**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: **Ассоциативные массивы**

|  |  |  |
| --- | --- | --- |
| Студент гр. 0309 |  | Головатюк К.А. |
| Преподаватель |  | Тутуева А.В |

Санкт-Петербург

2022

### Постановка задачи

Реализовать методы:

1. insert(ключ, значение); // добавление элемента с ключом и значением
2. remove(ключ); // удаление элемента по ключу
3. find(ключ); // поиск элемента по ключу
4. clear(); // очищение ассоциативного массива
5. get\_keys(); // возвращает список ключей
6. get\_values(); // возвращает список значений
7. print(); // вывод в консоль

Провести тестирование методов на случайных данных. Оценить временную сложность каждого метода.

### Описание реализованных методов и оценка их временной сложности

|  |  |  |
| --- | --- | --- |
| Название метода | Описание | Оценка временной сложности |
| void insert(T key, V value) | Вставка элемента | O(log(n)) |
| void remove(T key) | Удаление элемента | O(log(n)) |
| Node<T, V>\* find(T key) | Поиск элемента | О(log(n)) |
| void clear() | Очистить древо | O(1) |
| List<T> get\_keys() | Получить все ключи | О(n) |
| List<V> get\_values() | Получить все значения | O(n) |
| void print() | Вывод в консоль | О(n) |

### Описание реализованных unit-тестов

Список тестов:

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Тесты реализованы при помощи Google test. Все тесты представляют собой сравнения дерева, после выполнения функции, с контрольными данными.

### Пример работы методов

**print:**

**Изображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст

Автоматически созданное описание**

**get\_keys, get\_values:**

**Изображение выглядит как текст, закрыть

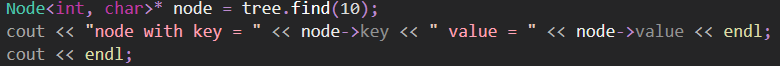
Автоматически созданное описание**

**Изображение выглядит как текст

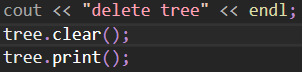
Автоматически созданное описание**

**find:**

****

****

**clear:**

**Изображение выглядит как текст

Автоматически созданное описание**

**Вывод**

При реализации была изучена такая структура данных как красно черное дерево, так же получены навыки создания шаблонных классов и структур.

### Листинг

Файл main.cpp:

#include"RBTree.h"

#include<iostream>

using namespace std;

int main()

{

vector<int> nums{ 10,40,30,60,90,70,20,50,80,100 };

vector<char> symbols{ 'q','w','e','r','t','y','u','i','o','p'};

RBtree<int, char> tree;

for (int i = 0; i < 10; i++)

tree.insert(nums[i], symbols[i]);

tree.print();

cout << endl;

List<int> listInt = tree.get\_keys();

cout << "List keys: ";

listInt.print();

cout << endl;

List<char> listChar = tree.get\_values();

cout << "List values: ";

listChar.print();

cout << endl;

Node<int, char>\* node = tree.find(10);

cout << "node with key = " << node->key << " value = " << node->value << endl;

cout << endl;

tree.remove(10);

cout << "remove node with key = 10 " << endl;

node = tree.find(10);

cout << "node equal nullptr = " << (node == nullptr) << endl;

cout << endl;

cout << "delete tree" << endl;

tree.clear();

tree.print();

return 0;

}

Файл RBTree.h:

#pragma once

#include <iostream>

#include <vector>

#include "List.h"

using namespace std;

enum RBColor { Black, Red };

template<typename T, typename V>

struct Node {

T key;

V value;

RBColor color;

Node<T, V>\* left = nullptr;

Node<T, V>\* right = nullptr;

Node<T, V>\* parent = nullptr;

Node(T key, V value, RBColor color, Node\* left, Node\* right, Node\* parent) :

key(key), value(value), color(color), left(left), right(right), parent(parent) {};

};

template <typename T, typename V>

class RBtree {

public:

RBtree() : root(nullptr) {

root = nullptr;

};

~RBtree() {

clear();

};

void insert(T key, V value) {

Node<T, V>\* node = new Node<T, V>(key, value, Red, nullptr, nullptr, nullptr);

insert(root, node);

};

void remove(T key) {

Node<T, V>\* deletenode = find(root, key);

if (deletenode != nullptr)

remove(root, deletenode);

};

Node<T, V>\* find(T key) {

return find(root, key);

};

void clear() {

destory(root);

};

List<T> get\_keys() {

List<T>\* list = new List<T>();

if (root == nullptr)

cout << "empty RBtree\n";

else {

return \*get\_keys(root, list);

}

return \*list;

};

List<V> get\_values() {

List<V>\* list = new List<V>();

if (root == nullptr)

cout << "empty RBtree\n";

else {

return \*get\_values(root, list);

}

return \*list;

};

void print() {

if (root == nullptr)

cout << "empty RBtree\n";

else

print(root);

};

private:

Node <T, V>\* root;

void insert(Node<T, V>\*& root, Node<T, V>\* node) {

Node<T, V>\* x = root;

Node<T, V>\* y = nullptr;

while (x != nullptr)

{

y = x;

if (node->key > x->key)

x = x->right;

else

x = x->left;

}

node->parent = y;

if (y != nullptr)

{

if (node->key > y->key)

y->right = node;

else

y->left = node;

}

else

root = node;

node->color = Red;

InsertFixUp(root, node);

};

void InsertFixUp(Node<T, V>\*& root, Node<T, V>\* node) {

Node<T, V>\* parent;

parent = node->parent;

while (node != RBtree::root && parent->color == Red)

{

Node<T, V>\* gparent = parent->parent;

if (gparent->left == parent)

{

Node<T, V>\* uncle = gparent->right;

if (uncle != nullptr && uncle->color == Red)

{

parent->color = Black;

uncle->color = Black;

gparent->color = Red;

node = gparent;

parent = node->parent;

}

else

{

if (parent->right == node)

{

leftRotate(root, parent);

swap(node, parent);

}

rightRotate(root, gparent);

gparent->color = Red;

parent->color = Black;

break;

}

}

else

{

Node<T, V>\* uncle = gparent->left;

if (uncle != nullptr && uncle->color == Red)

{

gparent->color = Red;

parent->color = Black;

uncle->color = Black;

node = gparent;

parent = node->parent;

}

else

{

if (parent->left == node)

{

rightRotate(root, parent);

swap(parent, node);

}

leftRotate(root, gparent);

parent->color = Black;

gparent->color = Red;

break;

}

}

}

root->color = Black;

};

void leftRotate(Node<T, V>\*& root, Node<T, V>\* x) {

Node<T, V>\* y = x->right;

x->right = y->left;

if (y->left != nullptr)

y->left->parent = x;

y->parent = x->parent;

if (x->parent == nullptr)

root = y;

else {

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

x->parent->left = y;

else

x->parent->right = y;

}

y->left = x;

x->parent = y;

};

void rightRotate(Node<T, V>\*& root, Node<T, V>\* y) {

Node<T, V>\* x = y->left;

y->left = x->right;

if (x->right != nullptr)

x->right->parent = y;

x->parent = y->parent;

if (y->parent == nullptr)

root = x;

else {

if (y == y->parent->right)

y->parent->right = x;

else

y->parent->left = x;

}

x->right = y;

y->parent = x;

};

void remove(Node<T, V>\*& root, Node<T, V>\* node) {

Node<T, V>\* child, \* parent;

RBColor color;

if (node->left != NULL && node->right != NULL)

{

Node<T, V>\* replace = node;

replace = node->right;

while (replace->left != NULL)

{

replace = replace->left;

}

if (node->parent != NULL)

{

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

node->parent->left = replace;

else

node->parent->right = replace;

}

else

root = replace;

child = replace->right;

parent = replace->parent;

color = replace->color;

if (parent == node)

parent = replace;

else

{

if (child != NULL)

child->parent = parent;

parent->left = child;

replace->right = node->right;

node->right->parent = replace;

}

replace->parent = node->parent;

replace->color = node->color;

replace->left = node->left;

node->left->parent = replace;

if (color == Black)

removeFixUp(root, child, parent);

delete node;

return;

}

if (node->left != NULL)

child = node->left;

else

child = node->right;

parent = node->parent;

color = node->color;

if (child)

{

child->parent = parent;

}

if (parent)

{

if (node == parent->left)

parent->left = child;

else

parent->right = child;

}

else

RBtree::root = child;

if (color == Black)

{

removeFixUp(root, child, parent);

}

delete node;

};

void removeFixUp(Node<T, V>\*& root, Node<T, V>\* node, Node<T, V>\* parent) {

Node<T, V>\* othernode;

while ((!node) || node->color == Black && node != RBtree::root)

{

if (parent->left == node)

{

othernode = parent->right;

if (othernode->color == Red)

{

othernode->color = Black;

parent->color = Red;

leftRotate(root, parent);

othernode = parent->right;

}

else

{

if (!(othernode->right) || othernode->right->color == Black)

{

othernode->left->color = Black;

othernode->color = Red;

rightRotate(root, othernode);

othernode = parent->right;

}

othernode->color = parent->color;

parent->color = Black;

othernode->right->color = Black;

leftRotate(root, parent);

node = root;

break;

}

}

else

{

othernode = parent->left;

if (othernode->color == Red)

{

othernode->color = Black;

parent->color = Red;

rightRotate(root, parent);

othernode = parent->left;

}

if ((!othernode->left || othernode->left->color == Black) && (!othernode->right || othernode->right->color == Black))

{

othernode->color = Red;

node = parent;

parent = node->parent;

}

else

{

if (!(othernode->left) || othernode->left->color == Black)

{

othernode->right->color = Black;

othernode->color = Red;

leftRotate(root, othernode);

othernode = parent->left;

}

othernode->color = parent->color;

parent->color = Black;

othernode->left->color = Black;

rightRotate(root, parent);

node = root;

break;

}

}

}

if (node)

node->color = Black;

};

void destory(Node<T, V>\*& node) {

if (node == nullptr)

return;

destory(node->left);

destory(node->right);

delete node;

node = nullptr;

};

Node<T, V>\* find(Node<T, V>\* node, T key) const {

if (node == nullptr || node->key == key)

return node;

else

if (key > node->key)

return find(node->right, key);

else

return find(node->left, key);

};

void print(Node<T, V>\* node)const {

if (node == nullptr)

return;

if (node->parent == nullptr)

cout << node->key << "(" << node->color << ") is root value = " << node->value << endl;

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

{

cout << node->key << "(" << node->color << ") is " << node->parent->key << " " << "left child value = " << node->value << endl;

}

else

{

cout << node->key << "(" << node->color << ") is " << node->parent->key << " " << "right child value = " << node->value << endl;

}

print(node->left);

print(node->right);

};

List<T>\* get\_keys(Node<T, V>\* node, List<T>\* list) {

if (node == nullptr)

return list;

list->push\_back(node->key);

get\_keys(node->left, list);

get\_keys(node->right, list);

return list;

};

List<V>\* get\_values(Node<T, V>\* node, List<V>\* list) {

if (node == nullptr)

return list;

list->push\_back(node->value);

get\_values(node->left, list);

get\_values(node->right, list);

return list;

};

};

Файл List.h

#pragma once

#include<iostream>

using namespace std;

template<typename T>

struct ListNode {

T val;

ListNode<T>\* next;

ListNode(T \_val) : val(\_val), next(nullptr) {}

};

template<typename T>

class List {

public:

List() : head(nullptr), tail(nullptr) {};

bool is\_empty() {

return head == nullptr;

};

void push\_back(T \_val) {

ListNode<T>\* p = new ListNode<T>(\_val);

if (is\_empty()) {

head = p;

tail = p;

return;

}

tail->next = p;

tail = p;

};

void print() {

if (is\_empty()) return;

ListNode<T>\* p = head;

while (p) {

cout << p->val << " ";

p = p->next;

}

cout << endl;

}

ListNode<T>\* take\_head() {

return head;

}

private:

ListNode<T>\* head;

ListNode<T>\* tail;

};

Файл test.cpp

#include "pch.h"

#include "RBtree.h"

#include <iostream>

#include <windows.h>

void TreeOut(RBtree<int, int>\* test, HANDLE hStdOut) {

SetConsoleTextAttribute(hStdOut, FOREGROUND\_GREEN | FOREGROUND\_INTENSITY);

cout << "[ Keys] ";

SetConsoleTextAttribute(hStdOut, FOREGROUND\_GREEN | FOREGROUND\_BLUE

| FOREGROUND\_RED | FOREGROUND\_INTENSITY);

List<int> keys = test->get\_keys();

keys.print();

SetConsoleTextAttribute(hStdOut, FOREGROUND\_GREEN | FOREGROUND\_INTENSITY);

cout << "[ Value] ";

SetConsoleTextAttribute(hStdOut, FOREGROUND\_GREEN | FOREGROUND\_BLUE

| FOREGROUND\_RED | FOREGROUND\_INTENSITY);

List<int> value = test->get\_values();

value.print();

}

void PrintTabs(HANDLE hStdOut) {

SetConsoleTextAttribute(hStdOut, FOREGROUND\_GREEN | FOREGROUND\_INTENSITY);

cout << "[ ] ";

SetConsoleTextAttribute(hStdOut, FOREGROUND\_GREEN | FOREGROUND\_BLUE

| FOREGROUND\_RED | FOREGROUND\_INTENSITY);

}

TEST(InputTest, insert) {

HANDLE hStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(hStdOut, FOREGROUND\_GREEN | FOREGROUND\_BLUE

| FOREGROUND\_RED | FOREGROUND\_INTENSITY);

vector<int> nums{ 10,40,30,60,90,70,20,50,80,100 };

RBtree<int, int> tree;

for (auto num : nums)

tree.insert(num, num);

TreeOut(&tree, hStdOut);

List<int> value = tree.get\_values();

ListNode<int>\* v = value.take\_head();

vector<int> n{ 60,30,10,20,40,50,80,70,90,100 };

for (int i = 0; i < 10; i++) {

ASSERT\_EQ(v->val, n[i]);

v = v->next;

}

tree.insert(0, 0);

value = tree.get\_values();

v = value.take\_head();

vector<int> m{ 60,30,10,0,20,40,50,80,70,90,100 };

for (int i = 0; i < 11; i++) {

ASSERT\_EQ(v->val, m[i]);

v = v->next;

}

}

TEST(DeleteTest, remove) {

HANDLE hStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(hStdOut, FOREGROUND\_GREEN | FOREGROUND\_BLUE

| FOREGROUND\_RED | FOREGROUND\_INTENSITY);

vector<int> nums{ 10,40,30,60,90,70,20,50,80,100 };

RBtree<int, int> tree;

for (auto num : nums)

tree.insert(num, num);

TreeOut(&tree, hStdOut);

tree.remove(10);

List<int> value = tree.get\_values();

ListNode<int>\* v = value.take\_head();

vector<int> n{ 60,30,20,40,50,80,70,90,100 };

for (int i = 0; i < 9; i++) {

ASSERT\_EQ(v->val, n[i]);

v = v->next;

}

}

TEST(DeleteTest, clear) {

HANDLE hStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(hStdOut, FOREGROUND\_GREEN | FOREGROUND\_BLUE

| FOREGROUND\_RED | FOREGROUND\_INTENSITY);

vector<int> nums{ 10,40,30,60,90,70,20,50,80,100 };

RBtree<int, int> tree;

for (auto num : nums)

tree.insert(num, num);

TreeOut(&tree, hStdOut);

tree.clear();

PrintTabs(hStdOut);

List<int> value = tree.get\_values();

ListNode<int>\* v = value.take\_head();

ASSERT\_EQ(v, nullptr);

}

TEST(OutputTest, get\_keys) {

HANDLE hStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(hStdOut, FOREGROUND\_GREEN | FOREGROUND\_BLUE

| FOREGROUND\_RED | FOREGROUND\_INTENSITY);

vector<int> nums{ 10,40,30,60,90,70,20,50,80,100 };

RBtree<int, int> tree;

for (auto num : nums)

tree.insert(num, num);

TreeOut(&tree, hStdOut);

List<int> key = tree.get\_keys();

ListNode<int>\* k = key.take\_head();

vector<int> n{ 60,30,10,20,40,50,80,70,90,100 };

for (int i = 0; i < 10; i++) {

ASSERT\_EQ(k->val, n[i]);

k = k->next;

}

}

TEST(OutputTest, get\_values) {

HANDLE hStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(hStdOut, FOREGROUND\_GREEN | FOREGROUND\_BLUE

| FOREGROUND\_RED | FOREGROUND\_INTENSITY);

vector<int> nums{ 10,40,30,60,90,70,20,50,80,100 };

RBtree<int, int> tree;

for (auto num : nums)

tree.insert(num, num);

TreeOut(&tree, hStdOut);

List<int> value = tree.get\_values();

ListNode<int>\* v = value.take\_head();

vector<int> n{ 60,30,10,20,40,50,80,70,90,100 };

for (int i = 0; i < 10; i++) {

ASSERT\_EQ(v->val, n[i]);

v = v->next;

}

}

TEST(SearchTest, find) {

HANDLE hStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(hStdOut, FOREGROUND\_GREEN | FOREGROUND\_BLUE

| FOREGROUND\_RED | FOREGROUND\_INTENSITY);

vector<int> nums{ 10,40,30,60,90,70,20,50,80,100 };

RBtree<int, int> tree;

for (auto num : nums)

tree.insert(num, num);

TreeOut(&tree, hStdOut);

for (int i = 10; i <= 100; i += 10) {

ASSERT\_EQ(tree.find(i)->value, i);

}

}