**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра CАПР**

отчет

**по лабораторной работе №1**

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

**Вариант 1**

**Тема: Ассоциативный массив на основе красно-черного дерева**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Бабкин И. А. |
| Преподаватель |  | Тутуева.А.В |

Санкт-Петербург

2021

## Постановка задачи

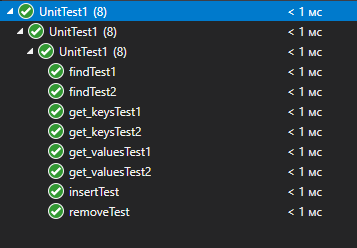
Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева. Наличие unit-тестов ко всем реализуемым методам является обязательным требованием.

## Описание и оценка временной сложности методов

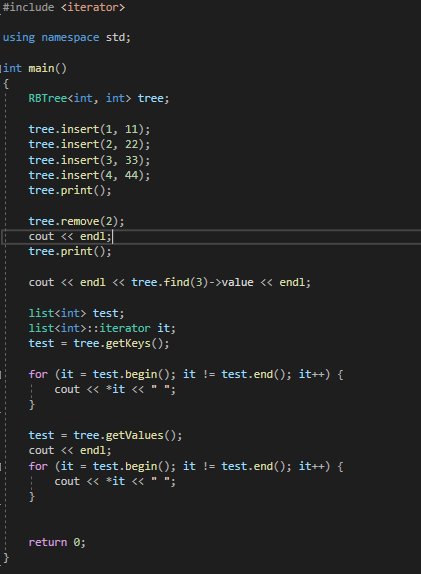
|  |  |  |
| --- | --- | --- |
| Название метода | Описание | Временная сложность |
| void leftRotate(Node\* x) | Левый поворот | O(lg(n)) |
| void rightRotate(Node\* y) | Правый поворот | O(lg(n)) |
| void insertRB(Node\* z) | Вставка элемента | O(lg(n)) |
| void insertFixUp(Node\* z) | Восстановление свойств после вставки | O(lg(n)) |
| void deleteNode(Node\* z) | Удаление элемента | O(h) |
| void deleteFixUp(Node\* x) | Восстановление свойств после удаления | O(h) |
| Node\* find(Key key) | Поиск элемента по ключу | O(lg(n)) |

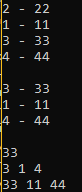
## Описание реализованный unit-тестов

|  |
| --- |
| Название теста |
| findTest |
| insertTest |
| removeTest |
| get\_keysTest |
| get\_valuesTest |



## Пример работы





## Листинг

RBTree.h

#pragma once

#include <iostream>

#include <list>

#include <queue>

using namespace std;

template <class Key, class Value>

class RBTree

{

class Node;

private:

enum class Color {

Red,

Black

};

class Node

{

public:

Node\* leftChild;

Node\* rightChild;

Node\* parent;

Key key;

Value value;

Color color;

Node(Node\* leftChild = nullptr, Node\* rightChild = nullptr, Node\* parent = nullptr,

Key key = NAN, Value value = NAN, Color color = Color::Black)

{

this->leftChild = leftChild;

this->rightChild = rightChild;

this->parent = parent;

this->key = key;

this->value = value;

this->color = color;

}

};

Node\* nil;

Node\* root;

public:

RBTree()

{

nil = new Node();

root = nil;

}

~RBTree()

{

}

void leftRotate(Node\* x)

{

Node\* y = x->rightChild;

x->rightChild = y->leftChild;

if (y->leftChild != nil) {

y->leftChild->parent = x;

}

if (x->parent == nil) {

root = y;

}

else if (x == x->parent->leftChild) {

x->parent->leftChild = y;

}

else {

x->parent->rightChild = y;

}

y->leftChild = x;

x->parent = y;

}

void rightRotate(Node\* y)

{

Node\* x = y->leftChild;

y->leftChild = x->rightChild;

if (x->rightChild != nil) {

x->rightChild->parent = y;

}

if (y->parent == nil) {

root = x;

}

else if (y == y->parent->rightChild) {

y->parent->rightChild = x;

}

else {

y->parent->leftChild = x;

}

x->rightChild = y;

y->parent = x;

}

void insertFixUp(Node\* z)

{

while (z!= root && z->parent->color == Color::Red) {

if (z->parent == z->parent->parent->leftChild) {

Node\* y = z->parent->parent->rightChild;

if (y->color == Color::Red) {

z->parent->color = Color::Black;

y->color = Color::Black;

z->parent->parent->color = Color::Red;

z = z->parent->parent;

}

else {

if (z == z->parent->rightChild) {

z = z->parent;

leftRotate(z);

}

z->parent->color = Color::Black;

z->parent->parent->color = Color::Red;

rightRotate(z->parent->parent);

}

}

else {

Node\* y = z->parent->parent->leftChild;

if (y->color == Color::Red) {

z->parent->color = Color::Black;

y->color = Color::Black;

z->parent->parent->color = Color::Red;

z = z->parent->parent;

}

else {

if (z == z->parent->leftChild) {

z = z->parent;

rightRotate(z);

}

z->parent->color = Color::Black;

z->parent->parent->color = Color::Red;

leftRotate(z->parent->parent);

}

}

}

root->color = Color::Black;

}

void insertRB(Node\* z)

{

Node\* y = nil;

Node\* x = root;

while (x != nil) {

y = x;

if (z->key < x->key) {

x = x->leftChild;

}

else {

x = x->rightChild;

}

}

z->parent = y;

if (y == nil) {

root = z;

}

else if (z->key < y->key) {

y->leftChild = z;

}

else {

y->rightChild = z;

}

z->leftChild = nil;

z->rightChild = nil;

z->color = Color::Red;

insertFixUp(z);

}

void insert(Key key, Value value)

{

Node\* node = new Node(nullptr, nullptr, nullptr, key, value, Color::Red);

insertRB(node);

}

void deleteFixUp(Node\* x)

{

while (x != root && x->color == Color::Black) {

if (x == x->parent->leftChild) {

Node\* w = x->parent->rightChild;

if (w->color == Color::Red) {

w->color = Color::Black;

x->parent->color = Color::Red;

leftRotate(x->parent);

w = x->parent->rightChild;

}

if (w->leftChild->color == Color::Black && w->rightChild->color == Color::Black) {

w->color = Color::Red;

x = x->parent;

}

else {

if (w->rightChild->color == Color::Black) {

w->leftChild->color = Color::Black;

w->color = Color::Red;

rightRotate(w);

w = x->parent->rightChild;

}

w->color = x->parent->color;

x->parent->color = Color::Black;

w->rightChild->color = Color::Black;

leftRotate(x->parent);

x = root;

}

}

else {

Node\* w = x->parent->leftChild;

if (w->color == Color::Red) {

w->color = Color::Black;

x->parent->color = Color::Red;

rightRotate(x->parent);

w = x->parent->leftChild;

}

if (w->rightChild->color == Color::Black && w->leftChild->color == Color::Black) {

w->color = Color::Red;

x = x->parent;

}

else {

if (w->leftChild->color == Color::Black) {

w->rightChild->color = Color::Black;

w->color = Color::Red;

leftRotate(w);

w = x->parent->rightChild;

}

w->color = x->parent->color;

x->parent->color = Color::Black;

w->leftChild->color = Color::Black;

rightRotate(x->parent);

x = root;

}

}

}

x->color = Color::Black;

}

void deleteNode(Node\* z)

{

Node\* x;

Node\* y;

if (z == nil) {

return;

}

if (z->leftChild == nil || z->rightChild == nil) {

y = z;

}

else {

y = z->rightChild;

while (y->leftChild != nil) {

y = y->leftChild;

}

}

if (y->leftChild != nil) {

x = y->leftChild;

}

else {

x = y->rightChild;

}

x->parent = y->parent;

if (y->parent != nil) {

if (y == y->parent->leftChild) {

y->parent->leftChild = x;

}

else {

y->parent->rightChild = x;

}

}

else {

root = x;

}

if (y != z) {

z->key = y->key;

z->value = y->value;

}

if (y->color == Color::Black) {

deleteFixUp(x);

}

delete y;

}

Node\* find(Key key)

{

if (root == nil) {

throw("There is no such element");

}

else {

Node\* current;

current = root;

while (2 + 2 == 4) {

if (key <= current->key) {

if (key == current->key) {

return current;

}

if (current->leftChild == nil) {

throw("There is no such element");

}

current = current->leftChild;

}

else {

if (current->rightChild == nil) {

throw("There is no such element");

}

current = current->rightChild;

}

}

}

}

void remove(Key key)

{

deleteNode(find(key));

}

void clear(Node\* x)

{

if (x->leftChild != nil) {

clear(x->leftChild);

}

if (x->rightChild != nil) {

clear(x->rightChild);

}

delete x;

}

list<Key> getKeys()

{

if (root == nil) {

throw("There is no element");

}

queue<Node\*> queueKey;

list<Value> listKey;

queueKey.push(root);

while (!queueKey.empty()) {

Node\* temp = queueKey.front();

queueKey.pop();

listKey.push\_back(temp->key);

if (temp->leftChild != nil) {

queueKey.push(temp->leftChild);

}

if (temp->rightChild != nil) {

queueKey.push(temp->rightChild);

}

}

return listKey;

}

list<Value> getValues()

{

if (root == nil) {

throw("There is no element");

}

queue<Node\*> queueValues;

list<Value> listValues;

queueValues.push(root);

while (!queueValues.empty()) {

Node\* temp = queueValues.front();

queueValues.pop();

listValues.push\_back(temp->value);

if (temp->leftChild != nil) {

queueValues.push(temp->leftChild);

}

if (temp->rightChild != nil) {

queueValues.push(temp->rightChild);

}

}

return listValues;

}

void print()

{

if (root == nil) {

throw("There is no element");

}

queue<Node\*> queue;

queue.push(root);

while (!queue.empty()) {

Node\* temp = queue.front();

queue.pop();

cout << temp->key << " - " << temp->value << endl;

if (temp->leftChild != nil) {

queue.push(temp->leftChild);

}

if (temp->rightChild != nil) {

queue.push(temp->rightChild);

}

}

}

};