Прософт Системы

Контейнеры и итераторы в C++ (II)

Прософт Системы

СОДЕРЖАНИЕ

- Последовательные контейнеры
- Ассоциативные контейнеры
- Рекомендации по использованию контейнеров

ПОСЛЕДОВАТЕЛЬНЫЕ КОНТЕЙНЕРЫ

- *When choosing a container, remember vector is best. Leave a comment to explain if you choose from the rest*

(c) Tony van Eerd

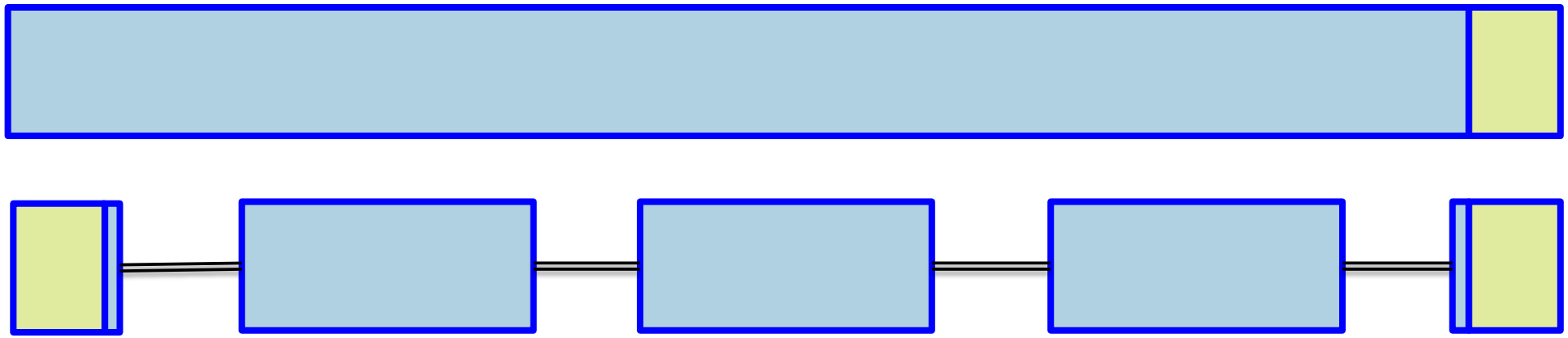
ПОСЛЕДОВАТЕЛЬНЫЕ КОНТЕЙНЕРЫ

- Контейнеры:
  - **vector** — массив с переменным размером и гарантией непрерывности памяти*
  - **array** — массив с фиксированным размером, известным в момент компиляции
  - **deque** — массив с переменным размером без гарантий по памяти
  - **list** — двусвязный список
  - **forward_list** — односвязный список
- Адаптеры:
  - **stack** — LIFO контейнер, чаще всего на базе deque
  - **queue** — FIFO контейнер, чаще всего на базе deque
  - **priority_queue** — очередь с приоритетами, чаще всего на базе vector

DEQUE

- Эффективно растёт в обоих направлениях
- Не требует больших реаллокаций с перемещениями, так как разбит на блоки
- Гораздо меньше фрагментирует кучу

DEQUE



VECTOR VS DEQUE


```cpp
#include <deque>

int main()
{
    std::deque<int> deq;
    for (size_t i = 0; i < 1'000'000; ++i)
    {
        deq.push_front(i);
        deq.push_back(i);
    }
}
```

```cpp
#include <vector>

int main()
{
    std::vector<int> deq;
    for (size_t i = 0; i < 1'000'000; ++i)
    {
        deq.push_back(i);
    }
}
```

А ТАК?


```cpp
#include <vector>

int main()
{
    std::vector<int> deq;
    deq.reserve(1'000'000);
    for (size_t i = 0; i < 1'000'000; ++i)
    {
        deq.push_back(i);
    }
}
```

VECTOR VS DEQUE

Vector:

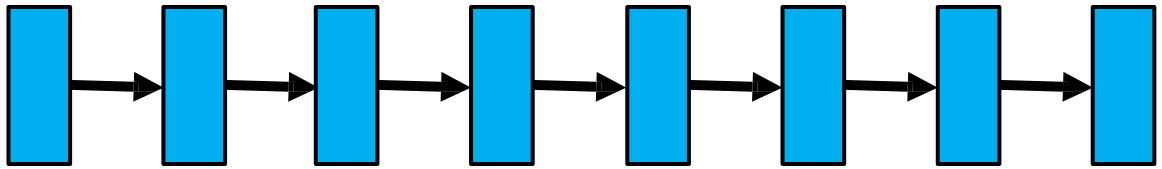
- Доступ к элементу O(1)
- Вставка в конец аморт. O(1)+
- Вставка в начало O(N)
- Вставка в середину O(N)
- Вычисление размера O(1)
- Есть гарантии по памяти
- Есть reserve / capacity

Deque:

- Доступ к элементу O(1)
- Вставка в конец O(1)
- Вставка в начало O(1)
- Вставка в середину O(N)
- Вычисление размера O(1)
- Нет гарантий по памяти
- Нет необходимости в reserve/capacity

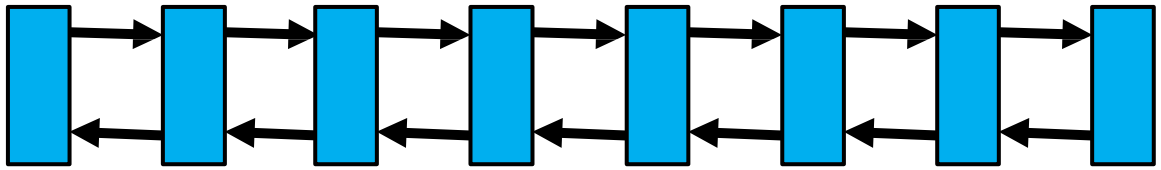
FORWARD_LIST

- Последовательный доступ
- Быстрая вставка в любое место
- НЕ инвалидируются ссылки и итераторы
- Написание этой штуки ваше домашнее задание btw

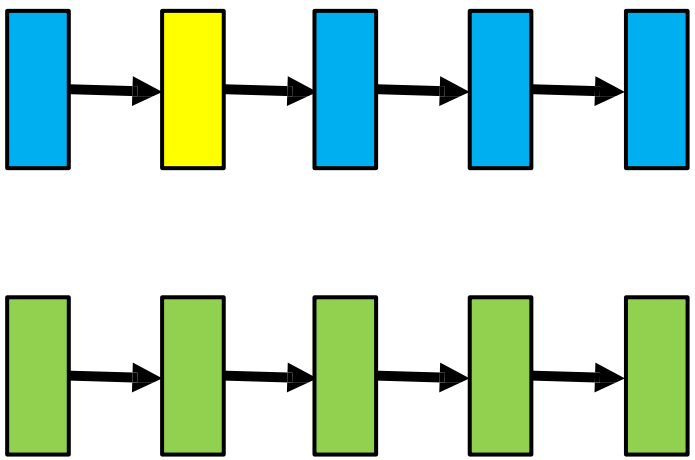


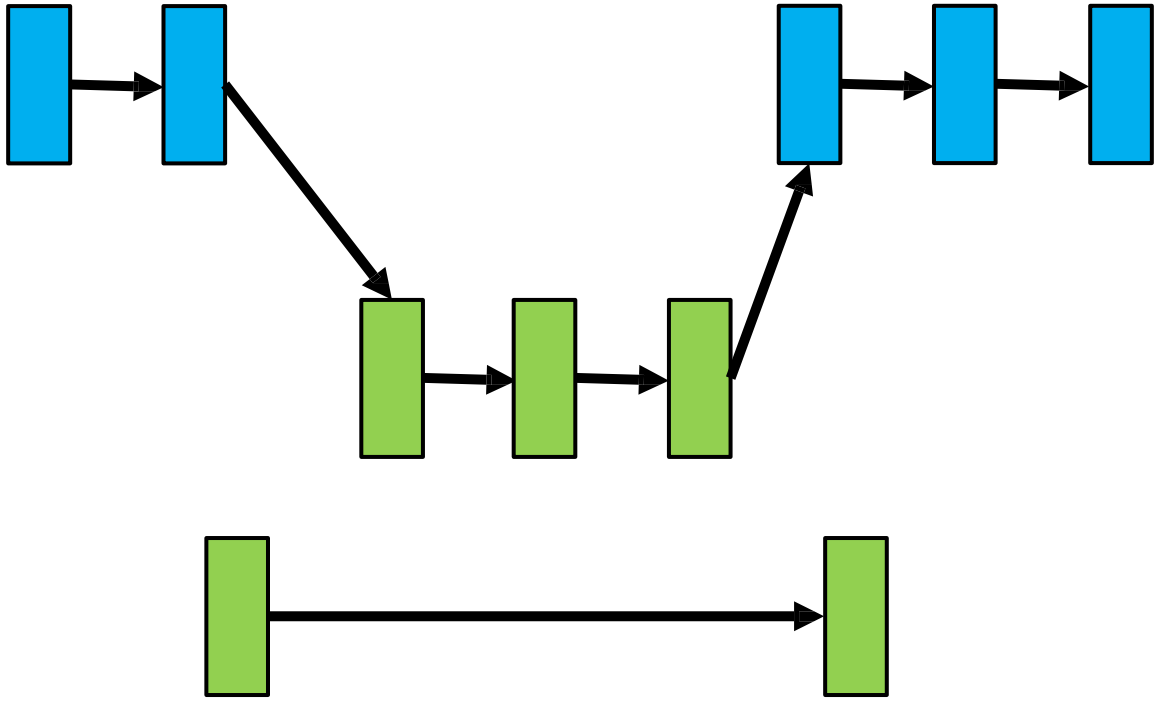
LIST

- Последовательный доступ
- Быстрая вставка в любое место
- НЕ инвалидируются ссылки и итераторы
- Итерация в обе стороны



ФИШЕЧКИ У LIST/FORWARD_LIST АКА СПЛАЙСЫ



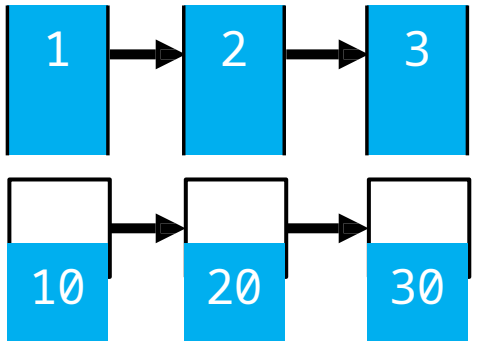


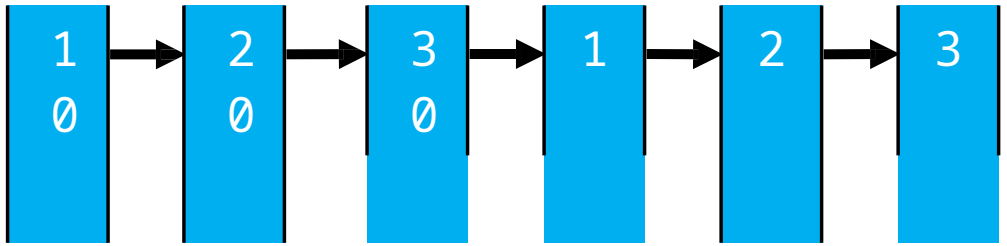
ФИШЕЧКИ У LIST/FORWARD_LIST АКА СПЛАЙСЫ


```cpp
#include <forward_list>

int main()
{
    std::forward_list<int> fst = { 1, 2, 3 };
    std::forward_list<int> snd = { 10, 20, 30 };
    auto it = fst.begin(); // указывает на 1
    // перемещаем second в начало first, it указывает на 1
    fst.splice_after(fst.before_begin(), snd);
}
```

ФИШЕЧКИ У LIST/FORWARD_LIST AKA СПЛАЙСЫ





ВСПОМНИМ БАЗУ, СТРОКИ


```
template <typename CharT> class basic_string
{
    // implementation
};
typedef basic_string<char> string;
typedef basic_string<u16char_t> u16string;
typedef basic_string<u32char_t> u32string;
typedef basic_string<wchar_t> wstring;
```

ВСПОМНИМ БАЗУ, СТРОКИ


```
template <typename CharT> class char_traits
{
    // assign, eq, lt, compare, find, eof
}
template <typename CharT, typename Traits = std::char_traits<CharT>>
class basic_string
{
    // implementation
};
typedef basic_string<char> string;
typedef basic_string<u16char_t> u16string;
typedef basic_string<u32char_t> u32string;
typedef basic_string<wchar_t> wstring;
```

ВСПОМНИМ БАЗУ, СТРОКИ

Является ли способ выделения памяти характеристикой символа из Traits?

АЛЛОКАТОРЫ


```
template <typename CharT,
typename Traits = std::char_traits<CharT>,
typename Allocator = std::allocator<CharT>>
class basic_string
{
    // implementation
};
```

СТАТИЧЕСКИЕ СТРОКИ


```
static const std::string kName = "very important string literal";
// .....
int foo(const std::string &arg);
// .....
foo(kName);
```

СТАТИЧЕСКИЕ СТРОКИ


```
constexpr std::string_view kName = "very important string literal";
// .....
int foo(std::string_view arg);
// .....
foo(kName);
```

СТАТИЧЕСКИЕ СТРОКИ


```
constexpr std::string_view kName = "very important string literal";
// .....
int foo(std::string_view arg);
// .....
foo(kName);
```

STRING_VIEW

- remove_prefix
- remove_suffix
- copy
- substr
- compare
- find
- data

```
std::string str = " trim me ";
std::string_view sv = str;

auto trimfst = sv.find_first_not_of(" ");
auto minsz = std::min(trimfst, sv.size());

sv.remove_prefix(minsz);

auto trimlst = sv.find_last_not_of(" ");
auto sz = sv.size() - 1;

minsz = std::min(trimlst, sz);
sv.remove_suffix(sz - minsz);
```

SPAN (C++20)


```
int oldArray[4] = {1, 2, 3, 4};
std::array<int, 4> newArray = {1, 2, 3, 4};
std::span<int, 4> viewArr = {1, 2, 3, 4};
std::span<int, 4> viewArr = {newArray};
```

SPAN (C++20)

- std::span для одномерных массивов то же, что string_view для строк

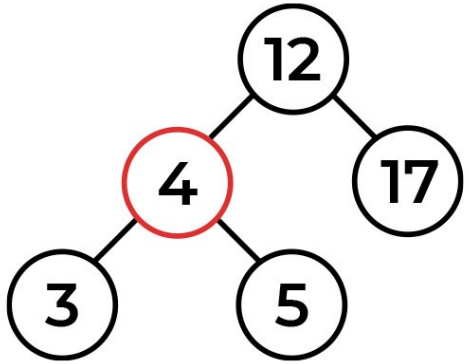
ВНИМАНИЕ ИНТЕРАКТИВЧИК!

АССОЦИАТИВНЫЕ КОНТЕЙНЕРЫ

- Контейнеры:
  - **map** — контейнер, содержащий упорядоченные пары ключ – значение, под капотом работает как красно-черное дерево
  - **unordered_map** – контейнер, содержащий пару ключ – значение, под капотом работает как хэш-таблица
  - **set** – контейнер, содержащий упорядоченные уникальные ключи, под капотом работает как красно-черное дерево
  - **unordered_set** – контейнер, содержащий уникальные ключи, под капотом работает как хэш-таблица

MAP

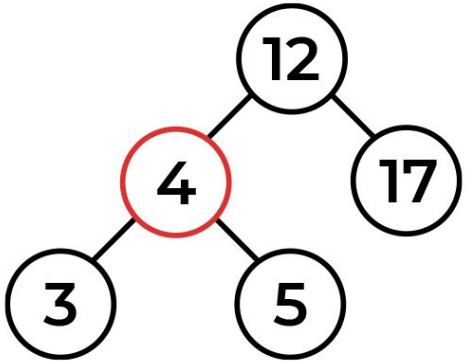
- Гарантированное время поиска, вставки и удаления - O(log n)
- Элементы хранятся в отсортированном порядке
- Автоматическая балансировка при модификациях



MAP

ЧЕМ ДАЛЬШЕ В ЛЕС...

- Red-black tree (наше родненькое stdлибное)
- AVL tree
- B-дерево (Google lib, Abseil => abseil_map)
- Splay tree
- Treap tree aka Декартово дерево (multithread implementations)



ПОЙДЕМ С САМОГО НАЧАЛА, ДЕРЕВО 2-3

Два-три дерево - абстрактный тип данных, напоминающий по структуре дерево. В нодах два-три дерева может быть одно или два значения и два или три потомка

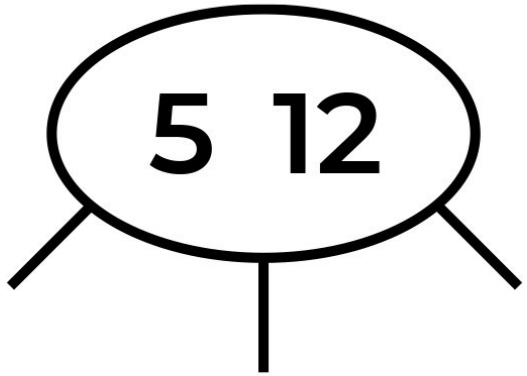
Ноду с одним значением и двумя потомками будем называть 2-нода, ноду с двумя значениями и тремя потомками – 3-нода.

ПОЙДЕМ С САМОГО НАЧАЛА, ДЕРЕВО 2-3

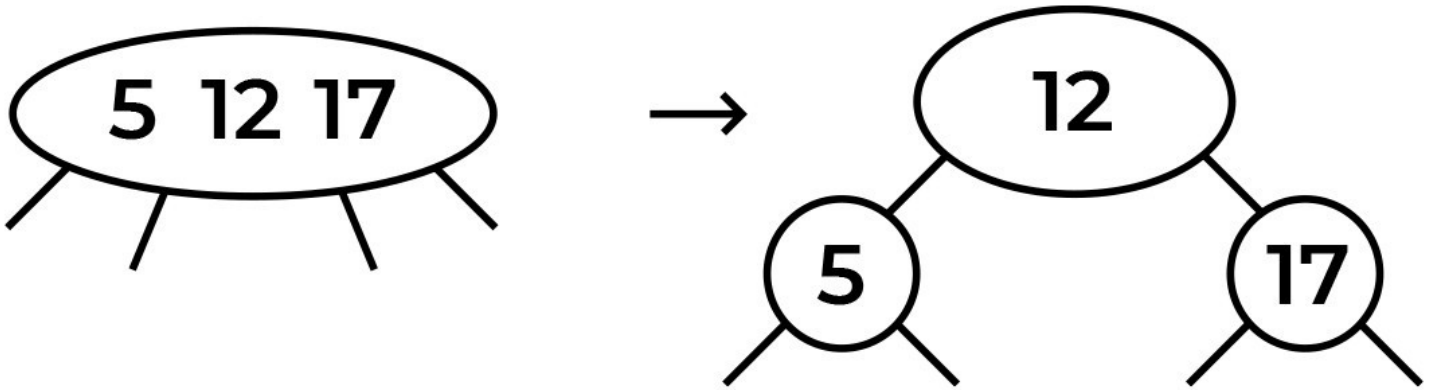
- Добавляя элемент, мы всегда спускаемся вниз по дереву.
- Дерево отсортировано классически - меньшие значения находятся слева, бОльшие - справа.
- Два-три дерево - отсортированное, сбалансированное дерево.

ПОЙДЕМ С САМОГО НАЧАЛА, ДЕРЕВО 2-3

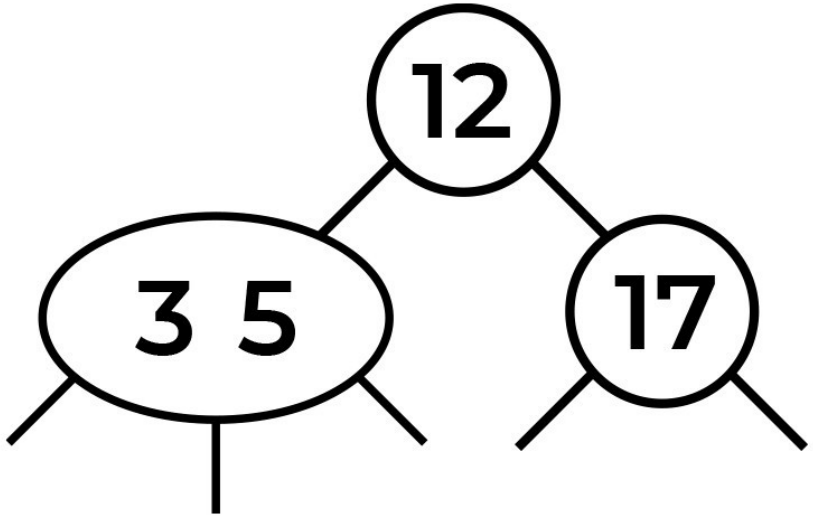
ПОЙДЕМ С САМОГО НАЧАЛА, ДЕРЕВО 2-3



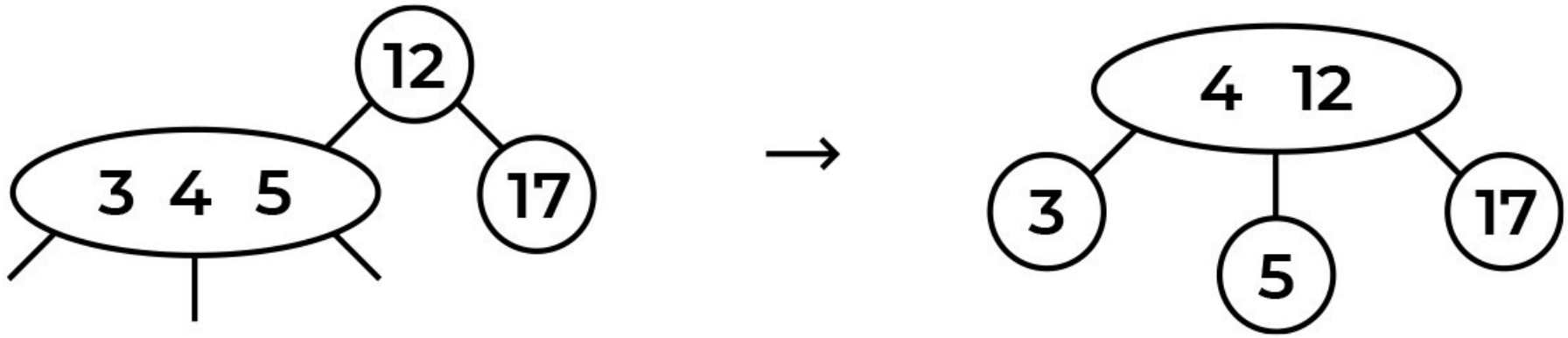
ПОЙДЕМ С САМОГО НАЧАЛА, ДЕРЕВО 2-3



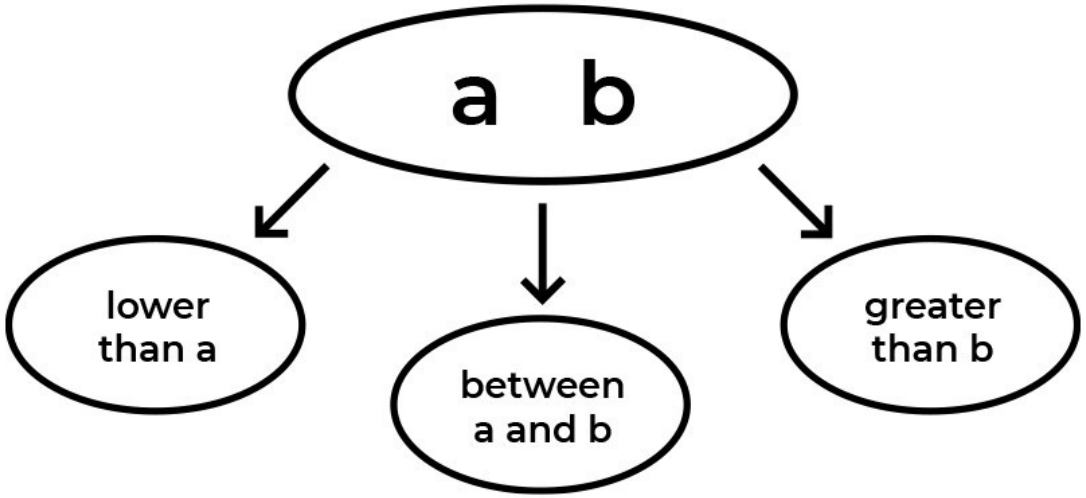
ПОЙДЕМ С САМОГО НАЧАЛА, ДЕРЕВО 2-3



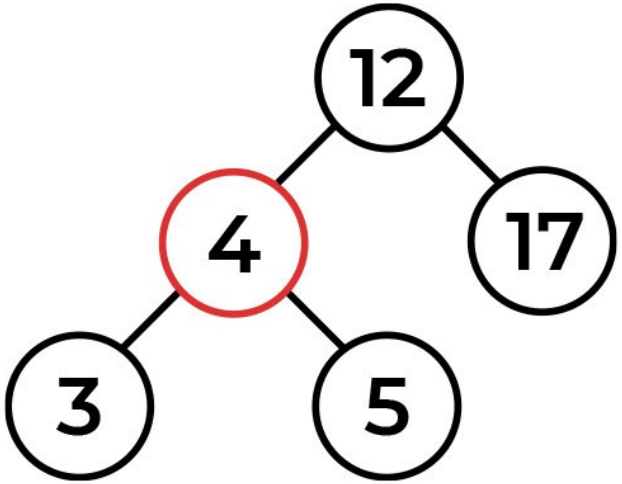
ПОЙДЕМ С САМОГО НАЧАЛА, ДЕРЕВО 2-3



ПОЙДЕМ С САМОГО НАЧАЛА, ДЕРЕВО 2-3



ИЗОБРЕТАЕМ КРАСНО-ЧЕРНОЕ ДЕРЕВО



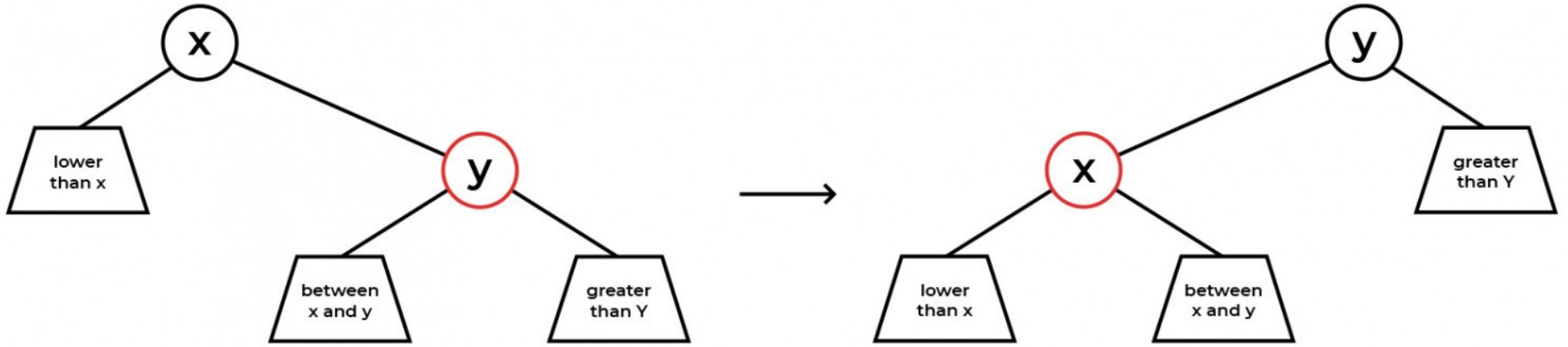
ИЗОБРЕТАЕМ КРАСНО-ЧЕРНОЕ ДЕРЕВО

- Две красные ноды не могут идти подряд. Это свойство приобретает смысл, если мы знаем, что красная нода - это по сути часть 3-ноды в 2-3 дереве
- Корень дерева всегда черный. Опять же, тут все понятно: красная нода не может существовать без родителя.
- Все null-ноды (ноды, которые не имеют потомков) – черные.
- Высота дерева измеряется только по черным нодам и называется "черной высотой".

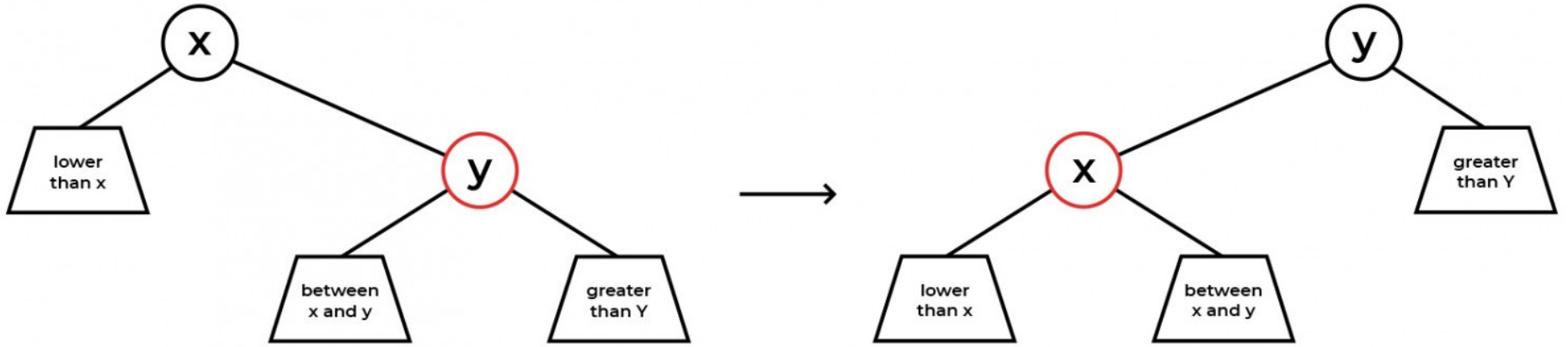
ИЗОБРЕТАЕМ КРАСНО-ЧЕРНОЕ ДЕРЕВО

- Все вставленные ноды, кроме корня дерева, вставляются с красным цветом. Объясняется это тем, что мы всегда сначала добавляем значение к уже существующей ноде и только после этого занимаемся балансировкой
- Мы разбираем левостороннее красно-черное дерево, из этого следует, что красные ноды могут лежать только слева (обратный случай требует балансировки).

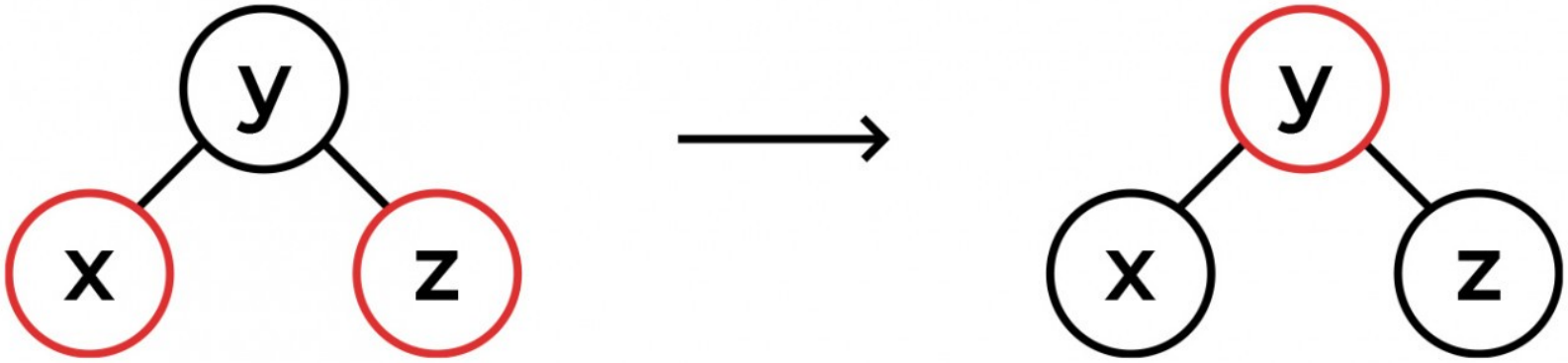
ВСПЫШКА СЛЕВА, ЛОЖИСЬ! АКА ЛЕВОСТОРОННИЙ ПОВОРТ



ВСПЫШКА СПРАВА, ЛОЖИСЬ! АКА ПРАВОСТОРОННИЙ ПОВОРТ

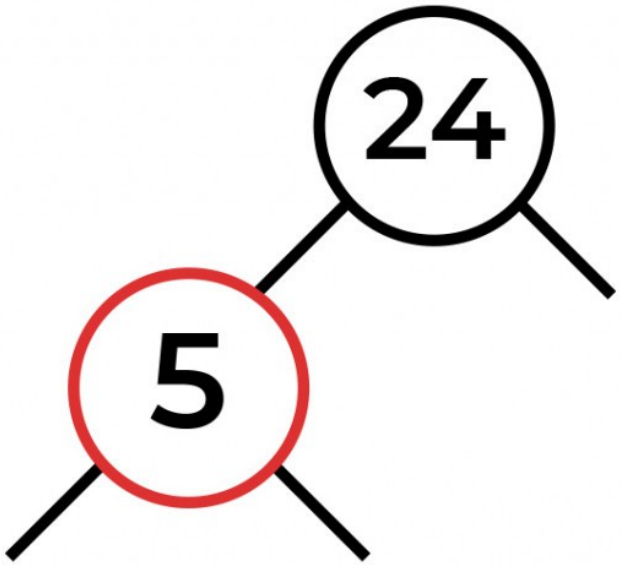


ЗИМОЙ И ЛЕТОМ ОДНИМ ЦВЕТОМ

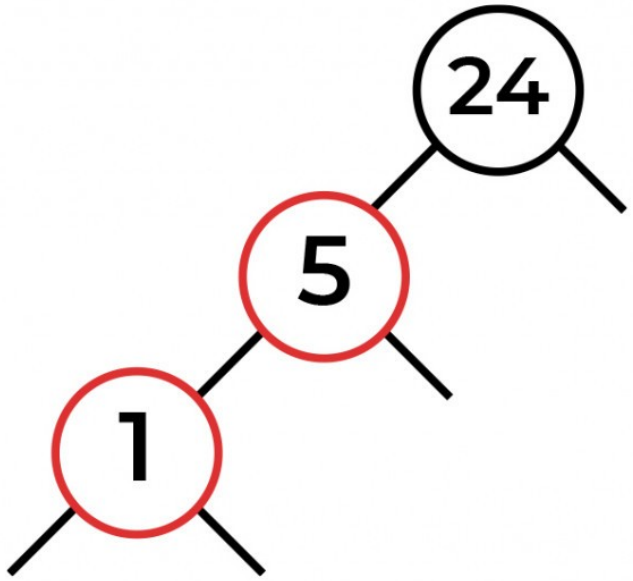


ПОСТРОИМ ДЕРЕВО

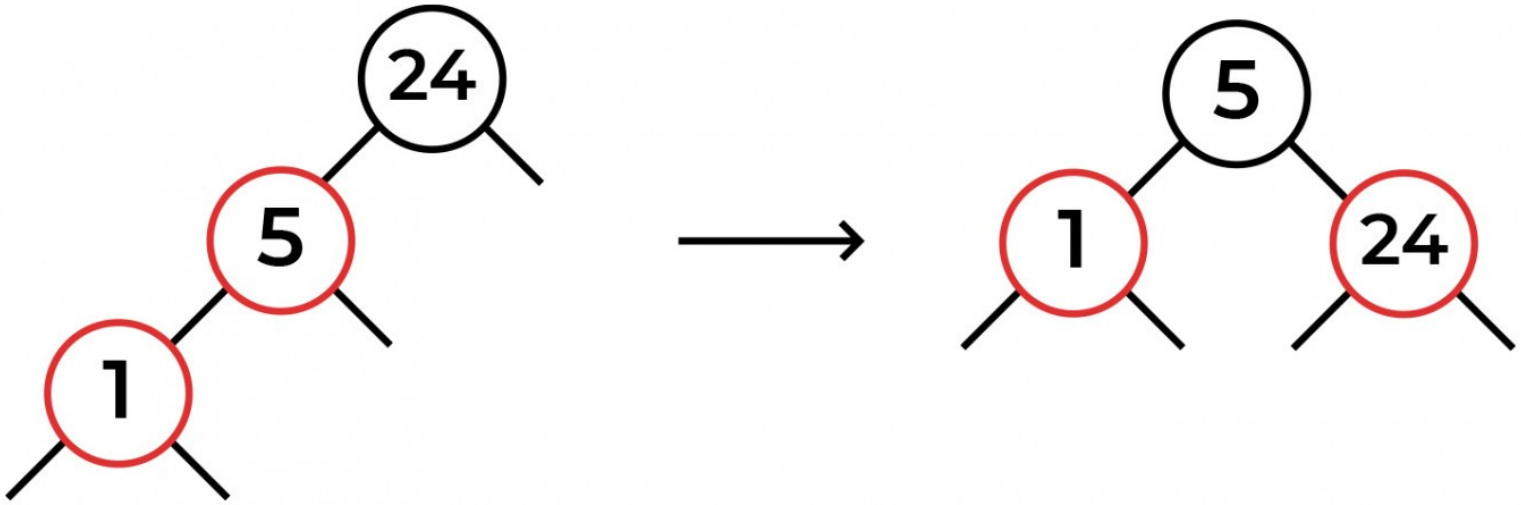
ПОСТРОИМ ДЕРЕВО



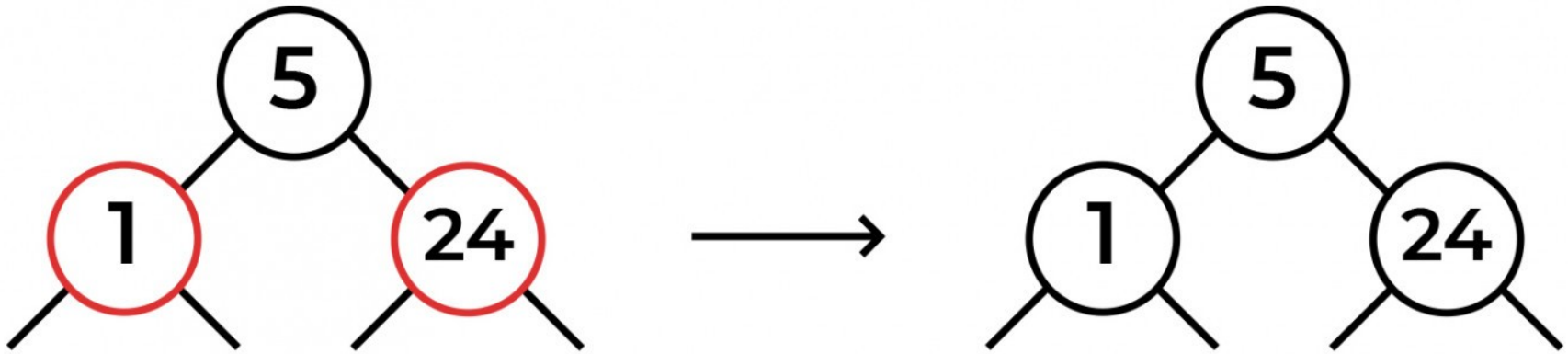
ПОСТРОИМ ДЕРЕВО



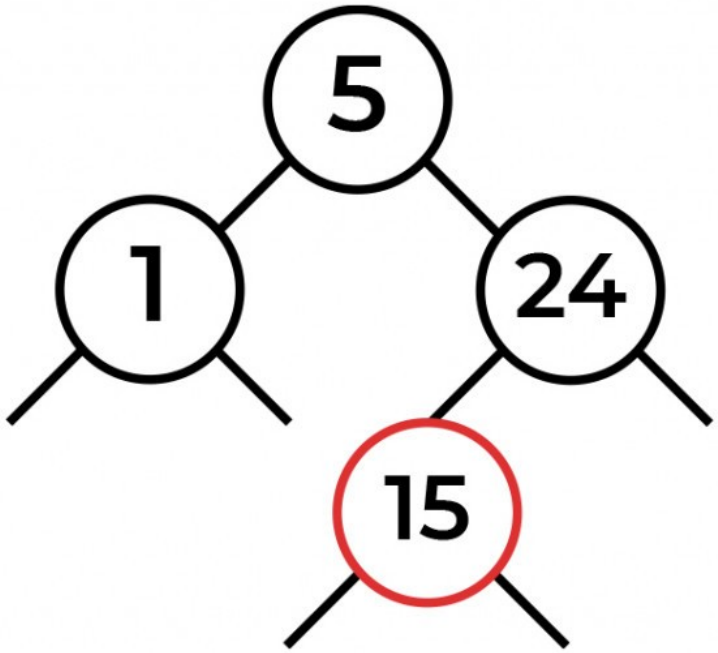
ПОСТРОИМ ДЕРЕВО



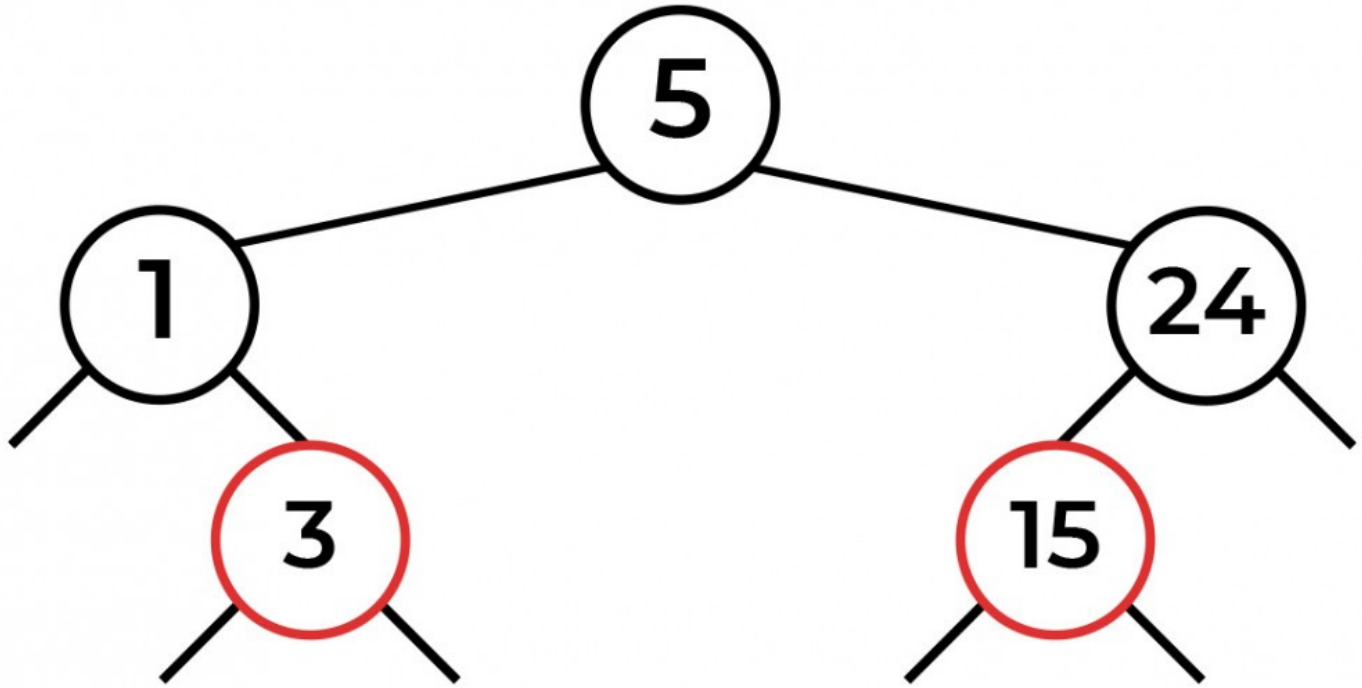
ПОСТРОИМ ДЕРЕВО



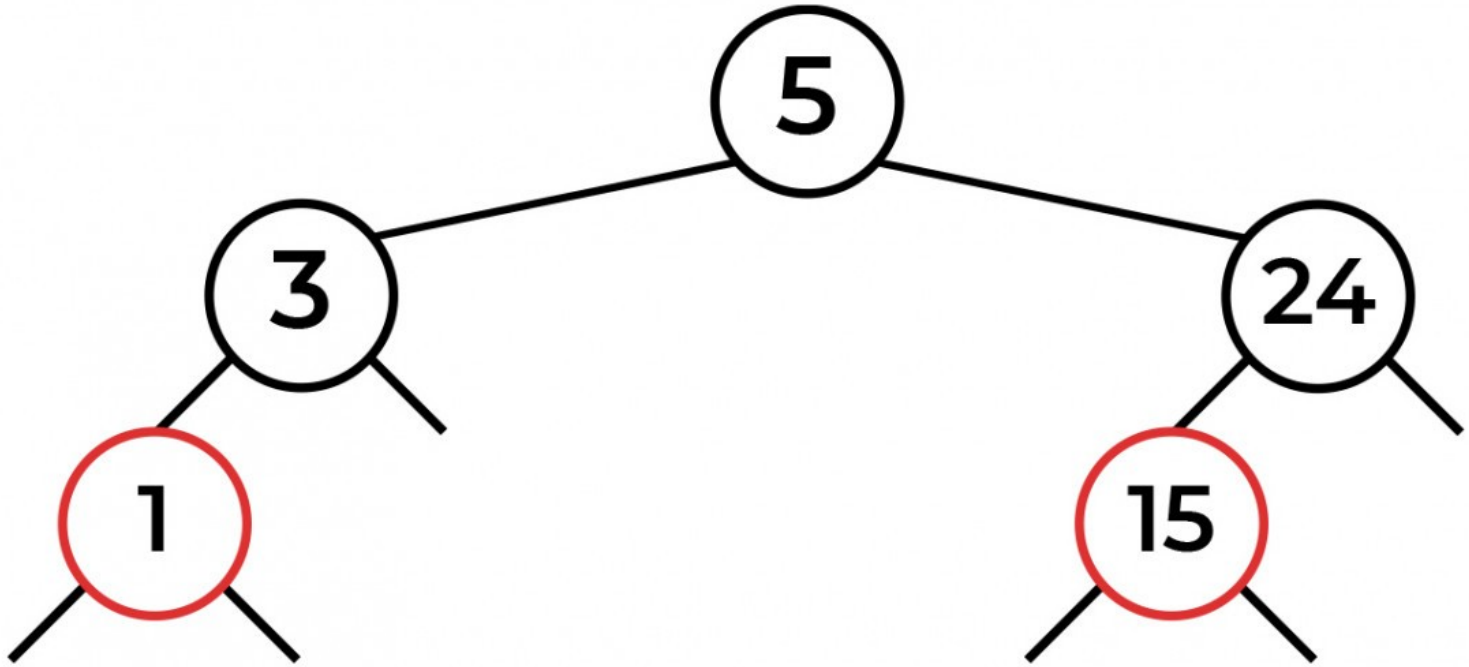
ПОСТРОИМ ДЕРЕВО



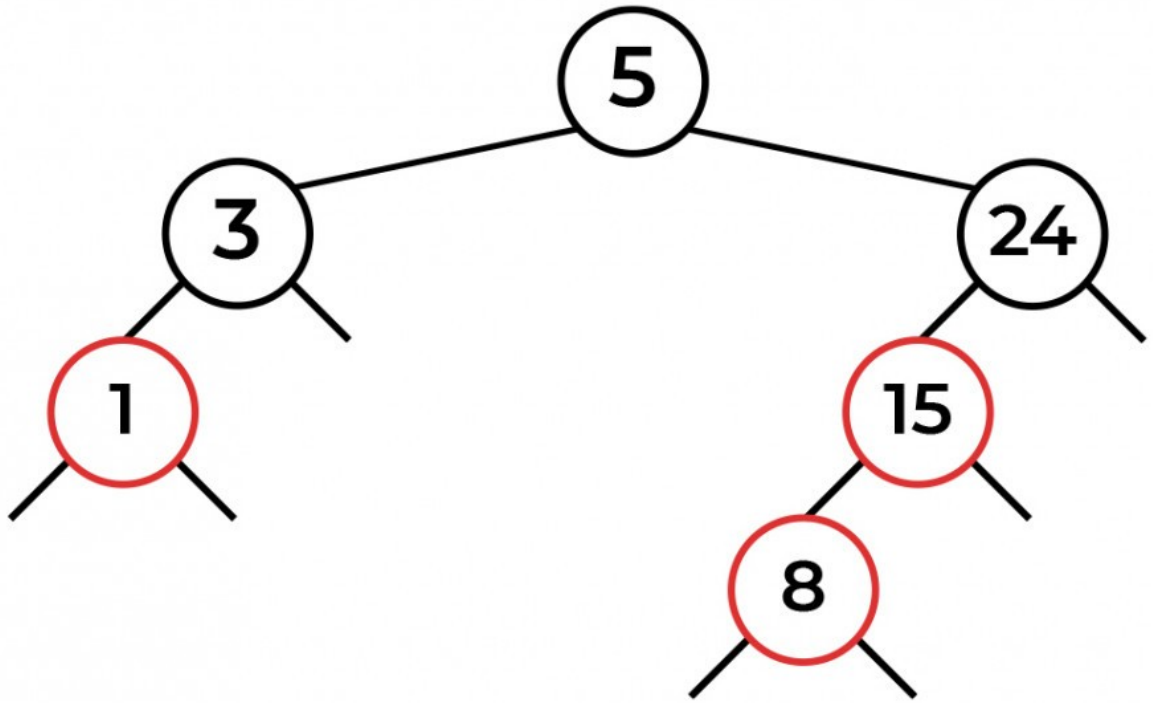
ПОСТРОИМ ДЕРЕВО



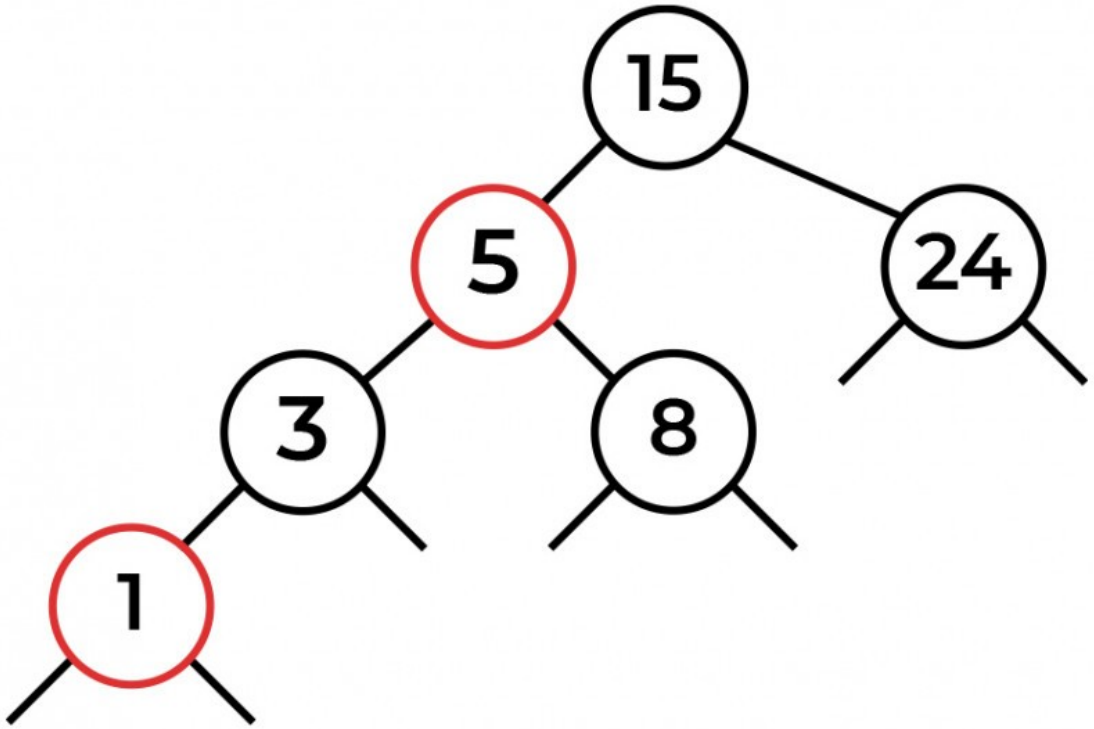
ПОСТРОИМ ДЕРЕВО



ПОСТРОИМ ДЕРЕВО



ПОСТРОИМ ДЕРЕВО



ВЕРНЕМСЯ К C++

- Придумайте интерфейс стандартной мапы, прям с шаблонами и вот этим всем.

ОТВЕТ УБИЛ...


```
template<
class Key,
class T,
class Compare = std::less<Key>,
class Allocator = std::allocator<std::pair<const Key, T>>>
class map;
```