Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

**Кафедра «Компьютерные системы и программные технологии»**

**К У Р С О В А Я РА Б О Т А**

**Программирование на языке C**

по дисциплине «Низкоуровневое программирование»

Выполнил

студент гр. 23531/5

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Иванов И.Д.

(подпись)

Руководитель

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Петров М.А.

(подпись)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г.

Санкт-Петербург   
2019

Санкт-Петербургский политехнический университет Петра Великого

**ЗАДАНИЕ**

**НА ВЫПОЛНЕНИЕ КУРСОВОГО ПРОЕКТА**

**(КУРСОВОЙ РАБОТЫ)**

студенту группы 23531/5 Иванову Илье Дмитриевичу

*(номер группы)* *(фамилия, имя, отчество)*

1. ***Тема проекта (работы):*** Программирование на языкеC
2. ***Срок сдачи законченного проекта (работы):*** 04.06.2019
3. ***Исходные данные к проекту (работе)***:вариант3:Красно-черное дерево

***4. Содержание пояснительной записки*** :введение,основнаячасть(описание программы: описание API библиотеки, описание форматов файлов, параметров командной строки, руководство программиста; тестирование программы), заключение, список используемых источников, приложения.

***Дата получения задания***: « 12 »февраля2019г.

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Петров М.А.

*(подпись)*

Задание принял к исполнению \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Иванов И.Д.

*(подпись студента)*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(дата)*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **СОДЕРЖАНИЕ** |  |
| Введение. | . . . . . . | . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 4 |
| 1.1. | Постановка задачи. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 4 |
| Основная часть. . | | . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 4 |
| 2.1. | Описание программы. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 4 |
|  | 2.1.1 | Описание API библиотеки. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 4 |
|  | 2.1.2 | Описание форматов файлов, параметров командной строки. . | 5 |
|  | 2.1.3 | Руководство программиста. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 5 |
| 2.2. | Тестирование программы. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 5 |
| Заключение. . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 6 |
| Список используемых источников. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | | 6 |
| Приложение 1. Код программы. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | | 7 |

**1. Введение**

Дерево – структура данных, эмулирующая древовидную структуру в виде набора связанных узлов. Является связным графом, не содержащим циклы.

Двоичное дерево - иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей).

Красно-чёрное дерево - самобалансирующееся двоичное дерево поиска, гарантирующее логарифмический рост высоты дерева от числа узлов и быстро выполняющее основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева — «цвета». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный».

**1.1 Постановка задачи**

1. Разработать статическую библиотеку, реализующую определенный вариантом задания абстрактный тип данных.
2. Разработать демонстрационную программу – консольное приложение, обеспечивающее ввод данных из файла (файлов), их обработку и вывод в файл (файлы); имена файлов передаются в качестве параметров командной строки.

Определенный вариантом задания абстрактный тип данных:

Красно-черное дерево.

**2. Основная часть**

**2.1 Описание программы**

**2.1.1 Описание API библиотеки**

Реализация типа данных предусматривает динамическое выделение памяти и функцию деинициализации, обеспечивающую освобождение всей выделенной памяти. Разработанный исходный код компилируется gcc без ошибок со следующими параметрами: -std=c11 -pedantic –Wall - Wextra. Сборка библиотеки, демонстрационной программы и модульных тестов осуществляется утилитой make.

Библиотека librb\_tree.a содержит файл rb\_tree.c, содержащий в себе описание структуры красно-чёрного дерева, и заголовочный файл rb\_tree.h, содержащий объявление функций. Структура дерева состоит из полей, описывающих корень дерева, функцию для сравнения узлов дерева и его размера:

struct rb\_tree {

struct rb\_node \*root;

rb\_tree\_comparator cmp;

size\_t size;

};

Реализуемые функции:

* int rb\_tree\_node\_compare(struct rb\_tree \*self, struct rb\_node \*a, struct rb\_node \*b) –сравнение узлов дерева
* struct rb\_node \*rb\_node\_init(void \*value); -инициализация узла дерева
* void rb\_node\_dealloc(struct rb\_node \*self) –деинициализация узла дерева
* struct rb\_tree \*rb\_tree\_init(rb\_tree\_comparator cmp) - инициализация дерева
* void rb\_tree\_dealloc(struct rb\_tree \*self) – деинициализация дерева
* void \*rb\_tree\_find(struct rb\_tree \*self, void \*value) –поиск элемента
* int rb\_tree\_insert(struct rb\_tree \*self, void \*value)–вставка нового элемента
* int rb\_tree\_remove(struct rb\_tree \*self, void \*value) – удаление элемента
* size\_t rb\_tree\_size(struct rb\_tree \*self) –получение размера дерева

• void print\_tree\_to\_file(struct rb\_node \*root, char \*filename, char \*mode, char \*msg)–печать дерева в файл

* int rb\_tree\_insert\_node(struct rb\_tree \*self, struct rb\_node \*node); - вставка уже созданного узла
* struct rb\_tree \*read\_from\_file(char \*filename)–чтение дерева из файла

**2.1.2 Описание форматов файлов, параметров командной строки** Ввод и вывод дерева осуществляется через текстовые файлы. Имена входного и выходного файла передаются в качестве аргументов командной строки.

**2.1.3 Руководство программиста**

Последовательность действий демонстрационной программы:

* Определяет имена входного и выходного файлов из аргументов командной строки
* Считывает дерево из текстового файла
* Создает переменную реализуемого типа и выводит в выходной текстовый файл дерево
* Вызывает функции добавления, удаления и поиска элемента, результаты работы функций печатаются в выходной файл
* Вызывает функцию освобождения памяти

**2.2 Тестирование программы**

Тесты программы описаны в файле test.c.

При успешном прохождении очередного теста в консоль выводится сообщение о том, что тест пройден, в противном случае, что он не пройден.

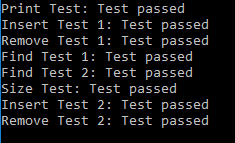


Рис.1.

**3. Заключение**

В ходе курсовой работы был реализован абстрактный тип данных -

красно-чёрное дерево. Все требования к

работе были выполнены: предусмотрено динамическое выделение памяти и функция деинициализации, разработанный исходный код компилируется gcc без ошибок, сборка библиотеки, демонстрационной программы и модульных тестов осуществляется утилитой make.

**4. Список используемых источников**

1. Ахо, Альфред В., Лам, Моника С, Сети, Рави, Ульман, Джеффри Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.
2. Ахо, Альфред, В., Хопкрофт, Джон, Ульман, Джеффри, Д. Структуры данных и алгоритмы.
3. <https://learnc.info/c/memory_allocation.html>
4. <https://prog-cpp.ru/c-files/>

**Приложение 1. Код программы**

**main.c**

#include <stdio.h>

#include <stdlib.h>

#include "rb\_tree.h"

int main1(int argc, char \*argv[]) {

if (argc != 3) {

printf("Укажите входной и выходной файлы");

return 0;

}

FILE \*fclear;

fclear = fopen(argv[2], "w");

fclose(fclear);

struct rb\_tree \*tree = read\_from\_file(argv[1]);

print\_tree\_to\_file(tree->root, argv[2],"a","Input file:\n");

rb\_tree\_insert(tree, 30);

print\_tree\_to\_file(tree->root, argv[2], "a", "Adding 30:\n");

rb\_tree\_remove(tree,6);

print\_tree\_to\_file(tree->root, argv[2], "a","Removing 6:\n");

FILE \*fout;

fout = fopen(argv[2], "a");

fprintf(fout,"Finding 45:");

if (rb\_tree\_find(tree, 45)) {

fprintf(fout, "Found!\n");

}

else {

fprintf(fout, "No...!\n");

}

fprintf(fout, "Finding 121:");

if (rb\_tree\_find(tree, 121)) {

fprintf(fout, "Found!\n");

}

else {

fprintf(fout, "No...\n");

}

}

**test.c**

#include <stdio.h>

#include <stdlib.h>

#include "rb\_tree.h"

void assertEquals(char \*actual, char \*expected) {

FILE \*file1, \*file2;

char c1, c2;

file1 = fopen(actual, "r");

file2 = fopen(expected, "r");

if (file1 == NULL || file2 == NULL) {

printf("Cannot open file");

}

else {

c1 = (char)fgetc(file1);

c2 = (char)fgetc(file2);

while (c1 != EOF && c2 != EOF) {

if (c1 != c2) {

printf("Test failed\n");

}

c1 = (char)fgetc(file1);

c2 = (char)fgetc(file2);

}

}

if (c1 == EOF && c2 != EOF && c2 != '\n') {

printf("Test failed 1\n");

}

if (c2 == EOF && c1 != EOF && c1 != '\n') {

printf("Test failed 2\n");

}

printf("Test passed\n");

fclose(file1);

fclose(file2);

}

void printTreeTest(struct rb\_tree \*self) {

print\_tree\_to\_file(self->root,"test\_print\_fact.txt","w","");

printf("Print Test: ");

assertEquals("test\_print\_fact.txt", "test\_print\_exp.txt");

}

void insertTest\_1(struct rb\_tree \*self) {

rb\_tree\_insert(self,5);

print\_tree\_to\_file(self->root, "test\_insert\_fact\_1.txt", "w","");

printf("Insert Test 1: ");

assertEquals("test\_insert\_fact\_1.txt", "test\_insert\_exp\_1.txt");

}

void insertTest\_2(struct rb\_tree \*self) {

struct rb\_node \*nodeTest = rb\_node\_init(4);

nodeTest->red = 1;

rb\_tree\_insert\_node(self, nodeTest);

print\_tree\_to\_file(self->root, "test\_insert\_fact\_2.txt", "w", "");

printf("Insert Test 2: ");

assertEquals("test\_insert\_fact\_2.txt", "test\_insert\_exp\_2.txt");

}

void findTest\_1(struct rb\_tree \*self) {

printf("Find Test 1: ");

if (rb\_tree\_find(self, 5)) {

printf("Test passed\n");

}

else

{

printf("Test failed\n");

}

}

void findTest\_2(struct rb\_tree \*self) {

printf("Find Test 2: ");

if (rb\_tree\_find(self, 22)) {

printf("Test failed\n");

}

else

{

printf("Test passed\n");

}

}

void removeTest\_1(struct rb\_tree \*self) {

rb\_tree\_remove(self,0);

print\_tree\_to\_file(self->root, "test\_remove\_fact\_1.txt", "w", "");

printf("Remove Test 1: ");

assertEquals("test\_remove\_fact\_1.txt", "test\_remove\_exp\_1.txt");

}

void removeTest\_2(struct rb\_tree \*self) {

rb\_tree\_remove(self, 4);

print\_tree\_to\_file(self->root, "test\_remove\_fact\_2.txt", "w","");

printf("Remove Test 2: ");

assertEquals("test\_remove\_fact\_2.txt", "test\_remove\_exp\_2.txt");

}

void sizeTest(struct rb\_tree \*self) {

printf("Size Test: ");

if (rb\_tree\_size(self)==6) {

printf("Test passed\n");

}

else

{

printf("Test failed\n");

}

}

int main(int argc, char \*argv[]) {

struct rb\_tree \*tree = rb\_tree\_init(NULL);

rb\_tree\_insert(tree, 10);

rb\_tree\_insert(tree, 6);

rb\_tree\_insert(tree, 45);

rb\_tree\_insert(tree, 4);

rb\_tree\_insert(tree, 8);

rb\_tree\_insert(tree, 0);

printTreeTest(tree);

insertTest\_1(tree);

removeTest\_1(tree);

findTest\_1(tree);

findTest\_2(tree);

sizeTest(tree);

rb\_tree\_dealloc(tree);

struct rb\_tree \*tree2 = rb\_tree\_init(NULL);

struct rb\_node \*node1 = rb\_node\_init(10);

node1->red = 0;

struct rb\_node \*node2 = rb\_node\_init(6);

node2->red = 1;

struct rb\_node \*node3 = rb\_node\_init(45);

node3->red = 0;

rb\_tree\_insert\_node(tree2, node1);

rb\_tree\_insert\_node(tree2, node2);

rb\_tree\_insert\_node(tree2, node3);

insertTest\_2(tree2);

rb\_tree\_dealloc(tree2);

struct rb\_tree \*tree3 = rb\_tree\_init(NULL);

struct rb\_node \*node4 = rb\_node\_init(3);

node4->red = 0;

struct rb\_node \*node5 = rb\_node\_init(2);

node5->red = 0;

struct rb\_node \*node6 = rb\_node\_init(4);

node6->red = 0;

struct rb\_node \*node7 = rb\_node\_init(1);

node7->red = 1;

rb\_tree\_insert\_node(tree3, node4);

rb\_tree\_insert\_node(tree3, node5);

rb\_tree\_insert\_node(tree3, node6);

rb\_tree\_insert\_node(tree3, node7);

removeTest\_2(tree3);

rb\_tree\_dealloc(tree3);

scanf("%d");

return 0;

}

**rb\_tree.c**

#include <stdio.h>

#include <string.h>

#include "rb\_tree.h"

int pow2 = 1;

int counter = 0;

struct rb\_node \*rb\_node\_init (void \*value) {

struct rb\_node \*self = malloc(sizeof(struct rb\_node));

if (self) {

self->red = 1;

self->link[0] = self->link[1] = NULL;

self->value = value;

}

return self;

}

void rb\_node\_dealloc (struct rb\_node \*self) {

if (self) free(self);

}

static int rb\_node\_is\_red (const struct rb\_node \*self) {

return self ? self->red : 0;

}

static struct rb\_node \*rb\_node\_rotate (struct rb\_node \*self, int dir) {

struct rb\_node \*result = NULL;

if (self) {

result = self->link[!dir];

self->link[!dir] = result->link[dir];

result->link[dir] = self;

self->red = 1;

result->red = 0;

}

return result;

}

static struct rb\_node \*rb\_node\_rotate2 (struct rb\_node \*self, int dir) {

struct rb\_node \*result = NULL;

if (self) {

self->link[!dir] = rb\_node\_rotate(self->link[!dir], !dir);

result = rb\_node\_rotate(self, dir);

}

return result;

}

int rb\_tree\_node\_compare (struct rb\_tree \*self, struct rb\_node \*a, struct rb\_node \*b) {

return (a->value > b->value) - (a->value < b->value);

}

struct rb\_tree \*rb\_tree\_init (rb\_tree\_comparator node\_cmp\_cb) {

struct rb\_tree \*self= malloc(sizeof(struct rb\_tree));

if (self) {

self->root = NULL;

self->size = 0;

self->cmp = node\_cmp\_cb ? node\_cmp\_cb : rb\_tree\_node\_compare;

}

return self;

}

void rb\_tree\_dealloc (struct rb\_tree \*self) {

if (self) free(self);

}

void \*rb\_tree\_find(struct rb\_tree \*self, void \*value) {

void \*result = NULL;

if (self) {

struct rb\_node node = { .value = value };

struct rb\_node \*it = self->root;

int cmp = 0;

while (it) {

if ((cmp = self->cmp(self, it, &node))) {

it = it->link[cmp < 0];

} else {

break;

}

}

result = it ? it->value : NULL;

}

return result;

}

int rb\_tree\_insert (struct rb\_tree \*self, void \*value) {

return rb\_tree\_insert\_node(self, rb\_node\_init(value));

}

int rb\_tree\_insert\_node (struct rb\_tree \*self, struct rb\_node \*node) {

int result = 0;

if (self && node) {

if (self->root == NULL) {

node->red = 0;

self->root = node;

result = 1;

} else {

struct rb\_node head = { 0 };

struct rb\_node \*g, \*t;

struct rb\_node \*p, \*q;

int dir = 0, last = 0;

t = &head;

g = p = NULL;

q = t->link[1] = self->root;

while (1) {

if (q == NULL) {

p->link[dir] = q = node;

}else if (rb\_node\_is\_red(q->link[0]) && rb\_node\_is\_red(q->link[1])) {

q->red = 1;

q->link[0]->red = 0;

q->link[1]->red = 0;

}

if (rb\_node\_is\_red(q) && rb\_node\_is\_red(p)) {

int dir2 = t->link[1] == g;

if (q == p->link[last]) {

t->link[dir2] = rb\_node\_rotate(g, !last);

} else {

t->link[dir2] = rb\_node\_rotate2(g, !last);

}

}

if (self->cmp(self, q, node) == 0) {

break;

}

last = dir;

dir = self->cmp(self, q, node) < 0;

if (g != NULL) {

t = g;

}

g = p, p = q;

q = q->link[dir];

}

self->root = head.link[1];

}

self->root->red = 0;

++self->size;

}

return result;

}

int rb\_tree\_remove (struct rb\_tree \*self, void \*value) {

if (self->root != NULL) {

struct rb\_node head = {0};

struct rb\_node node = { .value = value };

struct rb\_node \*q, \*p, \*g;

struct rb\_node \*f = NULL;

int dir = 1;

q = &head;

g = p = NULL;

q->link[1] = self->root;

while (q->link[dir] != NULL) {

int last = dir;

g = p, p = q;

q = q->link[dir];

dir = self->cmp(self, q, &node) < 0;

if (self->cmp(self, q, &node) == 0) {

f = q;

}

if (!rb\_node\_is\_red(q) && !rb\_node\_is\_red(q->link[dir])) {

if (rb\_node\_is\_red(q->link[!dir])) {

p = p->link[last] = rb\_node\_rotate(q, dir);

} else if (!rb\_node\_is\_red(q->link[!dir])) {

struct rb\_node \*s = p->link[!last];

if (s) {

if (!rb\_node\_is\_red(s->link[!last]) && !rb\_node\_is\_red(s->link[last])) {

p->red = 0;

s->red = 1;

q->red = 1;

} else {

int dir2 = g->link[1] == p;

if (rb\_node\_is\_red(s->link[last])) {

g->link[dir2] = rb\_node\_rotate2(p, last);

} else if (rb\_node\_is\_red(s->link[!last])) {

g->link[dir2] = rb\_node\_rotate(p, last);

}

q->red = g->link[dir2]->red = 1;

g->link[dir2]->link[0]->red = 0;

g->link[dir2]->link[1]->red = 0;

}

}

}

}

}

if (f) {

void \*tmp = f->value;

f->value = q->value;

q->value = tmp;

p->link[p->link[1] == q] = q->link[q->link[0] == NULL];

if (self) {

if (q) {

rb\_node\_dealloc(q);

}

}

q = NULL;

}

else {

return 0;

}

self->root = head.link[1];

if (self->root != NULL) {

self->root->red = 0;

}

--self->size;

}

return 1;

}

void print\_tree\_to\_file(struct rb\_node \*root, char \*filename,char \*mode,char \*msg)

{

pow2 = 1;

counter = 0;

FILE \*fout;

fout = fopen(filename, mode);

if (fout == NULL) {

printf("Cannot open file");

exit(0);

}

fprintf(fout, "%s",msg);

int n = 0;

while (print\_layer(root, n,fout)) ++n;

fprintf(fout, "\n\n");

fclose(fout);

}

static int print\_layer(struct rb\_node \*root, int n, FILE \*fout)

{

while (1)

{

if (!root) {

fprintf(fout, "null ");

counter++;

if (counter == pow2) {

fprintf(fout, "\n");

counter = 0;

pow2 = pow2 \* 2;

}

return 0;

}

if (n == 0)

{

fprintf(fout, "%d(%d) ", root->value,root->red);

counter++;

if (counter == pow2) {

fprintf(fout, "\n");

counter = 0;

pow2 = pow2 \* 2;

}

return root->link[0] || root->link[1];

}

int l = print\_layer(root->link[0], n - 1, fout);

int r = print\_layer(root->link[1], n - 1, fout);

return l || r;

}

}

struct rb\_tree \*read\_from\_file(char \*filename) {

FILE \*fin;

fin = fopen(filename, "r");

if (fin == NULL) {

printf("Cannot open file ");

printf(filename);

exit(0);

}

struct rb\_tree \*res = rb\_tree\_init(NULL);

char c;

char str[128];

for (int i = 0; i < 128; i++) {

str[i] = NULL;

}

while ((c = (char)getc(fin)) != EOF) {

if (c == ' '||c=='\n') {

if (str[0] == ' '|| str[0] == '\n'||str[0]==NULL) {

memset(str, 0, 128);

continue;

}

if (str[0] != 'n'||str[1] != 'u'||str[2] != 'l'||str[3] != 'l') {

int result = 0;

int r\_or\_b = -1;

for (int i = 0; i < 128; i++) {

if (str[i] == NULL) break;

if (str[i] == '(') {

r\_or\_b = str[i+1]%48;

break;

}

result = result \* 10;

result = result + (int)str[i]%48;

}

struct rb\_node \*resNode = rb\_node\_init(&result);

resNode->red = r\_or\_b;

rb\_tree\_insert\_node(res, resNode);

}

memset(str, 0, 128);

continue;

}

for (int i = 0; i < 128; i++) {

if (str[i] == NULL) {

str[i] = c;

break;

}

}

}

fclose(fin);

return res;

}

size\_t rb\_tree\_size (struct rb\_tree \*self) {

size\_t result = 0;

if (self) {

result = self->size;

}

return result;

}

**rb\_tree.h**

#ifndef RB\_TREE\_H\_

#define RB\_TREE\_H\_ 1

#include <stdio.h>

#include <stdint.h>

#include <stddef.h>

#include <stdlib.h>

struct rb\_node;

struct rb\_tree;

typedef int(\*rb\_tree\_comparator)(struct rb\_tree \*self, struct rb\_node \*a, struct rb\_node \*b);

struct rb\_node {

int red;

struct rb\_node \*link[2];

void \*value;

};

struct rb\_tree {

struct rb\_node \*root;

rb\_tree\_comparator cmp;

size\_t size;

};

int rb\_tree\_node\_compare(struct rb\_tree \*self, struct rb\_node \*a, struct rb\_node \*b);

struct rb\_node \*rb\_node\_init(void \*value);

void rb\_node\_dealloc(struct rb\_node \*self);

struct rb\_tree \*rb\_tree\_init(rb\_tree\_comparator cmp);

void rb\_tree\_dealloc(struct rb\_tree \*self);

void \*rb\_tree\_find(struct rb\_tree \*self, void \*value);

int rb\_tree\_insert(struct rb\_tree \*self, void \*value);

int rb\_tree\_remove(struct rb\_tree \*self, void \*value);

size\_t rb\_tree\_size(struct rb\_tree \*self);

void print\_tree\_to\_file(struct rb\_node \*root, char \*filename, char \*mode, char \*msg);

int rb\_tree\_insert\_node(struct rb\_tree \*self, struct rb\_node \*node);

struct rb\_tree \*read\_from\_file(char \*filename);

#endif