**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №3**

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

## Вариант №1

Студент гр. 9309 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Корягин Е.А.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Тутуева А.В.

Санкт-Петербург

2020

## Постановка задачи

## Реализовать класс дерева двоичного поиска и класс итератора с набором методов и наличием unit-тестов.

## Описание реализуемого класса и метода

Описание реализуемого класса:

Классы BinaryTree и Tree хранят в себе все необходимые указатели и данные дерева двоичного поиска.

Класс Iterator – Класс итератора.

Класс bft\_iter – Класс итератора выполняющего обход дерева в ширину используя очередь. Хранит в себе указатель на элемент, с которым в данный момент работает итератор и функции перехода к следующему элементу дерева.

Класс dft\_iter – Класс итератора выполняющего обход дерева в глубину используя стек. Хранит в себе указатель на элемент, с которым в данный момент работает итератор и функции перехода к следующему элементу дерева.

**Описание методов:**

Метод insert – Метод добавляет элемент в дерево.

Метод contains – Метод проверяет наличие элемента в дереве.

Метод remove – Метод удаляет выбранный элемент дерева.

**Оценка временной сложности методов:**

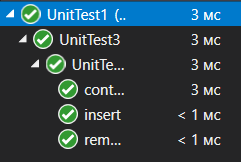
1. insert = O(logN).
2. contains = O(log N).
3. remove = O(log N).
4. Queue = O(N^2)
5. Stack =O(logN)
6. bft\_iter = O(N).
7. dft\_iter = O(N).

## Описание реализованных unit-тестов

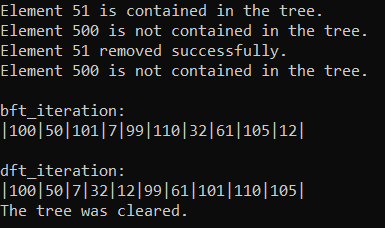
insert – Проверка функции вставки элемента в дерево, с помощью сравнения элементов дерева с элементами другого созданного и заполненного дерева.

remove – Проверка функции удаления элемента из дерева, с помощью сравнения элементов дера с элементами другого созданного дерева без удаленного элемента.

contains – Проверка функции, показывающей наличие выбранного элемента в дереве.



## Пример работы



## Листинг

**Source.cpp**

#include "func\_n\_class.h"

#include <iostream>

#include <locale>

using namespace std;

int main()

{

Tree\* tree = new Tree;

tree->get\_root(100);

tree->insert(50);

tree->insert(101);

tree->insert(110);

tree->insert(99);

tree->insert(105);

tree->insert(7);

tree->insert(32);

tree->insert(61);

tree->insert(12);

tree->insert(51);

if (tree->contains(51) == true)

cout << "Element " << 51 << " is contained in the tree.\n";

else

cout << "Element " << 51 << " is not contained in the tree.\n";

if (tree->contains(500) == true)

cout << "Element " << 500 << " is contained in the tree.\n";

else

cout << "Element " << 500 << " is not contained in the tree.\n";

if (tree->contains(50) == true)

{

tree->remove(51);

cout << "Element " << 51 << " removed successfully.\n";

}

else

cout << "Element " << 51 << " is not contained in the tree.\n";

if (tree->contains(500) == true)

tree->remove(500);

else

cout << "Element " << 500 << " is not contained in the tree.\n";

tree->Queue();

tree->cout\_bft();

tree->Stack();

tree->cout\_dft();

tree->deleting\_tree();

}

**func\_n\_class.h**

#pragma once

#include <iomanip>

#include <iostream>

#include <locale>

using namespace std;

class Iterator

{

public:

virtual int next() = 0;

virtual bool has\_next() = 0;

};

class BinaryTree

{

int key;

class BinaryTree\* left;

class BinaryTree\* right;

class BinaryTree\* next;

friend class Tree;

friend class Iterator;

public:

BinaryTree()

{

left = NULL;

right = NULL;

next = NULL;

}

};

class Tree

{

class BinaryTree\* cur;

class BinaryTree\* root;

class BinaryTree\* go;

int size = 0;

Iterator\* bft\_it;

Iterator\* dft\_it;

friend class Iterator;

public:

void get\_root(int key)

{

if (root == NULL)

{

BinaryTree\* el = new BinaryTree;

el->key = key;

root = el;

root->left = NULL;

root->right = NULL;

}

}

void insert(int key)

{

cur = root;

if (key == cur->key)

{

cout << "Error! This key is already contained in the tree!\n";

return;

}

r\_n:

while (cur->left != NULL && cur->right != NULL)

{

if (key < cur->key)

{

cur = cur->left;

}

else if (key > cur->key)

{

cur = cur->right;

}

else

{

cout << "Error! This key is already contained in the tree!\n";

return;

}

}

if (key < cur->key)

{

if (cur->left == NULL)

{

BinaryTree\* el = new BinaryTree;

el->key = key;

el->left = NULL;

el->right = NULL;

cur->left = el;

}

else

{

cur = cur->left;

goto r\_n;

}

}

else if (key > cur->key)

{

if (cur->right == NULL)

{

BinaryTree\* el = new BinaryTree;

el->key = key;

el->left = NULL;

el->right = NULL;

cur->right = el;

}

else

{

cur = cur->right;

goto r\_n;

}

}

else

{

if (key == cur->key)

{

cout << "Error! This key is already contained in the tree!\n";

return;

}

}

}

bool contains(int key)

{

cur = root;

if (key == cur->key)

{

return true;

}

r\_n:

while (cur->left != NULL && cur->right != NULL)

{

if (key < cur->key)

{

cur = cur->left;

}

else if (key > cur->key)

{

cur = cur->right;

}

else

{

return true;

}

}

if (key < cur->key)

{

if (cur->left == NULL)

{

return 0;

}

else

{

cur = cur->left;

goto r\_n;

}

}

else if (key > cur->key)

{

if (cur->right == NULL)

{

return 0;

}

else

{

cur = cur->right;

goto r\_n;

}

}

else

{

if (key == cur->key)

{

return true;

}

}

}

void remove(int key)

{

cur = root;

while (true)

{

if (key < cur->key && cur->left->key != key)

{

cur = cur->left;

}

else if (key > cur->key && cur->right->key != key)

{

cur = cur->right;

}

else

{

if (cur->key == key)

{

if (cur->left == NULL && cur->right == NULL)

{

cur = NULL;

root = NULL;

}

else if (cur->left == NULL && cur->right != NULL)

{

cur = cur->right;

}

else if (cur->left != NULL && cur->right == NULL)

{

cur = cur->left;

}

else

{

BinaryTree\* pt;

pt = cur;

pt = pt->right;

if (pt->left == NULL)

{

cur->key = pt->key;

cur->right = pt->right;

return;

}

BinaryTree\* pt1;

pt1 = cur;

pt1 = pt1->right;

while (pt1->left != NULL)

pt1 = pt1->left;

while (pt->left != pt1)

pt = pt->left;

if (pt1->right == NULL)

{

cur->key = pt1->key;

pt->left = NULL;

}

else

{

cur->key = pt1->key;

pt->left = pt1->right;

}

pt = NULL;

pt1 = NULL;

delete pt, pt1;

}

return;

}

else if (cur->right != NULL && cur->right->key == key)

{

BinaryTree\* pt;

pt = cur;

pt = pt->right;

if (pt->left == NULL && pt->right == NULL)

{

cur->right = NULL;

}

else if (pt->left == NULL && pt->right != NULL)

{

cur->right = pt->right;

}

else if (pt->left != NULL && pt->right == NULL)

{

cur->right = pt->left;

}

else

{

pt = pt->right;

if (pt->left == NULL)

{

cur->right->key = pt->key;

cur->right->right = pt->right;

pt = NULL;

delete pt;

return;

}

BinaryTree\* pt1;

pt1 = pt;

while (pt1->left != NULL)

pt1 = pt1->left;

while (pt->left != pt1)

pt = pt->left;

if (pt1->right == NULL)

{

cur->right->key = pt1->key;

pt->left = NULL;

}

else

{

cur->right->key = pt1->key;

pt->left = pt1->right;

}

}

pt = NULL;

delete pt;

return;

}

else

{

BinaryTree\* pt;

pt = cur;

pt = pt->left;

if (pt->left == NULL && pt->right == NULL)

{

cur->left = NULL;

}

else if (pt->left == NULL && pt->right != NULL)

{

cur->left = pt->right;

}

else if (pt->left != NULL && pt->right == NULL)

{

cur->left = pt->left;

}

else

{

pt = pt->right;

if (pt->left == NULL)

{

cur->left->key = pt->key;

cur->left->right = pt->right;

pt = NULL;

delete pt;

return;

}

BinaryTree\* pt1;

pt1 = pt;

while (pt1->left != NULL)

pt1 = pt1->left;

while (pt->left != pt1)

pt = pt->left;

if (pt1->right == NULL)

{

cur->left->key = pt1->key;

pt->left = NULL;

}

else

{

cur->left->key = pt1->key;

pt->left = pt1->right;

}

pt = NULL;

pt1 = NULL;

delete pt, pt1;

}

return;

}

}

}

}

void Queue()

{

BinaryTree\* mrk;

BinaryTree\* fnsh;

go = root;

if (go == NULL)

{

return;

}

go->next = NULL;

fnsh = go;

mrk = go;

if (mrk->left != NULL)

{

fnsh->next = mrk->left;

fnsh = fnsh->next;

fnsh->next = NULL;

}

if (mrk->right != NULL)

{

fnsh->next = mrk->right;

fnsh = fnsh->next;

fnsh->next = NULL;

}

mrk = mrk->next;

while (true)

{

if (mrk->left != NULL)

{

fnsh->next = mrk->left;

fnsh = fnsh->next;

fnsh->next = NULL;

}

if (mrk->right != NULL)

{

fnsh->next = mrk->right;

fnsh = fnsh->next;

fnsh->next = NULL;

}

if (mrk->left == NULL && mrk->right == NULL && mrk->next == NULL)

{

break;

}

mrk = mrk->next;

}

mrk = go;

while (mrk->next != NULL)

{

size++;

mrk = mrk->next;

}

mrk = NULL;

fnsh = NULL;

delete mrk;

delete fnsh;

return;

}

void Stack()

{

BinaryTree\* stack;

BinaryTree\* parent;

BinaryTree\* fnsh;

BinaryTree\* mrk;

stack = NULL;

go = root;

fnsh = go;

mrk = root;

fnsh->next = NULL;

if (mrk->right != NULL)

{

stack = mrk->right;

stack->next = NULL;

}

if (mrk->left != NULL)

{

mrk = mrk->left;

fnsh->next = mrk;

fnsh = fnsh->next;

}

while (true)

{

if (mrk->right != NULL)

{

if (stack == NULL)

{

stack = mrk->right;

stack->next = NULL;

}

else

{

parent = mrk->right;

parent->next = stack;

stack = parent;

}

}

if (mrk->left != NULL)

{

mrk = mrk->left;

fnsh->next = mrk;

fnsh = fnsh->next;

}

else if (stack != NULL)

{

mrk = stack;

fnsh->next = mrk;

fnsh = fnsh->next;

stack = stack->next;

}

else

{

break;

}

}

stack = NULL;

parent = NULL;

fnsh = NULL;

mrk = NULL;

delete stack;

delete parent;

delete fnsh;

delete mrk;

}

void deleting\_tree()

{

Queue();

while (go != NULL)

{

BinaryTree\* dltr;

dltr = go;

dltr->left = NULL;

dltr->right = NULL;

go = go->next;

delete dltr;

}

cout << "The tree was cleared." << endl;

}

void cout\_bft()

{

cout << "\nbft\_iteration:\n";

Iterator\* iter;

iter = bft\_creator();

cout << "|" << go->key << "|";

while (iter->has\_next())

cout << iter->next() << "|";

cout << endl;

iter = NULL;

delete iter;

}

void cout\_dft()

{

cout << "\ndft\_iteration:\n";

Iterator\* iter;

iter = dft\_iterator();

cout << "|" << go->key << "|";

while (size != 0 && iter->has\_next())

{

cout << iter->next() << "|";

size--;

}

cout << endl;

iter = NULL;

delete iter;

}

class bft\_iter : public Iterator

{

class BinaryTree\* begin;

public:

bft\_iter(BinaryTree\* go)

{

begin = go;

}

int next() override

{

if (!has\_next())

throw out\_of\_range("no more elements\n");

if (begin->next != NULL)

{

begin = begin->next;

}

return begin->key;

}

bool has\_next() override

{

if (begin->next == NULL)

{

return false;

}

else return true;

}

~bft\_iter()

{

delete begin;

}

};

class dft\_iter : public Iterator

{

class BinaryTree\* begin;

public:

dft\_iter(BinaryTree\* go)

{

begin = go;

}

int next() override

{

if (!has\_next())

throw out\_of\_range("no more elements\n");

if (begin->next != NULL)

{

begin = begin->next;

}

return begin->key;

}

bool has\_next() override

{

if (begin->next == NULL)

{

return false;

}

else return true;

}

dft\_iter()

{

delete begin;

}

};

Iterator\* bft\_creator()

{

bft\_it = new bft\_iter(go);

return bft\_it;

}

Iterator\* dft\_iterator()

{

dft\_it = new dft\_iter(go);

return dft\_it;

}

Tree()

{

root = NULL;

cur = NULL;

go = NULL;

}

~Tree()

{

delete root;

delete cur;

delete go;

delete bft\_it;

delete dft\_it;

}

};

**UnitTest1.cpp**

#include "pch.h"

#include "CppUnitTest.h"

#include <stdexcept>

#include "../Lab3Aistrd\_Koryagin/func\_n\_class.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest3

{

TEST\_CLASS(UnitTest3)

{

public:

TEST\_METHOD(insert)

{

Tree\* tree = new Tree();

Tree\* tree2 = new Tree();

tree->get\_root(10);

tree->insert(5);

tree->insert(15);

tree->insert(100);

tree->insert(20);

tree2->get\_root(10);

tree2->insert(5);

tree2->insert(15);

tree2->insert(100);

tree2->insert(20);

tree->Queue();

tree2->Queue();

Iterator\* iter;

iter = tree->bft\_creator();

Iterator\* iter2;

iter2 = tree2->bft\_creator();

for (int i = 0; i < 4; i++)

{

Assert::AreEqual(iter2->next(), iter->next());

}

tree->deleting\_tree();

tree2->deleting\_tree();

}

TEST\_METHOD(remove)

{

Tree\* tree = new Tree();

Tree\* tree2 = new Tree();

tree->get\_root(10);

tree->insert(5);

tree->insert(15);

tree->insert(100);

tree->insert(20);

tree->remove(20);

tree2->get\_root(10);

tree2->insert(5);

tree2->insert(15);

tree2->insert(100);

tree->Queue();

tree2->Queue();

Iterator\* iter;

iter = tree->bft\_creator();

Iterator\* iter2;

iter2 = tree2->bft\_creator();

for (int i = 0; i < 3; i++)

{

Assert::AreEqual(iter2->next(), iter->next());

}

tree->deleting\_tree();

tree2->deleting\_tree();

}

TEST\_METHOD(contains)

{

Tree\* tree = new Tree();

tree->get\_root(10);

tree->insert(5);

tree->insert(15);

tree->insert(100);

tree->insert(20);

Assert::AreEqual(true, tree->contains(15));

}

};

}