

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

ОТЧЕТ
по лабораторной работе №5 (24 вариант)
по дисциплине «Алгоритмы и структуры данных»
Тема: «Поддержка произвольной последовательности в структуре
данных для множеств»

Студент гр. 7308

Замыслов Н.Ю.

Студент гр. 7308

Цебульский С.А.

Преподаватель

Колинько П.Г.

Санкт-Петербург
2019

Цель работы

Получить практические навыки по работе с деревьями двоичного поиска и алгоритмами над ними с поддержкой произвольных последовательностей.

Постановка задачи

Дополнить программу с реализацией красно-черного дерева двоичного поиска операциями над последовательностями: `mul`, `excl` и `erase`. Выбрать такой способ доработки структуры данных, чтобы получились эффективные алгоритмы.

Реализованная программа

На экран выводятся исходные данные в виде множества и в виде последовательности. В множествах красные элементы дерева обозначены звездочкой, а последовательности на основе множества задают исходный порядок элементов. Последовательность на основе множества представлена полем `vector<node*>` и является вектором указателей на элементы дерева, благодаря чему в дереве не хранятся повторяющиеся элементы.

Программа реализует цепочку операций над множествами (рисунок 1). Для этого в прошлой лабораторной были написаны функции для множеств: объединение, пересечение, разность и симметрическая разность. В данной реализации добавлен вывод данных в виде последовательности.

```
A:
Tree [2] :      [ 0*      4      ]
Seq [5] :      [ 4      0      0      4      0      ]

B:
Tree [3] :      [ 0*      1      3*      ]
Seq [5] :      [ 0      1      1      1      3      ]

C:
Tree [4] :      [ 0      2      3      4*      ]
Seq [5] :      [ 0      3      2      4      0      ]

D:
Tree [3] :      [ 0*      2      3*      ]
Seq [5] :      [ 3      0      2      3      2      ]

E:
Tree [2] :      [ 0      1*      ]
Seq [5] :      [ 0      1      1      0      0      ]

A | B:
Tree [4] :      [ 0      1      3*      4      ]
Seq [4] :      [ 1      0      4      3      ]

C & D:
Tree [3] :      [ 0*      2      3*      ]
Seq [3] :      [ 2      0      3      ]

C & D / E:
Tree [2] :      [ 2      3*      ]
Seq [2] :      [ 2      3      ]

(A | B) ^ ((C & D) / E):
Tree [4] :      [ 0      1      2*      4      ]
Seq [4] :      [ 1      0      4      2      ]
```

Рисунок 1 – Цепочка операций над множествами

Как видно, размеры множеств не превышают размеров последовательностей.

При реализации функций (добавление, удаление, поворот, балансировка) класса красно-чёрного дерева были использованы и интегрированы в программу некоторые идеи и реализации из статьи Википедии [1].

Таблица 1

Оценки сложности алгоритмов над множествами

Алгоритм	Сложность в среднем случае
Пересечение	$O(n \log n)$
Объединение	$O(n \log n)$
Разность	$O(n \log n)$
Симметрическая разность	$O(n \log n)$

Программа реализует три операции над последовательностями:

- Erase – Из последовательности исключается часть, ограниченная порядковыми номерами от $p1$ до $p2$
- Mul – Последовательность сцепляется сама с собой заданное количество раз.
- Excl – Вторая последовательность исключается из первой, если она является её частью.

Для наглядности перед выполнением операции выводится первоначальная последовательность (рисунок 2).

```

S1:
Tree [4] :      [ 0      2      3*      4      ]
Seq [5] :      [ 0      4      2      3      2      ]

S2:
Tree [2] :      [ 0*      2      ]
Seq [5] :      [ 2      0      0      0      2      ]

Erase S2 [0; 3] :
Current S2:
Tree [2] :      [ 0*      2      ]
Seq [5] :      [ 2      0      0      0      2      ]
New S2:
Tree [1] :      [ 2      ]
Seq [1] :      [ 2      ]

Mul(1) S1:
Current S1:
Tree [4] :      [ 0      2      3*      4      ]
Seq [5] :      [ 0      4      2      3      2      ]
New S1:
Tree [4] :      [ 0      2      3*      4      ]
Seq [10] :     [ 0      4      2      3      2      0      4      2      3      2      ]

Excl S2 from S1:
Current S1:
Tree [4] :      [ 0      2      3*      4      ]
Seq [10] :     [ 0      4      2      3      2      0      4      2      3      2      ]
Current S2:
Tree [1] :      [ 2      ]
Seq [1] :      [ 2      ]
New S1:
Tree [3] :      [ 0*      3      4*      ]
Seq [6] :      [ 0      4      3      0      4      3      ]

```

Рисунок 2 – Цепочка операций над последовательностями

Как видно, при применении операций к последовательности также изменяется дерево и элементы в нём.

Таблица 2

Оценки сложности алгоритмов над последовательностями

Алгоритм	Сложность в худшем случае
mul	$O(n^2)$
excl	$O(n^2)$
erase	$O(n)$

Вывод

В результате выполнения лабораторной работы была доработана программа для работы с красно-черными деревьями с поддержкой произвольных последовательностей.

Для хранения нужного порядка использовался вектор указателей на элементы дерева. Программа реализована так, что повторяющиеся элементы не записываются в дерево, а указатель на него просто записывается в вектор.

Использованные источники

- 1) Красно-чёрное дерево // Википедия. [2019—2019]. Дата обновления: 05.05.2019. URL: <https://ru.wikipedia.org/?oldid=99613396> (дата обращения: 05.05.2019).
- 2) Колинко П. Г. Алгоритмы и структуры данных. Часть 2: Методические указания к практическим занятиям на ПЭВМ и курсовому проектированию. Вып. 1902. — СПб.: СПбГЭТУ «ЛЭТИ», 2019. — **Ошибка! Закладка не определена.** с.: ил.

Приложения

Source.cpp

```
#include "b_r_tree.h"
#include <time.h>
#include <iostream>
#include <conio.h>

b_r_tree* generate();

int main()
{
    srand(time(NULL));

    b_r_tree A, B, C, D, E, Temp1, Temp2, Temp3, result;

    // Генерация множеств
    A = *generate();
    B = *generate();
    C = *generate();
    D = *generate();
    E = *generate();

    // Вывод множеств
    cout << "A:\n" << A << '\n';
    cout << "B:\n" << B << '\n';
    cout << "C:\n" << C << '\n';
    cout << "D:\n" << D << '\n';
    cout << "E:\n" << E << '\n';

    // Цепочка операций
    Temp1 = A | B;
    cout << "A | B:\n" << Temp1 << '\n';

    Temp2 = C & D;
    cout << "C & D:\n" << Temp2 << '\n';

    Temp3 = Temp2 / E;
    cout << "C & D / E:\n" << Temp3 << '\n';

    result = Temp1 ^ Temp3;
    cout << "(A | B) ^ ((C & D) / E):\n" << result << '\n';
```



```

//.....
ДЕМОНСТРАЦИЯ РАБОТЫ С ПОСЛЕДОВАТЕЛЬНОСТЯМИ
b_r_tree S1, S2;
size_t left, right; // Границы для операции erase
size_t count; // Количество вставок mul

left = 0;
right = 3;
count = 1;
S1 = *generate();
S2 = *generate();

// Вывод последовательностей
cout << "\n-----\n\n\n";
cout << "S1: \n" << S1;
cout << "\nS2: \n" << S2;

// Операции над последовательностями

// ERASE
cout << "\n\nErase S2 [" << left << "; " << right << "] : \n";
cout << "Current S2: \n" << S2;
S2.erase(left, right);
cout << "New S2: \n" << S2;

// MUL
cout << "\n\nMul(" << count << ") S1: \n";
cout << "Current S1: \n" << S1;
S1.mul(count);
cout << "New S1: \n" << S1;

// EXCL
cout << "\n\nExcl S2 from S1: \n";
cout << "Current S1: \n" << S1;
cout << "Current S2: \n" << S2;
S1.excl(S2);
cout << "New S1: \n" << S1;

_getch();

return 0;
}

```

```
b_r_tree* generate()
{
    b_r_tree* result = new b_r_tree();

    for (size_t i = 0; i < N; ++i)
        result->add(rand() % POWER);

    return result;
}
```

b_r_tree.h

```
#pragma once
#include <iostream>
#include <vector>

class b_r_tree;

using namespace std;

const size_t N = 5; // Размер последовательностей
const size_t POWER = 5; // Мощность множества размещаемых элементов

struct node {

    node* child[2];
    node* parent;
    int key;
    bool red = false;

    node(int key) :key(key) { child[0] = nullptr; child[1] =
    nullptr; parent = nullptr; };
    ~node()
    {
        delete child[0];delete child[1];
        child[0] = nullptr; child[1] = nullptr;
    };

    node* brother();

    void rot_one(b_r_tree& tree, bool dir);
};
```

```

class b_r_tree {

    node *root;
    int size;
    vector<node *> seq;

public:

    b_r_tree() : root(nullptr), size(0) {};
    b_r_tree(b_r_tree &&);
    b_r_tree(const b_r_tree &);
    ~b_r_tree();

    friend struct node;
    friend ostream & operator<<(ostream& os, b_r_tree& set);

    bool add(int key);

    node* make_node(int data);
    node* search(int key, node* temp = nullptr) const;
    node* next_node(node* victim);

    void rebalance_add(node* inserted);
    void rebalance_delete(node* inserted);
    void rebalance_delete_2(node* inserted);
    void rebalance_delete_3(node* inserted);
    void remove(node* victim);
    void copy_tree(const node* n);
    void difference(const node* n);
    void AND(const node* parent, const b_r_tree& other);
    void put_all(ostream& os, node* temp);
    void put_Seq(ostream& os);
    void erase(size_t, size_t);
    void excl(b_r_tree &);
    void mul(size_t);

    b_r_tree& operator=(const b_r_tree &);
    b_r_tree operator |(const b_r_tree&)const;
    b_r_tree operator &(const b_r_tree&)const;
    b_r_tree operator ^(const b_r_tree&)const;
    b_r_tree operator /(const b_r_tree&)const;

};

```

b_r_tree.cpp

```
#include "b_r_tree.h"

b_r_tree::b_r_tree(const b_r_tree & set)
{
    root = nullptr;
    size = 0;
    copy_tree(set.root);
}

b_r_tree::b_r_tree(b_r_tree && set)
{
    root = set.root;
    size = set.size;
}

b_r_tree::~b_r_tree()
{
    delete root;
}

void b_r_tree::copy_tree(const node* n) { // O(n log n)
    if (n) {
        add(n->key);
        copy_tree(n->child[0]);
        copy_tree(n->child[1]);
    }
}
```

```

bool b_r_tree::add(int key) // O(log n)
{
    node* inserted_node = nullptr;

    if (!root) {
        root = make_node(key);
        size++;
        inserted_node = root;
        root->parent = nullptr;
    }
    else {
        for (node* temp = root; temp;) {
            if (temp->key == key) {
                seq.push_back(temp);
                temp = nullptr;
                return false;
            }
            else if (temp->key < key) {
                if (!temp->child[1]) {
                    temp->child[1] = make_node(key);
                    size++;
                    temp->child[1]->parent = temp;
                    inserted_node = temp->child[1];
                    temp = nullptr;
                }
                else
                    temp = temp->child[1];
            }
            else {
                if (!temp->child[0]) {
                    temp->child[0] = make_node(key);
                    size++;
                    temp->child[0]->parent = temp;
                    inserted_node = temp->child[0];
                    temp = nullptr;
                }
                else
                    temp = temp->child[0];
            }
        }
        seq.push_back(inserted_node);
        rebalance_add(inserted_node);
        return true;
    }
}

```

```

node* b_r_tree::search(int key, node* temp) const
{
    node* n = temp;
    if (!n)    n = root;
    if (n) {
        if (n->key == key)    return n;
        else if (!(n->key < key ? n->child[1] : n->child[0]))
            return nullptr;
        else search(key, n->key < key ? n->child[1] : n->
>child[0]);
    }
    else return nullptr;
}

// функция для однократного поворота узла
void node::rot_one(b_r_tree& tree, bool dir) {
    node* pivot = child[!dir];

    pivot->parent = parent;
    if (parent != nullptr) {
        if (parent->child[dir] == this)
            parent->child[dir] = pivot;
        else
            parent->child[!dir] = pivot;
    }

    child[!dir] = pivot->child[dir];
    if (pivot->child[dir] != nullptr)
        pivot->child[dir]->parent = this;

    parent = pivot;
    pivot->child[dir] = this;
    if (this == tree.root)
        tree.root = pivot;
}

node* b_r_tree::make_node(int data)
{
    node *red_node = new node(data);

    if (red_node != nullptr) {
        red_node->red = true;
    }
    return red_node;
}

```

```

void b_r_tree::rebalance_add(node* inserted) {
    if (root == inserted)
        inserted->red = false;
    else {
        if (inserted->parent->red) {
            if (inserted->parent->brother() && inserted->parent-
>brother()->red) {
                inserted->parent->red = false; inserted-
>parent->brother()->red = false;
                inserted->parent->parent->red = true;
                rebalance_add(inserted->parent->parent);
            }
            else {
                node* granda = inserted->parent->parent;
                if ((inserted == inserted->parent->child[1]) &&
(inserted->parent == granda->child[0])) {
                    inserted->parent->rot_one(*this, 0);
                    inserted = inserted->child[0];
                }
                else if ((inserted == inserted->parent-
>child[0]) && (inserted->parent == granda->child[1])) {
                    inserted->parent->rot_one(*this, 1);
                    inserted = inserted->child[1];
                }
                inserted->parent->red = false;
                granda->red = true;
                if ((inserted == inserted->parent->child[0]) &&
(inserted->parent == granda->child[0])) {
                    granda->rot_one(*this, 1);
                }
                else {
                    granda->rot_one(*this, 0);
                }
            }
        }
        root->red = false;
    }
}

```



```

void b_r_tree::rebalance_delete_3(node* son) {
    node* bro = son->brother();
    if (bro->red == false) {
        if ((son == son->parent->child[0]) &&
            (bro->child[1]->red == false) &&
            (bro->child[0]->red == true)) {
            bro->red = true;
            bro->child[0]->red = false;
            bro->rot_one(*this, 1);
        }
        else {
            if ((son == son->parent->child[1]) &&
                (bro->child[0]->red == false) &&
                (bro->child[1]->red == true)) {
                bro->red = true;
                bro->child[1]->red = false;
                bro->rot_one(*this, 0);
            }
        }
    }
    bro->red = son->parent->red;
    son->parent->red = false;

    if (son == son->parent->child[0]) {
        bro->child[1]->red = false;
        son->parent->rot_one(*this, 0);
    }
    else {
        bro->child[0]->red = false;
        son->parent->rot_one(*this, 1);
    }
}

```

```

void b_r_tree::rebalance_delete_2(node* son) {
    node* bro = son->brother();
    bool bro_left_child_black, bro_rigth_child_black;

    if (bro->child[0]) bro_left_child_black = bro->child[0]->red ==
false;
    else bro_left_child_black = true;

    if (bro->child[1]) bro_rigth_child_black = bro->child[1]->red
== false;
    else bro_rigth_child_black = true;

    if ((son->parent->red == true) &&
        (bro->red == false) &&
        (bro_left_child_black) &&
        (bro_rigth_child_black)) {
        bro->red = true;
        son->parent->red = false;
    }
    else
        rebalance_delete_3(son);
}

```

```

void b_r_tree::rebalance_delete(node* son) {
    if (son && son != root) {
        node* bro = son->brother();
        if (!bro) son->red = true;
        else {
            if (bro->red) {
                son->parent->red = true;
                bro->red = false;
                if (son == son->parent->child[0])
                    son->parent->rot_one(*this, 0);
                else
                    son->parent->rot_one(*this, 1);
            }
            bool bro_left_child_black, bro_rigth_child_black;

            if (bro->child[0]) bro_left_child_black = bro-
>child[0]->red == false;
            else bro_left_child_black = true;

            if (bro->child[1]) bro_rigth_child_black = bro-
>child[1]->red == false;
            else bro_rigth_child_black = true;

            if (!son->parent->red && !bro->red &&
bro_left_child_black && bro_rigth_child_black) {
                bro->red = true;
                rebalance_delete(son->parent);
            }
            else {
                rebalance_delete_2(son);
            }
        }
    }
}

```

```

void b_r_tree::remove(node* victim) {
    if (victim->child[0] != nullptr && victim->child[1] != nullptr)
    {
        node* next = next_node(victim);
        victim->key = next->key;
        victim = next;
    }
    node* son = victim->child[0] != nullptr ? victim->child[0] :
victim->child[1];
    bool black = !victim->red;
    if (!(victim->child[0] || victim->child[1])) {
        if (victim == root)
            root = nullptr;
        else{
            if (victim == (victim->parent->child[0]))
                victim->parent->child[0] = nullptr;
            else
                victim->parent->child[1] = nullptr;
        }
        delete victim; size--;
    }
    else if (victim->child[0] != nullptr ^ victim->child[1] !=
nullptr) {
        if (!victim->red) {
            if (victim->child[0] != nullptr)
                victim->child[0]->red = false;
            else
                victim->child[1]->red = false;
        }
        if (victim->child[0] != nullptr)
            victim->child[0]->parent = victim->parent;
        else
            victim->child[1]->parent = victim->parent;
        if (victim->parent) {
            if (victim->parent->child[0] == victim)
                victim->parent->child[0] = victim->child[0] !=
nullptr ? victim->child[0] : victim->child[1];
            else
                victim->parent->child[1] = victim->child[0] !=
nullptr ? victim->child[0] : victim->child[1];
        }
        if (victim == root) {
            root = victim->child[0] != nullptr ? victim-
>child[0] : victim->child[1];
        }
        victim->child[0] = victim->child[1] = nullptr;
        delete victim; size--;
    }
    if (black)
        rebalance_delete(son);
}

```

```

node* node::brother()
{
    if (this == parent->child[0])
        return parent->child[1];
    else
        return parent->child[0];
}

node* b_r_tree::next_node(node* victim) {
    node* iterator = victim->child[1];
    while (iterator->child[0]) iterator = iterator->child[0];
    return iterator;
}

b_r_tree & b_r_tree::operator=(const b_r_tree& other)
{
    if (&other != this) {
        delete root;
        seq.clear();
        root = nullptr;
        size = 0;
        copy_tree(other.root);

        seq.clear();

        for (size_t i = 0; i < other.seq.size(); ++i)
            seq.push_back(search(other.seq.at(i)->key));
    }
    return *this;
}

b_r_tree b_r_tree::operator |(const b_r_tree& other) const
{
    b_r_tree result = b_r_tree(*this);

    result.copy_tree(other.root);

    return b_r_tree(result);
}

b_r_tree b_r_tree::operator /(const b_r_tree& other) const
{
    b_r_tree result = b_r_tree(*this);

    result.difference(other.root);

    return b_r_tree(result);
}

```

```

void b_r_tree::difference(const node* n) {
    if (n) {
        node* victim = search(n->key);
        if (victim) remove(victim);
        difference(n->child[0]);
        difference(n->child[1]);
    }
}

b_r_tree b_r_tree::operator &(const b_r_tree& other) const
{
    b_r_tree result = b_r_tree();
    result.AND(root, other);
    return b_r_tree(result);
}

void b_r_tree::AND(const node* parent, const b_r_tree& other) {
    if (parent) {
        node* victim = other.search(parent->key);
        if (victim) add(victim->key);
        AND(parent->child[0], other);
        AND(parent->child[1], other);
    }
}

b_r_tree b_r_tree::operator ^(const b_r_tree& other) const
{
    b_r_tree result = b_r_tree(*this);
    result = (result | other) / (result & other);
    return b_r_tree(result);
}

ostream& operator<<(ostream& os, b_r_tree& tree)
{
    os << "Tree [" << tree.size << "]" : \t[ ";

    tree.put_all(os, tree.root);

    os << " ]\n";

    tree.put_Seq(os);

    os << " ]\n";

    return os;
}

```

```

void b_r_tree::put_all(ostream& os, node* n) {
    if (n) {
        if (n->child[0])
            put_all(os, n->child[0]);
        os << n->key;
        if (n->red)
            os << '*';
        os << " \t ";
        if (n->child[1])
            put_all(os, n->child[1]);
    }
}

void b_r_tree::put_Seq(ostream & os) {
    os << "Seq [" << seq.size() << "] : \t[ ";

    for (size_t i = 0; i < seq.size(); ++i)
        os << seq.at(i)->key << " \t";
}

void b_r_tree::erase(size_t left, size_t right)
{
    if (right >= seq.size())
        right = seq.size() - 1;

    if (left <= right) {
        vector<int> tempSeq;

        for (size_t i = 0; i < seq.size(); ++i)
            tempSeq.push_back(seq.at(i)->key);

        seq.clear();
        delete root;
        root = nullptr;
        size = 0;

        for (size_t i = 0; i < tempSeq.size(); ++i)
            if (!(i >= left && i <= right))
                add(tempSeq.at(i));
    }
}

```

```

void b_r_tree::excl(b_r_tree & exclSeq) {

    if (seq.size() >= exclSeq.seq.size() && exclSeq.seq.size() !=
0) {

        vector<int> tempSeq;

        for (size_t i = 0; i < seq.size(); ++i)
            tempSeq.push_back(seq.at(i)->key);

        seq.clear();
        delete root;
        root = nullptr;
        size = 0;

        int j = 0;

        for (int i = 0; i < tempSeq.size(); ++i) {

            if (tempSeq.at(i) == exclSeq.seq.at(j)->key) {
                ++j;
            }
            else {
                i -= j;
                j = 0;
            }

            if (j == exclSeq.seq.size()) {

                for (int g = i; g > i - j; --g)
                    tempSeq.erase(tempSeq.begin() + g);

                i -= j;
                j = 0;
            }
        }

        for (size_t i = 0; i < tempSeq.size(); ++i)
            add(tempSeq.at(i));
    }
}

```



```
void b_r_tree::mul(size_t n)
{
    size_t tempSize = seq.size();
    for (size_t i = 0; i < n; ++i) {
        for (size_t j = 0; j < tempSize; ++j) {
            seq.push_back(seq[j]);
        }
    }
}
```