|  |
| --- |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА – Российский технологический университет"**  **РТУ МИРЭА** |

Институт комплексной безопасности и специального приборостроения

Кафедра КБ-3 «Управление и моделирование систем»

**ОТЧЕТ   
о выполнении лабораторной работы №2**

**«Реализация деревьев двоичного поиска»**

**по дисциплине   
«Программная реализация нелинейных структур»**

**Вариант № 44**

Выполнил: студент 2 курса

группы БСБО-13-18

шифр 18Б1544

\_\_**Маслов А.Е.**\_\_\_\_  
*(фио студента)*

Проверил:

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Москва 2020 г.

**Задание на лабораторную работу № 2.**

В рамках лабораторной работы №1 требуется программно реализовать (с помощью указателей (дерево двоичного поиска, оптимальное дерево двоичного поиска, рандомизированное дерево двоичного поиска - по варианту) абстрактный тип данных (АТД) в соответствии с заданием.

Абстрактный тип данных должен позволять осуществлять только операции, присущие типу деревьев двоичного поиска:

* добавить элемент,
* удалить элемент из списка,
* обнулить (проинициализировать) дерево (при необходимости).
* деструктор (при необходимости)

Используя разработанный АТД и указанный набор операций, необходимо реализовать заданный алгоритм сортировки последовательности элементов заданного типа.

На основе исходного текста программы получить аналитическую оценку трудоемкости работы алгоритма сортировки, используя О-символику для каждого реализованного метода АТД и сортировки в целом.

**Вариант № 44.**

**Тип дерева: оптимальное дерево двоичного поиска**

**Способ реализации дерева: левый сын, правый брат (указатели)**

**Операция:** А = A ⋂ B

**Вывод деревьев на экран: А– прямой, В – симметричный**

**Теория об оптимизированном дереве двоичного поиска.**

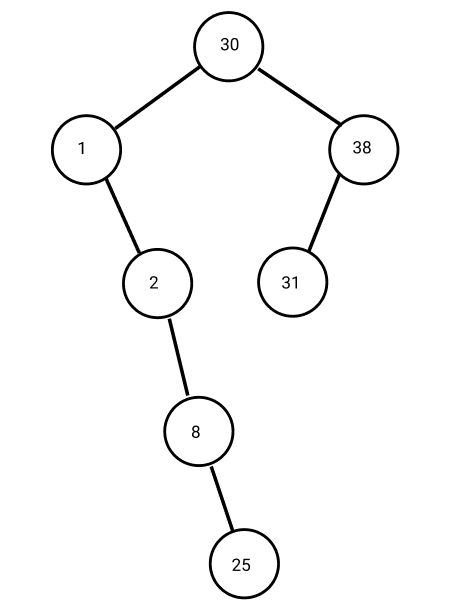
Оптимизированное дерево двоичного представляет из себя простое дерево двоичного поиска, за исключением того, что эти деревья сбалансированы. Сбалансированное двоичного дерево – это двоичное дерево поиска (граф, где узел имеет левого сына, тот в свою очередь имеет правого брата), у которого время поиска оптимальна.

Пример оптимизированного двоичного дерева.

*Входящая последовательность чисел:*

30, 1, 2, 8, 38, 25, 31

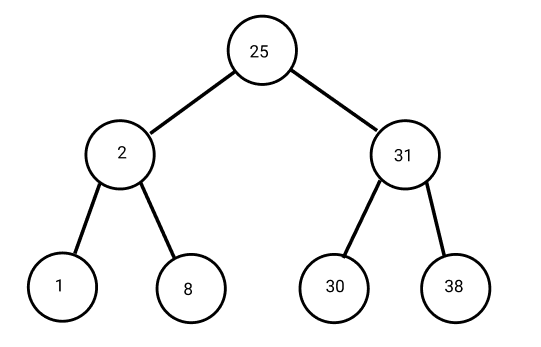
На рис 1 продемонстрировано обыкновенное дерево двоичного поиска.



*Рис.1. – построение двоичного дерева*

Как можем заметить, граф довольно громоздкий и неэффективный по поиску. У него есть преимущества перед другими графами, а конкретно – простота реализации. Если этот граф использовать в качестве сортировки линейной структуры данных, то он вполне подойдёт, т. к. он автоматически сортирует элементы.

На рис 2 изображено оптимальное двоичное дерево поиска.

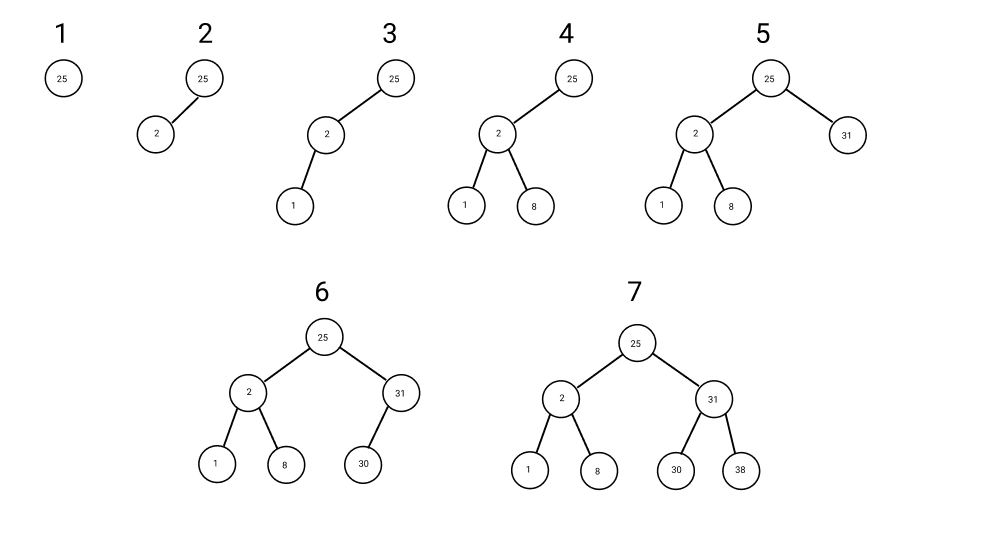


*Рис.2. –построение оптимального дерева двоичного поиска*

Как мы можем заметить, это дерево не обладает теми же недостатками, что и обыкновенное дерево двоичного поиска. Он эффективен по поиску элементов, но построение такого графа имеет некоторые проблемы. Чтобы построить такой граф, необходимо рассмотреть всю последовательность числе в отсортированном формате, что можно сделать с помощью симметричного обхода двоичного дерева.

1, 2, 8, 25, 30, 31, 38

Алгоритм построения оптимального дерева продемонстрирован на рис. 3.



*Рис.3. Алгоритм построения оптимального дерева*

Суть в следующем: в отсортированной последовательности находим середину, вставляем в дерево. Также поступает с остальными участками последовательности.

**Листинг программы с расчетами.**

#include <iostream>  
#include <vector>  
#include <iterator>  
#include <algorithm>  
#include "time.h"  
  
using namespace std;  
  
class tree {  
public:  
 long long N\_op = 0;  
  
 class node{  
 public:  
 int value;  
 node\* right;  
 node\* left;  
 node\* parent;  
 node(int value){ // 4  
 this->value = value; // 1  
 this->right = nullptr; // 1  
 this->left = nullptr; // 1  
 this->parent = nullptr; // 1  
 }  
 node(int value, node\* parent){  
 this->value = value; // 1  
 this->right = nullptr; // 1  
 this->left = nullptr; // 1  
 this->parent = parent; // 1  
 }  
 };  
  
 void insert(int value) { //12 + log(20n)  
 node \*tmp = nullptr; // 1  
 node \*ins = nullptr; // 1  
 size++ // 2  
 if (head == nullptr) { // если дерево пустое  
 head = getFreeNode(value, nullptr); // 7  
 return;  
 }  
 tmp = head; // 1  
 while (tmp) { // log(20n)  
 // если значение больше  
 if (value > tmp->value) {  
 if (tmp->right) {  
 tmp = tmp->right; // 1  
 continue;  
 } else {  
 tmp->right = getFreeNode(value, tmp); // 1  
 return;  
 }  
 //если меньше  
 } else if (value < tmp->value) {  
 if (tmp->left) {  
 tmp = tmp->left; // 1  
 continue;  
 } else {  
 tmp->left = getFreeNode(value, tmp); // 7  
 return;  
 }  
 }  
 }  
  
 }  
  
 void erase(int value) { // log(4n) + 31  
 node\* current = find\_node(value); // 3 + log(4n)  
 if (current != nullptr) {  
  
 // если есть правый потомок  
 if (current->right != nullptr) {  
 // если у правого потомка нету левого потомка  
 if (current->right->left == nullptr) {  
 current->value = current->right->value; // 3  
 node\* tmp = current->right; // 2  
 current->right = current->right->right; //3  
  
 delete tmp;  
  
 } else {  
 // найти самого левого потомка у правого потомка  
 node\* most\_left = current->right; // 1  
 while (most\_left->left != nullptr) {  
 most\_left = most\_left->left; // 2  
 }  
 current->value = most\_left->value; // 3  
 most\_left->parent->left = most\_left->right; //4  
  
 delete most\_left;  
  
 }  
 }  
 else {  
 // если этот потомок у предка слева  
 if (current->parent->left->value == current->value) {  
 current->parent->left = current->left; // 4  
 }  
 else {  
 current->parent->right = current->left; //4  
 }  
 delete current;  
  
 }  
 size--; // 2;  
 balanced(); //   
 }  
 }  
  
 void print\_direct(){  
 std::vector<int> data;  
 node\* current = head;  
 direct\_bypass(current, data);  
 for(auto it = data.begin(); it != data.end(); it++){  
 std::cout << \*it << " ";  
 }  
 std::cout << std::endl;  
 }  
  
 void print\_simmetric(){  
 std::vector<int> data;  
 node\* current = head;  
 simmetric\_bypass(current, data);  
 for(auto it = data.begin(); it != data.end(); it++){  
 std::cout << \*it << " ";  
 }  
 std::cout << std::endl;  
 }  
  
 void print\_reverse(){  
 std::vector<int> data;  
 node\* current = head;  
 reverse\_bypass(current, data);  
 for(auto it = data.begin(); it != data.end(); it++){  
 std::cout << \*it << " ";  
 }  
 std::cout << std::endl;  
 }  
  
 void balanced(){ // 1 + log(5n) + 2 + 3 + n(18 + log(20n) + 1) = n(18 + log(20n)) + log(5n) + 6  
 std::vector<int> data;  
 node\* current = head; // 1  
 simmetric\_bypass(current, data); // log(5n)  
 if(data.size() > 2) { // 2  
 tree temp;   
 int start = 0, end = data.size(); // 3  
 optimize(temp, start, end, data); // n(18 + log(20n)  
 \*this = temp; // 1  
 }  
 }  
  
 std::vector<int> transfer\_vec\_direct(){  
 std::vector<int> data;  
 node\* current = head;  
 direct\_bypass(current, data);  
 return data;  
 }  
  
 std::vector<int> transfer\_vec\_sim(){  
 std::vector<int> data;  
 node\* current = head;  
 simmetric\_bypass(current, data);  
 return data;  
 }  
  
 std::vector<int> transfer\_vec\_reverse(){  
 std::vector<int> data;  
 node\* current = head;  
 reverse\_bypass(current, data);  
 return data;  
 }

private:  
 void optimize(tree &temp, int start, int end, std::vector<int> data){ // n(18 + log(20n)  
 int middle = (start + end) / 2, lock = (end - start) / 2; // 6  
 temp.insert(data[middle]); // 12 + log(20n)  
 if(start != 0) {  
 if ((middle - start) != 1) {  
 optimize(temp, start, middle, data);   
 }  
 } else {  
 if((middle - start) != 0)  
 optimize(temp, start, middle, data);  
 }  
 if((end - middle) != 1){  
 optimize(temp, middle, end, data);  
 }  
 }  
  
 void simmetric\_bypass(node\* current, std::vector<int> &data){ // log(5n)  
 if(current->left != nullptr) // 2  
 simmetric\_bypass(current->left,data);   
 data.push\_back(current->value); //1  
 if(current->right != nullptr) // 2  
 simmetric\_bypass(current->right, data);  
 }  
  
 void reverse\_bypass(node\* current, std::vector<int> &data){  
 if(current->right != nullptr)  
 direct\_bypass(current->right, data);  
 data.push\_back(current->value); N\_op+=1;  
 if(current->left != nullptr)  
 direct\_bypass(current->left,data);  
 }  
  
 void direct\_bypass(node\* current, std::vector<int> &data){  
 data.push\_back(current->value); N\_op+=1;  
 if(current->left != nullptr)  
 direct\_bypass(current->left,data);  
 if(current->right != nullptr)  
 direct\_bypass(current->right, data);  
 }  
  
 node\* getFreeNode(int value, node \*parent) {  
 node\* tmp = new node(value, parent); N\_op+=5;  
 return tmp;  
 }

node\* find\_node(int value) { // 3 + log(4n)  
 node\* current=head; // 1  
 while(current != nullptr && current->value != value) { // log(4n)  
 if (value < current->value)  
 current = current->left; //2  
 else  
 current = current->right; //2  
 }  
 return current;  
 }  
  
 node\* \_head = nullptr; // 1  
 int size = 0; // 1  
};  
  
tree operator ^ (tree A, tree B){  
 std::vector<int> vec\_A = A.transfer\_vec\_sim();  
 std::vector<int> vec\_B = B.transfer\_vec\_sim();  
 for(auto it = vec\_A.begin(); it != vec\_B.end(); it++){  
 if(!(std::binary\_search(vec\_B.begin(), vec\_B.end(), \*it) ) ){  
 A.erase(\*it);  
 }  
 }  
 A.balanced();  
 return A;  
}  
  
int main()  
{  
 tree FirstTree;  
 std::vector<int> x(10000); // вектор из десяти элементов  
  
 for (int i = 0; i < 10000; i++){  
 x[i] = i; // инициализация диапазоном от 0 до 10 (здесь вы можете указать свой диапазон)  
 }  
  
 srand(unsigned(time(0)));  
 random\_shuffle(x.begin(), x.end()); // перемешивающем  
  
 for (int i = 0; i < 10000; i++){  
 FirstTree.insert(x[i]);  
 }  
  
 FirstTree.balanced();  
/\*  
std::vector<int> values\_first = {30, 1, 2, 8, 38, 25, 31};  
std::vector<int> values\_second = {30, 1, 5, 20, 38, 31, 22};  
tree FirstTree;  
FirstTree.insert(30);  
FirstTree.insert(1);  
FirstTree.insert(2);  
FirstTree.insert(8);  
FirstTree.insert(38);  
FirstTree.insert(25);  
FirstTree.insert(31);  
FirstTree.balanced();  
tree SecondTree;  
SecondTree.insert(30);  
SecondTree.insert(1);  
SecondTree.insert(5);  
SecondTree.insert(20);  
SecondTree.insert(38);  
SecondTree.insert(31);  
SecondTree.insert(22);  
SecondTree.balanced();  
std::cout << "First values: { ";  
for(auto it = values\_first.begin(); it != values\_first.end(); it++)  
 std::cout << \*it << " ";  
std::cout << "};";  
std::cout << " A = ";  
FirstTree.print\_direct();  
std::cout << "Second values: { ";  
for(auto it = values\_second.begin(); it != values\_second.end(); it++)  
 std::cout << \*it << " ";  
std::cout << "};";  
std::cout << " B = ";  
SecondTree.print\_simmetric();  
std::cout << std::endl << "A ^ B = ";  
(FirstTree^SecondTree).print\_direct();  
return 0;

\*/  
 // FirstTree.print\_simmetric();  
 // FirstTree.print\_direct();  
 std::cout << "N\_op: " << FirstTree.N\_op << ";" << std::endl;  
 //FirstTree.print\_reverse();  
 return 0;  
}

F(n)=

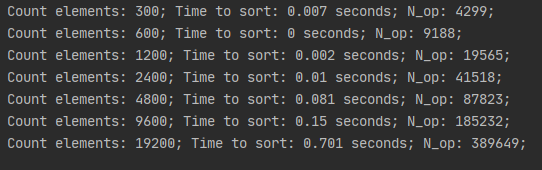
O(F(n))=

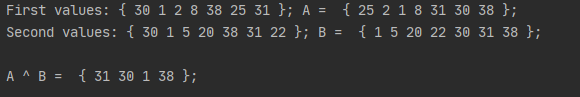
**Таблица результата экспериментов и графики зависимостей**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество | F(n) | O(F(n)) | T(n) | N\_op |
| 300 | 9189 | 2467 | 0,007 | 4299 |
| 600 | 18956 | 5538 | 0,0001 | 9188 |
| 1200 | 39088 | 12274 | 0,002 | 19565 |
| 2400 | 80551 | 26950 | 0,01 | 41518 |
| 4800 | 145146 | 58700 | 0,081 | 87823 |
| 9600 | 298662 | 126999 | 0,15 | 185232 |
| 19200 | 559471 | 273197 | 0,701 | 389649 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 1312715 | 352429 | 2,1374738 | 0,573854385 |
| 189560000 | 55380000 | 2,0631258 | 0,602742708 |
| 19544000 | 6137000 | 1,9978533 | 0,627344748 |
| 8055100 | 2695000 | 1,9401464 | 0,649116046 |
| 1791926 | 724692 | 1,6527106 | 0,66838983 |
| 1991080 | 846660 | 1,6123672 | 0,685621275 |
| 798105 | 389725 | 1,4358333 | 0,701136151 |

**Скриншот работы программы:**

****

****

**Выводы.**

По результатам экспериментов было установлено, что деревья двоичного поиска эффективнее линейных структур данных, в сортировке и нахождения данных. К сожалению, сложно конструировать на языке программирования оптимальное двоичное дерево, так как нужно учитывать не только особенности двоичного дерева, как графа, но и дополнительно заниматься оптимизацией такого дерева, что подразумевает перестановку элементов в графе. По результатам этой лабораторной работы мы можем понять, какую структуру данных используют IT-компании, чтобы повысить скорость ответа пользователю.

**Литература:**

1. Структуры данных и алгоритмы. Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман. – М.: Издательский дом «Вильямс», 2000

2. Д. Кнут. Искусство программирования для ЭВМ.

**Приложение 1. Применение счетчика операций N\_op.**

#include <iostream>  
#include <vector>  
#include <iterator>  
#include <algorithm>  
#include "time.h"  
  
using namespace std;  
  
class tree {  
public:  
 long long N\_op = 0;  
  
 class node{  
 public:  
 int value;  
 node\* right;  
 node\* left;  
 node\* parent;  
 node(int value){  
 this->value = value;  
 this->right = nullptr;  
 this->left = nullptr;  
 this->parent = nullptr;  
 }  
 node(int value, node\* parent){  
 this->value = value;  
 this->right = nullptr;  
 this->left = nullptr;  
 this->parent = parent;  
 }  
 };  
  
 void insert(int value) {  
 node \*tmp = nullptr; N\_op+=1;  
 node \*ins = nullptr; N\_op+=1;

size++;  
 // если дерево пустое  
 if (head == nullptr) {  
 head = getFreeNode(value, nullptr); N\_op+=1;  
 return;  
 }  
 tmp = head; N\_op+=1;  
 while (tmp) {  
 // если значение больше  
 if (value > tmp->value) {  
 if (tmp->right) {  
 tmp = tmp->right; N\_op+=1;  
 continue;  
 } else {  
 tmp->right = getFreeNode(value, tmp); N\_op+=1;  
 return;  
 }  
 //если меньше  
 } else if (value < tmp->value) {  
 if (tmp->left) {  
 tmp = tmp->left; N\_op+=1;  
 continue;  
 } else {  
 tmp->left = getFreeNode(value, tmp); N\_op+=1;  
 return;  
 }  
 }  
 }  
  
 }  
  
 void erase(int value) {  
 node\* current = find\_node(value); N\_op+=1;  
 if (current != nullptr) {  
  
 // если есть правый потомок  
 if (current->right != nullptr) {  
 // если у правого потомка нету левого потомка  
 if (current->right->left == nullptr) {  
 current->value = current->right->value; N\_op+=1;  
 node\* tmp = current->right; N\_op+=1;  
 current->right = current->right->right; N\_op+=1;  
  
 delete tmp;  
  
 } else {  
 // найти самого левого потомка у правого потомка  
 node\* most\_left = current->right; N\_op+=1;  
 while (most\_left->left != nullptr) {  
 most\_left = most\_left->left; N\_op+=1;  
 }  
 current->value = most\_left->value; N\_op+=1;  
 most\_left->parent->left = most\_left->right; N\_op+=1;  
  
 delete most\_left;  
  
 }  
 }  
 else {  
 // если этот потомок у предка слева  
 if (current->parent->left->value == current->value) {  
 current->parent->left = current->left; N\_op+=1;  
 }  
 else {  
 current->parent->right = current->left; N\_op+=1;  
 }  
 delete current;  
  
 }  
 size--; N\_op+=2;  
 balanced();  
 }  
 }  
  
 void print\_direct(){  
 std::vector<int> data;  
 node\* current = head;  
 direct\_bypass(current, data);  
 for(auto it = data.begin(); it != data.end(); it++){  
 std::cout << \*it << " ";  
 }  
 std::cout << std::endl;  
 }  
  
 void print\_simmetric(){  
 std::vector<int> data;  
 node\* current = head;  
 simmetric\_bypass(current, data);  
 for(auto it = data.begin(); it != data.end(); it++){  
 std::cout << \*it << " ";  
 }  
 std::cout << std::endl;  
 }  
  
 void print\_reverse(){  
 std::vector<int> data;  
 node\* current = head;  
 reverse\_bypass(current, data);  
 for(auto it = data.begin(); it != data.end(); it++){  
 std::cout << \*it << " ";  
 }  
 std::cout << std::endl;  
 }  
  
 void balanced(){  
 std::vector<int> data;  
 node\* current = head; N\_op+=1;  
 simmetric\_bypass(current, data);  
 if(data.size() > 2) {  
 tree temp;  
 int start = 0, end = data.size(); N\_op+=2;  
 optimize(temp, start, end, data);  
 \*this = temp; N\_op+=1;  
 }  
 }  
  
 std::vector<int> transfer\_vec\_direct(){  
 std::vector<int> data;  
 node\* current = head;  
 direct\_bypass(current, data);  
 return data;  
 }  
  
 std::vector<int> transfer\_vec\_sim(){  
 std::vector<int> data;  
 node\* current = head;  
 simmetric\_bypass(current, data);  
 return data;  
 }  
  
 std::vector<int> transfer\_vec\_reverse(){  
 std::vector<int> data;  
 node\* current = head;  
 reverse\_bypass(current, data);  
 return data;  
 }  
  
 bool find(int value) {  
 node\* current = head; N\_op+=1;  
 while(current != nullptr && current->value != value) {  
 if (value < current->value)  
 current = current->left;  
 else  
 current = current->right;  
 N\_op+=1;  
 }  
  
 }  
private:  
 void optimize(tree &temp, int start, int end, std::vector<int> data){  
 int middle = (start + end) / 2, lock = (end - start) / 2; N\_op += 3;  
 temp.insert(data[middle]);  
 if(start != 0) {  
 if ((middle - start) != 1) {  
 optimize(temp, start, middle, data);  
 }  
 } else {  
 if((middle - start) != 0)  
 optimize(temp, start, middle, data);  
 }  
 if((end - middle) != 1){  
 optimize(temp, middle, end, data);  
 }  
 }  
  
 void simmetric\_bypass(node\* current, std::vector<int> &data){  
 if(current->left != nullptr)  
 simmetric\_bypass(current->left,data);  
 data.push\_back(current->value); N\_op+=1;  
 if(current->right != nullptr)  
 simmetric\_bypass(current->right, data);  
 }  
  
 void reverse\_bypass(node\* current, std::vector<int> &data){  
 if(current->right != nullptr)  
 direct\_bypass(current->right, data);  
 data.push\_back(current->value); N\_op+=1;  
 if(current->left != nullptr)  
 direct\_bypass(current->left,data);  
 }  
  
 void direct\_bypass(node\* current, std::vector<int> &data){  
 data.push\_back(current->value); N\_op+=1;  
 if(current->left != nullptr)  
 direct\_bypass(current->left,data);  
 if(current->right != nullptr)  
 direct\_bypass(current->right, data);  
 }  
  
 node\* getFreeNode(int value, node \*parent) {  
 node\* tmp = new node(value, parent); N\_op+=5;  
 return tmp;  
 }  
  
 node\* find\_node(int value) {  
 node\* current=head; N\_op+=1;  
 while(current != nullptr && current->value != value) {  
 if (value < current->value)  
 current = current->left;  
 else  
 current = current->right;  
 N\_op+=1;  
 }  
 return current;  
 }  
  
 node\* head = nullptr;  
 int size = 0;  
};  
  
tree operator ^ (tree A, tree B){  
 std::vector<int> vec\_A = A.transfer\_vec\_sim();  
 std::vector<int> vec\_B = B.transfer\_vec\_sim();  
 for(auto it = vec\_A.begin(); it != vec\_B.end(); it++){  
 if(!(std::binary\_search(vec\_B.begin(), vec\_B.end(), \*it) ) ){  
 A.erase(\*it);  
 }  
 }  
 A.balanced();  
 return A;  
}  
  
int main()  
{  
 tree FirstTree;  
 std::vector<int> x(10000); // вектор из десяти элементов  
  
 for (int i = 0; i < 10000; i++){  
 x[i] = i; // инициализация диапозоном от 0 до 10 (здесь вы можете указать свой диапозон)  
 }  
  
 srand(unsigned(time(0)));  
 random\_shuffle(x.begin(), x.end()); // перемешиваеаем  
  
 for (int i = 0; i < 10000; i++){  
 FirstTree.insert(x[i]);  
 }  
  
 FirstTree.balanced();  
/\*  
 std::vector<int> values\_first = {30, 1, 2, 8, 38, 25, 31};  
std::vector<int> values\_second = {30, 1, 5, 20, 38, 31, 22};  
tree FirstTree;  
FirstTree.insert(30);  
FirstTree.insert(1);  
FirstTree.insert(2);  
FirstTree.insert(8);  
FirstTree.insert(38);  
FirstTree.insert(25);  
FirstTree.insert(31);  
FirstTree.balanced();  
tree SecondTree;  
SecondTree.insert(30);  
SecondTree.insert(1);  
SecondTree.insert(5);  
SecondTree.insert(20);  
SecondTree.insert(38);  
SecondTree.insert(31);  
SecondTree.insert(22);  
SecondTree.balanced();  
std::cout << "First values: { ";  
for(auto it = values\_first.begin(); it != values\_first.end(); it++)  
 std::cout << \*it << " ";  
std::cout << "};";  
std::cout << " A = ";  
FirstTree.print\_direct();  
std::cout << "Second values: { ";  
for(auto it = values\_second.begin(); it != values\_second.end(); it++)  
 std::cout << \*it << " ";  
std::cout << "};";  
std::cout << " B = ";  
SecondTree.print\_simmetric();  
std::cout << std::endl << "A ^ B = ";  
(FirstTree^SecondTree).print\_direct();

\*/  
 // FirstTree.print\_simmetric();  
 // FirstTree.print\_direct();  
 std::cout << "N\_op: " << FirstTree.N\_op << ";" << std::endl;  
 //FirstTree.print\_reverse();  
 return 0;  
}