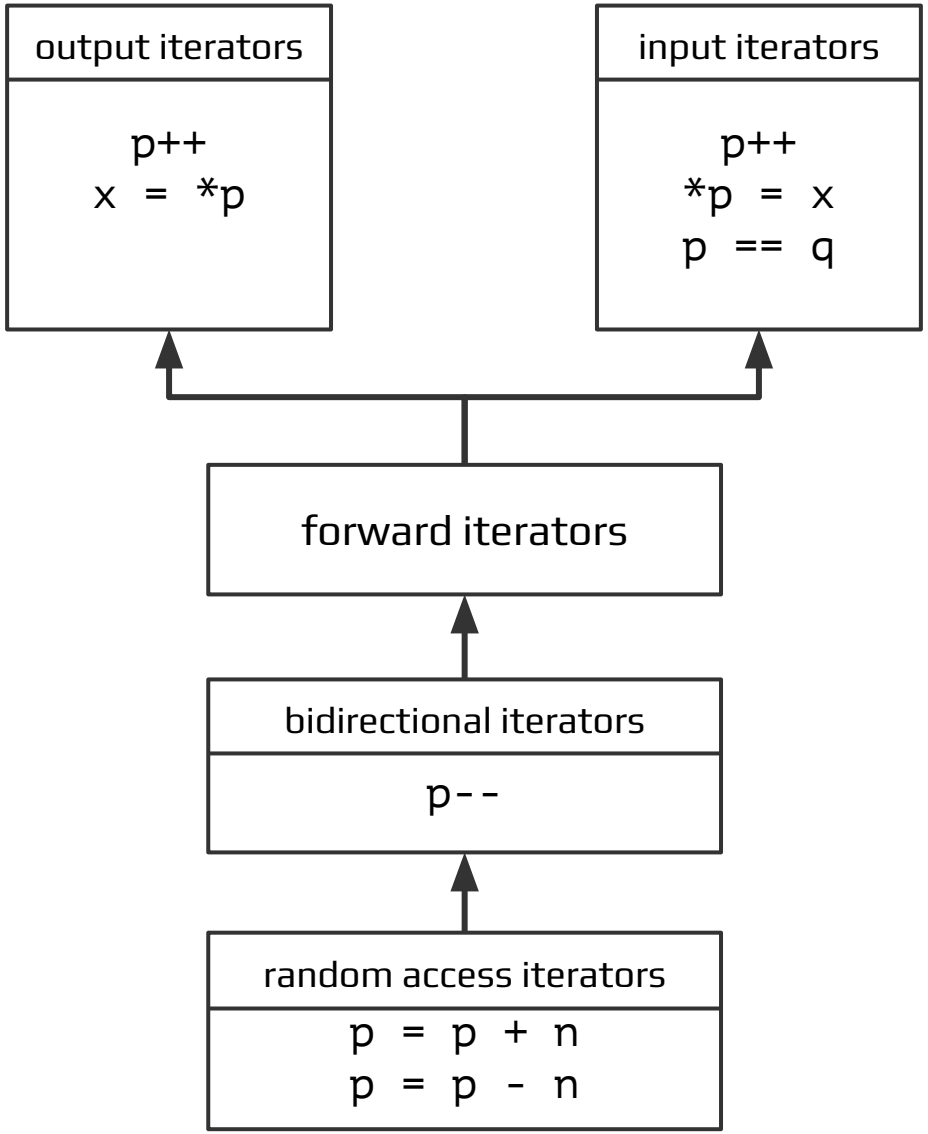
Объект предоставляющий доступ к элементам коллекции и осуществляющий навигацию по ним.

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)



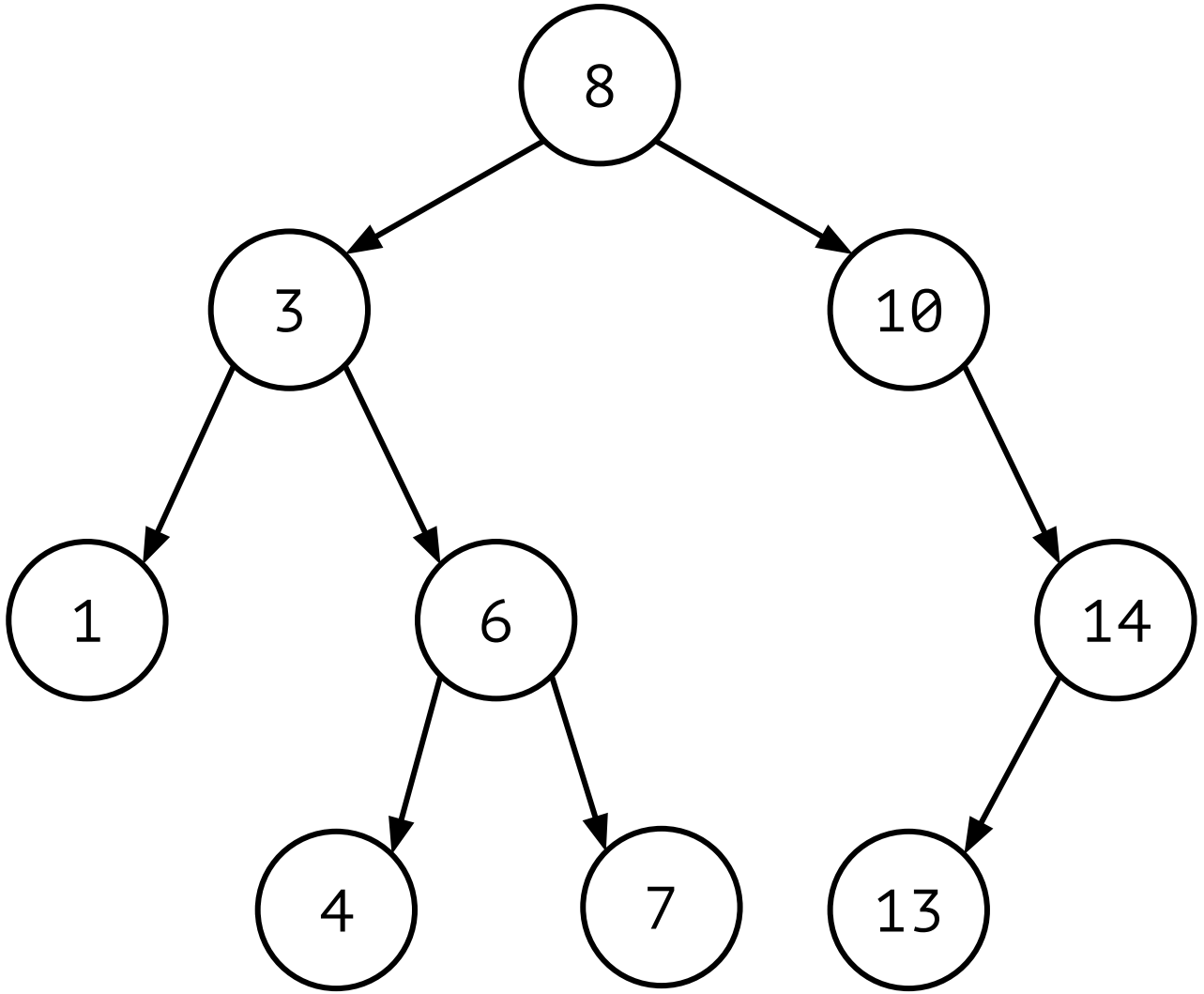
Итераторы (iterators)

```
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;
};
```

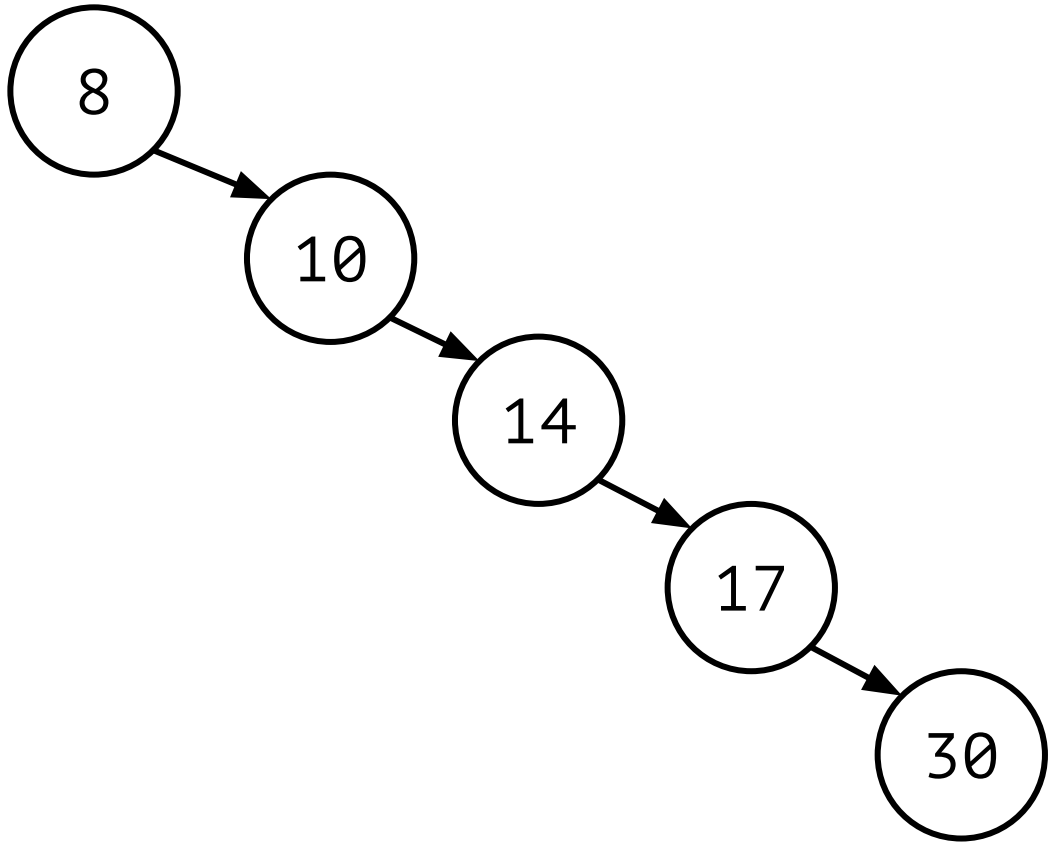
Code time

- Пишем свой итератор!

Бинарное дерево поиска (BST)



Бинарное дерево поиска



Красно-чёрные деревья

Красно-чёрное дерево (Red-black tree)

Бинарное дерево поиска с одним дополнительным битом цвета в каждом узле.

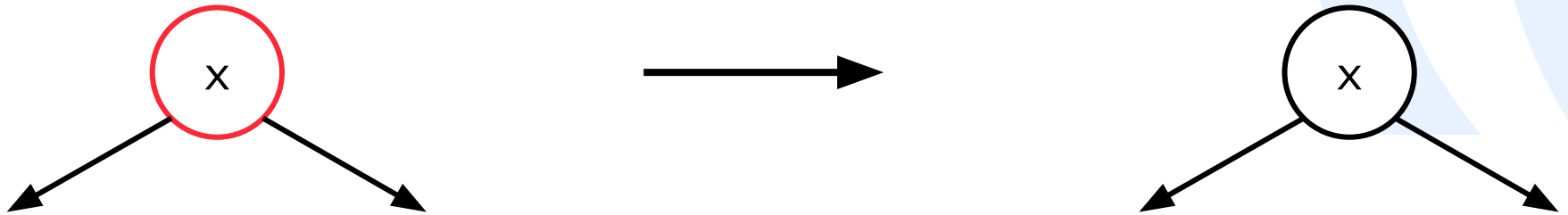
Свойства:

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.

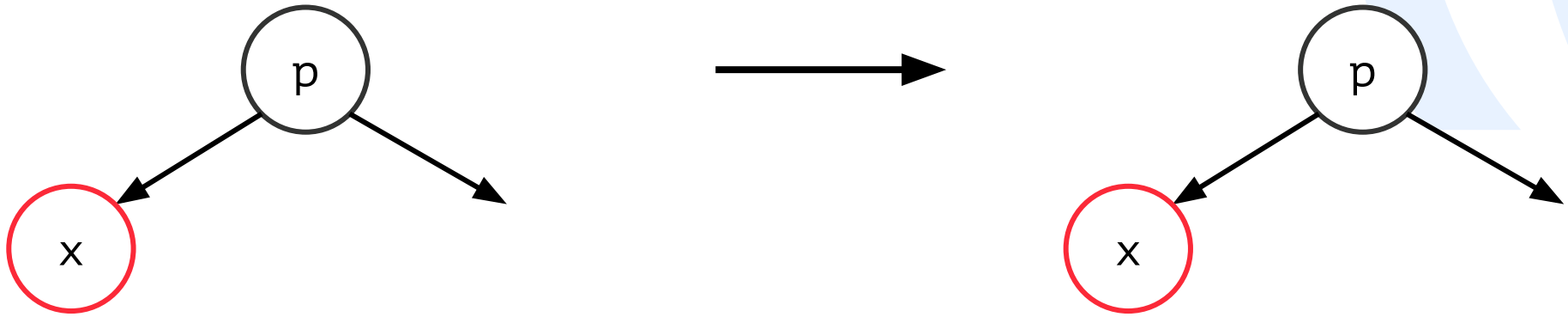
Вставка

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



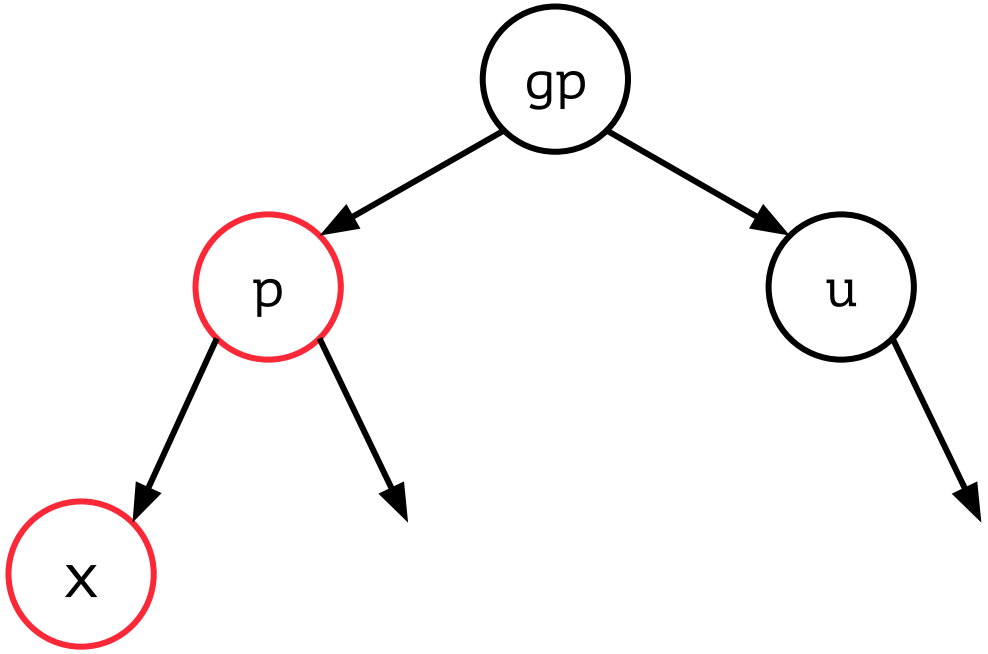
Вставка

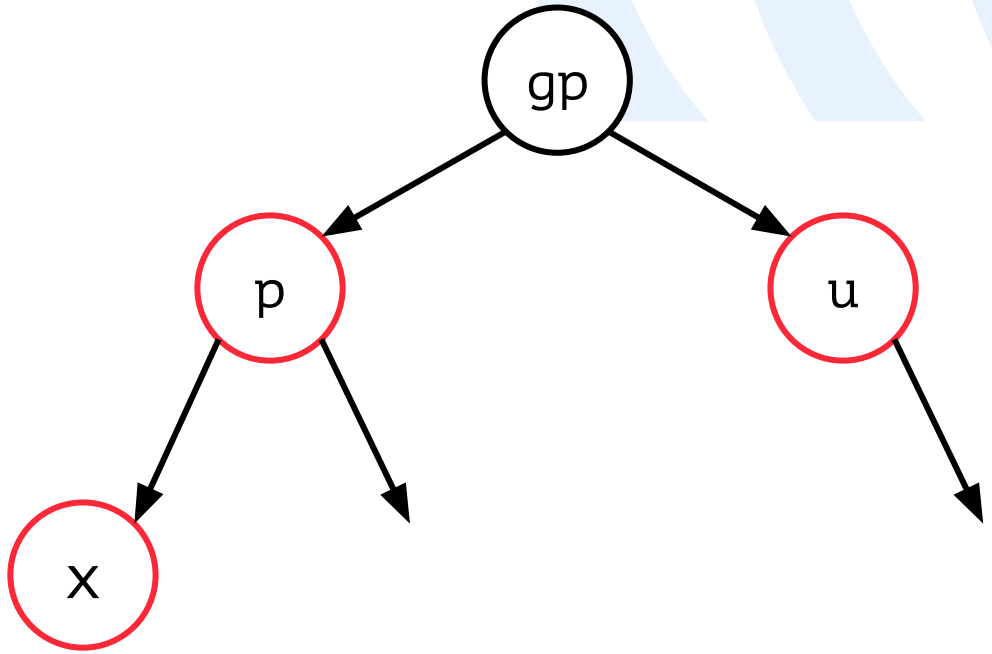
Случай 2. Родитель вставляемого узла – чёрный. Ничего не делаем



Вставка

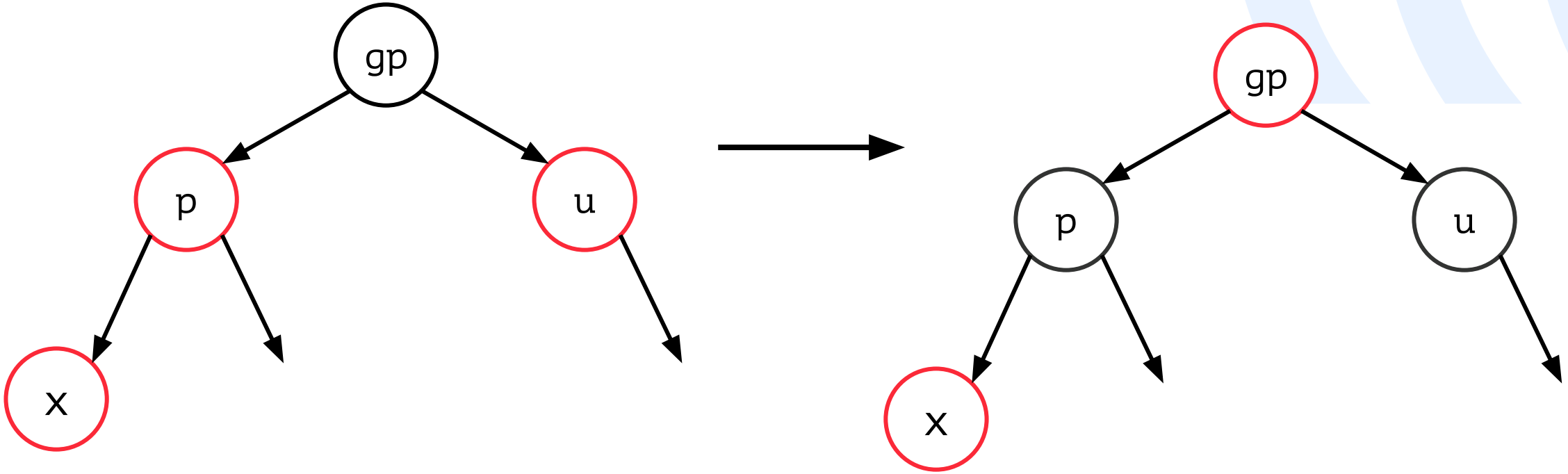
Случай 3 и 4. Родитель вставляемого узла – красный. Смотрим на цвет дяди.



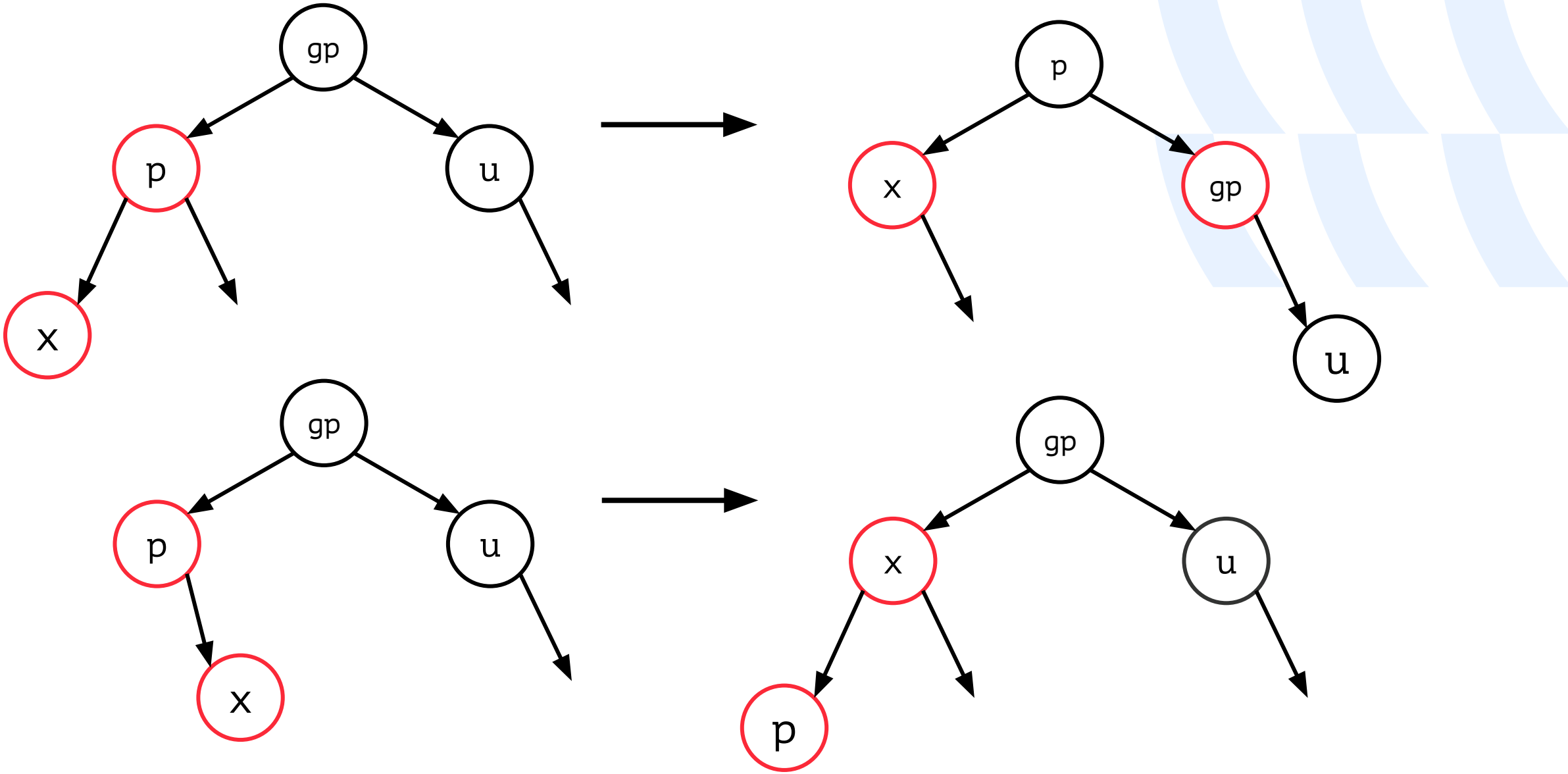


Вставка

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

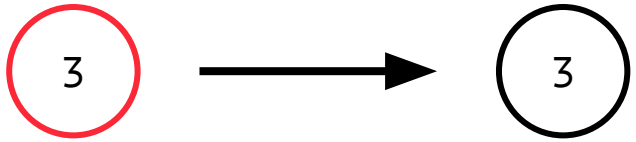


Вставка



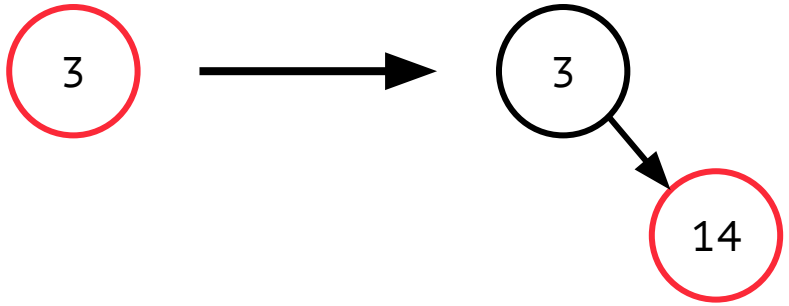
Вставка

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



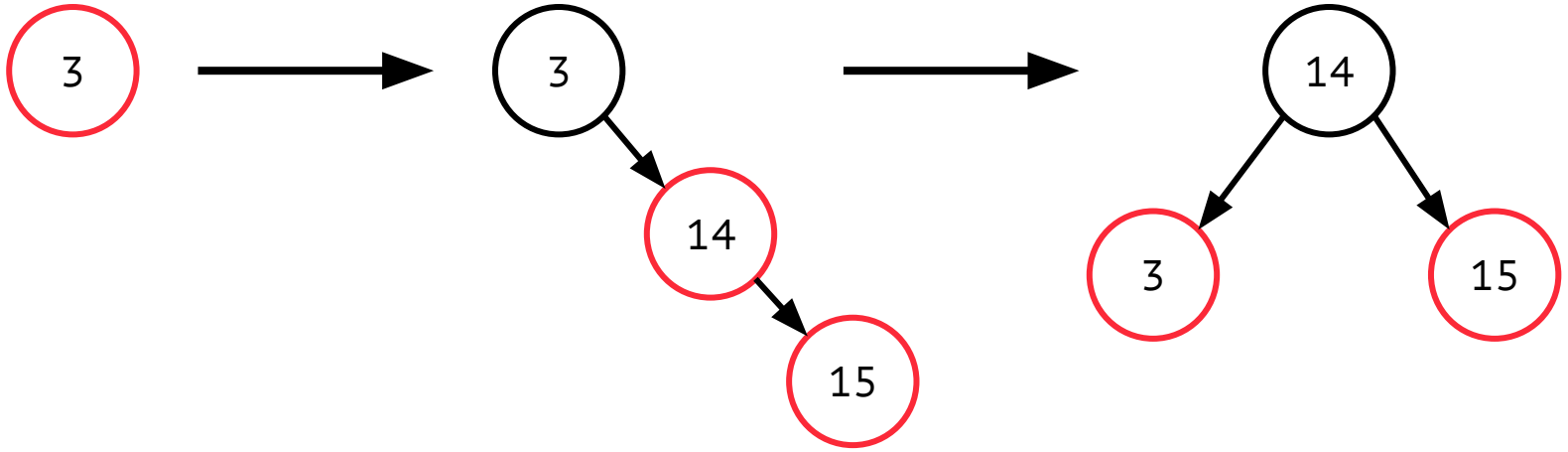
Вставка

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



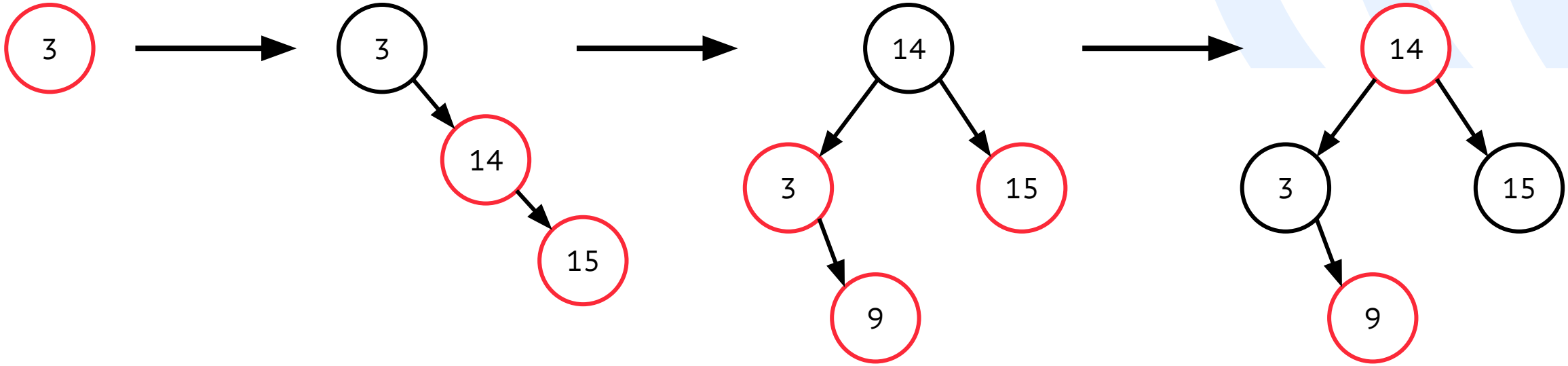
Вставка

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



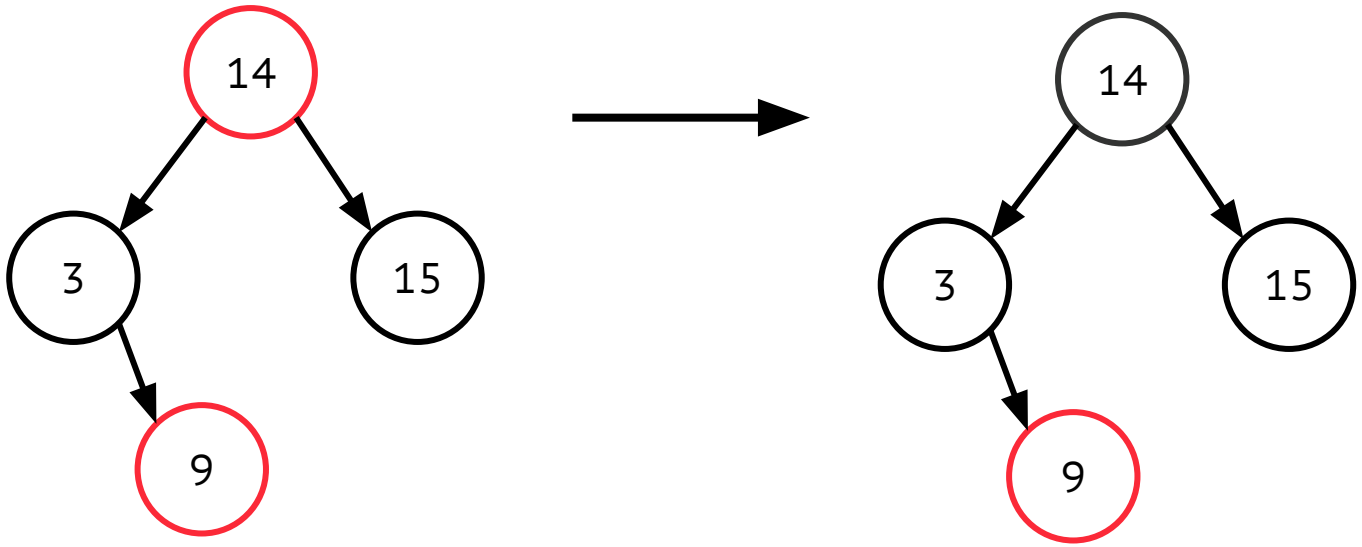
Вставка

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



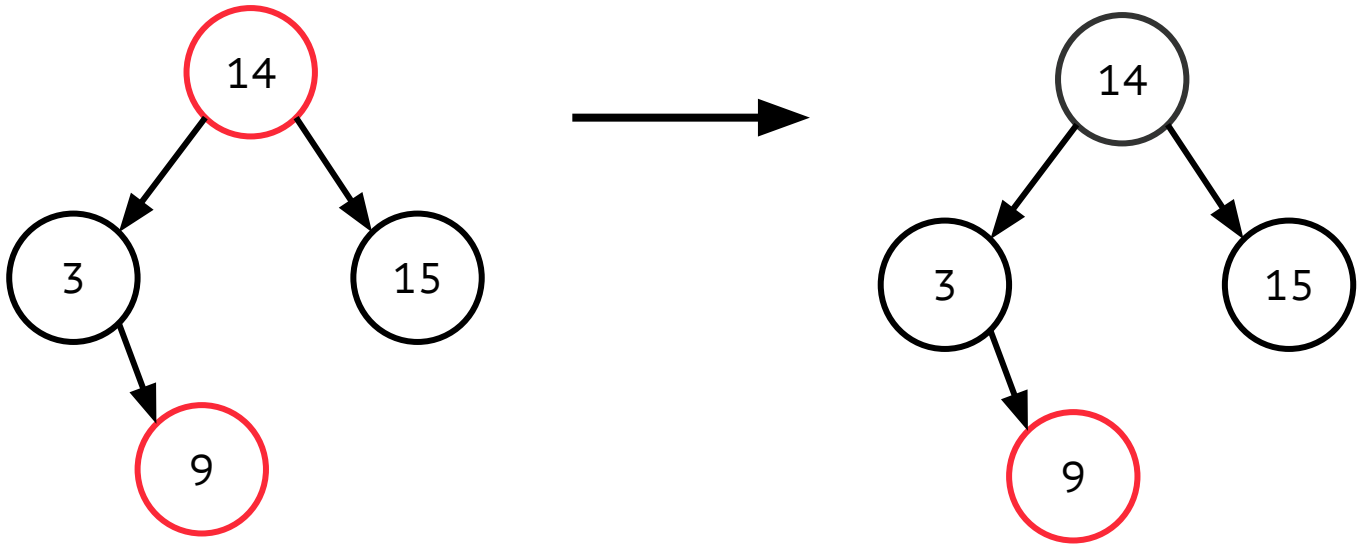
Вставка

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



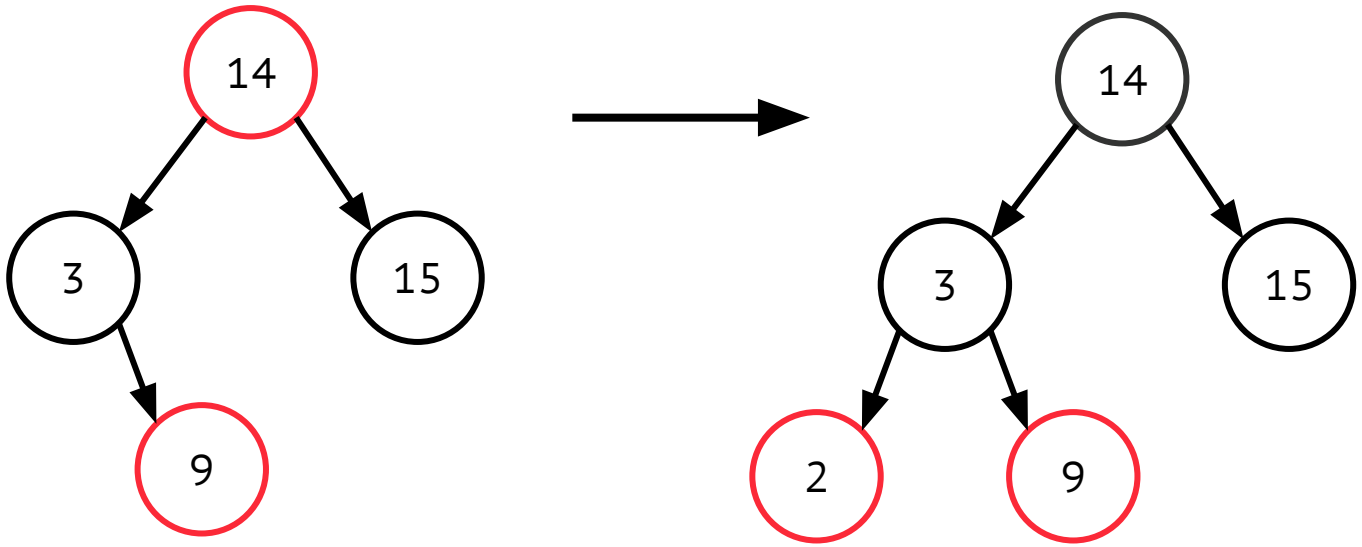
Вставка

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



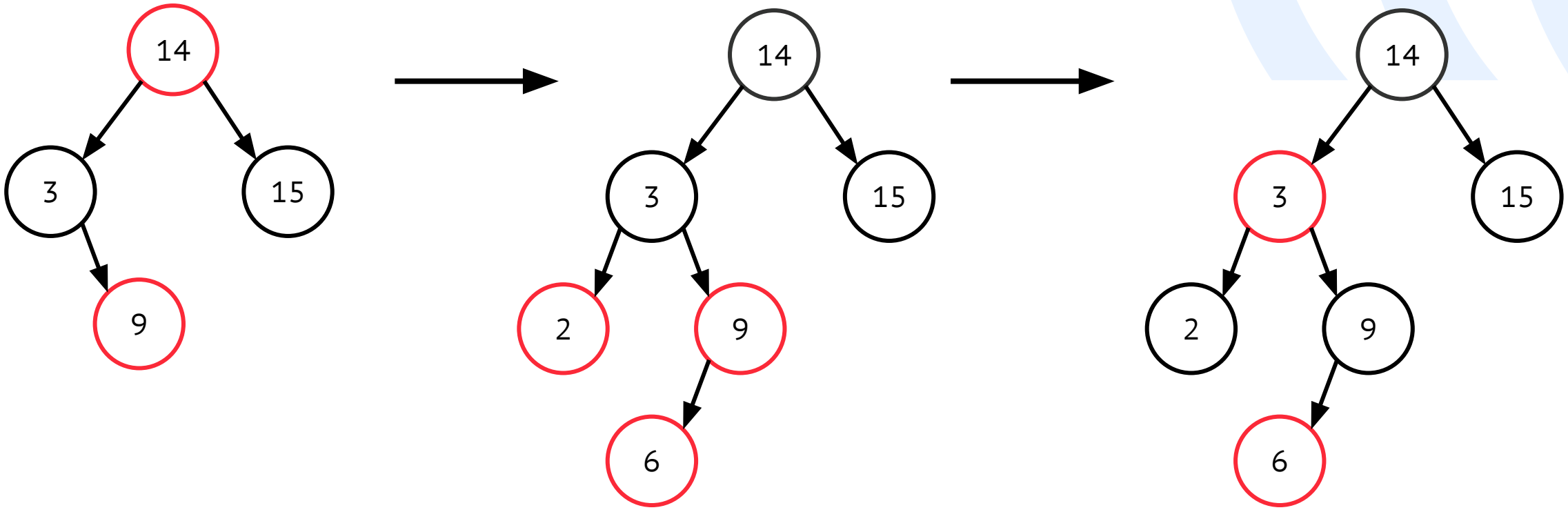
Вставка

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



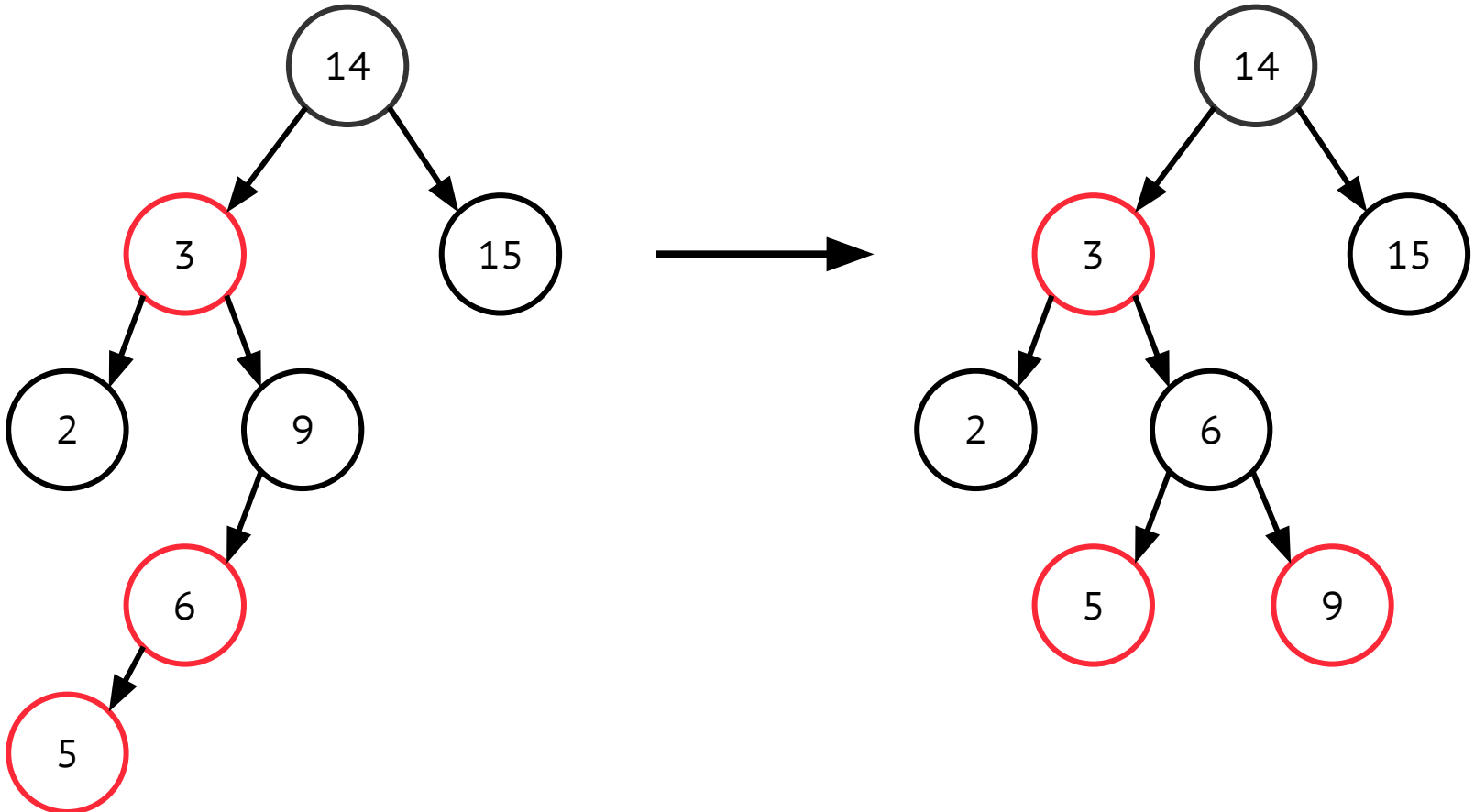
Вставка

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



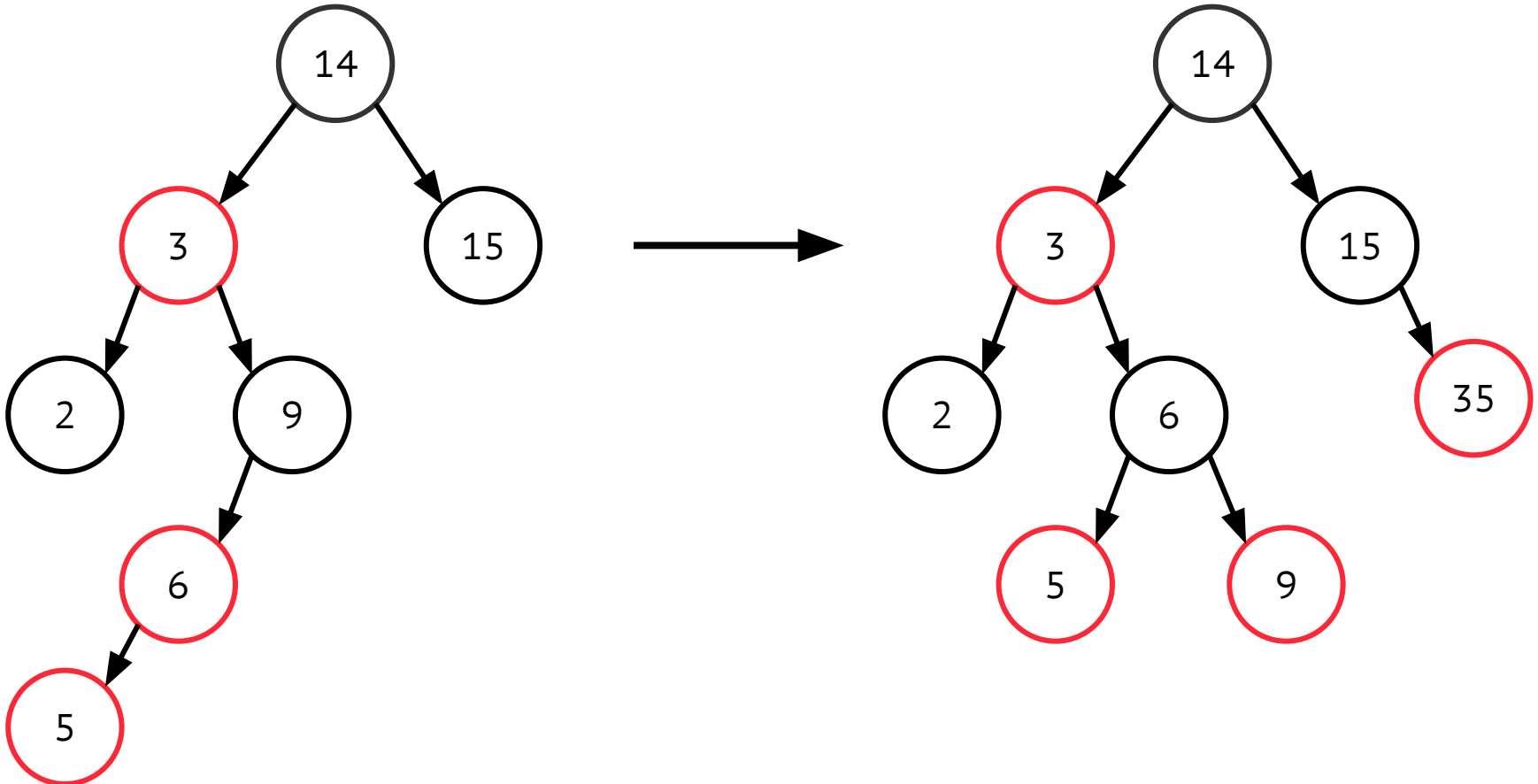
Вставка

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



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
```

1. Елена Сагалаева - std::remove и std::remove_if на самом деле ничего не удаляют https://alenacpp.blogspot.com/2005/10/stdremove-stdremoveif.html

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`.

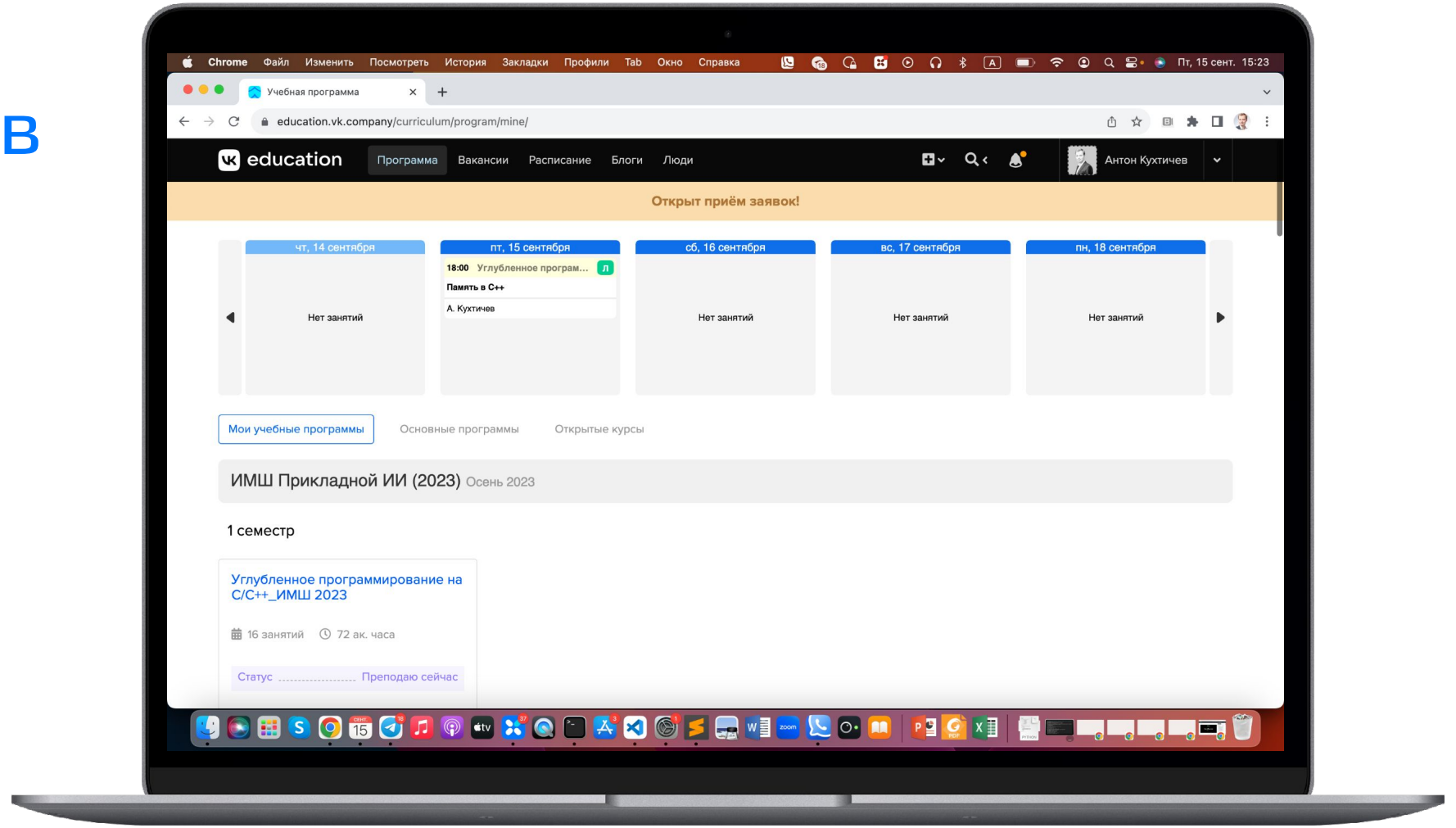
Интерфейс аллокатора и итератора смотрите в документации.

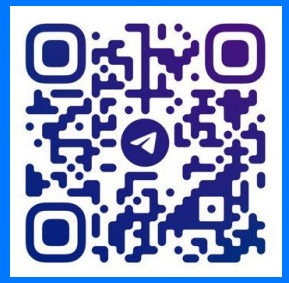
Полезная литература в помощь

- Документация стандартной библиотеки

Напоминание оставить отзыв

Это правда важно





Спасибо
за внимание!