Novosibirsk State University
*THE REAL SCIENCE

Программирование на C++ и Python

Лекция 3

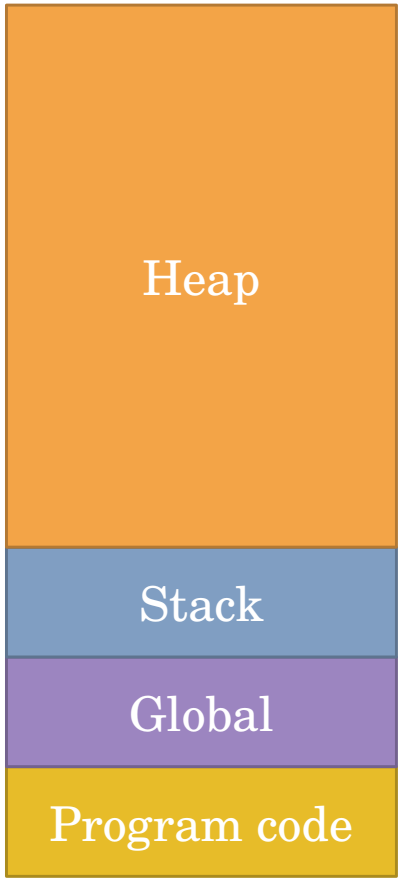
Динамическая память.
Объектно-ориентированное программирование

Воробьев Виталий Сергеевич (ИЯФ, НГУ)

Новосибирск, 29 сентября 2021

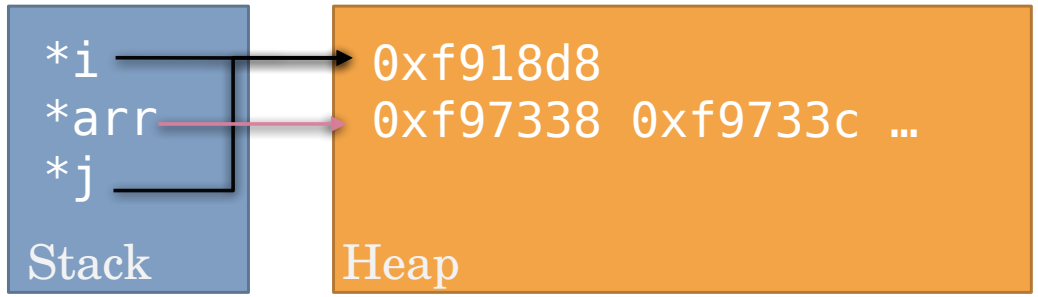
Память в программе *C++*

- Стек (stack) – быстрая «автоматическая» память
  - Отвечает за вызовы функций и хранение автоматических переменных
  - Является стеком
  - Имеет небольшой размер ()
  - При выходе из области видимости автоматические переменные уничтожаются в порядке обратном созданию
- Куча (heap) – большая доступная для динамического выделения память
  - Работа с кучей ведется с помощью *указателей*
  - Значительно *медленнее*, чем стек
  - В Си программист отвечает за освобождение динамической памяти. В стандартной библиотеке *C++* есть «умные» указатели



Динамическая память

- Зачем нужна динамическая память?
  - Чтобы хранить большие структуры данных
  - Чтобы лучше контролировать время жизни объектов
- Динамическая память на низком уровне: `new`, `delete`, `delete[]`
  - Не забыть вызвать `delete` для каждого `new`
  - Не перепутать `delete` и `delete[]`
- Контейнеры `vector`, `set` и др. хранят данные в куче




```
int main() {
    int* i = new int(9);
    int* arr = new int[10];
    for (int i = 0; i < 10; ++i)
{
        cout << arr + i << endl;
        arr[i] = i;
    }


cout << i << ": " << *i << endl;
    int *j = i;
    *j += 1;  // *i == 10
    cout << j << ": " << *i << en
dl;

    delete i;
    delete[] arr;

    return 0;
}
```

```
> 0xf97338
> 0xf9733c
> 0xf97340
> 0xf97344
> 0xf97348
> 0xf9734c
> 0xf97350
> 0xf97354
> 0xf97358
> 0xf9735c
> 0xf918d8: 9
> 0xf918d8:
10
```

<memory>

Идиома RAII

Получение ресурса есть инициализация
Resource acquisition is initialization

- Программы работают с ресурсами: память, файловые дескрипторы, сетевые соединения, …
- *Идея*: для каждого ресурса иметь *объект на стеке*, при инициализации которого ресурс выделяется, а при удалении – освобождается
- Контейнеры стандартной библиотеки *C++* и объект `fstream` – примеры реализации идиомы RAII
- Объекты `unique_ptr` и `shared_ptr` из библиотеки `<memory>` реализуют идиому RAII для динамической памяти

```
using Record = tuple<string, int, double>;

int main() {
    auto iptr = make_shared<Record>(
        "NSU", 2021, 3.1415);

    cout << iptr << ": " << get<0>(*iptr) << en
dl;
    shared_ptr<Record> jptr = iptr;
    cout << jptr << ": " << get<1>(*iptr) << en
dl;

    return 0;
```

```
> 0xfb7348: NSU
> 0xfb7348: 2021
```

<memory>

Умные указатели

- `shared_ptr`
  - Содержит *счетчик ссылок* на объект
  - Может быть скопирован
  - Освобождает память, когда счетчик ссылок достиг нуля
- `unique_ptr`
  - *Владеет* объектом
  - Не может быть скопирован
  - Может передать владение (`std::move`)
  - Не уступает по эффективности низкоуровневым указателям
- Вспомогательные функции для создания умных указателей
  - `make_unique`
  - `make_shared`
- Есть еще `weak_ptr`, но я не буду о нем говорить

Пример

```
using Record = tuple<string, int, double>;

unique_ptr<Record> make_record() {
    return make_unique<Record>("Sam", 6, 4.1)
;
}

void process_record(unique_ptr<Record> rec) {
    auto [s, d, x] = *rec;
    cout << '[' << s << ", "
         << d << ", " << x << "]\n";
}

int main() {
    auto rptr = make_record();
    process_record(move(rptr));


    return 0;
}
```

Передача владения

```
> [Sam, 6, 4.1]
```

```
using Record = tuple<string, int, double>;

shared_ptr<Record> make_record() {
    return make_shared<Record>("Sam", 6, 4.1)
;
}

void process_record(shared_ptr<Record> rec) {
    auto [s, d, x] = *rec;
    cout << '[' << s << ", "
         << d << ", " << x << "]\n";
}

int main() {
    auto rptr = make_record();
    process_record(rptr);
    process_record(rptr);


    return 0;
}
```

```
> [Sam, 6, 4.1]
> [Sam, 6, 4.1]
```

lvalue и rvalue

- lvalue – это выражение, для которого существует адрес. Все выражения *слева* от оператора присваивания являются lvalue
- rvalue – это безымянное* выражение, которое существует только до тех пор, пока оно вычисляется
- Оператор & обозначает ссылку на lvalue выражение
- Оператор && обозначает ссылку на rvalue выражение
- Функция std::move переводит lvalue в rvalue

```
int x = 1;       // x - lvalue, 1 - rvalue
int y = x + 1;   // y - lvalue, x + 1 - rval
ue
int z = x + y;   // z - lvalue, x + y - rval
ue

string getName() {
    return "Name";
}
// string& sref = getName();  // error
string&& sref = getName();
```

Классы

- *C++* позволяет создавать пользовательские типы данных – классы
- Класс состоит из элементов двух типов:
  - Поля – переменные класса
  - Методы – функции класса
- Элементы класса могут быть приватными (`private`) или публичными (`public`)
- Элементы класса могут быть константными
  - Константные поля не могут менять значение
  - Константные методы могут вызываться только у константных объектов
- Элементы класса могут быть статическими (`static`). В этом случае они относятся к самому классу, а не к его *объектам*.

```
class Point {
    double m_x;
    double m_y;

 public:
    Point(double x, double y) :
        m_x(x), m_y(y) {}

    double x() const {
        return m_x;
    }
    double y() const {
        return m_y;
    }
    double r() const {
        return sqrt(m_x *
m_x + m_y * m_y);
    }
};
```

Использование класса

```
int main() {
    Point p(3, 4);
    // cout << p.m_x;  // m_x is private
    cout << p.x() << ", " << p.y() << ". Radius " << p.r() << e
ndl;
    return 0;
}
```

```
> 3, 4. Radius 5
```

Статические поля и методы

```
int main() {
    vector<Point> pvec;
    for (int i = 0; i < 10; ++i)
        pvec.emplace_back(i, 2 * i);
    cout << Point::nobjects() << endl;
    return 0;
}
```

```
> 10
```

- В какой области памяти располагаются статические поля?
- Каково время жизни статических полей?
- Могут ли статические методы быть константными?

```
class Point {
    static size_t m_count;

 public:
    Point(double x, double y) :
            m_x(x), m_y(y) {
        ++m_count;
    }
    static size_t nobjects() {
        return m_count;
    }
};

// нужна инициализация
size_t Point::m_count = 0;
```

Константные объекты

```
int main() {
    const Point v(3, 4);
    Point w(5, 8);
    w.setx(6);
    // v.setx(2);
    cout << w.x() << ", " << w.y()
         << ", " << w.r() << endl;
    cout << v.x() << ", " << v.y()
      /* << ", "  << v.r() */  << endl;

    return 0;
}
```

```
> 6, 8, 10
> 3, 4
```

- Используйте константность. Это простой способ избежать многих ошибок

```
class Point {
    double m_x;
    double m_y;

 public:
    Point(double x, double y) :
        m_x(x), m_y(y) {}

    void setx(double val) {m_x = val;}
    double x() const {
        return m_x;
    }
    double y() const {
        return m_y;
    }
    double r() /* const */ {
        return sqrt(m_x *
m_x + m_y * m_y);
    }
};
```

Специальные методы

- Есть шесть специальных методов. Компилятор может создавать все или часть из них автоматически
- Конструктор вызывается при создании объекта
  - Конструкторов – с разными аргументами – может быть несколько
- Деструктор вызывается при уничтожении объекта
- *Перемещение* может быть эффективнее копирования
- && обозначает ссылку на `rvalue`
- Для специальных методов можно указывать идентификаторы `default` и `delete`

```
class Point {
 public:
    // 1. Конструктор по умолчанию
    Point() : m_x(0), m_y(0) {}
    // 2. Копирующий конструктор
    Point(const Point& p) :
        m_x(p.m_x), m_y(p.m_y) {}
    // 3. Перемещающий конструктор
    Point(const Point&& p) :

m_x(move(p.m_x)), m_y(move(p.m_y)) {}
    // 4. Оператор копирования
    Point& operator=(const Point& p) {
        m_x = p.m_x;
        m_y = p.m_y;
        return *this;
    }
    // 5. Оператор перемещения
    Point& operator=(Point&& p) {
        m_x = move(p.m_x);
        m_y = move(p.m_y);
        return *this;
    }
    // 6. Деструктор
    ~Point() = default;
};
```

Сколько раз вызван конструктор?

```
int main() {
    const Point a(3, 4);
    Point b;
    const Point c = a;
    b = a;

    vector<Point> pvec(3);
    double i = 1;
    generate(pvec.begin(), pvec.end(), [&i]() {
        return Point{i *= 2, 2 * i};
    });

    pvec.push_back({7, 8});
    pvec.emplace_back(7, 8);

    cout << Point::nobjects() << endl;
    return 0;
}
```

```
> Custom ctor
> Default ctor
> Copy ctor
> Copy operator
```

```
> 3 x Default ctor
```

```
> 3 x (Custom ctor + Move
operator)
```

```
> Custom ctor + Move ctor
  + 3 x Copy ctor (???)
> Custom ctor
```

```
> 15
```

Перегрузка операторов

- Оператор – это функция со специальным способом вызова
- Перегрузка операторов позволяет комфортно работать с пользовательскими классами

```
ostream& operator<<(ostream& os, const Point& p) {
    return os << '(' << p.x() << ", " << p.y() << '
)';
}

Point operator+
(const Point& lhs, const Point& rhs) {
    return {lhs.x() + rhs.x(), lhs.y() + rhs.y()};
}
```

```
const Point a(3, 4);
const Point b(5, 6);
const Point c = a + b;
cout << a << " " << b << " " << c << endl
;
```

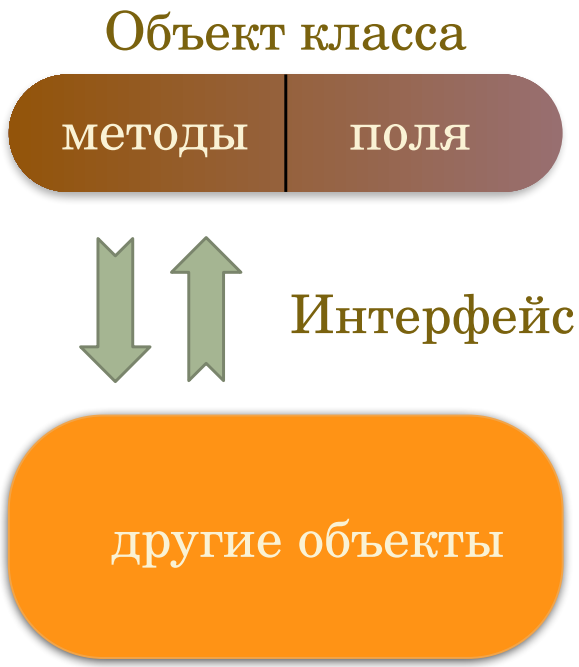
```
> (3, 4) (5, 6) (8, 10)
```

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

- Если оператор должен иметь доступ к приватным полям и методам, его можно сделать членом класса или использовать идентификатор `friend`

Инкапсуляция

- Классы реализуют инкапсуляцию – одну из фундаментальных идей объектно-ориентированного программирования (ООП)
- *Идея*:
  1. Объединяем данные и методы работы с ними в одном объекте
  2. Ясно определяем *интерфейс* – все способы взаимодействия с этих объектом
- Достоинства инкапсуляции:
  1. *Модульность*. Сохранив интерфейс, можно полностью изменить способ хранения и работы с данными в объекте. Остальной код программы может останется неизменным
  2. *Безопасность*. Приватность полей защищает их от некорректного изменения



Пример: бинарное дерево поиска

```
class TreeNode;
using TreeNodePtr = std::shared_ptr<TreeNod
e>;

class TreeNode {
    const int m_value;
    TreeNodePtr m_right;
    TreeNodePtr m_left;
 public:
    TreeNode(int value) : m_value(value)
        m_right(nullptr), m_left(nullptr)
{}
    int value() const {return m_value;}
    TreeNodePtr insert(int value) {…}
    TreeNodePtr find (int value) const {…}
    TreeNodePtr erase (int value) {…}
};
```

- Инварианты BST:
  1. Все элементы левого поддерева меньше данного элемента
  2. Все элементы правого поддерева больше данного элемента

```
class Tree {
    TreeNodePtr m_root;
 public:
    Tree () : m_root(nullptr) {}
    void insert(int value) {…}
    void erase (int value) {…}
    bool contains (int value) const
{…}
    // …
};
```

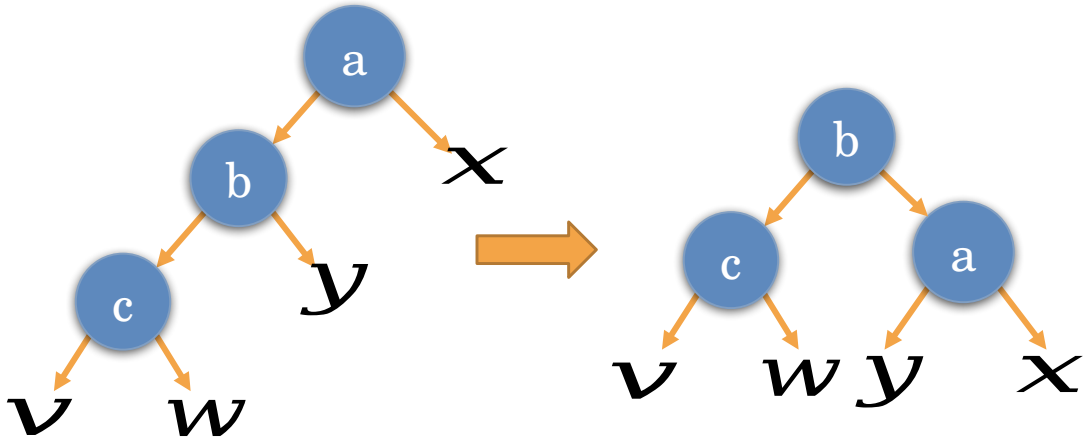
Поиск и вставка элементов

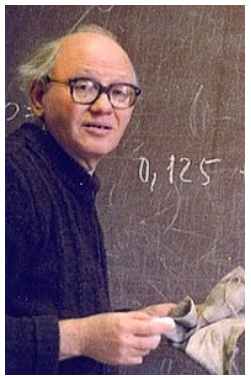
```
TreeNodePtr insert(int value) {
    if (value < m_value) {
        if (m_left) return m_left->insert(value);
        return m_left = make_shared<TreeNode>(value);
    }
    if (m_value < value) {
        if (m_right) return m_right->insert(value);
        return m_right = make_shared<TreeNode>(value);
    }
    return nullptr;
}
TreeNodePtr find(int value) const {
    if (value < m_value)
        return m_left ? m_left->find(value) : nullptr;
    if (m_value < value)
        return m_right ? m_right-
>find(value) : nullptr;
    return this;
}
```

- Любое поддерево BST является BST
- *Какова алгоритмическая сложность этих операций?*

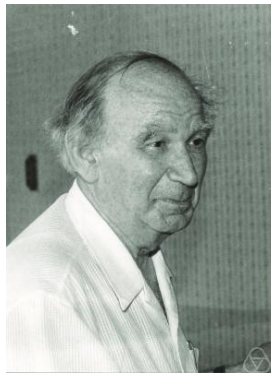
Сбалансированные деревья

- Высота сбалансированного дерева пропорциональна
- АВЛ деревья (1962)
  - Поддерживают инвариант: разность высот левого и правого поддеревьев не превосходит 1
- Red-black tree (1972)
  - Сложнее, но быстрее, чем АВЛ
  - `std::set` и `std::map` реализованы как red-black tree

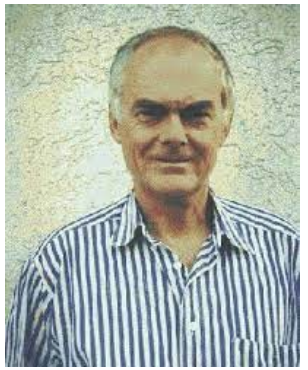




Георгий Максимович **А**дельсон-**В**ельский

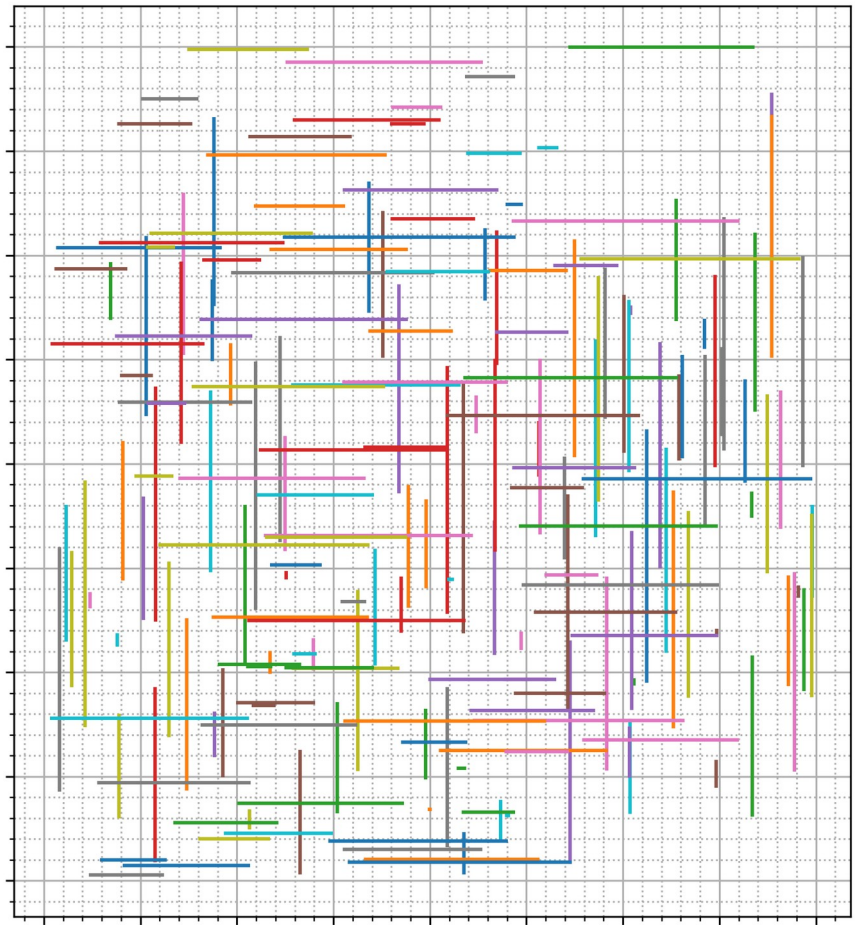


Евгений Михайлович **Л**андис



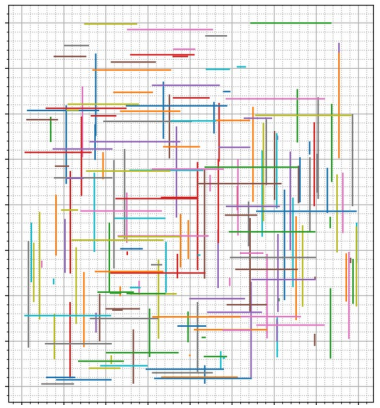
Rudolf Bayer

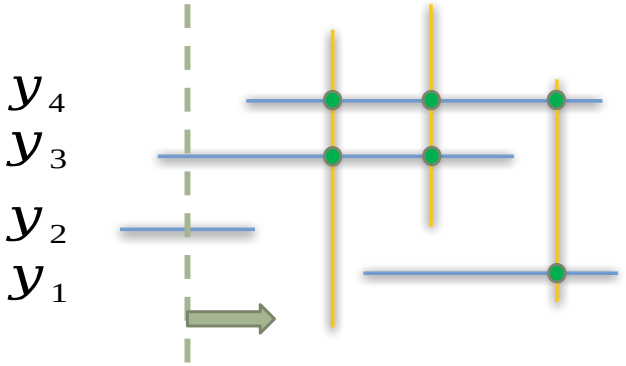
Подсчет пересечений отрезков



- На плоскости задано множество из горизонтальных и вертикальных отрезков
  - Вертикальные отрезки не пересекаются друг с другом
  - Горизонтальные отрезки не пересекаются друг с другом
- Подсчитать количество пересечений отрезков
- Как определить пересекаются ли данные вертикальный и горизонтальный отрезки?
- *Каково максимально возможное количество пересечений?*

Подсчет пересечений отрезков



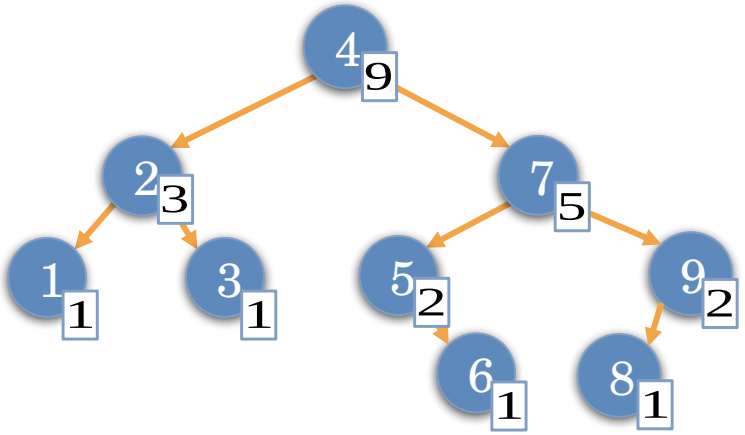


$status\{y_1, y_3, y_4\}$ $n = 3$

- Наивный алгоритм: перебираем все пары отрезков и проверяем их пересечение
  - *Какова сложность этого алгоритма?*
- Менее наивный алгоритм
  1. Сортируем отрезки по координате
  2. Сканируем плоскость вертикальной прямой слева направо
  3. Поддерживаем контейнер `status`, который содержит координаты горизонтальных отрезков, пересекающих сканирующую прямую
  4. При достижении вертикального отрезка , подсчитываем количество элементов в `status` в диапазоне

Ранг элемента в дереве

- Структура `status` может быть реализована как BST
- Чтобы эффективно искать количество элементов в заданном диапазоне необходимо в каждом узле хранить размер поддерева
- `std::set` позволяет найти количество элементов в диапазоне лишь за линейное время:




```
std::distance(
    status.lower_bound(ylo),
    status.upper_bound(yhi)
);
```

- Я написал АВЛ дерево с поддержкой эффективного подсчета элементов в диапазоне значений. Посмотрим насколько это позволит ускорить работу алгоритма поиска пересечений

```
size_t lcount() const {
    return m_left ? m_left->m_count : 0;
}
size_t rrank(int value) const {
    return m_right ? m_right-
>rank(value) : 0;
}
size_t rank(int value) const {
    if (value < m_value) return lrank(value);
    if (m_value < value)
        return 1 + lcount() + rrank(value);
    return lcount();
}
```

```cpp
#include <chrono>
#include <functional>
using namespace std::chrono;

auto duration_ms(std::function<void()> cb) {
    auto start = high_resolution_clock::now();
    cb();
    auto stop = high_resolution_clock::now();
    return duration_cast<microseconds>(stop - start).count()
;
}
```

```cpp
RankAVLTree<int> tr;
std::vector<int> vec(nitems);
std::iota(vec.begin(), vec.end(), 0);
std::random_shuffle(vec.begin(), vec.end());
auto time = duration_ms([&tr, &vec]() {
    std::for_each(vec.begin(), vec.end(),
        [&tr](int i) {tr.insert(i);}
    );
});
```

Как измерить время?

RankAVL vs `std::set`

$N = 10^5$

Контейнер	insert	distance(2, N-10)	erase
RankAVL			
std::set			
std::unordered_set			

$N = 10^6$

Контейнер	insert	distance(2, N-10)	erase
RankAVL			
std::set			
std::unordered_set			

- Процедура:
  1. Добавляем элементы от до последовательно (метод `insert`)
  2. Определяем разность рангов значений и
  3. Удаляем все элементы последовательно

RankAVL vs `std::set`

$N = 10^5$

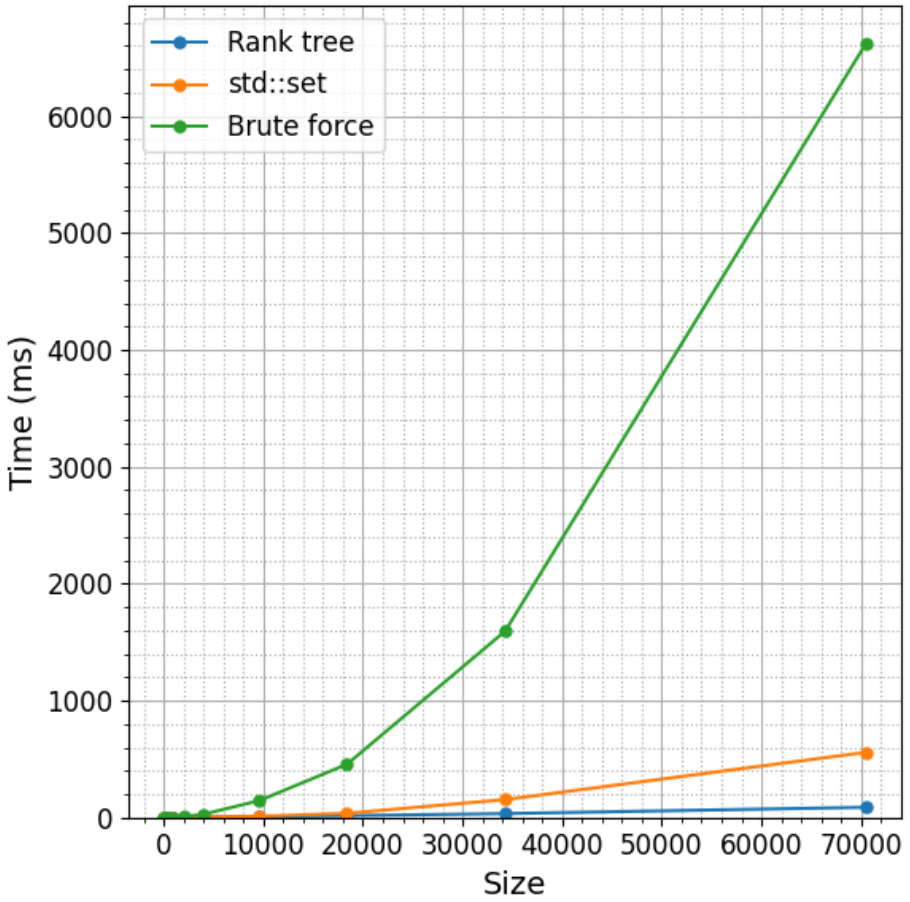
Контейнер	insert	distance(2, N-10)	erase
RankAVL			
std::set			
std::unordered_set			

$N = 10^6$

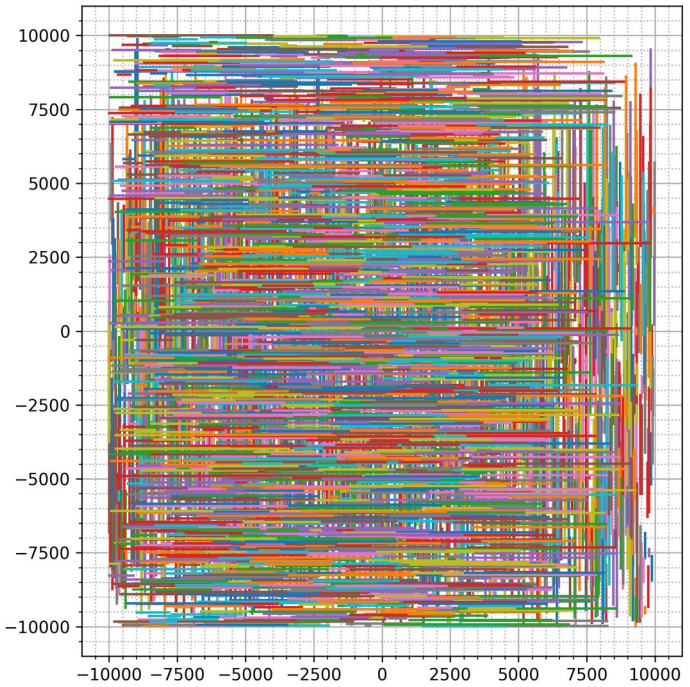
Контейнер	insert	distance(2, N-10)	erase
RankAVL			
std::set			
std::unordered_set			

- Процедура:
  1. Добавляем в случайном порядке элементы от до (метод `insert`)
  2. Определяем разность рангов значений и
  3. Удаляем все элементы в (новом) случайном порядке

Что там с пересечениями?



отрезков



Для отрезков:

- В раз быстрее, чем наивный алгоритм
- В раз быстрее, чем с `std::set`

Заключение

1. Программа *C++* работает с разными типами памяти:
   - Глобальная (статическая)
   - Стек
   - Куча
2. Идиома RAII позволяет программе безопасно работать с ресурсами. Для работы с динамической памятью используют `shared_ptr` и `unique_ptr`
3. *C++* поддерживает парадигму объектно-ориентированного программирования.
   1. Класс – пользовательский тип данных
   2. Одна из ключевых идей ООП – инкапсуляция
4. При реализации и оптимизации алгоритма выполняйте измерения, находите узкие места в программе

Обработка исключений

- В некоторых ситуациях программа не может корректно выполнить свою логику:
  - Деление на ноль
  - ip адрес передан в неверном формате
  - …
- Такие ситуации необходимо отслеживать и обрабатывать
- В *C++ обработка исключений* выполнятся в блоке `try-catch`
- Через `throw` можно передать объект любого типа
- В библиотеке `<stdexcept>` определены часто встречающиеся исключения

```
#include <iostream>
#include <stdexcept>
using namespace std;

int divide (int i, int j) {
    if (!
j) throw overflow_error("Division by zero");
    return i / j;
}
int main() {
    try {
        int k = divide(1, 0);
        cout << "k = " << k << endl;
    } catch (overflow_error& ex) {
        cout << ex.what() << endl;
    } catch (...) {
        cout << "Unknown exception" << endl;
    }
    return 0;
}
```

```
> Division by zero
```