

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

Кафедра информатики и программирования

АНАЛИЗ КРАСНО-ЧЁРНОГО ДЕРЕВА
ОТЧЕТ

студентки 2 курса 211 группы
направления 02.03.02 — Фундаментальная информатика и информационные
технологии
факультета компьютерных наук и информационных технологий
Никитенко Яны Валерьевны

СОДЕРЖАНИЕ

| | | |
|---|-------------------------|----|
| 1 | Текст программы | 3 |
| 2 | Операция вставки | 20 |
| 3 | Операция удаления | 21 |
| 4 | Операция поиска | 22 |
| 5 | Обходы дерева..... | 23 |
| 6 | Расход памяти | 24 |

1 Текст программы

```
#define RED true
#define BLACK false

enum RBTCOLOR { Black, Red };

// Структура для дерева. Определена как шаблон функции
template<class KeyType>
struct RBTNode {
    KeyType key;
    RBTCOLOR color;
    RBTNode<KeyType>* left;
    RBTNode<KeyType>* right;
    RBTNode<KeyType>* parent;
    RBTNode(KeyType k, RBTCOLOR c, RBTNode* p, RBTNode* l, RBTNode* r) :
        key(k), color(c), parent(p), left(l), right(r) { };
};
//

// Класс для дерева. Шаблон. Тут почти список функций
template<class T>
class RBTree {
public:
    RBTree();
    ~RBTree();

    void insert(T key);
    void remove(T key);
    RBTNode<T>* search(T key);
    void print();
    void preOrder();
    void inOrder();
    void postOrder();
};
```

private:

```
void leftRotate(RBTreeNode<T>*& root, RBTreeNode<T>* x);
void rightRotate(RBTreeNode<T>*& root, RBTreeNode<T>* y);
void insert(RBTreeNode<T>*& root, RBTreeNode<T>* node);
void InsertFixUp(RBTreeNode<T>*& root, RBTreeNode<T>* node);
void destroy(RBTreeNode<T>*& node);
void remove(RBTreeNode<T>*& root, RBTreeNode<T>* node);
void removeFixUp(RBTreeNode<T>*& root, RBTreeNode<T>* node,
RBTreeNode<T>* parent);
RBTreeNode<T>* search(RBTreeNode<T>* node, T key) const;
void print(RBTreeNode<T>* node) const;
void preOrder(RBTreeNode<T>* tree) const;
void inOrder(RBTreeNode<T>* tree) const;
void postOrder(RBTreeNode<T>* tree) const;
```

```
RBTreeNode<T>* root;
```

```
HANDLE outp = GetStdHandle(STD_OUTPUT_HANDLE);
```

```
CONSOLE_SCREEN_BUFFER_INFO csbInfo;
```

```
void max_height(RBTreeNode<T>* x, short& max, short deepness = 1) {
    if (deepness > max) max = deepness;
    if (x->left) max_height(x->left, max, deepness + 1);
    if (x->right) max_height(x->right, max, deepness + 1);
}
```

```
// Измененная функция print_helper
```

```
void print_helper(RBTreeNode<T>* x, const COORD pos, const short offset) {
    SetConsoleTextAttribute(outp, x->color == RED ? 12 : 8);
    SetConsoleCursorPosition(outp, pos);
    cout << setw(offset + 1) << x->key;
    if (x->left) print_helper(x->left, { pos.X, short(pos.Y + 1) }, offset >> 1);
    if (x->right) print_helper(x->right, { short(pos.X + offset), short(pos.Y + 1) },
        offset >> 1);
}
```

```

bool isSizeOfConsoleCorrect(const short& width, const short& height) {
    GetConsoleScreenBufferInfo(outp, &csbInfo);
    COORD szOfConsole = csbInfo.dwSize;
    if (szOfConsole.X < width && szOfConsole.Y < height)
        cout << "Please increase the height and width of the terminal. ";
    else if (szOfConsole.X < width)
        cout << "Please increase the width of the terminal. ";
    else if (szOfConsole.Y < height)
        cout << "Please increase the height of the terminal. ";
    if (szOfConsole.X < width || szOfConsole.Y < height) {
        cout << "Size of your terminal now:
            " << szOfConsole.X << ' ' << szOfConsole.Y
                << ". Minimum required: " << width << ' ' << height << ".\n";
        return false;
    }
    return true;
}

};
//

// Конструктор
template<class T>
RBTree<T>::RBTree() : root(nullptr) {
    root = nullptr;
}
//

// Деструктор
template<class T>
RBTree<T>::~~RBTree() {
    destroy(root);
}
//

```

```

//
template<class T>
void RBTNode<T>::leftRotate(RBTNode<T>* & root, RBTNode<T>* x) {
    RBTNode<T>* y = x->right;
    x->right = y->left;
    if (y->left != NULL)
        y->left->parent = x;
    y->parent = x->parent;
    if (x->parent == NULL)
        root = y;
    else {
        if (x == x->parent->left)
            x->parent->left = y;
        else
            x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
};
//

//
template<class T>
void RBTNode<T>::rightRotate(RBTNode<T>* & root, RBTNode<T>* y) {
    RBTNode<T>* x = y->left;
    y->left = x->right;
    if (x->right != NULL)
        x->right->parent = y;
    x->parent = y->parent;
    if (y->parent == NULL)
        root = x;
    else {
        if (y == y->parent->right)

```

```

        y->parent->right = x;
    else
        y->parent->left = x;
    }
    x->right = y;
    y->parent = x;
};
//

// Публичный метод для вставки
template<class T>
void RBTNode<T>::insert(T key) {
    RBTNode<T>* z = new RBTNode<T>(key, Red, NULL, NULL, NULL);
    insert(root, z);
};
//

// Добавить узел. Основная функция
template<class T>
void RBTNode<T>::insert(RBTNode<T>*& root, RBTNode<T>* node) {
    RBTNode<T>* x = root;
    RBTNode<T>* y = NULL;
    while (x != NULL) {
        y = x;
        if (node->key > x->key)
            x = x->right;
        else
            x = x->left;
    }
    node->parent = y;
    if (y != NULL) {
        if (node->key > y->key)
            y->right = node;
        else

```

```

        y->left = node;
    }
    else
        root = node;
    node->color = Red;
    InsertFixUp(root, node);
};
//

// Восстановление дерева после вставки
template<class T>
void RBTNode<T>::InsertFixUp(RBTNode<T>*& root,
RBTNode<T>* node) {
    RBTNode<T>* parent;
    parent = node->parent;
    while (node != root && parent->color == Red) {
        RBTNode<T>* gparent = parent->parent;
        if (gparent->left == parent) {
            RBTNode<T>* uncle = gparent->right;
            if (uncle != NULL && 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 {
    RBTNode<T>* uncle = gparent->left;
    if (uncle != NULL && 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;
}
//

// Функция для рекурсивного удаления указанного узла
template<class T>
void RBTree<T>::destroy(RBTNode<T>*& node) {
    if (node == NULL)
        return;
    destroy(node->left);

```

```

    destroy(node->right);
    delete node;
    node = nullptr;
}
//

// Публичный метод для следующей функции
template<class T>
void RBTNode<T>::remove(T key) {
    RBTNode<T>* deletenode = search(root, key);
    if (deletenode != NULL)
        remove(root, deletenode);
}
//

// Удаление узла, и все все проблемы которые могут быть с этим связаны
template<class T>
void RBTNode<T>::remove(RBTNode<T>*& root, RBTNode<T>* node) {
    RBTNode<T>* child, * parent;
    RBTColor color;
    if (node->left != NULL && node->right != NULL) {
        RBTNode<T>* 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
        root = child;
    if (color == Black) {
        removeFixUp(root, child, parent);
    }
    delete node;
}
//

// Восстановление дерева после удаления узла
template<class T>
void RBTree<T>::removeFixUp(RBTreeNode<T>*& root,
RBTreeNode<T>* node, RBTreeNode<T>* parent) {
    RBTreeNode<T>* othernode;
    while ((!node) || node->color == Black && node != 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;
}
//

// Публичный метод для поиска
template<class T>
RBTreeNode<T>* RBTree<T>::search(T key) {
    return search(root, key);
}
//

// Функция для поиска узла
template<class T>
RBTreeNode<T>* RBTree<T>::search(RBTreeNode<T>* node, T key) const {
    if (node == NULL || node->key == key)
        return node;
    else
        if (key > node->key)
            return search(node->right, key);
        else
            return search(node->left, key);
}
//

// Функция для вывода
template<class T>
void RBTree<T>::print() {
    if (root == NULL)
        cout << "Пусто.\n";
    else {
        short max = 1;

```

```

    max_height(root, max);
    short width = 1 << max + 1, max_w = 128;
    if (width > max_w) width = max_w;
    while (!isSizeOfConsoleCorrect(width, max)) system("pause");
    for (short i = 0; i < max; ++i) cout << '\n';
    GetConsoleScreenBufferInfo(outp, &csbInfo);
    COORD endPos = csbInfo.dwCursorPosition;
    print_helper(root, { 0, short(endPos.Y - max) }, width >> 1);
    SetConsoleCursorPosition(outp, endPos);
    SetConsoleTextAttribute(outp, 7); // чтоб интерфейс не окрашивался
}
}
//

// Обход
template<class T>
void RBTNode<T>::preOrder() {
    if (root == NULL)
        cout << "Пусто.\n";
    else
        preOrder(root);
};
//

// Обход в прямом порядке
template<class T>
void RBTNode<T>::preOrder(RBTNode<T>* tree) const {
    if (tree != NULL) {
        cout << tree->key << (tree->color == RED ? 12 : 8);
        preOrder(tree->left);
        preOrder(tree->right);
    }
}
//

```

```

//
template<class T>
void RBTre<T>::inOrder() {
    if (root == NULL)
        cout << "Пусто.\n";
    else
        inOrder(root);
};
//

// Обход в симметричном порядке
template<class T>
void RBTre<T>::inOrder(RBTNode<T>* tree) const {
    if (tree != NULL) {
        inOrder(tree->left);
        cout << tree->key << (tree->color == tree->color == RED ? 12 : 8);
        inOrder(tree->right);
    }
}
//

//
template<class T>
void RBTre<T>::postOrder() {
    if (root == NULL)
        cout << "Пусто.\n";
    else
        postOrder(root);
};
//

// Обход в обратном порядке
template<class T>

```



```

void RBTree<T>::postOrder(RBTreeNode<T>* tree) const {
    if (tree != NULL) {
        postOrder(tree->left);
        postOrder(tree->right);
        cout << tree->key << (tree->color == tree->color == RED ? 12 : 8);
    }
}
//

```

```

// Функция для вывода меню
void menu() {
    cout << "1. Добавить узлы\n";
    cout << "2. Удалить узел\n";
    cout << "3. Вывести дерево\n";
    cout << "4. Поиск узла\n";
    cout << "5. Обход в прямом порядке\n";
    cout << "6. Обход в симметричном порядке\n";
    cout << "7. Обход в обратном порядке\n";
    cout << "0. Выход\n";
}
//

```

```

int main()
{
    setlocale(LC_ALL, "Russian");
    RBTree<int> rbtree;
    int choice, key;
    while (true) {
        menu();
        cout << "Выберите действие: ";
        cin >> choice;
    }
}

```

```

switch (choice) {
case 1: {
    cout << "(-1 - ВЫХОД):\n";
    while (true) {
        cin >> key;
        if (key == -1) break;
        rbtree.insert(key);
        cout << key << " добавлен.\n";
    }
    break;
}
case 2: {
    cin >> key;
    rbtree.remove(key);
    cout << key << " удалён.\n";
    break;
}
case 3:
    rbtree.print();
    break;
case 4:{
    cout << "КЛЮЧ ДЛЯ ПОИСКА: ";
    cin >> key;
    if (rbtree.search(key)){
        cout << "узел с ключом " << key << " найден.\n";
    }
    else {
        cout << "узел с ключом " << key << " не найден.\n";
    }
    break;
}
case 5:
    cout << "Обход в прямом порядке: ";

```

```

    rbtree.preOrder();
    cout << endl;
    break;
case 6:
    cout << "Обход в симметричном порядке: ";
    rbtree.inOrder();
    cout << endl;
    break;
case 7:
    cout << "Обход в обратном порядке: ";
    rbtree.postOrder();
    cout << endl;
    break;
case 0:
    return 0;
default:
    cout << "Неверный выбор!\n";
}
}
return 0;

}

```

2 Операция вставки

Вставка ключа в непустое дерево состоит из трёх этапов:

- вставка узла на своё место согласно арифметическому правилу, как и в любом дереве. Так как красно-чёрное дерево является сбалансированным, операция вставки занимает $O(\text{высота дерева})$, что равно $O(\log N)$.
- новый узел окрашивается в красный цвет, что занимает константное время $O(1)$, так как происходит только изменение значения поля цвета одного узла.
- восстанавливаются любые свойства красно-чёрного дерева, если они были нарушены. Здесь возникает разделение на случаи.

Лучший случай

Никакие свойства не были нарушены либо произошло просто перекрашивание, занимающее константное время. Сложность — $O(\log N)$.

Худший случай

Красно-чёрным деревьям требуется не более 2 поворотов для восстановления баланса, каждый из которых занимает константное время. Так что в худшем случае при вставке будет 2 оборота, и временная сложность составит $O(\log N)$.

Средний случай

Средний случай является средним значением всех возможных случаев, следовательно, временная сложность вставки в этом случае также составит $O(\log N)$. Таким образом, временная сложность для всех случаев равна $O(\log N)$.

3 Операция удаления

Нахождение узла удаления пропорционально высоте дерева, поэтому сложность равна $O(\log N)$, так как дерево сбалансировано по степеням двойки и растёт в высоту. Обмен и непосредственно удаление занимают константное время, так как в основном время затрачивается на поиск элемента. Каждое исправление дерева в соответствии со свойствами занимает константное время.

Лучший случай

В лучшем случае вращений нет, и может происходить разве что перекрашивание за константное время, потому что мы просто обращаемся к областям памяти. Временная сложность составит $O(\log N)$.

Худший случай

Красно-чёрным деревьям требуется не более 3 поворотов во время удаления. Так что в худшем случае при удалении будет 3 поворота и временная сложность составит $O(\log N)$.

Средний случай

Средний случай является средним значением всех возможных случаев, следовательно, временная сложность вставки в этом случае также составит $O(\log N)$. Таким образом, временная сложность для всех случаев равна $O(\log N)$.

4 Операция поиска

Пусть у красно-чёрного дерева будет высота h . Так как у красной вершины чёрные дети (по свойству 3), количество красных вершин не больше $h/2$. Тогда чёрных вершин не меньше, чем $h/2 - 1$. Для количества внутренних вершин в дереве выполняется неравенство $N \leq h/2 - 1$. Прологарифмировав неравенство, имеем: $\log(N + 1) \geq h/2 \rightarrow 2\log(N + 1) \geq h \rightarrow h \leq 2\log(N + 1)$.

Во всех случаях операция поиска займёт не более $O(\log N)$ времени, так как красно-чёрное дерево является сбалансированным.

5 Обходы дерева

Красно-чёрное дерево имеет 3 основных обхода: прямой, обратный и симметричный. Их разница заключается в том, в каком порядке мы обращаемся к элементам. Каждый из них будет иметь временную сложность $O(N)$, так как процедура вызывается ровно два раза для каждого узла дерева.

6 Расход памяти

Расход памяти в красно-чёрном дереве происходит так же, как в двоичном дереве поиска, и определяется общим количеством узлов. Поэтому получаем $O(N)$, потому что нам не нужно дополнительное пространство для хранения повторяющихся структур данных. Данный вывод следует из того, что каждый узел имеет три указателя: левый ребёнок, правый ребёнок и родитель. Каждый узел занимает $O(1)$ памяти. Для отслеживания цвета каждого узла требуется только один бит информации на каждый узел. Во многих случаях дополнительный бит данных может храниться без дополнительных затрат памяти. В результате сложность по памяти будет равна $O(N)$, где N – количество узлов в дереве.

Вращение и перекрашивание происходят за время $O(1)$. Временная сложность всех функций равна $O(\log N)$, потому что дерево всегда сбалансированное.