**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

**Отчет**

**По лабораторной работе №3**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Исследование бинарных деревьев»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 2309 |  | Савин П. А. |
| Преподаватель |  | Пестерев Д. О. |

Санкт-Петербург

2023

1. Постановка задачи

Реализовать известные деревья поиска.

1. Теоретическое описание  
     
   1. Бинарное дерево поиска. Базовая структура данных, на которой основаны все остальные модификации деревьев. Состоит из узлов, каждый из которых содержит значение-ключ и ссылки на левый и правый потомки. Узел, от которого исходит все дерево — корень дерева, крайние узлы дерева, не имеющие ни одного потомка — листовые элементы. В бинарном дереве поиска, если взять любой узел, в левом поддереве находятся элементы, меньшие узла по ключу, а в правом большие, и этот критерий рекурсивен — от корня дерева до листовых элементов. Поиск узла по значению зависит от высоты дерева, теоретическая оценка — . Высота дерева может очень сильно колебаться, так как никаких механизмов балансировки для бинарного дерева поиска не предусмотрено: худший случай — когда все узлы находятся по одну сторону, и дерево становится похожим на вытянутый список — ; лучший случай — когда дерево сбалансированно, т.е. высота левого поддерева для любого взятого узла отличается от высоты правого поддерева не более, чем на один элемент —

2. Красно-черное дерево. Бинарное дерево поиска с дополнительными условиями:

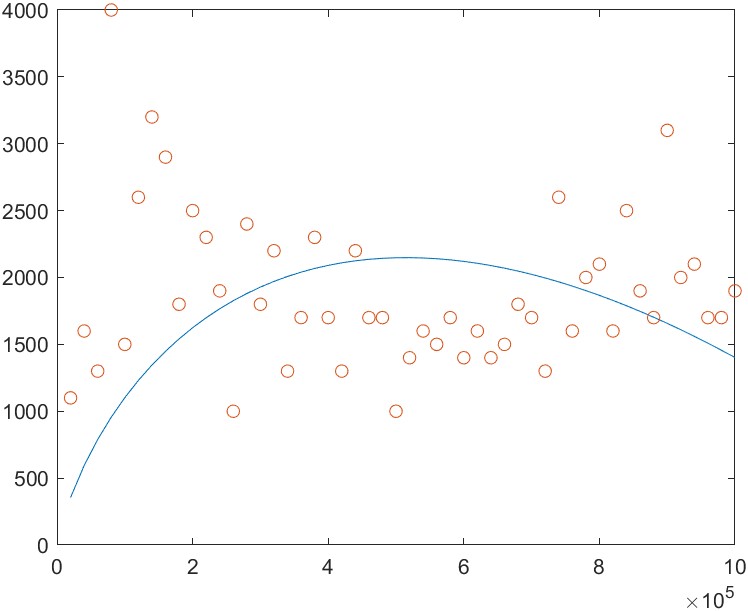
1. Каждый узел имеет свой цвет — красный или черный (следовательно, помимо ключа узлы хранят значение своего цвета);
2. Корень дерева — черный;
3. Листьями являются не крайние элементы в дереве, а потомки от них (так называемые фиктивные листья), и они должны быть черными;
4. Каждый красный узел должен иметь два черных дочерних узла. При этом у черного узла могут быть как красные дочерние узлы, так и черные. Красные узлы в качестве дочерних могут иметь только черные.
5. Пути от узла к его листьям должны содержать одинаковое количество черных узлов (так называемая черная высота).

За счет таких дополнительных условий обеспечивается самобалансировка дерева (общее условие баланса — высота поддеревьев, но у красно-черного дерева баланс достигается дополнительными условиями с цветами). Оценка сложности операций: вставка — ; удаление — .

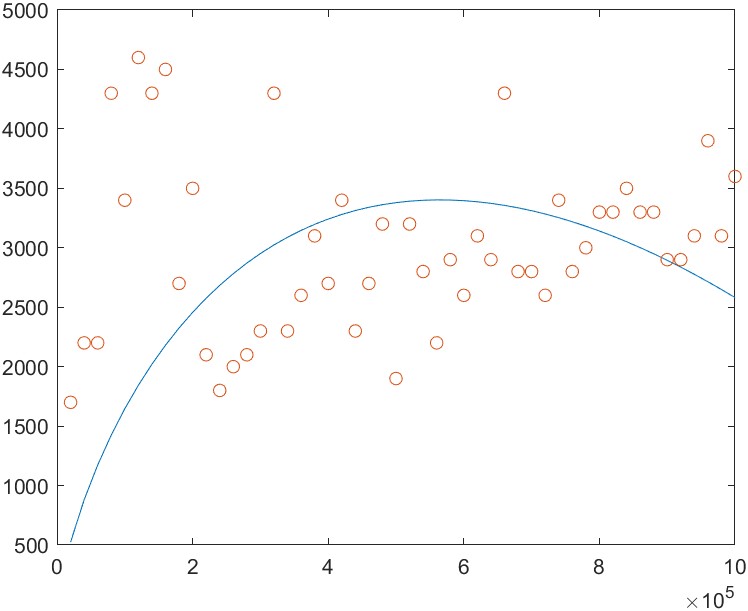
3. АВЛ-дерево. Бинарное дерево поиска, реализующее то самое общее условие баланса — узлы хранят значение высоты дерева, корнем которого они являются. Оценка сложности операций: вставка — ; удаление — .

1. Временная и практическая сложность  
   Было взято 1000000 чисел, перемешано, и вставлено в деревья; на каждый 20000-ый элемент был запущен таймер, который замерял время вставки и удаления элемента, также была посчитана высота деревьев на этих отметках. Для каждого полученного набора данных были составлены аппроксимированные графики зависимости времени от размера, а также аппроксимированные графики зависимости высоты от размера. Оси абсцисс соответствуют кол-ву вставленных в дерево элементов, оси ординат — время выполнения в наносекундах (высота дерева). Уравнение регрессионной кривой — .

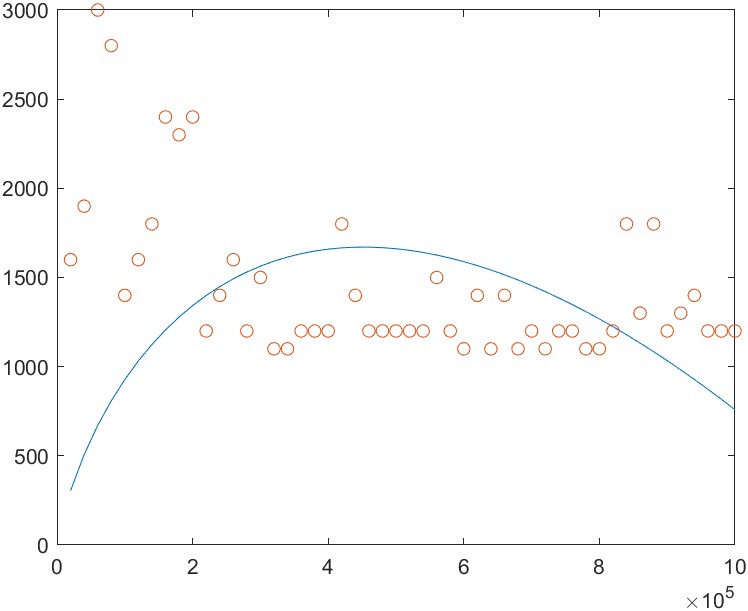
Операция вставки в красно-черное дерево с балансировкой



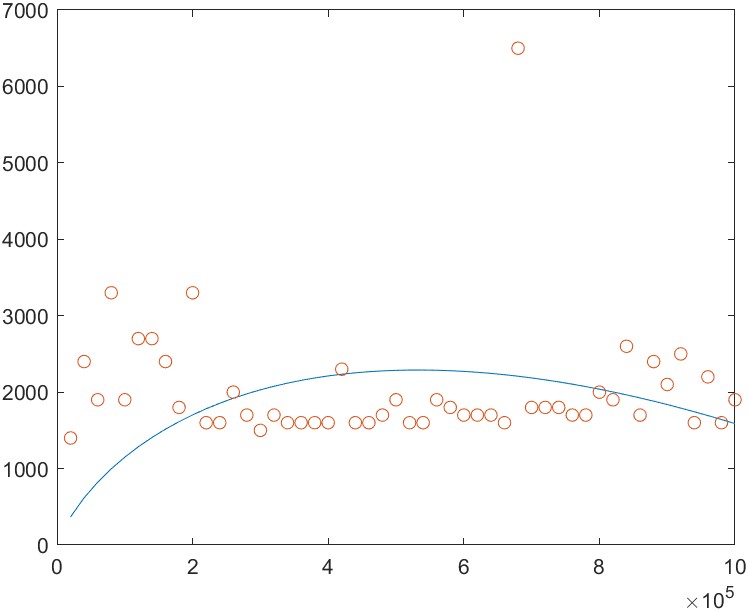
Операция вставки в АВЛ-дерево с балансировкой



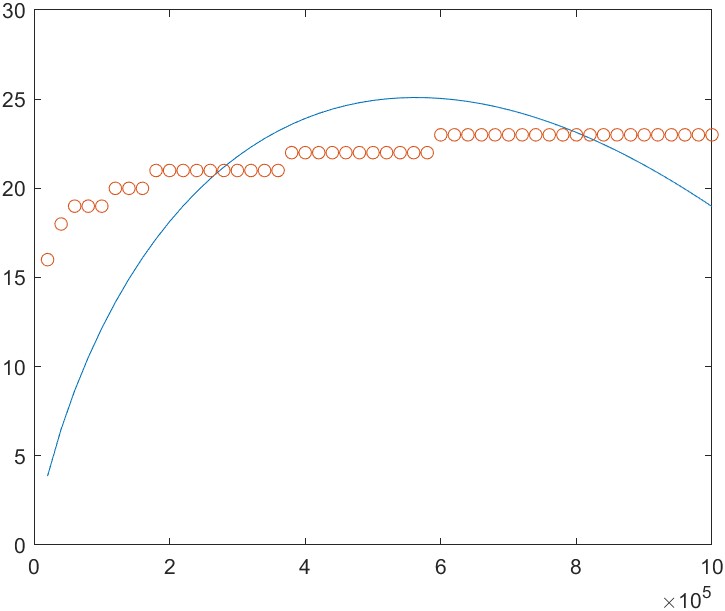
Операция удаления из красно-черного дерева с балансировкой



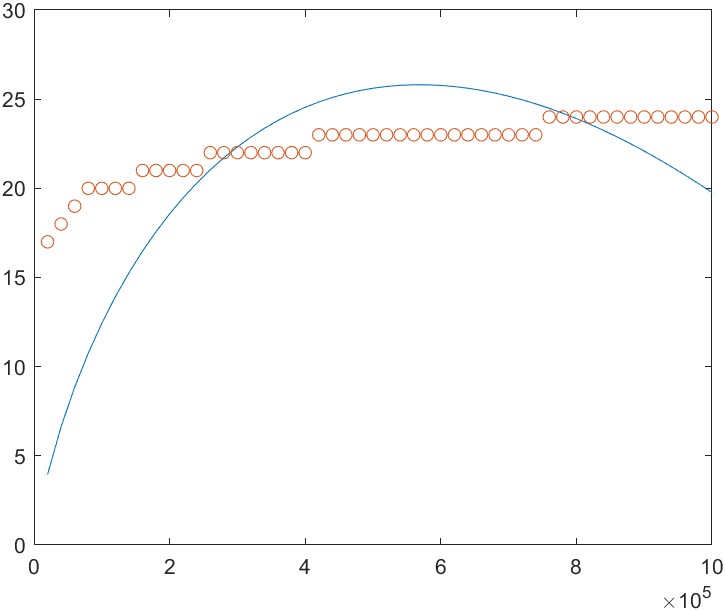
Операция удаления из АВЛ-дерева с балансировкой



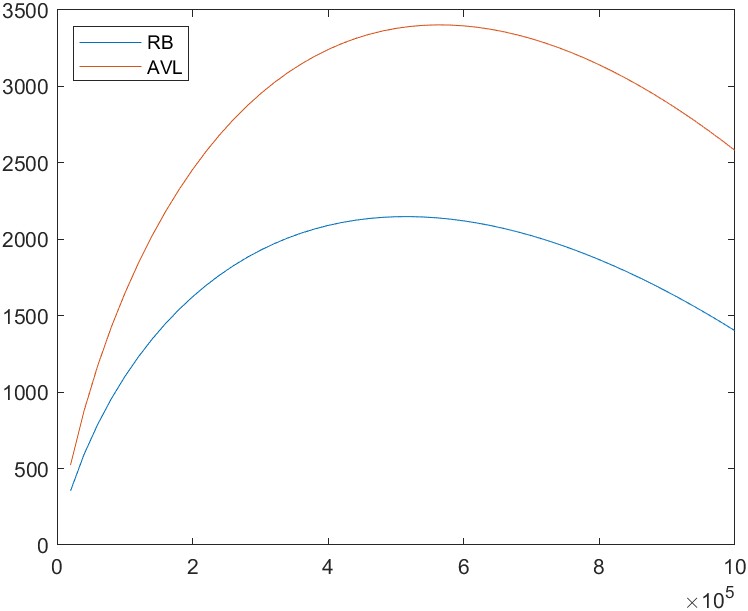
Высота красно-черного дерева при разном количестве элементов



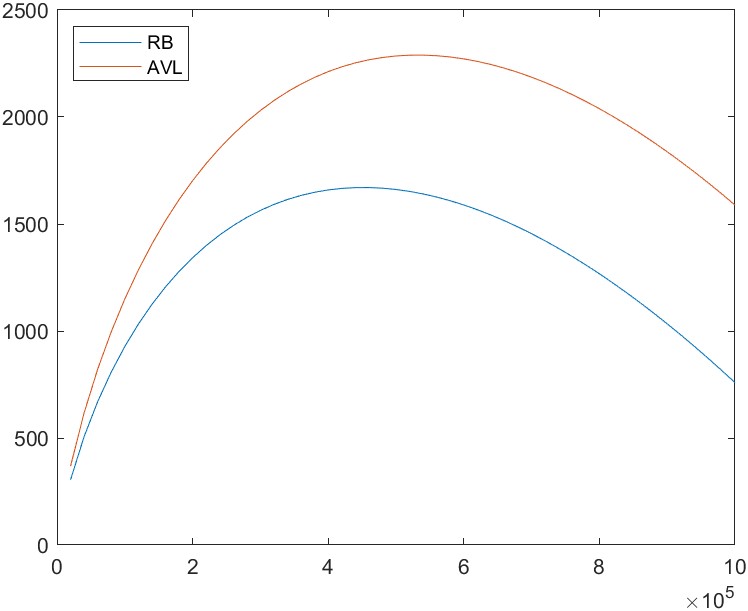
Высота АВЛ-дерева при разном количестве элементов



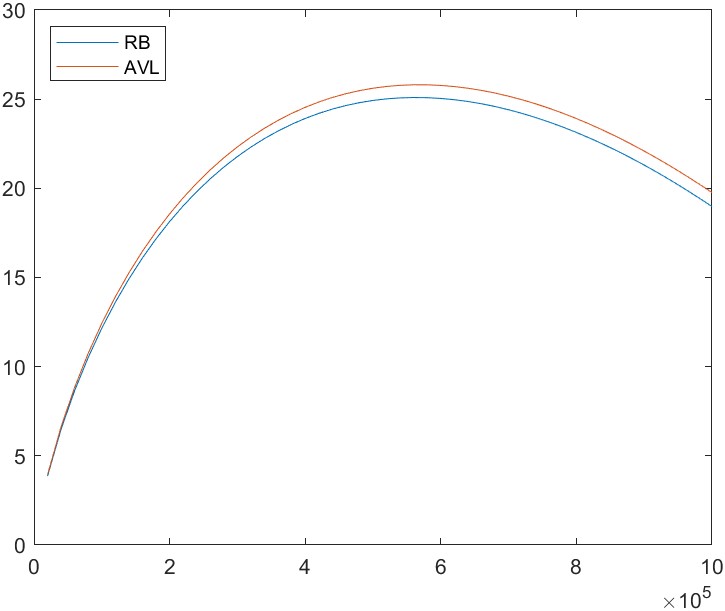
Сравнение вставки



Сравнение удаления



Сравнение высот



1. **Выводы**

Полученные данные показали, что красно-черное дерево отрабатывает лучше за счет более сложного, но более эффективного механизма балансировки. Что касается высот — даже несмотря на различие красно-черного и АВЛ-дерева в 1 элемент, по графику можно сказать, что их высоты растут сравнительно одинаково.

1. **Листинг**

BST.h

#pragma once

#include <iostream>

struct bst\_node {

int key;

bst\_node\* left;

bst\_node\* right;

bst\_node(int);

};

void preorder\_traversal(bst\_node\* node);

void postorder\_traversal(bst\_node\* node);

void inorder\_traversal(bst\_node\* node);

bst\_node\* insert(bst\_node\* root, int val);

bst\_node\* remove(bst\_node\* root, int key);

BST.cpp

#include "BST.h"

bst\_node::bst\_node(int val) {

key = val;

left = right = nullptr;

}

void preorder\_traversal(bst\_node\* node) {

if (node != nullptr) {

std::cout << node->key << " ";

preorder\_traversal(node->left);

preorder\_traversal(node->right);

}

}

void postorder\_traversal(bst\_node\* node) {

if (node != nullptr) {

postorder\_traversal(node->left);

postorder\_traversal(node->right);

std::cout << node->key << " ";

}

}

void inorder\_traversal(bst\_node\* node) {

if (node != nullptr) {

inorder\_traversal(node->left);

std::cout << node->key << " ";

inorder\_traversal(node->right);

}

}

bst\_node\* insert(bst\_node\* root, int val) {

if (root == nullptr) {

return new bst\_node(val);

}

if (val > root->key) {

root->right = insert(root->right, val);

}

else if (val < root->key) {

root->left = insert(root->left, val);

}

return root;

}

bst\_node\* remove(bst\_node\* root, int key) {

if (root == nullptr) {

return root;

}

if (root->key > key) {

root->left = remove(root->left, key);

return root;

}

else if (root->key < key) {

root->right = remove(root->right, key);

return root;

}

if (root->left == nullptr) {

bst\_node\* tmp = root->right;

delete root;

return tmp;

}

else if (root->right == NULL) {

bst\_node\* tmp = root->left;

delete root;

return tmp;

}

else {

bst\_node\* next\_p = root;

bst\_node\* next = root->right;

while (next->left != nullptr) {

next\_p = next;

next = next->left;

}

if (next\_p != root)

next\_p->left = next->right;

else

next\_p->right = next->right;

root->key = next->key;

delete next;

return root;

}

}

RB.h

#pragma once

#include <iostream>

struct rb\_node {

int key;

char color;

rb\_node\* left;

rb\_node\* right;

rb\_node\* parent;

rb\_node(int);

rb\_node\* uncle();

rb\_node\* sibling();

bool is\_left();

bool has\_red\_child();

void move\_down(rb\_node\* new\_parent);

};

struct rb\_tree {

rb\_node\* root;

rb\_tree();

rb\_node\* search(int val);

rb\_node\* successor(rb\_node\* node);

void swap\_colors(rb\_node\* u, rb\_node\* v);

void swap\_values(rb\_node\* u, rb\_node\* v);

void preorder\_traversal(rb\_node\* node);

void postorder\_traversal(rb\_node\* node);

void inorder\_traversal(rb\_node\* node);

void preorder();

void postorder();

void inorder();

void rotate\_left(rb\_node\* node);

void rotate\_right(rb\_node\* node);

void fix\_double\_red(rb\_node\* node);

void fix\_double\_black(rb\_node\* node);

rb\_node\* replace\_node(rb\_node\* node);

void insert(int val);

void remove\_node(rb\_node\* node);

void remove(int val);

};

RB.cpp

#include "RB.h"

rb\_node::rb\_node(int val) {

key = val;

color = 'R';

left = right = parent = nullptr;

}

rb\_node\* rb\_node::uncle() {

if (parent == nullptr || parent->parent == nullptr) {

return nullptr;

}

if (parent->is\_left()) {

return parent->parent->right;

}

else {

return parent->parent->left;

}

}

rb\_node\* rb\_node::sibling() {

if (parent == nullptr) {

return nullptr;

}

if (is\_left()) {

return parent->right;

}

else {

return parent->left;

}

}

bool rb\_node::is\_left() {

return this == parent->left;

}

bool rb\_node::has\_red\_child() {

return ((left != nullptr && left->color == 'R') || (right != nullptr && right->color == 'R'));

}

void rb\_node::move\_down(rb\_node\* new\_parent) {

if (parent != nullptr) {

if (is\_left()) {

parent->left = new\_parent;

}

else {

parent->right = new\_parent;

}

}

new\_parent->parent = parent;

parent = new\_parent;

}

rb\_tree::rb\_tree() {

root = nullptr;

}

rb\_node\* rb\_tree::search(int val) {

rb\_node\* tmp = root;

while (tmp != nullptr) {

if (val < tmp->key) {

if (tmp->left == nullptr)

break;

else

tmp = tmp->left;

}

else if (val == tmp->key) {

break;

}

else {

if (tmp->right == nullptr)

break;

else

tmp = tmp->right;

}

}

return tmp;

}

rb\_node\* rb\_tree::successor(rb\_node\* node) {

rb\_node\* tmp = node;

while (tmp->left != nullptr) {

tmp = tmp->left;

}

return tmp;

}

void rb\_tree::swap\_colors(rb\_node\* u, rb\_node\* v) {

char tmp;

tmp = u->color;

u->color = v->color;

v->color = tmp;

}

void rb\_tree::swap\_values(rb\_node\* u, rb\_node\* v) {

int tmp;

tmp = u->key;

u->key = v->key;

v->key = tmp;

}

void rb\_tree::preorder\_traversal(rb\_node\* node) {

if (node != nullptr) {

std::cout << node->key << node->color << " ";

preorder\_traversal(node->left);

preorder\_traversal(node->right);

}

}

void rb\_tree::postorder\_traversal(rb\_node\* node) {

if (node != nullptr) {

postorder\_traversal(node->left);

postorder\_traversal(node->right);

std::cout << node->key << node->color << " ";

}

}

void rb\_tree::inorder\_traversal(rb\_node\* node) {

if (node != nullptr) {

inorder\_traversal(node->left);

std::cout << node->key << node->color << " ";

inorder\_traversal(node->right);

}

}

void rb\_tree::preorder() {

preorder\_traversal(root);

}

void rb\_tree::postorder() {

postorder\_traversal(root);

}

void rb\_tree::inorder() {

inorder\_traversal(root);

}

void rb\_tree::rotate\_left(rb\_node\* node) {

rb\_node\* new\_parent = node->right;

if (node == root) {

root = new\_parent;

}

node->move\_down(new\_parent);

node->right = new\_parent->left;

if (new\_parent->left != nullptr) {

new\_parent->left->parent = node;

}

new\_parent->left = node;

}

void rb\_tree::rotate\_right(rb\_node\* node) {

rb\_node\* new\_parent = node->left;

if (node == root) {

root = new\_parent;

}

node->move\_down(new\_parent);

node->left = new\_parent->right;

if (new\_parent->right != nullptr) {

new\_parent->right->parent = node;

}

new\_parent->right = node;

}

void rb\_tree::fix\_double\_red(rb\_node\* node) {

if (node == root) {

node->color = 'B';

return;

}

rb\_node\* parent = node->parent, \* grandparent = parent->parent, \* uncle = node->uncle();

if (parent->color != 'B') {

if (uncle != NULL && uncle->color == 'R') {

parent->color = 'B';

uncle->color = 'B';

grandparent->color = 'R';

fix\_double\_red(grandparent);

}

else {

if (parent->is\_left()) {

if (node->is\_left()) {

swap\_colors(parent, grandparent);

}

else {

rotate\_left(parent);

swap\_colors(node, grandparent);

}

rotate\_right(grandparent);

}

else {

if (node->is\_left()) {

rotate\_right(parent);

swap\_colors(node, grandparent);

}

else {

swap\_colors(parent, grandparent);

}

rotate\_left(grandparent);

}

}

}

}

void rb\_tree::fix\_double\_black(rb\_node\* node) {

if (node == root) {

return;

}

rb\_node\* sibling = node->sibling(), \* parent = node->parent;

if (sibling == NULL) {

fix\_double\_black(parent);

}

else {

if (sibling->color == 'R') {

parent->color = 'R';

sibling->color = 'B';

if (sibling->is\_left()) {

rotate\_right(parent);

}

else {

rotate\_left(parent);

}

fix\_double\_black(node);

}

else {

if (sibling->has\_red\_child()) {

if (sibling->left != NULL and sibling->left->color == 'R') {

if (sibling->is\_left()) {

sibling->left->color = sibling->color;

sibling->color = parent->color;

rotate\_right(parent);

}

else {

sibling->left->color = parent->color;

rotate\_right(sibling);

rotate\_left(parent);

}

}

else {

if (sibling->is\_left()) {

sibling->right->color = parent->color;

rotate\_left(sibling);

rotate\_right(parent);

}

else {

sibling->right->color = sibling->color;

sibling->color = parent->color;

rotate\_left(parent);

}

}

parent->color = 'B';

}

else {

sibling->color = 'R';

if (parent->color == 'B')

fix\_double\_black(parent);

else

parent->color = 'B';

}

}

}

}

rb\_node\* rb\_tree::replace\_node(rb\_node\* node) {

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

return successor(node->right);

}

if (node->left == nullptr and node->right == nullptr) {

return nullptr;

}

if (node->left != nullptr) {

return node->left;

}

else {

return node->right;

}

}

void rb\_tree::insert(int val) {

rb\_node\* node = new rb\_node(val);

if (root == nullptr) {

node->color = 'B';

root = node;

}

else {

rb\_node\* tmp = search(val);

if (tmp->key == val) {

return;

}

node->parent = tmp;

if (val < tmp->key) {

tmp->left = node;

}

else {

tmp->right = node;

}

fix\_double\_red(node);

}

}

void rb\_tree::remove\_node(rb\_node\* node) {

rb\_node\* rep = replace\_node(node);

bool both\_black = ((rep == nullptr || rep->color == 'B') && (node->color == 'B'));

rb\_node\* parent = node->parent;

if (rep == nullptr) {

if (node == root) {

root = nullptr;

}

else {

if (both\_black) {

fix\_double\_black(node);

}

else {

if (node->sibling() != nullptr)

node->sibling()->color = 'R';

}

if (node->is\_left()) {

parent->left = nullptr;

}

else {

parent->right = nullptr;

}

}

delete node;

return;

}

if (node->left == nullptr || node->right == nullptr) {

if (node == root) {

node->key = rep->key;

node->left = node->right = nullptr;

delete rep;

}

else {

if (node->is\_left()) {

parent->left = rep;

}

else {

parent->right = rep;

}

delete node;

rep->parent = parent;

if (both\_black) {

fix\_double\_black(rep);

}

else {

rep->color = 'B';

}

}

return;

}

swap\_values(rep, node);

remove\_node(rep);

}

void rb\_tree::remove(int val) {

if (root == nullptr) {

return;

}

rb\_node\* node = search(val);

if (node->key != val) {

return;

}

remove\_node(node);

}

AVL.h

#pragma once

#include <iostream>

struct avl\_node {

int key;

int height;

avl\_node\* left;

avl\_node\* right;

avl\_node(int);

};

int max(int a, int b);

int height(avl\_node\* node);

int balance(avl\_node\* node);

void preorder\_traversal(avl\_node\* node);

void postorder\_traversal(avl\_node\* node);

void inorder\_traversal(avl\_node\* node);

avl\_node\* successor(avl\_node\* node);

avl\_node\* rotate\_left(avl\_node\* node);

avl\_node\* rotate\_right(avl\_node\* node);

avl\_node\* insert(avl\_node\* root, int val);

avl\_node\* remove(avl\_node\* root, int val);

AVL.cpp

#include "AVL.h"

avl\_node::avl\_node(int val) {

key = val;

height = 1;

left = right = nullptr;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

int height(avl\_node\* node) {

if (node == nullptr) {

return 0;

}

return node->height;

}

int balance(avl\_node\* node) {

if (node == nullptr) {

return 0;

}

return height(node->left) - height(node->right);

}

void preorder\_traversal(avl\_node\* node) {

if (node != nullptr) {

std::cout << node->key << "|" << balance(node) << " ";

preorder\_traversal(node->left);

preorder\_traversal(node->right);

}

}

void postorder\_traversal(avl\_node\* node) {

if (node != nullptr) {

postorder\_traversal(node->left);

postorder\_traversal(node->right);

std::cout << node->key << "|" << balance(node) << " ";

}

}

void inorder\_traversal(avl\_node\* node) {

if (node != nullptr) {

inorder\_traversal(node->left);

std::cout << node->key << "|" << balance(node) << " ";

inorder\_traversal(node->right);

}

}

avl\_node\* successor(avl\_node\* node) {

avl\_node\* cur = node->right;

while (cur->left != nullptr) {

cur = cur->left;

}

return cur;

}

avl\_node\* rotate\_left(avl\_node\* node) {

avl\_node\* x = node->right;

avl\_node\* y = x->left;

x->left = node;

node->right = y;

node->height = max(height(node->left), height(node->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

avl\_node\* rotate\_right(avl\_node\* node) {

avl\_node\* x = node->left;

avl\_node\* y = x->right;

x->right = node;

node->left = y;

node->height = max(height(node->left), height(node->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

avl\_node\* insert(avl\_node\* root, int val) {

if (root == nullptr) {

return new avl\_node(val);

}

if (val < root->key) {

root->left = insert(root->left, val);

}

else if (val > root->key) {

root->right = insert(root->right, val);

}

else {

return root;

}

root->height = 1 + max(height(root->left), height(root->right));

int bal = balance(root);

if (bal > 1 && val < root->left->key) {

return rotate\_right(root);

}

if (bal < -1 && val > root->right->key) {

return rotate\_left(root);

}

if (bal > 1 && val > root->left->key) {

root->left = rotate\_left(root->left);

return rotate\_right(root);

}

if (bal < -1 && val < root->right->key) {

root->right = rotate\_right(root->right);

return rotate\_left(root);

}

return root;

}

avl\_node\* remove(avl\_node\* root, int val) {

if (root == nullptr) {

return root;

}

if (val < root->key) {

root->left = remove(root->left, val);

}

else if (val > root->key) {

root->right = remove(root->right, val);

}

else {

if ((root->left == nullptr) || (root->right == nullptr)) {

avl\_node\* tmp = (root->left != nullptr ? root->left : root->right);

if (tmp == nullptr) {

tmp = root;

root = nullptr;

}

else {

\*root = \*tmp;

}

delete tmp;

}

else {

avl\_node\* tmp = successor(root);

root->key = tmp->key;

root->right = remove(root->right, tmp->key);

}

}

if (root == nullptr) {

return root;

}

root->height = 1 + max(height(root->left), height(root->right));

int bal = balance(root);

if (bal > 1 && balance(root->left) >= 0) {

return rotate\_right(root);

}

if (bal > 1 && balance(root->left) < 0) {

root->left = rotate\_left(root->left);

return rotate\_right(root);

}

if (bal < -1 && balance(root->right) <= 0) {

return rotate\_left(root);

}

if (bal < -1 && balance(root->right) > 0) {

root->right = rotate\_right(root->right);

return rotate\_left(root);

}

return root;

}

Ссылка на репозиторий: <https://github.com/pashkev14/A_DS_Lab3>