**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. |

**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.

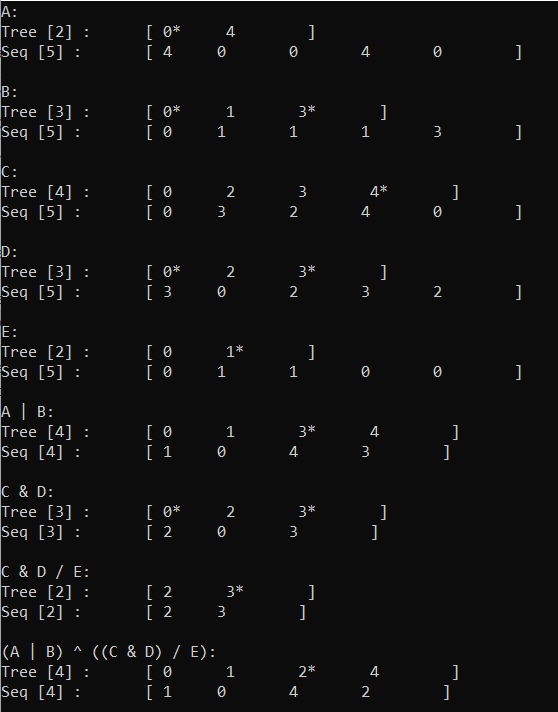


Рисунок 1 – Цепочка операций над множествами

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 *p*1 до *p*2
* 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).

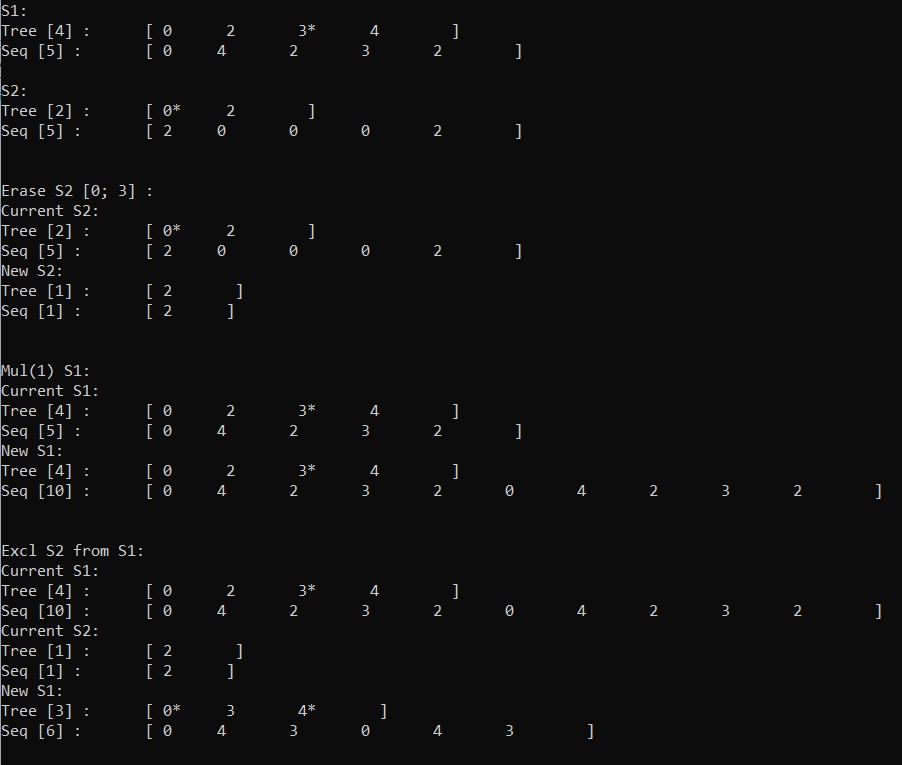


Рисунок 2 – Цепочка операций над последовательностями

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(n2) |
| excl | O(n2) |
| 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. Колинько 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]);

}

}

}