

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»
Тема: Красно-черное дерево

Студент гр. 9381

Авдеев Илья

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить структуру БДП: красно-чёрное дерево.

Задание

Вариант 27

По заданной последовательности элементов Elem построить структуру данных. Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если входит, то в скольких экземплярах. Добавить элемент e в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

Основные теоретические положения.

Красно-черное дерево - это еще одна форма сбалансированного бинарного поискового дерева. Впервые оно было представлено в 1972 году как еще одна разновидность сбалансированного бинарного дерева. Время поиска, вставки или удаления узла для красно-черного дерева является логарифмической функцией от числа узлов. Благодаря этим ограничениям, путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего и дерево примерно сбалансировано. Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска. Пусть для красно-чёрного дерева T число чёрных узлов от корня до листа равно B . Тогда кратчайший возможный путь до любого листа содержит B узлов и все они чёрные. Более длинный возможный путь может быть построен путём включения красных узлов. Однако, благодаря п.4 в дереве не может быть двух красных узлов подряд, а согласно пп. 2 и 3, путь начинается и кончается чёрным узлом. Поэтому самый длинный возможный путь состоит из $2B-1$ узлов, попеременно красных и чёрных.

1. ОПИСАНИЕ АЛГОРИТМА

Сначала проводится проверка наличия элемента структуры. Если ее нет, то функция возвращает 0.

Создается возвращаемое значение *res* типа *int* равное 0;

В цикле проводится сравнение содержимого узла с элементом *value*, если они равны к *i* прибавляется один. Если содержимое больше узла рассмотрим левое плечо, иначе правое.

2. ОПИСАНИЕ СТРУКТУР ДАННЫХ И ФУНКЦИЙ

Для решения поставленной задачи был написан класс *RBtree* для хранения БДП красно-черное дерева.

Результаты тестирования см. в приложении А.

Разработанный программный код см. в приложении Б.

2.1. Класс *RBtree*

Класс предоставляет функционал для хранения, построения, удаления и вывода БДП красно-черного дерева. Поля и методы класса приведены в таблицах 1 и 2.

Таблица 1 - Поля класса *RBtree*

Модификатор доступа	Тип и название поля	Предназначение	Значение по умолчанию
<i>private</i>	<i>Struct node</i>	Типа данных узла. Поля и методы приведены в таблице 3	-
<i>private</i>	<i>node* tree root</i>	Указатель на корень дерева	<i>NULL</i>
<i>private</i>	<i>int size</i>	Хранит число узлов дерева	<i>0</i>

Таблица 2 - Методы класса *RBtree*

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
<i>private</i>	<i>node*</i>	<i>make_node(int value)</i>
<i>private</i>	<i>void</i>	<i>del_node(node*)</i>
<i>private</i>	<i>void</i>	<i>clear(node*)</i>
<i>private</i>	<i>node*</i>	<i>rotate_right(node*)</i>
<i>private</i>	<i>node*</i>	<i>rotate_left(node*)</i>

<i>private</i>	<i>void</i>	<i>balance_insert(node**)</i>
<i>private</i>	<i>bool</i>	<i>balance_remove_case1(node**)</i>
<i>private</i>	<i>bool</i>	<i>balance_remove_case2(node**)</i>
<i>private</i>	<i>bool</i>	<i>insert(int, node**)</i>
<i>private</i>	<i>bool</i>	<i>getmin(node**, node**)</i>
<i>private</i>	<i>bool</i>	<i>remove(node**, int)</i>
<i>private</i>	<i>bool</i>	<i>print_tree(node*& ptr, int u)</i>
<i>public</i>	<i>void</i>	<i>clear()</i>
<i>public</i>	<i>void</i>	<i>insert(int)</i>
<i>public</i>	<i>void</i>	<i>remove(int);</i>
<i>public</i>	<i>void</i>	<i>print()</i>
<i>public</i>	<i>int</i>	<i>getsize()</i>
<i>public</i>		<i>RBtree()</i>
<i>public</i>		<i>~RBtree()</i>

Метод Rbtree::RBtree.

Конструктор. Ничего не принимает. Инициализирует корень дерева равным листом.

Метод Rbtree::~~RBtree.

Деструктор. Ничего не принимает. Освобождает память от дерева с помощью метода *clear(node*)*.

Метод Rbtree::make_node.

Принимает value - значение. Создает узел с этим значением и красит его в красный цвет. Возвращает созданный узел.

Метод RBtree::del_node.

Принимает node* - узел дерева и удаляет его.

Метод Rbtree::clear.

Является рекурсивным методом. Принимает на вход *root — корень дерева, которое нужно удалить. Удаляет дерево целиком с помощью метода *del_node*.

Memod RBtree::rotate_right.

Принимает на вход `node*` - узел. Осуществляет поворот поддеревы этого узла вправо. Возвращает родительский узел.

Memod RBtree::rotate_left.

Принимает на вход `node*` - узел. Осуществляет поворот поддеревы этого узла вправо. Возвращает родительский узел.

Memod RBtree::balance_insert.

Является рекурсивным методом. Принимает на вход `node**` - узел. Производит поиск ситуаций балансировки дерева после добавления нового узла.

Memod Rbtree::balance_remove_case1.

Является рекурсивным методом. Принимает на вход `node**` - узел. Производит поиск ситуаций балансировки дерева после удаления левого узла. Возвращает `true` если нужен баланс.

Memod Rbtree::balance_remove_case2.

Является рекурсивным методом. Принимает на вход `node**` - узел. Производит поиск ситуаций балансировки дерева после удаления правого узла. Возвращает `true` если нужен баланс.

Memod RBtree::insert.

Является рекурсивным методом. Принимает на вход `value` — значение и `node*` - узел. Находит место для вставки узла со значением `value`. При помощи метода `make_node` создает узел и с помощью метода `balance_insert` производит балансировку дерева. Возвращает `true` если нужен баланс.

Memod Rbtree::getmin.

Является рекурсивным методом. Принимает на вход *****root*** — корень дерева и ****res*** — узел который был удален. Удаляет узел с максимальным значением. Возвращает true если нужен баланс.

Memod Rbtree::remove.

Является рекурсивным методом. Принимает на вход ****root*** — корень дерева в котором нужно удалить узел со значением и ***value*** — значение, узел с которым нужно удалить. Вызывая себя находит узел, с помощью метода *del_node* удаляет его, возвращает true если нужен баланс и с помощью методов *balance_remove_case1* или *balance_remove_case2* балансирует дерево.

Memod Rbtree::print_tree.

Является рекурсивным методом. Принимает на вход *****ptr*** — корень дерева и ***u*** — глубина на котором сейчас обход. Выводит в консоль дерево.

Memod Rbtree::clear.

Ничего не принимает. Вспомогательный метод для запуска метода *clear(node*)*.

Memod Rbtree::remove.

Принимает на вход ***value*** — значение, узел с которым нужно удалить. Вспомогательный метод для запуска метода *remove(node*, int)*.

Memod Rbtree::print.

Ничего не принимает. Вспомогательный метод для запуска метода *print_tree(node**, int)*.

Memod Rbtree::getsize.

Ничего не принимает. Возвращает количество дерева.

2.2. Структура Node

Тип данных элемента списка. Поля структуры приведены в таблице 3.

Таблица 3 - Поля структуры node

Модификатор доступа	Тип и название поля	Предназначение	Значение по умолчанию
<i>public</i>	<i>struct node* left</i>	Указатель на левого потомка	<i>NULL</i>
<i>public</i>	<i>struct node* right</i>	Указатель на правого потомка	<i>NULL</i>
<i>public</i>	<i>int value</i>	Значение узла	-
<i>public</i>	<i>bool col</i>	Цвет узла	1

2.3. Функция main

Для начала объявляются следующие переменные:

- *n* — вводимое значение для записи в дерево;
- *tree* — Хранит дерево;
- *cin* – переменная для проверки ввода пользователя
- *c* — переменная хранящая выбор пользователя
- *exit* – переменная для проверки условия выхода из программы

Производится настройка русского языка для консоли.

Далее происходит вход в цикл выбора способа ввода. Считывается выбранное пользователем действие (цифра от 1 до 3). Если пользователь выбрал создать из случайных чисел дерево, программа входит в цикл ввода размера дерева. Затем программа входит в первый основной цикл программы. В нем, если пользователь выбрал консольный ввод, он будет вводить числа, из которых создается дерево. Если он выбрал случайное дерево, то в цикле от 0 до введенного размера, сгенерируются числа из которых будет состоять дерево. Если из файла, то дерево введется из текстового документа lin.txt.

Затем входит в цикл в основной цикл программы, где пользователю будет предложена узнать есть ли определенный элемент в дереве. Для этого он должен вводить интересующий его элемент, и если его нет, программа предложит ввести его в дерево.

Вывод

В ходе работы над поставленным заданием была изучена такая структура данных, как БДП красно-черное дерево, был разработан класс, включающий в себя методы работы с БДП красно-черное дерево, также была реализована функция проверки наличия введенного элемента в дереве. Программа была успешно протестирована на работоспособность.

ПРИЛОЖЕНИЕ А

ТЕСТИРОВАНИЕ

Таблица А.1 - Примеры тестовых случаев на некорректных данных

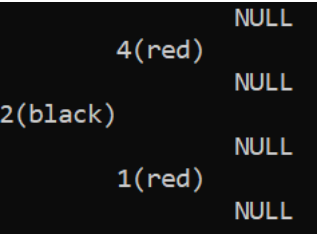
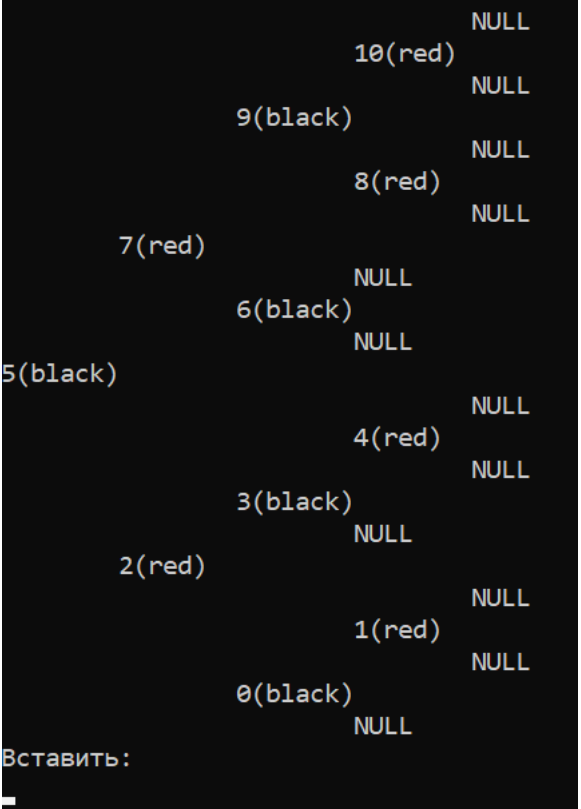
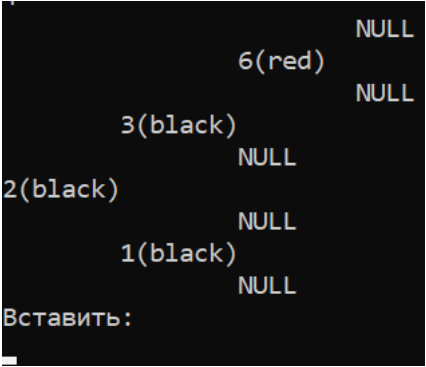
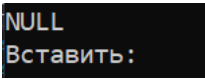
№ п/п	Входные данные	Выходные данные
1.	f	Нужно ввести число, попробуй еще
2.	9	Нужно ввести 1 или 2, чтобы выйти из программы нужно нажать esc
3.	2 2 4	 <pre> 4(red) NULL 2(black) NULL 1(red) NULL </pre>
4.	1 f	Нужно ввести число, попробуй еще
5.	3	 <pre> 10(red) NULL 9(black) NULL 8(red) NULL 7(red) NULL 6(black) NULL 5(black) NULL 4(red) NULL 3(black) NULL 2(red) NULL 1(red) NULL 0(black) NULL Вставить: </pre>

Таблица А.2 - Примеры тестовых случаев на корректных данных

№ п/п	Входные данные	Выходные данные
6.	2 4 0 y	 <p>Выход из программы</p>
7.	8 esc	Выход из программы
8.	2 200	Построено дерево из 200 элементов, заполненное случайными числами от 0 до 400
9.	-1	Нужно ввести число, попробуй еще
10.	1 0	 <p>Пустое дерево</p>

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <time.h>
#include <fstream>
#include <conio.h>

class RBtree
{
    struct node
    {
        node* left, * right;
        int value;
        bool red;
    };
    node* tree_root;
    int size;
private:
    node* make_node(int value);
    void del_node(node*);
    void clear(node*);
    node* rotate_right(node*);
    node* rotate_left(node*);
    void balance_insert(node**);
    bool balance_remove_case1(node**);
    bool balance_remove_case2(node**);
    bool insert(int, node**);
    bool getmin(node**, node**);
    bool remove(node**, int);
    void print_tree(node*& ptr, int u);
public:
    RBtree();
    ~RBtree();
    void clear();
    int find(int);
    void insert(int);
    void remove(int);
    void print();
    int getsize() { return size; }
};
```

```

RBtree::RBtree()
{
    tree_root = 0;
    size = 0;
}

RBtree::~~RBtree()
{
    clear(tree_root);
}

RBtree::node* RBtree::make_node(int value)
{
    size++;
    node* n = new node;
    n->value = value;
    n->left = n->right = NULL;
    n->red = true;
    return n;
}

void RBtree::del_node(node* node)
{
    size--;
    delete node;
}

void RBtree::clear(node* node)
{
    if (!node) return;
    clear(node->left);
    clear(node->right);
    del_node(node);
}

RBtree::node* RBtree::rotate_left(node* n)
{
    node* right = n->right;
    node* rleft = right->left;
    right->left = n;

```

```

        n->right = rleft;
        return right;
    }

RBtree::node* RBtree::rotate_right(node* n)
{
    node* left = n->left;
    node* lright = left->right;
    left->right = n;
    n->left = lright;
    return left;
}

void RBtree::balance_insert(node** root)
{
    node* left, * right, * lleft, * lright;
    node* node = *root;
    if (node->red) return;
    left = node->left;
    right = node->right;
    if (left && left->red)
    {
        lright = left->right;
        if (lright && lright->red)
            left = node->left = rotate_left(left);
        lleft = left->left;
        if (lleft && lleft->red)
        {
            node->red = true;
            left->red = false;
            if (right && right->red)
            {
                lleft->red = true;
                right->red = false;
                return;
            }
        }
        *root = rotate_right(node);
        return;
    }
}

```

```

    if (right && right->red)
    {
        lleft = right->left;
        if (lleft && lleft->red)
            right = node->right = rotate_right(right);
        lright = right->right;
        if (lright && lright->red)
        {
            node->red = true;
            right->red = false;
            if (left && left->red)
            {
                lright->red = true;
                left->red = false;
                return;
            }
            *root = rotate_left(node);
            return;
        }
    }
}

bool RBtree::balance_remove_case1(node** root)
{
    node* n = *root;
    node* left = n->left;
    node* right = n->right;
    if (right && left->red)
    {
        left->red = false; return false;
    }
    if (right && right->red)
    {
        n->red = true;
        right->red = false;
        n = *root = rotate_left(n);
        if (balance_remove_case1(&n->left)) n->left->red = false;
        return false;
    }
    unsigned int mask = 0;
    node* rleft = right->left;
    node* rright = right->right;

```

```

    if (rleft && rleft->red) mask |= 1;
    if (rright && rright->red) mask |= 2;
    switch (mask)
    {
    case 0:
        right->red = true;
        return true;
    case 1:
    case 3:
        right->red = true;
        rright->red = false;
        right = n->right = rotate_right(right);
        rright = right->right;
    case 2:
        right->red = n->red;
        rright->red = n->red = false;
        *root = rotate_left(n);
    }
    return false;
}

```

```

bool RBtree::balance_remove_case2(node** root)
{
    node* n = *root;
    node* left = n->left;
    node* right = n->right;
    if (right && right->red) { right->red = false; return false; }
    if (left && left->red)
    {
        n->red = true;
        left->red = false;
        n = *root = rotate_right(n);
        if (balance_remove_case2(&n->right)) n->right->red = false;
        return false;
    }
    unsigned int mask = 0;
    node* lleft = left->left;
    node* lright = left->right;
    if (lleft && lleft->red) mask |= 1;
    if (lright && lright->red) mask |= 2;
    switch (mask)
    {

```

```

        case 0:
            left->red = true;
            return true;
        case 2:
        case 3:
            left->red = true;
            lright->red = false;
            left = n->left = rotate_left(left);
            lleft = left->left;
        case 1:
            left->red = n->red;
            lleft->red = n->red = false;
            *root = rotate_right(n);
    }
    return false;
}

int RBtree::find(int value)
{
    node* n = tree_root;
    int i = 0;
    while (n)
    {
        if (n->value == value)
            i++;
        n = n->value > value ? n->left : n->right;
    }
    return i;
}

bool RBtree::insert(int value, node** root)
{
    node* n = *root;
    if (!n) *root = make_node(value);
    else
    {
        if (value == n->value) return true;
        if (insert(value, value < n->value ? &n->left : &n->right))
            return true;
        balance_insert(root);
    }
}

```



```

        return false;
    }

    bool RBtree::getmin(node** root, node** res)
    {
        node* node = *root;
        if (node->left)
        {
            if (getmin(&node->left, res)) return
balance_remove_case1(root);
        }
        else
        {
            *root = node->right;
            *res = node;
            return !node->red;
        }
        return false;
    }

    bool RBtree::remove(node** root, int value)
    {
        node* t, * node = *root;
        if (!node) return false;
        if (node->value < value)
        {
            if (remove(&node->right, value)) return
balance_remove_case2(root);
        }
        else if (node->value > value)
        {
            if (remove(&node->left, value)) return
balance_remove_case1(root);
        }
        else
        {
            bool res;
            if (!node->right)
            {
                *root = node->left;
                res = !node->red;
            }

```

```

        }
        else
        {
            res = getmin(&node->right, root);
            t = *root;
            t->red = node->red;
            t->left = node->left;
            t->right = node->right;
            if (res) res = balance_remove_case2(root);
        }
        del_node(node);
        return res;
    }
    return 0;
}

void RBtree::insert(int value)
{
    insert(value, &tree_root);
    if (tree_root) tree_root->red = false;
}

void RBtree::remove(int value)
{
    remove(&tree_root, value);
}

void RBtree::clear()
{
    clear(tree_root);
    tree_root = 0;
}

void RBtree::print_tree(node*& ptr, int u)
{
    if (ptr == nullptr)
    {
        u++;
        for (int i = 0; i < u - 1; ++i) std::cout << "\t";
        std::cout << "NULL\n";
        return;
    }

```

```

    }
    else
    {
        print_tree(ptr->right, ++u);
        for (int i = 0; i < u - 1; ++i) std::cout << "\t";
        std::cout << ptr->value << "(";
        if (ptr->red)
            std::cout << "red";
        else
            std::cout << "black";
        std::cout << ")\n";
        u--;
    }
    print_tree(ptr->left, ++u);
}

void RBtree::print()
{
    print_tree(tree_root, 0);
}

int main()
{
    int n = 1, c, j = 20;
    char cin[10], exit = 0;
    RBtree tree;
    setlocale(LC_ALL, "ru");

    std::cout << "Способ ввода из консоли 1, заполнить дерево случайными
числами 2, ввести из файла 3\n";
    std::cin >> cin;

    while (exit != 27)
    {
        if (isdigit(*cin))
        {
            c = atoi(cin);
            if ((c != 1) && (c != 2) && (c != 3))
            {
                std::cout << "нужно ввести 1 или 2 или 3, чтобы
выйти из программы нужно нажать esc\n";

```

```

        exit = _getch();
        if (exit == 27)
            break;
    }
    else
        break;
    std::cout << "Способ ввода из консоли 1, Заполнить
дерево случайными числами 2\n";
    std::cin >> cin;
    continue;
}
else
{
    std::cout << "Нужно ввести число, попробуйте еще\n";
    std::cin >> cin;
}
}

if (exit != 27)
    while (1)
    {
        if (c == 2)
        {
            std::cout << "сколько узлов создать?\n";
            std::cin >> cin;
            if (isdigit(*cin))
            {
                j = atoi(cin);
                break;
            }
            else
                std::cout << "Нужно ввести число, попробуйте
еще\n";
        }
        else
            break;
    }

    std::ifstream lin("lin.txt");
    if (c == 3)
        if (!lin.is_open())

```

```

        {
            std::cout << "Файл не открыт";
            return -1;
        }

srand(time(0));
while (n)
{
    switch (c)
    {
    case 1:
        std::cin >> cin;
        if (isdigit(*cin))
        {
            n = atoi(cin);
        }
        else
        {
            std::cout << "Нужно ввести число, попробуйте еще\
n";

            continue;
        }
        while (n != NULL)
        {
            tree.insert(n);
            std::cin >> cin;
            if (isdigit(*cin))
            {
                n = atoi(cin);
            }
            else
            {
                std::cout << "Нужно ввести число, попробуйте
еще\n";

                continue;
            }
        }
        break;
    case 2:
        while (tree.getsize() < j)
        {
            n = rand() % j + rand() % j;

```

```

        tree.insert(n);
    }
    n = 0;
    break;
case 3:
    while (!lin.eof())
    {
        lin >> n;
        tree.insert(n);
    }

default:
    n = 0;
    break;
}
}
n = 1;
if (exit != 27)
{

while (n)
{
    tree.print();
    std::cout << "Найти: \n";
    std::cin >> cin;
    system("cls");
    if (isdigit(*cin))
    {
        n = atoi(cin);
    }
    else
    {
        std::cout << "Нужно ввести число, попробуйте еще\
n";

        continue;
    }
    if (n == 0)
    {
        std::cout << "Закончить или найти 0?\n Y - выйти N
= найти\n";

        while (exit != 'y' && exit != 'n')

```

```

        exit = _getch();
    }
    if (exit != 'y')
    {
        if (!tree.find(n))
        {
            std::cout << "Числа в дереве нет, ввести
его?\n Y - да N - нет\n";

            while (*cin != 'y' && *cin != 'n')
                *cin = _getch();
            if (*cin == 'y')
                tree.insert(n);
        }
        else
            std::cout << "Число в дереве есть\n";
    }

    if (exit == 'n')
        n = 1;
    exit = 0;
}

}

lin.close();

getchar();
return 0;
}

```