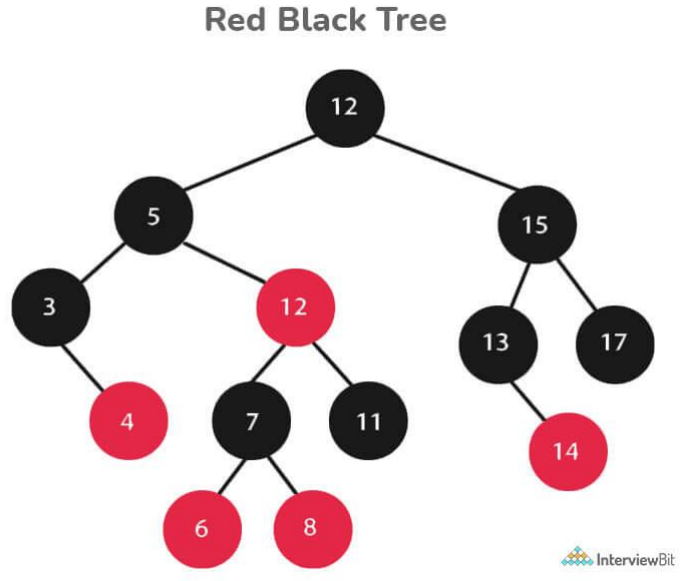
# BINARY TREE AUTOBALANCE

Выполнили: Сударкин Д.И, Борисов Н.О, Соболев Д.И

# BINARY TREE AUTOBALANCE

- **Сбалансированное двоичное дерево** (бинарное дерево с балансировкой по высоте) — это структура данных, в которой высота левого и правого поддеревьев любого узла отличается не более чем на 1.
- **Красно-чёрные деревья** (англ. red-black tree, RB tree). Используют более «мягкие» правила балансировки: сбалансированность достигается за счёт введения дополнительного атрибута узла — «цвета» (может принимать одно из двух значений — «чёрный» или «красный»).



# CMAKE

**CMake (Cross-Platform Make)** — система сборки, которая автоматизирует процесс сборки программного обеспечения из исходного кода.

**Она позволяет**:

- создавать проекты, которые могут компилироваться на различных платформах (Linux, Windows, macOS);
- генерировать файлы проектов для популярных IDE (Microsoft Visual Studio, Xcode, Qt и т. д.);
- управлять зависимостями проекта от сторонних библиотек.
- Важно: CMake не занимается непосредственно сборкой, а лишь генерирует файлы сборки из предварительно написанного файла сценария

# ITERATOR

**Итератор в C++** — это объект, который предоставляет последовательный доступ к элементам контейнера (например, векторов, списков, массивов) без раскрытия внутренней структуры данных. Итераторы похожи на указатели, но обладают дополнительными возможностями и ограничениями, заданными типом контейнера.

Контейнеры стандартной библиотеки C++ (STL) реализуют итераторы, обеспечивая разные типы доступа. Например:

- **vector** — динамический массив с произвольным доступом;
- **list** — двусвязный список, в котором элементы хранятся в неопределённом порядке, но имеют последовательную структуру;
- **deque** — двойная очередь, которая поддерживает доступ как с начала, так и с конца последовательности;
- **set/map** — ассоциативные контейнеры, которые предоставляют итераторы для обхода элементов в отсортированном порядке.

```cpp
#ifndef ITERATOR_HPP
#define ITERATOR_HPP

#include <iterator>
#include <stdexcept>

// Предварительное объявление Tree (Tree уже определен в tree.hpp)
template <typename T>
class Tree;

// Реализация итератора
template <typename T>
class TreeIterator {
public:
    using iterator_category = std::bidirectional_iterator_tag;
    using value_type = T;
    using difference_type = std::ptrdiff_t;
    using pointer = T*;
    using reference = T&;

private:
    using Node = typename Tree<T>::Node;
    Node* current;
    Node* nil;
    const Tree<T>* tree;

public:
    TreeIterator() : current(nullptr), nil(nullptr), tree(nullptr) {}
    TreeIterator(Node* node, Node* nilNode, const Tree<T>* t) : current(node), nil(nilNode), tree(t) {}
    TreeIterator(const TreeIterator& other) : current(other.current), nil(other.nil), tree(other.tree)
{}
```

```
    TreeIterator& operator=(const TreeIterator& other) {
        if (this != &other) {
            current = other.current;
            nil = other.nil;
            tree = other.tree;
        }
        return *this;
    }

    reference operator*() const {
        if (current == nil || current == nullptr) {
            throw std::runtime_error("Dereferencing end iterator");
        }
        return current->val;
    }

    pointer operator->() const {
        if (current == nil || current == nullptr) {
            throw std::runtime_error("Dereferencing end iterator");
        }
        return &(current->val);
    }
```

```
TreeIterator& operator++() {
    if (current == nil || current == nullptr) {
        return *this;
    }
    current = tree->successor(current);
    return *this;
}

TreeIterator operator++(int) {
    TreeIterator tmp = *this;
    ++(*this);
    return tmp;
}
```

```
    TreeIterator operator--(int) {
        TreeIterator tmp = *this;
        --(*this);
        return tmp;
    }

    bool operator==(const TreeIterator& other) const {
        return current == other.current;
    }

    bool operator!=(const TreeIterator& other) const {
        return !(*this == other);
    }

    Node* getNode() const { return current; }
};

#endif // ITERATOR_HPP
```

**Логика реализации красно-чёрных деревьев** основана на использовании двух цветов узлов: красного и чёрного. Эти цвета используются для поддержания баланса во время операций вставки и удаления.

**Некоторые свойства красно-чёрных деревьев**:

- Корневой узел всегда чёрного цвета.
- Две красные вершины не могут идти подряд ни на одном пути.
- Оба потомка каждого красного узла — чёрные.
- Для каждой вершины, в каждом исходящем из неё пути, одинаковое число чёрных вершин.

**Чтобы вставить узел**, нужно выполнить следующие шаги:

- Найти в дереве место, куда следует добавить новый узел.
- Новый узел всегда добавляется как лист, поэтому оба его потомка являются пустыми узлами и предполагаются чёрными.
- После вставки узел окрашивается в красный цвет.
- Проверяется, не нарушается ли свойства дерева.
- Если необходимо, узел перекрашивается и производится поворот, чтобы сбалансировать дерево.

```
определения Tree
#include "../iterator/iterator.hpp"

// Реализация методов Tree

template <typename T>
Tree<T>::Tree() : treeSize(0) {
    nil = new Node(T{});
    nil->color = Node::BLACK;
    nil->left = nil;
    nil->right = nil;
    nil->parent = nil;
    root = nil;
}

template <typename T>
Tree<T>::Tree(const Tree& other) : treeSize(0)
{
    nil = new Node(T{});
    nil->color = Node::BLACK;
    nil->left = nil;
    nil->right = nil;
    nil->parent = nil;
    root = nil;

    if (other.root != other.nil) {
        root = copyRecursive(other.root, nil);
        treeSize = other.treeSize;
    }
}

template <typename T>
Tree<T>::Tree(Tree&& other) noexcept
    : root(other.root), nil(other.nil),
treeSize(other.treeSize) {
    other.root = nullptr;
    other.nil = nullptr;
    other.treeSize = 0;
}

template <typename T>
Tree<T>::~Tree() {
    clear();
    if (nil) {
```

```
private:
    // Узел дерева
    struct Node {
        T val;
        Node* left;
        Node* right;
        Node* parent;
        enum Color { RED, BLACK } color;

        Node(const T& k) : val(k),
left(nullptr), right(nullptr), parent(nullptr),
color(RED) {}
    };

    Node* root;
    Node* nil;  // Sentinel node
    size_type treeSize;

    // Вращения
    void rotateLeft(Node* x);
    void rotateRight(Node* x);

    // Балансировка
    void fixInsert(Node* z);
    void fixDelete(Node* x);

    // Вспомогательные методы
    Node* search(Node* node, const T& value)
const;
    Node* minimum(Node* node) const;
    Node* maximum(Node* node) const;
    void transplant(Node* u, Node* v);
    Node* successor(Node* node) const;
    Node* predecessor(Node* node) const;

    // Рекурсивные методы
    void clearRecursive(Node* node);
    Node* copyRecursive(Node* node, Node*
parent);

    // Дружественный класс для итератора
    friend class TreeIterator<T>;
};
```

```
template <typename T>
class TreeIterator;

template <typename T>
class Tree {
public:
    // Типы
    using iterator = TreeIterator<T>;
    using value_type = T;
    using size_type = size_t;

    Tree();
    Tree(const Tree& other);
    Tree(Tree&& other) noexcept;
    ~Tree();

    Tree& operator=(const Tree& other);
    Tree& operator=(Tree&& other) noexcept;

    std::pair<iterator, bool> insert(const T&
value);
    size_type erase(const T& value);
    iterator find(const T& value);
    iterator begin();
    iterator end();
    iterator begin() const;
    iterator end() const;
    iterator cbegin() const;
    iterator cend() const;

    size_type size() const;
    bool empty() const;
    void clear();
```

```
    if (x->parent == nil) {
        root = y;
    } else if (x == x->parent->right) {
        x->parent->right = y;
    } else {
        x->parent->left = y;
    }

    y->right = x;
    x->parent = y;
}

template <typename T>
void Tree<T>::fixInsert(Node* z) {
    while (z->parent->color == Node::RED) {
        if (z->parent == z->parent->parent-
>left) {
            Node* y = z->parent->parent->right;
            if (y->color == Node::RED) {
                z->parent->color = Node::BLACK;
                y->color = Node::BLACK;
                z->parent->parent->color =
Node::RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    rotateLeft(z);
                }
                z->parent->color = Node::BLACK;
                z->parent->parent->color =
Node::RED;
                rotateRight(z->parent->parent);
            }
        } else {
            Node* y = z->parent->parent->left;
            if (y->color == Node::RED) {
                z->parent->color = Node::BLACK;
                y->color = Node::BLACK;
                z->parent->parent->color =
Node::RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
```

```
typename Tree<T>::Node*
Tree<T>::copyRecursive(Node* node, Node*
parent) {
    if (node == nullptr || node->left ==
nullptr) {
        return nil;
    }

    Node* newNode = new Node(node->val);
    newNode->color = node->color;
    newNode->parent = parent;
    newNode->left = copyRecursive(node->left,
newNode);
    newNode->right = copyRecursive(node->right,
newNode);

    if (newNode->left == nullptr) newNode->left
= nil;
    if (newNode->right == nullptr) newNode-
>right = nil;

    return newNode;
}

template <typename T>
void Tree<T>::rotateLeft(Node* x) {
    Node* y = x->right;
    x->right = y->left;

    if (y->left != nil) {
        y->left->parent = x;
    }

    y->parent = x->parent;

    if (x->parent == nil) {
        root = y;
    } else if (x == x->parent->left) {
        x->parent->left = y;
    } else {
        x->parent->right = y;
    }

    y->left = x;
```

```
template <typename T>
Tree<T>& Tree<T>::operator=(const Tree& other)
{
    if (this != &other) {
        clear();
        if (other.root != other.nil) {
            root = copyRecursive(other.root,
nil);
            treeSize = other.treeSize;
        } else {
            root = nil;
            treeSize = 0;
        }
    }
    return *this;
}

template <typename T>
Tree<T>& Tree<T>::operator=(Tree&& other)
noexcept {
    if (this != &other) {
        clear();
        if (nil) {
            delete nil;
        }
        root = other.root;
        nil = other.nil;
        treeSize = other.treeSize;
        other.root = nullptr;
        other.nil = nullptr;
        other.treeSize = 0;
    }
    return *this;
}
```

```
template <typename T>
void Tree<T>::fixDelete(Node* x) {
    while (x != root && x->color ==
Node::BLACK) {
        if (x == x->parent->left) {
            Node* w = x->parent->right;
            if (w->color == Node::RED) {
                w->color = Node::BLACK;
                x->parent->color = Node::RED;
                rotateLeft(x->parent);
                w = x->parent->right;
            }
            if (w->left->color == Node::BLACK
&& w->right->color == Node::BLACK) {
                w->color = Node::RED;
                x = x->parent;
            } else {
                if (w->right->color ==
Node::BLACK) {
                    w->left->color =
Node::BLACK;
                    w->color = Node::RED;
                    rotateRight(w);
                    w = x->parent->right;
                }
                w->color = x->parent->color;
                x->parent->color = Node::BLACK;
                w->right->color = Node::BLACK;
                rotateLeft(x->parent);
                x = root;
            }
        } else {
            Node* w = x->parent->left;
            if (w->color == Node::RED) {
                w->color = Node::BLACK;
                x->parent->color = Node::RED;
                rotateRight(x->parent);
                w = x->parent->left;
            }
            if (w->right->color == Node::BLACK
&& w->left->color == Node::BLACK) {
                w->color = Node::RED;
                x = x->parent;
            } else {
                if (w->left->color ==
Node::BLACK) {
```

```
template <typename T>
void Tree<T>::transplant(Node* u, Node* v) {
    if (u->parent == nil) {
        root = v;
    } else if (u == u->parent->left) {
        u->parent->left = v;
    } else {
        u->parent->right = v;
    }
    v->parent = u->parent;
}

template <typename T>
typename Tree<T>::Node* Tree<T>::minimum(Node*
node) const {
    while (node->left != nil) {
        node = node->left;
    }
    return node;
}

template <typename T>
typename Tree<T>::Node* Tree<T>::maximum(Node*
node) const {
    while (node->right != nil) {
        node = node->right;
    }
    return node;
}

template <typename T>
typename Tree<T>::Node*
Tree<T>::successor(Node* node) const {
    if (node->right != nil) {
        return minimum(node->right);
    }

    Node* y = node->parent;
    while (y != nil && node == y->right) {
        node = y;
        y = y->parent;
    }
    return y;
}
```

```
template <typename T>
std::pair<typename Tree<T>::iterator, bool>
Tree<T>::insert(const T& value) {
    Node* y = nil;
    Node* x = root;

    while (x != nil) {
        y = x;
        if (value < x->val) {
            x = x->left;
        } else if (x->val < value) {
            x = x->right;
        } else {
            // Элемент уже существует
            return std::make_pair(iterator(x,
nil, this), false);
        }
    }

    Node* z = new Node(value);
    z->parent = y;
    z->left = nil;
    z->right = nil;
    z->color = Node::RED;

    if (y == nil) {
        root = z;
    } else if (z->val < y->val) {
        y->left = z;
    } else {
        y->right = z;
    }

    treeSize++;
    fixInsert(z);

    return std::make_pair(iterator(z, nil,
this), true);
}
```

```
        return end();
    }
    return iterator(minimum(root), nil, this);
}

template <typename T>
typename Tree<T>::iterator Tree<T>::end() const
{
    return iterator(nil, nil, this);
}

template <typename T>
typename Tree<T>::iterator Tree<T>::cbegin()
const {
    return begin();
}

template <typename T>
typename Tree<T>::iterator Tree<T>::cend()
const {
    return end();
}

template <typename T>
typename Tree<T>::size_type Tree<T>::size()
const {
    return treeSize;
}

template <typename T>
bool Tree<T>::empty() const {
    return treeSize == 0;
}

template <typename T>
void Tree<T>::clearRecursive(Node* node) {
    if (node != nil && node != nullptr) {
        clearRecursive(node->left);
        clearRecursive(node->right);
        delete node;
    }
}

template <typename T>
void Tree<T>::clear() {
```

```
template <typename T>
typename Tree<T>::Node* Tree<T>::search(Node*
node, const T& value) const {
    while (node != nil && node->val != value) {
        if (value < node->val) {
            node = node->left;
        } else {
            node = node->right;
        }
    }
    return node;
}

template <typename T>
typename Tree<T>::iterator Tree<T>::find(const
T& value) {
    Node* node = search(root, value);
    if (node == nil) {
        return end();
    }
    return iterator(node, nil, this);
}

template <typename T>
typename Tree<T>::iterator Tree<T>::begin() {
    if (root == nil) {
        return end();
    }
    return iterator(minimum(root), nil, this);
}

template <typename T>
typename Tree<T>::iterator Tree<T>::end() {
    return iterator(nil, nil, this);
}
```

```
template <typename T>
typename Tree<T>::size_type
Tree<T>::erase(const T& value) {
    Node* z = search(root, value);
    if (z == nil) {
        return 0;
    }

    Node* y = z;
    Node* x;
    typename Node::Color yOriginalColor = y-
>color;

    if (z->left == nil) {
        x = z->right;
        transplant(z, z->right);
    } else if (z->right == nil) {
        x = z->left;
        transplant(z, z->left);
    } else {
        y = minimum(z->right);
        yOriginalColor = y->color;
        x = y->right;
        if (y->parent == z) {
            x->parent = y;
        } else {
            transplant(y, y->right);
            y->right = z->right;
            y->right->parent = y;
        }
        transplant(z, y);
        y->left = z->left;
        y->left->parent = y;
        y->color = z->color;
    }

    delete z;
    treeSize--;

    if (yOriginalColor == Node::BLACK) {
        fixDelete(x);
    }

    return 1;
}
```

# CONSTRUCTOR

Далее будет представлен конструктор, по которому мы будем считывать данные для нашего дерева из файла и преобразовывать его в нужный нам вид для работы с ним.

```hpp
#ifndef CONSTRUCTOR_UTILS_HPP
#define CONSTRUCTOR_UTILS_HPP

#include "../tree/tree.hpp"
#include <string>
#include <string_view>
#include <fstream>
#include <sstream>
#include <map>
#include <set>
#include <functional>
#include <vector>
#include <iostream>
#include <stdexcept>

template <typename T>
class ConstructorsUtil {
public:
    // Конструктор из файла со списком ребер
    // Формат: value left right (каждая строка)
    // Пример:
    // 5 2 7
    // 2 1 3
    // 7 6 8
    static Tree<T>
fromEdgeList(std::string_view filename);

private:
    struct EdgeData {
        T value;
        T left;
        T right;
    };

    static std::vector<EdgeData>
parseEdgeList(std::string_view filename);
};
```

```cpp
template <typename T>
std::vector<typename ConstructorsUtil<T>::EdgeData>
ConstructorsUtil<T>::parseEdgeList(std::string_view
filename) {
    std::vector<EdgeData> edges;
    std::string filenameStr(filename);
    std::ifstream file(filenameStr);

    if (!file.is_open()) {
        throw std::runtime_error("Cannot open file:
" + filenameStr);
    }

    std::string line;
    while (std::getline(file, line)) {
        if (line.empty() || line[0] == '#') {
            continue; // Пропускаем пустые строки и
комментарии
        }

        std::istringstream iss(line);
        EdgeData edge;

        if (iss >> edge.value >> edge.left >>
edge.right) {
            edges.push_back(edge);
        } else {
            std::cerr << "Warning: Invalid line
format: " << line << std::endl;
        }
    }

    file.close();
    return edges;
}
```


```cpp
    if (edges.empty()) {
        return tree;
    }

    // Создаем карту связей: значение -> (левое,
правое)
    std::map<T, std::pair<T, T>> connections;
    for (const auto& edge : edges) {
        connections[edge.value] = {edge.left,
edge.right};
    }

    // Находим корень (узел, который не является
потомком)
    std::set<T> children;
    for (const auto& edge : edges) {
        if (edge.left != T{})
children.insert(edge.left);
        if (edge.right != T{})
children.insert(edge.right);
    }

    T rootValue = T{};
    for (const auto& edge : edges) {
        if (children.find(edge.value) ==
children.end()) {
            rootValue = edge.value;
            break;
        }
    }

    if (rootValue == T{} && !edges.empty()) {
        rootValue = edges[0].value;
    }

    // Рекурсивно строим дерево
    std::function<void(T)> buildTree = [&](T val)
{
        if (val == T{}) return;

        tree.insert(val);

        auto it = connections.find(val);
        if (it != connections.end()) {
            buildTree(it->second.first);
            buildTree(it->second.second);
        }
    };
```

# MAIN

**empty()** — пусто ли дерево.

**size()** — количество элементов.

**insert()** — вставка.

**find()** — поиск элемента.

**erase()** — удаление:

- существующего элемента,
- Несуществующего элемента.

```cpp
void testBasicOperations() {
    std::cout << "=== Basic Operations Test ===" << std::endl;

    Tree<int> tree;

    // Тест empty и size
    std::cout << "Empty tree: empty=" << tree.empty() << ", size=" << tree.size() << std::endl;

    // Тест insert
    auto result1 = tree.insert(5);
    std::cout << "Insert 5: success=" << result1.second << std::endl;

    auto result2 = tree.insert(3);
    std::cout << "Insert 3: success=" << result2.second << std::endl;

    auto result3 = tree.insert(7);
    std::cout << "Insert 7: success=" << result3.second << std::endl;

    auto result4 = tree.insert(5); // Дубликат
    std::cout << "Insert 5 (duplicate): success=" << result4.second << std::endl;

    std::cout << "Size after inserts: " << tree.size() << std::endl;
```

```cpp
<< std::endl;
    }

    auto it2 = tree.find(10);
    if (it2 == tree.end()) {
        std::cout << "Element 10 not found (as expected)" << std::endl;
    }

    // Тест erase
    size_t erased = tree.erase(3);
    std::cout << "Erased elements: " << erased << ", new size: " << tree.size() << std::endl;

    erased = tree.erase(10); // Несуществующий
    std::cout << "Attempt to erase 10: erased=" << erased << std::endl;

    std::cout << std::endl;
}

void testIterator() {
    std::cout << "=== Iterator Test ===" << std::endl;

    Tree<int> tree;
    std::vector<int> values = {5, 3, 7, 2, 4, 6, 8};

    for (int val : values) {
        tree.insert(val);
    }

    std::cout << "Tree traversal (in-order): ";
    for (auto it = tree.begin(); it != tree.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    std::cout << "Reverse traversal: ";
    auto it = tree.end();
    --it; // Переходим к последнему элементу
    while (it != tree.begin()) {
        std::cout << *it << " ";
        --it;
```

```cpp
    Tree<int> tree;
    std::vector<int> values = {5, 3, 7, 2, 4, 6, 8, 1, 9};

    for (int val : values) {
        tree.insert(val);
    }

    // std::find
    auto it = std::find(tree.begin(), tree.end(), 4);
    if (it != tree.end()) {
        std::cout << "std::find found: " << *it << std::endl;
    }

    // std::count
    size_t count = std::count(tree.begin(), tree.end(), 5);
    std::cout << "std::count for 5: " << count << std::endl;

    // std::for_each
    std::cout << "std::for_each (doubling values): ";
    std::for_each(tree.begin(), tree.end(), [](int& val) {
        std::cout << val * 2 << " ";
    });
    std::cout << std::endl;

    // std::lower_bound (требует random_access, но попробуем)
    // Для bidirectional итератора используем ручной поиск
    int target = 5;
    auto lower = std::lower_bound(tree.begin(), tree.end(), target);
    if (lower != tree.end()) {
        std::cout << "std::lower_bound for " << target << ": " << *lower << std::endl;
    }
```

```cpp
void testConstructors() {
    std::cout << "=== Constructors Test ===" <<
std::endl;

    // Конструктор по умолчанию
    Tree<int> tree1;
    tree1.insert(1);
    tree1.insert(2);
    std::cout << "Default constructor: size="
<< tree1.size() << std::endl;

    // Копирующий конструктор
    Tree<int> tree2(tree1);
    std::cout << "Copy constructor: size=" <<
tree2.size() << std::endl;

    // Перемещающий конструктор
    Tree<int> tree3(std::move(tree1));
    std::cout << "Move constructor: size=" <<
tree3.size() << ", original tree: size=" <<
tree1.size() << std::endl;

    // Оператор присваивания
    Tree<int> tree4;
    tree4 = tree2;
    std::cout << "Copy assignment operator:
size=" << tree4.size() << std::endl;

    Tree<int> tree5;
    tree5 = std::move(tree2);
    std::cout << "Move assignment operator:
size=" << tree5.size() << std::endl;

    std::cout << std::endl;
}
```


```cpp
void testFromFile() {
    std::cout << "=== File Constructor Test ===" <<
std::endl;

    try {
        // Читаем дерево из файла tree_example.txt
        Tree<int> tree =
ConstructorsUtil<int>::fromEdgeList("tree_example.txt");
        std::cout << "Tree from file: size=" <<
tree.size() << std::endl;
        std::cout << "Elements: ";
        for (auto it = tree.begin(); it != tree.end();
++it) {
            std::cout << *it << " ";
        }
        std::cout << std::endl;
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << std::endl;

    }

    std::cout << std::endl;
}

int main() {
    try {
        testBasicOperations();
        testIterator();
        testSTLAlgorithms();
        testConstructors();
        testFromFile();

        std::cout << "All tests completed!" <<
std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;

        return 1;
    }

    return 0;
}
```

***СПАСИБО ЗА ВНИМАНИЕ!***