Санкт-Петербургский политехнический университет Петра Великого  
Институт машиностроения, материалов и транспорта   
Высшая школа автоматизации и робототехники

**КУРСОВАЯ РАБОТА**

По дисциплине «Объектно-ориентированное программирование»

**Red-Black Tree**

(семестр VI)

Студент группы Д.В. Брюханов   
3331506/20401   
 подпись, дата инициалы и фамилия

Оценка выполненной студентом работы:

Преподаватель, М.С. Ананьевский  
доцент, к.т.н.   
 подпись, дата инициалы и фамилия

Санкт-Петербург

2025

[Введение 3](#_Toc198607507)

[Актуальность 3](#_Toc198607508)

[Основная часть 4](#_Toc198607509)

[Правила красно-чёрного дерева: 4](#_Toc198607510)

[Балансировка красно-чёрного дерева 4](#_Toc198607511)

[Вставка узла в красно-чёрное дерево 4](#_Toc198607512)

[Удаление узлов из красно-чёрного дерева 9](#_Toc198607513)

[Тесты 17](#_Toc198607514)

[Заключение 18](#_Toc198607515)

[Список литературы 19](#_Toc198607516)

[Приложение 1 20](#_Toc198607517)

[Приложение 2 33](#_Toc198607518)

# Введение

Красно-чёрные деревья (Red-Black Trees, RBT) — это один из наиболее важных типов самобалансирующихся двоичных деревьев поиска (Binary Search Tree), обеспечивающих эффективное выполнение операций вставки, удаления и поиска за логарифмическое время. Они широко применяются в различных областях компьютерных наук, включая реализацию ассоциативных массивов, планировщиков задач и файловых систем.

Основная цель данной курсовой работы — изучить структуру красно-чёрных деревьев, их свойства и алгоритмы балансировки.

# Актуальность

В условиях постоянно растущих объёмов данных и требований к скорости обработки информации эффективные структуры данных становятся критически важными. Красно-чёрные деревья обеспечивают гарантированную сложность **O(log n)** для основных операций, что делает их предпочтительным выбором во многих приложениях, таких как:

* Реализация **std::map** и **std::set** в C++
* Базы данных и индексация
* Алгоритмы геометрического поиска
* Планирование процессов в операционных системах

Изучение RBT позволяет глубже понять принципы балансировки деревьев и их применение в реальных задачах, что делает данную тему актуальной для современных IT-специалистов.

# Основная часть

**Бинарное дерево поиска** – Red-Black Tree сохраняет свойства Binary Search Tree (левое поддерево содержит меньшие ключи, правое — большие).

## Правила красно-чёрного дерева:

1) Цвет узла **чёрный** или **красный**

2) Корень всегда **чёрный**

3) Листья всегда **чёрные** и null

4) Каждый **красный** узел должен иметь 2 **чёрных** узла, **чёрный** может иметь **чёрных** сыновей.

5) Пути от узла к его листьям должны содержать одинаковое количество **чёрных** узлов (**чёрную высоту**)

## Балансировка красно-чёрного дерева

**Балансировка дерева** – операции, благодаря которым дерево гарантированно сохраняет высот O(log n)

### Вставка узла в красно-чёрное дерево

Новый элемент, вставляемый в красно-чёрное дерево по умолчанию красный, так как его вставка в среднем ломает меньшее количество правил, чем при вставке чёрного, которая гарантировано меняет чёрную высоту.

5 возможных случаев при вставке нового элемента:

1. Новый корень
2. Красный дядя, дед не корень
3. Красный дядя, дед корень
4. Дядя чёрный, отец и дед зигзагом
5. Дядя чёрный, отец и дед на одной линии

Изображение выглядит как снимок экрана, диаграмма, текст, круг

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 1 – Все возможные нарушения правил, при вставке красного узла

В случае 1 перекрашиваем корень на чёрный.

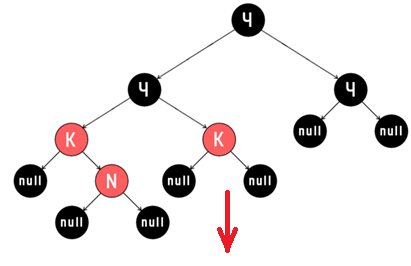
Изображение выглядит как круг, белый, часы, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.Изображение выглядит как круг, логотип, дизайн, иллюстрация

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2 – Способ балансировки при случае 1

В случае 2 красим перекрашиваем дядю, отца и деда.



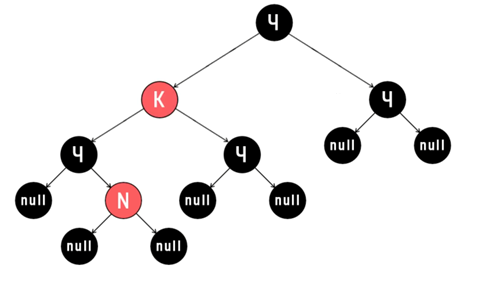


Рисунок 3 - Способ балансировки при случае 2

Изображение выглядит как круг, снимок экрана

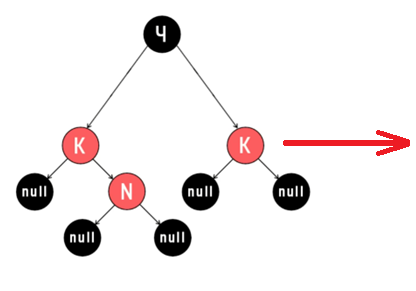
Контент, сгенерированный ИИ, может содержать ошибки.В случае 3 перекрашиваем отца и дядю, а дедушку не трогаем.

Рисунок 4 - Способ балансировки при случае 3

Изображение выглядит как круг, снимок экрана, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.В случае 4 делаем левый поворот относительно отца и переходим к 5 случаю.

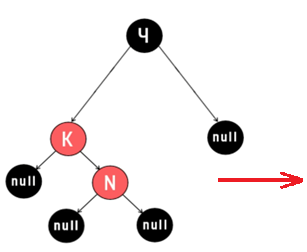


Рисунок 5 - Способ балансировки при случае 4

В случае 5 перекрашиваем отца и деда и делаем правый поворот относительно деда.

Изображение выглядит как круг, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.Изображение выглядит как круг

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как снимок экрана, круг

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 6 - Способ балансировки при случае 5

### Удаление узлов из красно-чёрного дерева

Удаление происходит если у узла 1 ребёнок или нет детей

Изображение выглядит как диаграмма, круг, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 7 – Случае, когда мы фактически удаляем узел

Если узел имеет двух детей, то мы заменяем его максимальным узлом по левой ветке, или минимальным по правой ветке. И только после этого удаляем узел с минимальным или максимальным числом.

Изображение выглядит как круг, диаграмма, снимок экрана, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как круг, диаграмма, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 8 – Способ “удаления” узла с двумя детьми

Если физически удаляемый узел красный, то у него ноль детей, поэтому мы можем его просто удалить.

Изображение выглядит как круг, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как круг, снимок экрана, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 9 – Способ удаления красного узла

Если мы удаляем чёрный узел, то чёрная высота стороны уменьшится на 1. При удалении чёрного узла с красным ребёнком

Изображение выглядит как снимок экрана, круг, диаграмма, линия

Контент, сгенерированный ИИ, может содержать ошибки.Изображение выглядит как круг, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 10 – Удаление чёрного узла с красным ребёнком

Чтобы сбалансировать получившиеся дерево, мы перекрасим красный пришедший узел в чёрный цвет.

Изображение выглядит как круг, снимок экрана, линия, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 11 – Перекрашиваем красный узел в чёрный, ради баланса

Если ребёнок у удаляемого узла чёрный, тогда мы смотрим на брата этого узла. Есть два варианта:  
1) Он чёрный и дети неизвестного цвета

2) Он красный и дети чёрного цвета, а родитель тогда чёрный

Изображение выглядит как снимок экрана, круг, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 12 – Случаи, когда удаляемый чёрный узел имеет чёрного ребёнка

Если брат чёрный, тогда есть 4 варианта:

* 1. 2 ребёнка красные
  2. Левый ребёнок чёрный, а правый красный
  3. Нет детей
  4. Левый ребёнок красный, а правый чёрный

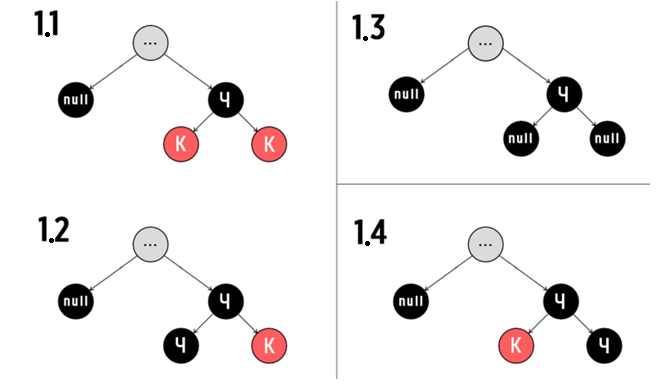
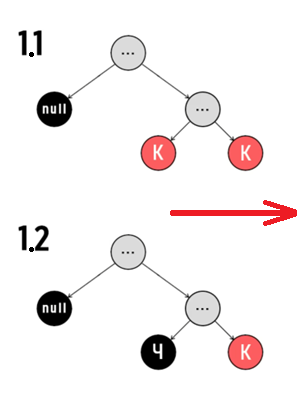
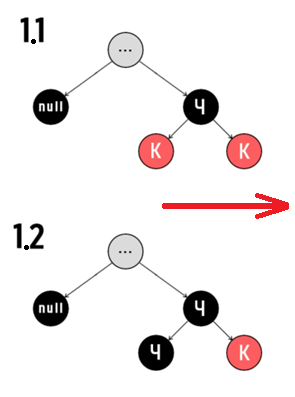


Рисунок 13 – Все возможные случае, когда брат чёрный

В случае 1.1 и 1.2 балансировка выполняется одинаково. Перекрашиваем брата в цвет родителя. Перекрашиваем родителя и красного ребёнка в чёрный и делаем левый поворот относительно родительского узла.



Изображение выглядит как круг, снимок экрана, часы, иллюстрация

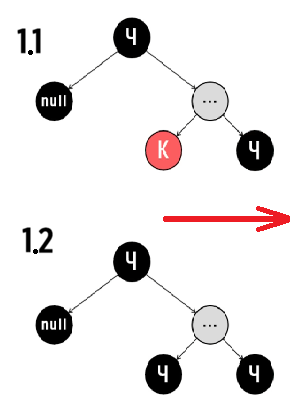
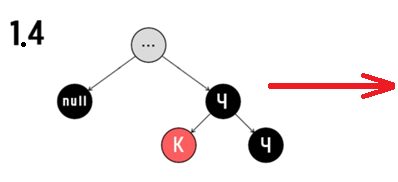
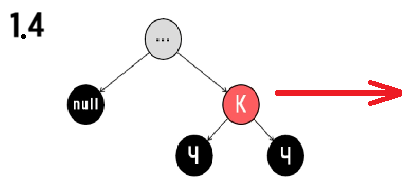
Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 14 – Балансировка дерева в случаях 1.1 и 1.2

В случае 1.4, перекрашиваем брата в красный цвет и левого ребёнка в чёрный. Далее совершаем правый поворот относительно брата и переходим к последней части случая 1.2.



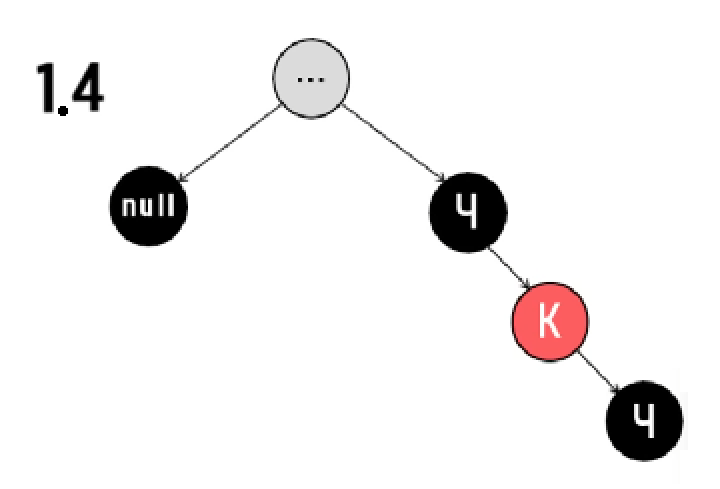


Рисунок 15 - Балансировка дерева в случае 1.4

В случае 1.3 есть 3 возможных случая

* + 1. Отец чёрный корень
    2. Отец красный не корень
    3. Отец чёрный не корень

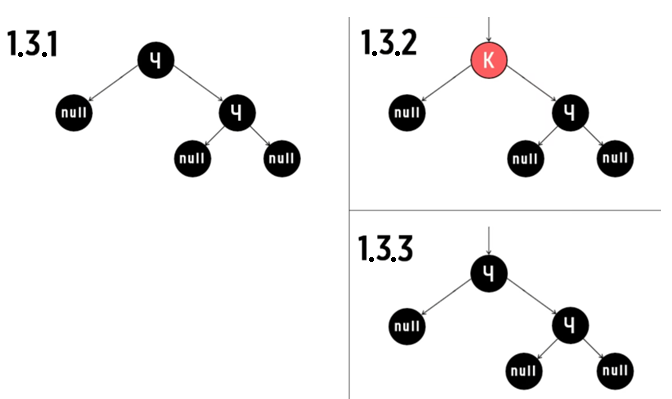


Рисунок 16 – Все возможные случае, когда у чёрного брата нет детей

В случае 1.3.1 красим брата в красный

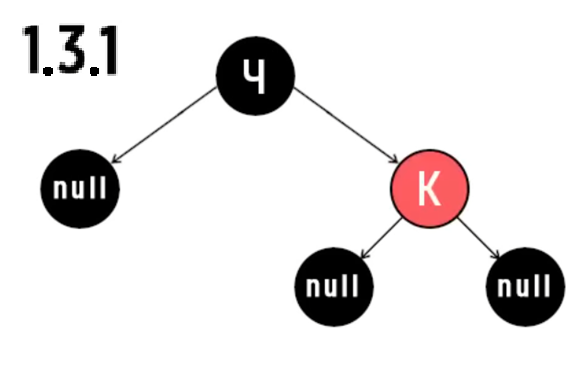


Рисунок 17 - Балансировка дерева в случае 1.3.1

В случае 1.3.2 красим брата в красный, а родителя в чёрный

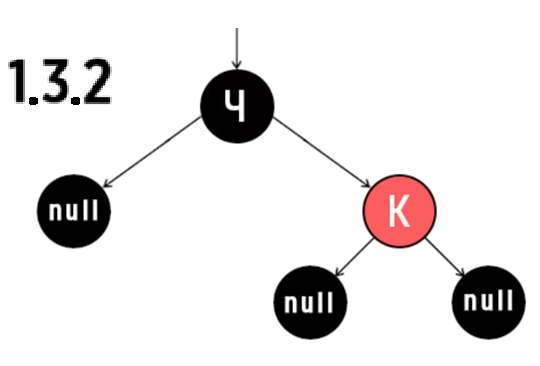


Рисунок 18 - Балансировка дерева в случае 1.3.2

В случае 1.3.3 родителя перекрашиваем в красный и совершаем левый поворот относительного него. Если возникает другое нарушение правила, то балансировка происходит по одному из случаев описанному выше.

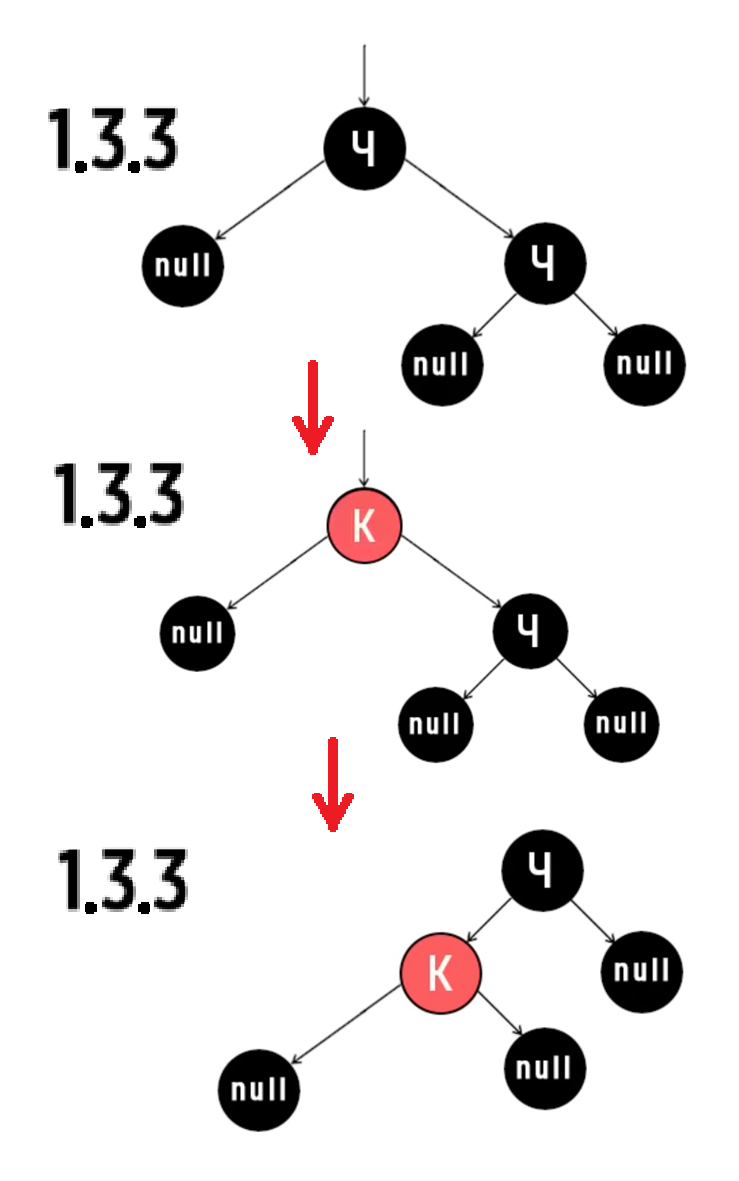


Рисунок 19 - Балансировка дерева в случае 1.3.3

В случае 2 красим родителя в красный, а брата в чёрный и делаем левый поворот относительно родителя

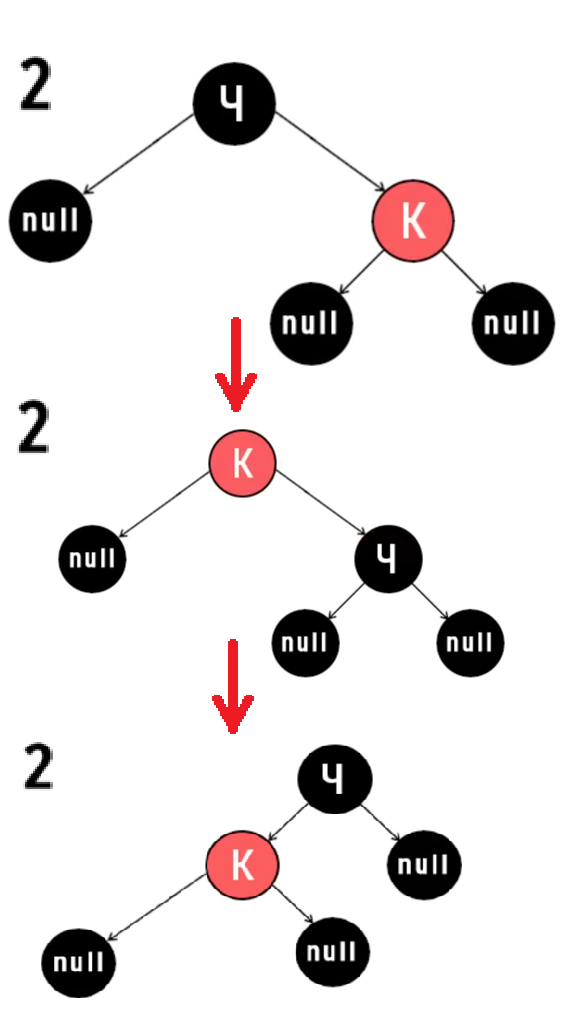


Рисунок 20 – Балансировка дерева, в случае если брат красный

## Тесты

Реализуем на C++ данную структуру и произведём тесты скорости работы данной структуры.

Результаты тестов приведены в таблице

|  |  |  |  |
| --- | --- | --- | --- |
| Размер n (узлов) | Вставка n узлов в дерево (мкс) | Поиск узла (мкс) | Удаление узла (мкс) |
| 100 | 24 | 0 | 1 |
| 1000 | 199 | 0 | 2 |
| 10000 | 1367 | 1 | 2 |
| 100000 | 16869 | 2 | 4 |
| 1000000 | 162227 | 3 | 4 |

Рисунок 21 – Графики зависимости скорости работы структуры от количества узлов

Как можно заметить скорость поиска и удаления имеет логарифмический характер.  
 С другой стороны вставка имеет линейный характер, хотя теоретически имеет сложность O(log n). Скорее всего это связано с тем, что запись производилась с помощью списка, и поэтому скорость вставки зависит от скорости работы списка, а не структуры.

# Заключение

В ходе работы были рассмотрены ключевые аспекты красно-чёрных деревьев: их свойства, алгоритмы вставки и удаления. Анализ показал, что Red-Black Tree являются мощным инструментом для эффективного хранения и обработки данных, что подтверждается их широким использованием в современных вычислительных системах.

# Список литературы

1. **Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К.** Алгоритмы: построение и анализ. — 3-е изд. — М.: Вильямс, 2022. — 1328 с.
2. **Седжвик, Р.** Алгоритмы на C++. — М.: Диалектика, 2020. — 1056 с.
3. **Wirth, N.** Algorithms and Data Structures. — Prentice Hall, 1985. — 288 p.
4. **Okasaki, C.** Purely Functional Data Structures. — Cambridge University Press, 1999. — 232 p.
5. **Weiss, M. A.** Data Structures and Algorithm Analysis in C++. — 4th ed. — Pearson, 2013. — 624 p.
6. **Knuth, D. E.** The Art of Computer Programming, Volume 3: Sorting and Searching. — 2nd ed. — Addison-Wesley, 1998. — 800 p.

# Приложение 1

#include <cstdlib>

#include <cstring>

#include <cassert>

#include <queue>

#include <vector>

#include <iostream>

enum Color {

    BLACK,

    RED

};

struct Node {

    int    key;

    int    val;

    Color   color;

    Node    \*parent;

    Node    \*left;

    Node    \*right;

    Node():parent(nullptr),left(nullptr),right(nullptr),color(RED){}

    ~Node()

    {

            if (left != nullptr)

                delete left;

            if (right != nullptr)

                delete right;

        }

};

class Tree {

    public:

        Tree():root(nullptr),size(0){};

        ~Tree();

        void    insert(int &key, int &val);

        bool    remove(const int &key);

        bool    search(const int &key, int &val) const;

        void    clear();

        void    printTree() const;

        int     getSize() const;

    private:

        int   size;

        Node \*root;

        int     cmp(const int &a, const int &b) const;

        void    leftRotate(Node \*node);

        void    rightRotate(Node \*node);

        void    removeNode(Node \*node);

};

Tree::~Tree()

{

    if (root != nullptr)

        delete root;

}

void Tree::insert(int &key, int &val)

{

    Node \*node = new Node; // Заменил auto

    node->key = key;

    node->val = val;

    if (root == nullptr) {

        root = node;

        node->color = BLACK;

        this->size++;

        return;

    }

    Node \*curr = root;

    while (curr->left != nullptr | curr->right != nullptr)

    {

        if (cmp(key, curr->key) == -1)

            curr = curr->left;

        else

            curr = curr->right;

    }

    node->parent = curr;

    if (cmp(key, curr->key) == -1)

        curr->left = node;

    else

        curr->right = node;

    while (curr->color == RED && curr->parent != nullptr)

    {

        bool isRight = (curr == curr->parent->right);

            Node \*uncle;

            if (isRight)

                    uncle = curr->parent->left;

            else

                    uncle = curr->parent->right;

            if (uncle == nullptr) {

                    curr->color = BLACK;

                    curr->parent->color = RED;

                    if (uncle == curr->parent->right) {

                        rightRotate(curr->parent);

                    }else {

                        leftRotate(curr->parent);

                    }

                    break;

            }else if (uncle->color == RED) {

            curr->color = BLACK;

            uncle->color = BLACK;

                curr->parent->color = RED;

            curr = curr->parent;

        }else {

            curr->color = BLACK;

            curr->parent->color = RED;

            if (isRight) {

                if (node == curr->left) {

                    rightRotate(curr);

                    curr = node;

                }

                leftRotate(curr->parent);

            }else {

                if (node == curr->right) {

                    leftRotate(curr);

                    curr = node;

                }

                rightRotate(curr->parent);

            }

        }

            root->color = BLACK;

    }

    this->size++;

}

bool Tree::remove(const int &key)

{

    Node \*curr = root; // Заменил auto

    while (curr->left != nullptr | curr->right != nullptr)

    {

        if (curr->key == key)

            break;

        if (cmp(key, curr->key) >= 0)

            curr = curr->right;

        else

            curr = curr->left;

    }

    if (curr->key != key)

        return 0;

    this->removeNode(curr);

    (this->size)--;

        return 1;

}

void Tree::removeNode(Node \*node)

{

    if (node->color == RED) {

        if (node->left != nullptr && node->right != nullptr) {

                    Node \*successor = node->right; // Заменил auto

                    while (successor->left != nullptr)

                        successor = successor->left;

                    node->key = successor->key;

                    node->val = successor->val;

                    this->removeNode(successor);

            }else if (node->left != nullptr) {

                    node->key = node->left->key;

                    node->val = node->left->val;

                    node->color = node->left->color;

                this->removeNode(node->left);

            }else if (node->right != nullptr) {

                    node->key = node->right->key;

                    node->val = node->right->val;

                    node->color = node->right->color;

                    this->removeNode(node->right);

            }else {

                    if (node->parent == nullptr) {

                        free(node);

                        root = nullptr;

                return;

                    }

                    if (node->parent->left == node)

                        node->parent->left = nullptr;

                    else

                            node->parent->right = nullptr;

            free(node);

        }

    }else {

            if (node->left != nullptr && node->right != nullptr) {

                Node \*successor = node->right; // Заменил auto

                    while (successor->left != nullptr)

                        successor = successor->left;

                    node->key = successor->key;

                    node->val = successor->val;

                    this->removeNode(successor);

            }else if (node->left != nullptr) {

                    node->key = node->left->key;

                    node->val = node->left->val;

                    this->removeNode(node->left);

            }else if (node->right != nullptr) {

                    node->key = node->right->key;

                    node->val = node->right->val;

                    this->removeNode(node->right);

            }else {

                    if (node->parent == nullptr) {

                        free(node);

                        root = nullptr;

                        return;

            }

                    if (node->parent->left == node) {

                            node->parent->left = nullptr;

                            if (node->parent->right != nullptr

                            && node->parent->right->color == RED) {

                                node->parent->right->color = BLACK;

                                leftRotate(node->parent);

                            }

                    }else {

                        node->parent->right = nullptr;

                            if (node->parent->left != nullptr

                            && node->parent->left->color == RED) {

                                node->parent->left->color = BLACK;

                                rightRotate(node->parent);

                            }

                    }

                    if (node->parent->left == nullptr

                        && node->parent->right == nullptr

                        && node->parent->parent != nullptr) {

                            rightRotate(node->parent->parent);

                    }

                    free(node);

            }

    }

}

bool Tree::search(const int &key, int &val) const

{

    Node \*curr = root; // Заменил auto

    while (curr->left != nullptr || curr->right != nullptr)

    {

        if (curr->key == key) {

            val = curr->val;

            break;

        }

        if (cmp(key, curr->key) < 0)

            curr = curr->left;

        else

            curr = curr->right;

    }

    if (curr->key == key){

        val = curr->val;

        return 1;

    }

    return 0;

}

int Tree::cmp(const int &a, const int &b) const

{

    if (a < b) return -1;

    if (a == b) return 0;

    return 1;

}

void Tree::leftRotate(Node \*node)

{

    assert( node->right != nullptr);

    Node \*temp = node->right; // Заменил auto

    node->right = temp->left;

    if (temp->left != nullptr)

        temp->left->parent = node;

    temp->left = node;

    temp->parent = node->parent;

    node->parent = temp;

    if (root == node) {

        root = temp;

        return;

    }

    if (temp->parent->left == node)

            temp->parent->left = temp;

        else

            temp->parent->right = temp;

}

void Tree::rightRotate(Node \*node)

{

    assert( node->left != nullptr);

    Node \*temp = node->left; // Заменил auto

    node->left = temp->right;

    if (temp->right != nullptr)

            temp->right->parent = node;

    temp->right = node;

    temp->parent = node->parent;

    node->parent = temp;

    if (root == node) {

        root = temp;

        return;

    }

    if (temp->parent->left == node)

        temp->parent->left = temp;

    else

            temp->parent->right = temp;

}

void Tree::printTree() const

{

    std::cout << "----------------" << std::endl;

    std::queue<Node\*> q;

    q.push(root);

    while (!q.empty())

    {

        Node \*top = q.front(); // Заменил auto

        q.pop();

        if (top->color == RED)

            std::cout << "R" ;

        else

            std::cout << "B" ;

        std::cout << top->key;

        std::cout << " ";

        if (top->left != nullptr) {

            q.push(top->left);

            if (top->left->color == RED)

                std::cout << "R" ;

            else

                std::cout << "B" ;

            std::cout << top->left->key;

            std::cout << " ";

        }else {

            std::cout << "NULL" << " ";

        }

        if (top->right != nullptr) {

            q.push(top->right);

            if (top->right->color == RED)

                std::cout << "R" ;

            else

                std::cout << "B" ;

            std::cout << top->right->key;

            std::cout << " ";

        }else {

            std::cout << "NULL" << " ";

        }

        std::cout << std::endl;

    }

    std::cout << std::endl;

    std::cout << "----------------" << std::endl;

}

int Tree::getSize() const

{

    return this->size;

}

void Tree::clear()

{

    delete this->root;

    this->root = nullptr;

    this->size = 0;

}

# Приложение 2

#include <iostream>

#include <chrono>

#include "Red-Black-Tea\_modify.h"

#include <vector>

#include <unordered\_set>

#include <random>

#include <algorithm>

#include <ctime>

std::vector<int> generateUniqueNumbers(int n) {

    std::vector<int> arr;

    for (int i = 0; i < n; i++){

        arr.push\_back(i);

    }

    std::vector<int> result;

    while (arr.size() > 0)

    {

        std::mt19937 gen(time(0));

        std::uniform\_int\_distribution<size\_t> dist(0, arr.size() - 1);

        int arr\_index = dist(gen);

        result.push\_back(arr[arr\_index]);

        arr.erase(arr.begin() + arr\_index);

    }

    return result;

}

int main()

{

  int n;

  std::cout << "Insert Tree size: ";

  std::cin >> n;

  Tree t;

  int key;

  int val;

  auto start = std::chrono::high\_resolution\_clock::now();

  std::vector<int> Massive = generateUniqueNumbers(n);

  auto end = std::chrono::high\_resolution\_clock::now();

  auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

  std::cout << "Time taken by generate Unique Numbers: " << duration.count() << " microseconds" << std::endl;

  start = std::chrono::high\_resolution\_clock::now();

  for (int i = 0; i < n; i++)

  {

    key = i;

    val = i;

    t.insert(key, val);

  }

  end = std::chrono::high\_resolution\_clock::now();

  duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

  std::cout << "Time taken by generating Tree: " << duration.count() << " microseconds" << std::endl;

  start = std::chrono::high\_resolution\_clock::now();

  key = 1;

  val = -1;

  if (t.search(key, val)) {

    end = std::chrono::high\_resolution\_clock::now();

    std::cout<<"Search result: "<<val<<std::endl;

  }

    duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

  std::cout << "Time taken by searching key: " << duration.count() << " microseconds" << std::endl;

  key = n-1;

  start = std::chrono::high\_resolution\_clock::now();

  t.remove(key);

  end = std::chrono::high\_resolution\_clock::now();

  duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

  std::cout << "Time taken by removing key: " << duration.count() << " microseconds" << std::endl;

  return 0;

}