**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: «Изучение и реализация различных алгоритмов для работы с двоичным деревом»**

**Вариант 1**

Студент гр. 9302 \_\_\_\_\_\_\_\_\_\_\_\_ Баязитов О.О.

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

Санкт-Петербург

2020

# Постановка задачи

Реализовать алгоритмы поиска, удаления и вставки. Так же реализовать итератор, для обхода дерева в глубину и ширину.

**Описание реализуемых алгоритмов**

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| bool contains(int); | Поиск элемента в дереве по ключу |
| void insert(int); | Добавление элемента в дерево по ключу |
| void remove(int); | Удаление элемента дерева по ключу |
| Iterator\* CreateWidthIterator () | Итератор для обхода в ширину |
| Iterator\* CreateDepthIterator () | Итератор для обхода в глубину |
| int next() | Возвращает следующее значение итератора |
| bool has\_next() | Проверяет наличие потомков узла |

# Оценка сложности работы алгоритмов

|  |  |
| --- | --- |
| **Функция** | **Оценка временной сложности** |
| bool contains(int); | O(logn) |
| void insert(int); | O(logn) |
| void remove(int); | O(logn) |
| Iterator\* CreateWidthIterator () | O(1) |
| Iterator\* CreateDepthIterator () | O(1) |
| int next() | O(1) |
| bool has\_next() | O(1) |

# Результат работы программы

# 

# Листинг

**Tree.h**

#ifndef Tree\_H

#define Tree\_H

#include"Node.h"

#include"Queue.h"

#include"Stack.h"

#include "Iterator.h"

class Tree

{

private:

Node\* Root;

public:

Tree();

~Tree();

bool contains(int);

void insert(int);

void remove(int);

Iterator\* CreateWidthIterator()

{

return new WidthIterator(Root);

}

Iterator\* CreateDepthIterator()

{

return new DepthIterator(Root);

}

class WidthIterator : public Iterator

{

private:

Node\* current;

queue Queue;

public:

WidthIterator(Node\* root);

~WidthIterator();

int next();

bool has\_next();

};

class DepthIterator : public Iterator

{

private:

Node\* current;

stack Stack;

bool check = true;

public:

DepthIterator(Node\* root);

~DepthIterator();

int next();

bool has\_next();

};

};

#endif

**Node.h**

#ifndef Node\_H

#define Node\_H

class Node

{

public:

int value;

Node\* Left;

Node\* Right;

Node\* parent;

Node(int value, Node\* parent, Node\* Left = nullptr, Node\* Right = nullptr)

{

this->parent = parent;

this->value = value;

this->Left = Left;

this->Right = Right;

};

~Node()

{

int key;

Left = nullptr;

Right = nullptr;

parent = nullptr;

}

};

#endif

**Stack.h**

#ifndef Stack\_H

#define Stack\_H

#include"Tree.h"

#include"Node.h"

class stack

{

private:

class stackNode {

public:

Node\* date;

stackNode\* next;

stackNode(Node\* date = nullptr, stackNode\* next = nullptr)

{

this->date = date;

this->next = next;

};

~stackNode()

{

}

};

public:

stackNode\* head;

stackNode\* tail;

unsigned int size;

stack() {

head = nullptr;

tail = nullptr;

size = 0;

}

~stack() {

clear();

}

void clear() {

while (size != 0)

{

pop\_front();

}

}

void push(Node\* date) {

if (size == 0) {

head = new stackNode(date);

tail = head;

}

else {

head = new stackNode(date, head);

}

size++;

}

void pop\_front() {

if (size == 0) {

return;

}

if (size == 1) {

delete head;

reset\_list();

}

else {

stackNode\* current = head;

head = head->next;

delete current;

}

size--;

}

void pop\_back() {

if (size == 0) return;

if (size == 1) {

delete head;

reset\_list();

}

else {

stackNode\* current = head;

while (current->next != tail) {

current = current->next;

}

current->next = nullptr;

delete tail;

tail = current;

}

size--;

}

void reset\_list()

{

head = nullptr;

tail = nullptr;

}

};

#endif

Queue.h

#ifndef Queue\_H

#define Queue\_H

#include"Tree.h"

#include"Node.h"

class queue

{

private:

class QNode

{

public:

Node\* Element;

QNode\* next;

QNode(Node\* element = nullptr, QNode\* next = nullptr)

{

this->Element = element;this->next = next;

};

~QNode(){}

};

public:

QNode\* head;

QNode\* tail;

unsigned int size;

queue()

{

tail = nullptr;

head = nullptr;

size = 0;

}

~queue()

{

clear();

}

void clear()

{

while (size != 0)

{

pop();

}

}

void push(Node\* date)

{

if (size == 0)

{

head = new QNode(date);

tail = head;

}

else

{

tail->next = new QNode(date);

tail = tail->next;

}

size++;

}

void pop()

{

if (size == 0)

{

return;

}

if (size == 1)

{

delete head;

reset\_list();

}

else

{

QNode\* current = head;

head = head->next;

delete current;

}

size--;

}

void reset\_list()

{

head = nullptr;

tail = nullptr;

}

};

#endif

**Iterator.h**

#pragma once

class Iterator

{

public:

virtual int next() = 0;

virtual bool has\_next() = 0;

};

**Tree.cpp**

#include"Tree.h"

#include<iostream>

#include"stack.h"

Tree::Tree()

{

Root = nullptr;

}

Tree::~Tree(){}

void Tree::insert(int val)

{

if (Root == NULL)

{

Root = new Node(val, NULL, NULL);

}

else

{

Node\* current;

current = Root;

while(true)

{

if (val <= current->value)

{

if (current->Left == NULL)

{

current->Left = new Node(val, current);

break;

}

current = current->Left;

}

else {

if (current->Right == NULL)

{

current->Right = new Node(val, current);

break;

}

current = current->Right;

}

}

}

}

void Tree::remove(int val)

{

if (!contains(val))

{

throw "element is absent";

}

else

{

Node\* current;

current = Root;

if (current->value == val) {

if (current->Left == NULL && current->Right == NULL)

{

delete current;

return;

}

if (current->Left != NULL && current->Right == NULL)

{

current->Left->parent = current->parent;

Root = current->Left;

delete current;

return;

}

if (current->Left == NULL && current->Right != NULL)

{

current->Right->parent = current->parent;

Root = current->Right;

delete current;

return;

}

else

{

Node\* supCurrent;

supCurrent = current->Right;

if (supCurrent->Left != NULL)

{

while (supCurrent->Left != NULL)

{

supCurrent = supCurrent->Left;

}

supCurrent->parent->Left = NULL;

}

else

{

supCurrent->parent->Right = supCurrent->Right;

}

current->Left->parent = supCurrent;

supCurrent->Left = current->Left;

supCurrent->parent = current->parent;

while (supCurrent->Right != NULL)

{

supCurrent = supCurrent->Right;

}

current->Right->parent = supCurrent;

supCurrent->Right = current->Right;

return;

}

}

while (true)

{

if (val <= current->value)

{

current = current->Left;

if (current->parent->Left->value == val)

{

if (current->Left == NULL && current->Right == NULL)

{

current->parent->Left = NULL;

delete current;

break;

}

if (current->Left != NULL && current->Right == NULL)

{

current->parent->Left = current->Left;

current->Left->parent = current->parent;

delete current;

break;

}

if (current->Left == NULL && current->Right != NULL)

{

current->parent->Left = current->Right;

current->Right->parent = current->parent;

delete current;

break;

}

else

{

Node\* supCurrent;

supCurrent = current->Right;

if (supCurrent->Left != NULL)

{

while (supCurrent->Left != NULL)

{

supCurrent = supCurrent->Left;

}

supCurrent->parent->Left = NULL;

}

else

{

supCurrent->parent->Right = supCurrent->Right;

}

current->parent->Left = supCurrent;

current->Left->parent = supCurrent;

supCurrent->Left = current->Left;

supCurrent->parent = current->parent;

while (supCurrent->Right != NULL)

{

supCurrent = supCurrent->Right;

}

current->Right->parent = supCurrent;

supCurrent->Right = current->Right;

delete current;

break;

}

}

}

else

{

current = current->Right;

if (current->parent->Right->value == val)

{

if (current->Left == NULL && current->Right == NULL)

{

current->parent->Right = NULL;

delete current;

break;

}

if (current->Left != NULL && current->Right == NULL)

{

current->parent->Right = current->Left;

current->Left->parent = current->parent;

delete current;

break;

}

if (current->Left == NULL && current->Right != NULL)

{

current->parent->Right = current->Right;

current->Right->parent = current->parent;

delete current;

break;

}

else

{

Node\* supCurrent;

supCurrent = current->Right;

if (supCurrent->Left != NULL)

{

while (supCurrent->Left != NULL)

{

supCurrent = supCurrent->Left;

}

supCurrent->parent->Left = NULL;

}

else

{

supCurrent->parent->Right = supCurrent->Right;

}

current->parent->Right = supCurrent;

current->Left->parent = supCurrent;

supCurrent->Left = current->Left;

supCurrent->parent = current->parent;

while (supCurrent->Right != NULL)

{

supCurrent = supCurrent->Right;

}

supCurrent->Right = current->Right;

current->Right->parent = supCurrent;

delete current;

break;

}

}

}

}

}

}

bool Tree::contains(int val)

{

if (Root == NULL)

{

return 0;

}

else {

Node\* current;

current = Root;

for (;;) {

if (val <= current->value) {

if (current->value == val)

{

return 1;

}

if (current->Left == NULL)

{

return 0;

}

current = current->Left;

}

else {

if (current->value == val)

{

return 1;

}

if (current->Right == NULL)

{

return 0;

}

current = current->Right;

}

}

}

}

Tree::DepthIterator::DepthIterator(Node\* root)

{

current = root;

if (current != nullptr)

{

Stack.push(current);

}

else {

throw "Tree is empty";

}

}

Tree::DepthIterator::~DepthIterator(){}

int Tree::DepthIterator::next()

{

int tmp = current->value;

if (!has\_next()) {

throw "Tree is empty";

}

if (current->Right != nullptr)

{

if (check) {

Stack.pop\_front();

check = false;

}

Stack.push(current->Right);

}

if (current->Left != nullptr)

{

current = current->Left;

}

else

{

if (check) {

Stack.pop\_front();

}

if (has\_next()) {

current = Stack.head->date;

}

check = true;

}

return tmp;

}

bool Tree::DepthIterator::has\_next()

{

if (current->Left == nullptr && current->Right == nullptr && Stack.size == 0) return false;

else return true;

}

Tree::WidthIterator::WidthIterator(Node\* root)

{

current = root;

if (current != nullptr) {

Queue.push(current);

}

else {

throw "Tree is empty";

}

}

Tree::WidthIterator::~WidthIterator()

{

delete current;

Queue.clear();

}

int Tree::WidthIterator::next()

{

if (!has\_next())

{

throw "Tree is empty";

}

current = Queue.head->Element;

Queue.pop();

if (current->Left != nullptr)

{

Queue.push(current->Left);

}

if (current->Right != nullptr)

{

Queue.push(current->Right);

}

return current->value;

}

bool Tree::WidthIterator::has\_next()

{

if (current->Left == nullptr && current->Right == nullptr && Queue.size == 0) return false;

else return true;

}

**Main.cpp**

#include "Tree.h"

#include<iostream>

int main()

{

setlocale(LC\_ALL, "Russian");

Iterator\* I1;

Tree\* Tree1 = new Tree;

Tree1->insert(5);

Tree1->insert(3);

Tree1->insert(4);

Tree1->insert(7);

Tree1->insert(6);

Tree1->insert(9);

Tree1->insert(11);

Tree1->insert(10);

Tree1->remove(11);

Tree1->remove(10);

I1 = Tree1->CreateWidthIterator();

std::cout << "Вывод итератора в ширину:";

for (int i = 0; i < 6; i++)

{

std::cout << I1->next() << " ";

}

std::cout << std::endl;

I1 = Tree1->CreateDepthIterator();

std::cout << "Вывод итератора в длинну: ";

for (int i = 0; i < 6; i++)

{

std::cout << I1->next() << " ";

}

return 0;

}