**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»**

**Тема: «Двоичные деревья»**

**Вариант 19(1)**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9302 |  | Плюснина Е.Ю. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

# Постановка задачи

Реализовать двоичное дерево поиска.

Функции:

bool contains(int) - поиск элемента в дереве по ключу.

void insert(int) - добавление элемента в дерево по ключу. Должен работать за O(logN).

void remove(int) - удаление элемента дерева по ключу.

Iterator create\_dft\_iterator() - создание итератора, реализующего один из методов обхода в глубину (depth-first traverse).

Iterator create\_bft\_iterator() - создание итератора, реализующего методы обхода в ширину (breadth-first traverse).

# Описание реализуемых классов и оценка временной сложности

Iterator – класс итератора

Node – класс Node

BinaryTree – класс двоичного дерева

Stack – класс стека

Node\_S – вложенный класс стека

Queue – класс очереди

Node\_Q – вложенный класс очереди

Node\* left\_branch; - указатель на следующий левый элемент дерева.

Node\* right\_branch; - указатель на следующий правый элемент дерева.

Node\* previous\_branch; - указатель на предыдущий элемент дерева.

Node\* next\_branch; - указатель на следующий элемент дерева.

int data; - данные в элементе дерева.

Node\_Q\* first\_Q; - указатель на первый элемент очереди.

Node\_Q\* last\_Q; - указатель на последний элемент очереди.

Node\* elem; - указатель на элемент дерева.

Node\_Q\* next; - указатель на следующий элемент очереди.

Node\_S\* first\_S; - указатель на первый элемент стека.

Node\_S\* last\_S; - указатель на последний элемент стека.

Node\* elem; - указатель на элемент дерева.

Node\_S\* next; - указатель на следующий элемент стека.

Node\* begin; - указатель на корень дерева.

Queue\* Queue\_elem; - очередь для обхода в ширину.

Stack\* Stack\_elem; - стек для обхода в глубину.

bool BinaryTree::contains(int FindElement) - поиск элемента в дереве по ключу. Оценка временной сложности: O(logN).

void BinaryTree::insert(int FindElement) - добавление элемента в дерево по ключу. Должен работать за O(logN). Оценка временной сложности: O(logN).

void BinaryTree::remove(int FindElement) - удаление элемента дерева по ключу.

Iterator\* BinaryTree::create\_dft\_iterator()- создание итератора, реализующего один из методов обхода в глубину (depth-first traverse). Оценка временной сложности: O(logN).

Iterator\* BinaryTree::create\_bft\_iterator()- создание итератора, реализующего методы обхода в ширину (breadth-first traverse). Оценка временной сложности: О(1)

int next() override; - переход к следующему элементу. Оценка временной сложности: О(1)

bool has\_next() override; - проверка на наличие следующего элемента. Оценка временной сложности: О(1)

bool Stack::IsEmpty() - проверка на пустоту. Оценка временной сложности: О(1)

void Stack::push(Node\* new\_elem) - добавление элемента в стек. Оценка временной сложности: О(1)

Node\* Stack::pop() – удаление первого элемента стека. Оценка временной сложности: О(1)

Node\* Stack::front()- возвращение первого элемента стека. Оценка временной сложности: О(1)

bool Queue::IsEmpty() - проверка на пустоту. Оценка временной сложности: О(1)

void Queue::push(Node\* new\_elem) - добавление элемента в очередь. Оценка временной сложности: О(1)

Node\* Queue::pop()- удаление первого элемента очереди. Оценка временной сложности: О(1)

Node\* Queue::front()- возвращение первого элемента очереди. Оценка временной сложности: О(1)

# Описание реализованных unit-тестов

TEST\_METHOD(check\_contains\_Find) - проверка функции contains, искомый элемент существует.

TEST\_METHOD(check\_contains\_NotFind) - проверка функции contains, искомый элемент НЕ существует.

TEST\_METHOD(check\_insert) - проверка функции insert.

TEST\_METHOD(check\_insert\_IsEmpty) - проверка функции insert при пустом дереве.

TEST\_METHOD(check\_insert\_IsNotEmpty) - проверка функции insert при НЕ пустом дереве.

TEST\_METHOD(check\_remove) - проверка функции remove.

TEST\_METHOD(check\_remove\_begin) - проверка функции remove, удаление начала дерева.

TEST\_METHOD(check\_remove\_IsEmpty) - проверка функции remove при пустом дереве.

TEST\_METHOD(check\_remove\_IsNotEmpty) - проверка функции remove при НЕ пустом дереве.

TEST\_METHOD(check\_remove\_impossible) - проверка функции remove, искомый (удаляемый) элемент НЕ существует.

TEST\_METHOD(dft\_iterator) - проверка итератора обхода в глубину.

TEST\_METHOD(bft\_iterator) - проверка итератора обхода в ширину.

# Листинг

**Iterator.h**

#pragma once

class Iterator {

public:

virtual int next() = 0;

virtual bool has\_next() = 0;

};

**Node.h**

#pragma once

class Node {

public:

Node\* left\_branch;

Node\* right\_branch;

Node\* previous\_branch;

Node\* next\_branch;

int data;

Node(int data, Node\* left\_branch = nullptr, Node\* right\_branch = nullptr, Node\* previous\_branch = nullptr, Node\* next\_branch = nullptr) {

this->left\_branch = left\_branch;

this->right\_branch = right\_branch;

this->previous\_branch = previous\_branch;

this->next\_branch = next\_branch;

this->data = data;

};

~Node() {};

};

**Stack.h**

#pragma once

#include "Node.h"

class Stack {

private:

class Node\_S {

public:

Node\_S(Node\* elem, Node\_S\* next = nullptr) {

this->elem = elem;

this->next = next;

};

~Node\_S() {};

Node\* elem;

Node\_S\* next;

};

Node\_S\* first\_S;

Node\_S\* last\_S;

public:

Stack();

bool IsEmpty();

void push(Node\*);

Node\* pop();

Node\* front();

~Stack();

};

**Queue.h**

#pragma once

#include "Node.h"

class Queue {

private:

class Node\_Q {

public:

Node\_Q(Node\* elem, Node\_Q\* next = nullptr) {

this->elem = elem;

this->next = next;

};

~Node\_Q() {};

Node\* elem;

Node\_Q\* next;

};

Node\_Q\* first\_Q;

Node\_Q\* last\_Q;

public:

Queue();

bool IsEmpty();

void push(Node\*);

Node\* pop();

Node\* front();

~Queue();

};

**BinaryTree.h**

#pragma once

#include "Iterator.h"

#include "Node.h"

#include "Queue.h"

#include "Stack.h"

class BinaryTree

{

private:

Node\* begin;

public:

BinaryTree();

bool contains(int);

void insert(int);

void remove(int);

~BinaryTree();

Iterator\* create\_dft\_iterator();

Iterator\* create\_bft\_iterator();

class dft\_Iterator : public Iterator {

public:

dft\_Iterator(Node\* begin\_dft);

int next() override;

bool has\_next() override;

private:

Stack\* Stack\_elem;

};

class bft\_Iterator : public Iterator {

public:

bft\_Iterator(Node\* begin\_bft);

int next() override;

bool has\_next() override;

private:

Queue\* Queue\_elem;

};

};

**Iterator.cpp**

#include "BinaryTree.h"

#include "Iterator.h"

#include "Node.h"

#include "Queue.h"

#include "Stack.h"

#include <stdexcept>

using namespace std;

Iterator\* BinaryTree::create\_bft\_iterator() {

return new bft\_Iterator(begin);

}

BinaryTree::bft\_Iterator::bft\_Iterator(Node\* begin\_bft) {

Queue\_elem = new Queue();

Queue\_elem->push(begin\_bft);

}

int BinaryTree::bft\_Iterator::next() {

Node\* this\_branch;

this\_branch = Queue\_elem->pop();

if (this\_branch->left\_branch != nullptr) {

Queue\_elem->push(this\_branch->left\_branch);

}

if (this\_branch->right\_branch != nullptr) {

Queue\_elem->push(this\_branch->right\_branch);

}

return this\_branch->data;

}

bool BinaryTree::bft\_Iterator::has\_next() {

return !(Queue\_elem->IsEmpty());

}

Iterator\* BinaryTree::create\_dft\_iterator() {

return new dft\_Iterator(begin);

}

BinaryTree::dft\_Iterator::dft\_Iterator(Node\* begin\_dft) {

Stack\_elem = new Stack();

Stack\_elem->push(begin\_dft);

}

int BinaryTree::dft\_Iterator::next() {

Node\* this\_branch;

this\_branch = Stack\_elem->pop();

if (this\_branch->left\_branch != nullptr) {

Stack\_elem->push(this\_branch->left\_branch);

}

if (this\_branch->right\_branch != nullptr) {

Stack\_elem->push(this\_branch->right\_branch);

}

return this\_branch->data;

}

bool BinaryTree::dft\_Iterator::has\_next() {

return !(Stack\_elem->IsEmpty());

}

**Stack.cpp**

#include "Stack.h"

#include <stdexcept>

using namespace std;

Stack::Stack() {

first\_S = nullptr;

last\_S = nullptr;

}

bool Stack::IsEmpty() {

return (first\_S == nullptr);

}

void Stack::push(Node\* new\_elem) {

first\_S = new Node\_S(new\_elem, first\_S);

}

Node\* Stack::pop() {

Node\* a = first\_S->elem;

first\_S = first\_S->next;

return a;

}

Node\* Stack::front() {

return first\_S->elem;

}

Stack::~Stack() {

while (!IsEmpty()) {

pop();

}

}

**Queue.cpp**

#include "Queue.h"

#include <stdexcept>

using namespace std;

Queue::Queue() {

first\_Q = nullptr;

last\_Q = nullptr;

}

bool Queue::IsEmpty() {

return (first\_Q == nullptr);

}

void Queue::push(Node\* new\_elem) {

if (!IsEmpty()) {

last\_Q->next = new Node\_Q(new\_elem);

last\_Q = last\_Q->next;

}

else {

first\_Q = new Node\_Q(new\_elem);

last\_Q = first\_Q;

}

}

Node\* Queue::pop() {

Node\* a = first\_Q->elem;

first\_Q = first\_Q->next;

return a;

}

Node\* Queue::front() {

return first\_Q->elem;

}

Queue::~Queue() {

while (first\_Q != nullptr) {

Node\_Q\* a = first\_Q;

first\_Q = first\_Q->next;

delete a;

}

}

**BinaryTree.cpp**

#include "BinaryTree.h"

#include "Iterator.h"

#include "Node.h"

#include "Queue.h"

#include "Stack.h"

#include <stdexcept>

using namespace std;

BinaryTree::BinaryTree() {

begin = nullptr;

}

bool BinaryTree::contains(int FindElement) {

Node\* this\_branch = begin;

while (this\_branch != nullptr) {

if (FindElement < this\_branch->data) {

this\_branch = this\_branch->left\_branch;

}

else

if (FindElement > this\_branch->data) {

this\_branch = this\_branch->right\_branch;

}

else

if (FindElement == this\_branch->data) {

return true;

}

}

return false;

}

void BinaryTree::insert(int FindElement) {

if (begin == nullptr) {

begin = new Node(FindElement);

}

else {

Node\* this\_branch = begin;

while (!((this\_branch->left\_branch == nullptr) && !(FindElement > this\_branch->data)) && !((this\_branch->right\_branch == nullptr) && !(FindElement < this\_branch->data))) {

if (this\_branch == nullptr) {

break;

}

if (!(FindElement > this\_branch->data)) {

this\_branch = this\_branch->left\_branch;

}

else {

if (!(FindElement < this\_branch->data)) {

this\_branch = this\_branch->right\_branch;

}

}

}

if (this\_branch != nullptr) {

if ((FindElement < this\_branch->data) && (this\_branch->left\_branch == nullptr)) {

this\_branch->left\_branch = new Node(FindElement, nullptr, nullptr, this\_branch, nullptr);

}

if ((FindElement > this\_branch->data) && (this\_branch->right\_branch == nullptr)) {

this\_branch->right\_branch = new Node(FindElement, nullptr, nullptr, this\_branch, nullptr);

}

}

}

}

void BinaryTree::remove(int FindElement) {

if (begin == nullptr) {

throw out\_of\_range("Tree is empty");

}

else {

if (begin->left\_branch == nullptr) {

if (begin->right\_branch == nullptr) {

if (begin->data == FindElement) {

begin = nullptr;

}

else {

throw out\_of\_range("Impossible to find");

}

}

}

else {

Node\* this\_branch = begin;

Node\* next\_branch = nullptr;

while (this\_branch != nullptr) {

if (this\_branch->data == FindElement) {

if (this\_branch->left\_branch == nullptr) {

if (this\_branch->right\_branch == nullptr) {

if ((this\_branch->previous\_branch)->left\_branch == this\_branch) {

(this\_branch->previous\_branch)->left\_branch = nullptr;

}

else {

(this\_branch->previous\_branch)->right\_branch = nullptr;

}

delete this\_branch;

}

}

else {

if (this\_branch->left\_branch != nullptr) {

if (this\_branch->right\_branch == nullptr) {

next\_branch = this\_branch->left\_branch;

this\_branch->left\_branch = next\_branch->left\_branch;

this\_branch->right\_branch = next\_branch->right\_branch;

this\_branch->data = next\_branch->data;

if (this\_branch->left\_branch != nullptr) {

this\_branch->left\_branch->previous\_branch = this\_branch;

}

if (this\_branch->right\_branch != nullptr) {

this\_branch->right\_branch->previous\_branch = this\_branch;

}

delete next\_branch;

}

}

else {

if (this\_branch->left\_branch == nullptr) {

if (this\_branch->right\_branch != nullptr) {

next\_branch = this\_branch->right\_branch;

this\_branch->left\_branch = next\_branch->left\_branch;

this\_branch->right\_branch = next\_branch->right\_branch;

this\_branch->data = next\_branch->data;

if (this\_branch->left\_branch != nullptr)

this\_branch->left\_branch->previous\_branch = this\_branch;

if (this\_branch->right\_branch != nullptr)

this\_branch->right\_branch->previous\_branch = this\_branch;

delete next\_branch;

}

}

else {

if (this\_branch->left\_branch != nullptr) {

if (this\_branch->right\_branch != nullptr) {

next\_branch = this\_branch->right\_branch;

if (next\_branch->left\_branch == nullptr) {

this\_branch->right\_branch = next\_branch->right\_branch;

if (next\_branch->right\_branch != nullptr) {

(next\_branch->right\_branch)->previous\_branch = this\_branch;

}

}

else {

while ((next\_branch->left\_branch != nullptr) && (next\_branch != nullptr)) {

next\_branch = next\_branch->left\_branch;

}

(next\_branch->previous\_branch)->left\_branch = next\_branch->right\_branch;

if (next\_branch->right\_branch != nullptr) {

next\_branch->right\_branch->previous\_branch = next\_branch->previous\_branch;

}

}

this\_branch->data = next\_branch->data;

delete next\_branch;

}

}

}

return;

}

}

}

else {

if (this\_branch->data > FindElement) {

this\_branch = this\_branch->left\_branch;

}

else {

this\_branch = this\_branch->right\_branch;

}

}

}

throw out\_of\_range("Impossible to find");

}

}

}

BinaryTree::~BinaryTree() {

while (begin != nullptr) {

remove(begin->data);

}

delete begin;

}

# Вывод

Я изучила реализацию двоичных деревьев поиска.