**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра математического обеспечения и применения ЭВМ**

**Лабораторная работа №5**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: Случайные бинарные деревья с рандомизацией**

|  |  |  |
| --- | --- | --- |
| Студент гр. 7381 |  | Судакова П.С |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2018

**Задание.  
Вариант 11.**

По заданному файлу F (типа file of *Elem*), все элементы которого различны, построить случайное БДП с рандомизацией. Для построенного БДП проверить, входит ли в него элемент *е* типа *Elem*, и если входит, то удалить элемент *е* в дерево поиска.

**Описание** **алгоритма.**

В данной лабораторной работе каждый узел двоичного дерева поиска содержит ключ *key*, который управляет всеми процессами, и пару указателей *left* и *righ*t на левое и правое поддеревья. Если какой-либо указатель нулевой, то соответствующее дерево является пустым (т.е. отсутствует). Помимо этого, для реализации рандомизированного дерева потребовалось еще одно поле *size*, в котором храниться размер дерева с корнем в данном узле. Узлы представлены структурами.

Основное свойство дерева поиска — любой ключ в левом поддереве меньше корневого ключа, а в правом поддереве — больше корневого ключа. Это свойство позволяет нам очень просто организовать поиск заданного ключа, перемещаясь от корня вправо или влево в зависимости от значения корневого ключа.

Вставка в корень дерева. Необходимой составляющей рандомизации является применение специальной вставки нового ключа, в результате которой новый ключ оказывается в корне дерева. Для реализации вставки в корень нам потребовалась функция поворота, которая производит локальное преобразование дерева. Сначала рекурсивно вставляем новый ключ в корень левого или правого поддеревьев (в зависимости от результата сравнения с корневым ключом) и выполняем правый (левый) поворот, который поднимает нужный нам узел в корень дерева.

Удаление ключей. Прежде чем удалять узлы, необходимо соединить два дерева. Даны два дерева поиска с корнями, причем любой ключ первого дерева меньше любого ключа во втором дереве. Эти два дерева объединяем в одно. В качестве корня нового дерева можно взять любой из двух корней деревьев. Тогда левое поддерево можно оставить как есть, а справа подвесить объединение двух деревьев — правого поддерева и всего левого дерева. Удаление происходит по ключу — ищется узел с заданным ключом и удаляется этот узел из дерева. Стадия поиска такая же, как и при поиске, далее объединятся левое и правое поддеревья найденного узла, удаляется узел, возвращается корень объединенного дерева.

**Тестирование программы.**

* light without removal test: "demonstation.txt"

removal value: inserted values: tree before removal:

values: [ 1 2 3 4 5 ]

tree shape:

5

        4

                3

            2

    1

removal process:

current value: 5

removal value less then current value - turn to the left

current value: 1

removal time: 53000 ns.

tree after removal:

values: [ 1 2 3 4 5 ]

tree shape:

5

        4

                3

            2

    1

* light without removal test: "empty.txt"

removal value: inserted values: tree before removal:

values: tree shape:

removal process:

removal time: 1000 ns.

tree after removal:

values: tree shape:

* light without removal test: "little.txt"

removal value: inserted values: tree before removal:

values: [ 1 2 3 4 5 6 7 ]

tree shape:

    7

            6

        5

4

        3

    2

        1

removal process:

current value: 4

removal value less then current value - turn to the left

current value: 2

removal value less then current value - turn to the left

current value: 1

removal time: 75000 ns.

tree after removal:

values: [ 1 2 3 4 5 6 7 ]

tree shape:

    7

            6

        5

4

        3

    2

        1

* light without removal test: "random\_demonstation\_duplicate1.txt"

removal value: inserted values: tree before removal:

values: [ 1 2 3 4 5 6 7 8 9 ]

tree shape:

9

    8

            7

                    6

                5

        4

                    3

                2

            1

removal process:

current value: 9

removal value less then current value - turn to the left

current value: 8

removal value less then current value - turn to the left

current value: 4

removal value less then current value - turn to the left

current value: 1

removal time: 63000 ns.

tree after removal:

values: [ 1 2 3 4 5 6 7 8 9 ]

tree shape:

9

    8

            7

                    6

                5

        4

                    3

                2

            1

* light without removal test: "random\_demonstation\_duplicate2.txt"

removal value: inserted values: tree before removal:

values: [ 1 2 3 4 5 6 7 8 9 ]

tree shape:

9

                8

            7

                6

        5

            4

    3

        2

            1

removal process:

current value: 9

removal value less then current value - turn to the left

current value: 3

removal value less then current value - turn to the left

current value: 2

removal value less then current value - turn to the left

current value: 1

removal time: 62000 ns.

tree after removal:

values: [ 1 2 3 4 5 6 7 8 9 ]

tree shape:

9

                8

            7

                6

        5

            4

    3

        2

            1

* light with removal test: "demonstation.txt"

removal value: inserted values: tree before removal:

values: [ 1 2 3 4 5 ]

tree shape:

5

    4

        3

            2

                1

removal process:

current value: 5

removal value less then current value - turn to the left

current value: 4

removal value less then current value - turn to the left

current value: 3

current value equal removal value - time to merge

removal time: 63000 ns.

tree after removal:

values: [ 1 2 4 5 ]

tree shape:

5

    4

        2

            1

* light with removal test: "empty.txt"

removal value: inserted values: tree before removal:

values: tree shape:

removal process:

removal time: 1000 ns.

tree after removal:

values: tree shape:

* light with removal test: "little.txt"

removal value: inserted values: tree before removal:

values: [ 1 2 3 4 5 6 7 ]

tree shape:

                7

            6

        5

    4

        3

            2

1

removal process:

current value: 1

current value equal removal value - time to merge

removal time: 64000 ns.

tree after removal:

values: [ 2 3 4 5 6 7 ]

tree shape:

            7

        6

    5

4

    3

        2

* light with removal test: "one\_node.txt"

removal value: inserted values: tree before removal:

values: [ 1 ]

tree shape:

1

removal process:

current value: 1

current value equal removal value - time to merge

removal time: 37000 ns.

tree after removal:

values: tree shape:

* invalid test: "meow.txt"

removal value: inserted values: you entered not a number, sorry

* invalid test: "normal\_all\_but\_meow.txt"

removal value: inserted values: you entered not a number, sorry

* invalid test: "punctuation.txt"

removal value: inserted values: you entered not a number, sorry

* invalid test: "remove\_normal\_and\_meow.txt"

removal value: inserted values: you entered not a number, sorry

**Выводы.**

В процессе выполнения лабораторной работы были получены навыки по построению случайных Бинарных Деревьев Поиска с рандомизацией, а также были приобретены навыки применения способа хранение таких данных на практике.

**ПРИЛОЖЕНИЕ А**

**ИСХОДНЫЙ КОД(Main)**

#include "randomized\_binary\_search\_tree.hpp"

#include <iostream>

#include <chrono>

int main() {

RandomizedBinarySearchTree<int> rbst;

int removal\_value;

std::cout << "removal value: ";

std::cin >> removal\_value;

std::cout << "inserted values: ";

while (!std::cin.eof()) {

if (std::cin.fail()) {

std::cout << "you entered not a number, sorry" << std::endl;

return EXIT\_FAILURE;

}

int inserted\_value;

std::cin >> inserted\_value;

rbst.insert(inserted\_value);

}

#ifndef ONLY\_TIME

std::cout << "tree before removal: " << std::endl;

std::cout << "values: ";

rbst.display\_like\_list();

std::cout << "tree shape: " << std::endl;

rbst.display\_like\_tree();

std::cout << std::endl;

#endif

#ifdef DETAILEDEXPLANATION

std::cout << "removal process: " << std::endl;

#endif

auto begin = std::chrono::steady\_clock::now();

rbst.remove(removal\_value);

auto end = std::chrono::steady\_clock::now();

auto elapsed\_ms = std::chrono::duration\_cast<std::chrono::nanoseconds>(end - begin);

std::cout << "removal time: " << elapsed\_ms.count() << " ns." << std::endl;

#ifndef ONLY\_TIME

std::cout << "tree after removal: " << std::endl;

std::cout << "values: ";

rbst.display\_like\_list();

std::cout << "tree shape: " << std::endl;

rbst.display\_like\_tree();

#endif

std::cout << std::endl << std::endl;

return EXIT\_SUCCESS;

}

**Приложение Б  
Код randomized\_binary\_search\_tree\_node**

#ifndef \_\_RANDOMIZED\_BINARY\_SEARCH\_TREE\_\_

#define \_\_RANDOMIZED\_BINARY\_SEARCH\_TREE\_\_

#include "randomized\_binary\_search\_tree\_node.hpp"

template <class Type>

class RandomizedBinarySearchTree {

public:

RandomizedBinarySearchTree();

void insert(const Type &value);

void remove(const Type &value);

void display\_like\_list();

void display\_like\_tree();

~RandomizedBinarySearchTree();

private:

RandomizedBinarySearchTreeNode<Type>\* m\_root;

};

template <class Type>

RandomizedBinarySearchTree<Type>::RandomizedBinarySearchTree()

: m\_root(nullptr)

{

}

template <class Type>

void RandomizedBinarySearchTree<Type>::insert(const Type &value)

{

if (m\_root)

m\_root = m\_root->insert(value);

else

m\_root = new RandomizedBinarySearchTreeNode<Type>(value, nullptr, nullptr);

}

template <class Type>

void RandomizedBinarySearchTree<Type>::remove(const Type &value)

{

if (m\_root)

m\_root = m\_root->remove(value);

}

template <class Type>

void RandomizedBinarySearchTree<Type>::display\_like\_list()

{

if (m\_root)

m\_root->display\_like\_list();

}

template <class Type>

void RandomizedBinarySearchTree<Type>::display\_like\_tree()

{

if (m\_root)

m\_root->display\_like\_tree();

}

template <class Type>

RandomizedBinarySearchTree<Type>::~RandomizedBinarySearchTree()

{

if (m\_root)

delete m\_root;

}

#endif

**Приложение В  
Код**

#ifndef \_\_RANDOMIZED\_BINARY\_SEARCH\_TREE\_NODE\_\_

#define \_\_RANDOMIZED\_BINARY\_SEARCH\_TREE\_NODE\_\_

#define DETAILEDEXPLANATION

#include <iostream>

#include <utility>

#include <random>

#include <chrono>

template <class Type>

class RandomizedBinarySearchTreeNode {

public:

RandomizedBinarySearchTreeNode(const Type &value, RandomizedBinarySearchTreeNode \*left, RandomizedBinarySearchTreeNode \*right);

RandomizedBinarySearchTreeNode<Type> \*insert(const Type &value);

RandomizedBinarySearchTreeNode<Type> \*remove(const Type &value);

void display\_like\_list();

void display\_like\_tree();

~RandomizedBinarySearchTreeNode();

private:

RandomizedBinarySearchTreeNode<Type> \*rotate\_left();

RandomizedBinarySearchTreeNode<Type> \*rotate\_right();

RandomizedBinarySearchTreeNode<Type> \*insert\_like\_root(const Type& value);

static RandomizedBinarySearchTreeNode<Type> \*merge(RandomizedBinarySearchTreeNode<Type> \*that, RandomizedBinarySearchTreeNode<Type> \*other);

void size\_update();

std::size\_t size() const;

void \_display\_like\_list();

void \_display\_like\_tree(std::size\_t depth);

/\*

(!) members only (!)

\*/

static std::default\_random\_engine m\_generator;

Type m\_value;

RandomizedBinarySearchTreeNode<Type> \*m\_left;

RandomizedBinarySearchTreeNode<Type> \*m\_right;

std::size\_t m\_size;

};

template <class Type>

std::default\_random\_engine RandomizedBinarySearchTreeNode<Type>::m\_generator = std::default\_random\_engine(std::chrono::system\_clock::now().time\_since\_epoch().count());

template <class Type>

RandomizedBinarySearchTreeNode<Type>::RandomizedBinarySearchTreeNode(const Type &value, RandomizedBinarySearchTreeNode<Type> \*left, RandomizedBinarySearchTreeNode<Type> \*right)

: m\_value(value), m\_left(left), m\_right(right)

{

size\_update();

}

template <class Type>

std::size\_t RandomizedBinarySearchTreeNode<Type>::size() const

{

return m\_size;

}

template <class Type>

void RandomizedBinarySearchTreeNode<Type>::size\_update()

{

m\_size = 1 + (m\_left ? m\_left->size() : 0) + (m\_right ? m\_right->size() : 0);

}

template <class Type>

void RandomizedBinarySearchTreeNode<Type>::display\_like\_list()

{

std::cout << "[ ";

\_display\_like\_list();

std::cout << "]";

std::cout << std::endl;

}

template <class Type>

void RandomizedBinarySearchTreeNode<Type>::\_display\_like\_list()

{

if (m\_left)

m\_left->\_display\_like\_list();

std::cout << m\_value << " ";

if (m\_right)

m\_right->\_display\_like\_list();

}

template <class Type>

void RandomizedBinarySearchTreeNode<Type>::display\_like\_tree()

{

\_display\_like\_tree(0);

}

template <class Type>

void RandomizedBinarySearchTreeNode<Type>::\_display\_like\_tree(std::size\_t depth)

{

if (m\_right)

m\_right->\_display\_like\_tree(depth + 1);

for (std::size\_t i = 0; i < depth; ++i)

std::cout << '\t';

std::cout << m\_value << std::endl;

if (m\_left)

m\_left->\_display\_like\_tree(depth + 1);

}

template <class Type>

RandomizedBinarySearchTreeNode<Type> \*RandomizedBinarySearchTreeNode<Type>::rotate\_left()

{

auto p\_right = m\_right;

if (!p\_right)

return this;

m\_right = p\_right->m\_left;

p\_right->m\_left = this;

p\_right->m\_size = m\_size;

size\_update();

return p\_right;

}

template <class Type>

RandomizedBinarySearchTreeNode<Type> \*RandomizedBinarySearchTreeNode<Type>::rotate\_right()

{

auto p\_left = m\_left;

if (!p\_left)

return this;

m\_left = p\_left->m\_right;

p\_left->m\_right = this;

p\_left->m\_size = m\_size;

size\_update();

return p\_left;

}

template <class Type>

RandomizedBinarySearchTreeNode<Type> \*RandomizedBinarySearchTreeNode<Type>::insert\_like\_root(const Type& value)

{

if (value < m\_value) {

if (m\_left)

m\_left = m\_left->insert\_like\_root(value);

else

m\_left = new RandomizedBinarySearchTreeNode<Type>(value, nullptr, nullptr);

return rotate\_right();

}

if (value > m\_value) {

if (m\_right)

m\_right = m\_right->insert\_like\_root(value);

else

m\_right = new RandomizedBinarySearchTreeNode<Type>(value, nullptr, nullptr);

return rotate\_left();

}

return this;

}

template <class Type>

RandomizedBinarySearchTreeNode<Type> \*RandomizedBinarySearchTreeNode<Type>::insert(const Type& value)

{

if (m\_generator() % (m\_size + 1) == 0)

return insert\_like\_root(value);

if (value < m\_value) {

if (m\_left)

m\_left = m\_left->insert(value);

else

m\_left = new RandomizedBinarySearchTreeNode<Type>(value, nullptr, nullptr);

}

if (value > m\_value) {

if (m\_right)

m\_right = m\_right->insert(value);

else

m\_right = new RandomizedBinarySearchTreeNode<Type>(value, nullptr, nullptr);

}

size\_update();

return this;

}

template <class Type>

RandomizedBinarySearchTreeNode<Type> \*RandomizedBinarySearchTreeNode<Type>::merge(RandomizedBinarySearchTreeNode<Type> \*that, RandomizedBinarySearchTreeNode<Type> \*other)

{

if (!that)

return other;

if (!other)

return that;

if (m\_generator() % (that->m\_size + other->m\_size) < that->m\_size) {

#ifdef DETAILEDEXPLANATION

std::cout << "randomly selected left subtree: " << that->m\_value << std::endl;

#endif

that->m\_right = merge(that->m\_right, other);

that->size\_update();

return that;

} else {

#ifdef DETAILEDEXPLANATION

std::cout << "randomly selected right subtree: " << other->m\_value << std::endl;

#endif

other->m\_left = merge(that, other->m\_left);

other->size\_update();

return other;

}

}

template <class Type>

RandomizedBinarySearchTreeNode<Type> \*RandomizedBinarySearchTreeNode<Type>::remove(const Type &value)

{

#ifdef DETAILEDEXPLANATION

std::cout << "current value: " << m\_value << std::endl;

#endif

if (value == m\_value) {

#ifdef DETAILEDEXPLANATION

std::cout << "current value equal removal value - time to merge" << std::endl;

#endif

auto new\_root = merge(m\_left, m\_right);

m\_left = m\_right = nullptr;

delete this;

return new\_root;

}

if (value < m\_value && m\_left) {

#ifdef DETAILEDEXPLANATION

std::cout << "removal value less then current value - turn to the left" << std::endl;

#endif

m\_left = m\_left->remove(value);

}

if (value > m\_value && m\_right) {

#ifdef DETAILEDEXPLANATION

std::cout << "removal value greater then current value - turn to the right" << std::endl;

#endif

m\_right = m\_right->remove(value);

}

return this;

}

template <class Type>

RandomizedBinarySearchTreeNode<Type>::~RandomizedBinarySearchTreeNode()

{

if (m\_left)

delete m\_left;

if (m\_right)

delete m\_right;

}

#endif

**Приложение Г  
Код Make файла**

CODE = ./source/

OBJ = main.o randomized\_binary\_search\_tree.o randomized\_binary\_search\_tree\_node.o

EXE = randomized\_binary\_search\_tree

CXX = g++

CFLAGS = -std=c++14 -Wall -Wextra -c

all: $(OBJ)

$(CXX) $(OBJ) -o $(EXE)

main.o: $(CODE)main.cpp $(CODE)randomized\_binary\_search\_tree.hpp

$(CXX) $(CFLAGS) $(CODE)main.cpp

randomized\_binary\_search\_tree.o: $(CODE)randomized\_binary\_search\_tree.cpp $(CODE)randomized\_binary\_search\_tree.hpp

$(CXX) $(CFLAGS) $(CODE)randomized\_binary\_search\_tree.cpp

randomized\_binary\_search\_tree\_node.o: $(CODE)randomized\_binary\_search\_tree\_node.cpp $(CODE)randomized\_binary\_search\_tree\_node.hpp

$(CXX) $(CFLAGS) $(CODE)randomized\_binary\_search\_tree\_node.cpp

clean:

rm $(OBJ)

cleanest:

rm $(OBJ) $(EXE)