**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 \_\_\_\_\_\_\_\_\_\_\_\_ Квитко Д.В.

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

Санкт-Петербург

2020

# Постановка задачи и описание реализуемых класса и алгоритмов

Реализовать алгоритмы поиска, удаления и вставки. Для их реализации создаются 2 класса.

Class Node с помощью него мы создаем узел дерева и ссылку разветвлением на правое и левое поддерево

Class BinaryTree этот класс создает с помощью Class Node дерево бинарного поиска.

Для этого используем следующие функции:

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Оценка временной сложности** |
| bool contains(int); | Поиск элемента в дереве по ключу | O(logn) |
| void insert(int); | Добавление элемента в дерево по ключу | O(logn) |
| void remove(int); | Удаление элемента дерева по ключу | O(logn) |
| Iterator\* create\_bft\_iterator() | Создает итератор, реализующий обход в ширину | O(1) |
| Iterator\* create\_dft\_iterator() | Создает итератор, реализующий обход в глубину | O(1) |
| int next() override | Возвращает следующее значение итератора | O(1) |
| bool has\_next() override | Проверяет наличие дочерних элементов у узла | O(1) |

# Описание реализованных unit-тестов

|  |  |
| --- | --- |
| Имя теста | Описание |
| BinarTree\_Test1 | Вставка в пустое дерева и создание итератора для обхода в глубину |
| BinarTree\_Test2 | Создание итератора в пустом дереве, проверка на возвращение исключения |
| BinarTree\_Test3 | Проверка корректной работы итератора обхода в ширину |
| BinarTree\_Test4 | Проверка корректной работы итератора обхода в глубину |
| BinarTree\_Test5 | Проверка поиска существующего элемента в дереве |
| BinarTree\_Test6 | Проверка поиска несуществующего элемента в непустом дереве |
| BinarTree\_Test7 | Поиск элемента в пустом дереве, проверка на исключение |
| BinarTree\_Test8 | Удаление элемента в дереве |
| BinarTree\_Test9 | Удаление несуществующего элемента в дереве, проверка на исключение |
| BinarTree\_Test10 | Удаление элемента в пустом дереве, проверка на исключение |

# Код программы

**Iterator.h**

class Iterator {

public:

virtual int next() = 0;

virtual bool has\_next() = 0;

};

**Queue.h**

#ifndef queue\_H

#define queue\_H

#include"Tree.h"

#include"TreeNode.h"

class queue

{

private:

class queueNode {

public:

Node\* date;

queueNode\* next;

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

{

this->date = date;

this->next = next;

};

~queueNode()

{

}

};

public:

queueNode\* head;

queueNode\* 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 queueNode(date);

tail = head;

}

else {

tail->next = new queueNode(date);

tail = tail->next;

}

size++;

}

void pop() {

if (size == 0) {

return;

}

if (size == 1) {

delete head;

reset\_list();

}

else {

queueNode\* current = head;

head = head->next;

delete current;

}

size--;

}

void reset\_list()

{

head = nullptr;

tail = nullptr;

}

};

#endif

**Stack.h**

#ifndef stack\_H

#define stack\_H

#include"Tree.h"

#include"TreeNode.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

**Tree.h**

#ifndef Tree\_H

#define Tree\_H

#include "iterator.h"

#include"stack.h"

#include"TreeNode.h"

#include"queue.h"

#include <stdexcept>

using namespace std;

class BinaryTree

{

private:

Node\* Root;

public:

BinaryTree();

~BinaryTree();

void preOrder(Node\*);

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\* create\_bft\_iterator() {

return new bft\_iterator(Root);

}

Iterator\* create\_dft\_iterator() {

return new dft\_iterator(Root);

}

class bft\_iterator : public Iterator {

private:

Node\* cur;

queue Queue;

public:

bft\_iterator(Node\* root);

~bft\_iterator();

int next();

bool has\_next();

};

class dft\_iterator : public Iterator {

private:

Node\* cur;

stack Stack;

bool flag = true;

public:

dft\_iterator(Node\* root);

~dft\_iterator();

int next();

bool has\_next();

};

};

#endif

**TreeNode.h**

#ifndef TreeNode\_H

#define TreeNode\_H

class Node {

public:

int key;

Node\* Lnext;

Node\* Rnext;

Node\* parent;

Node(int key, Node\* parent, Node\* Lnext = nullptr, Node\* Rnext = nullptr)

{

this->parent = parent;

this->key = key;

this->Lnext = Lnext;

this->Rnext = Rnext;

};

~Node()

{

int key;

Lnext = nullptr;

Rnext = nullptr;

parent = nullptr;

}

};

#endif

**Tree.cpp**

#include"Tree.h"

#include"stack.h"

#include <stdexcept>

#include<iostream>

BinaryTree::BinaryTree()

{

Root = nullptr;

}

BinaryTree::~BinaryTree()

{

preOrder(Root);

}

void BinaryTree::insert(int key)

{

if (Root == NULL) {

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

}

else {

Node\* iter;

iter = Root;

for (;;) {

if (key <= iter->key) {

if (iter->Lnext == NULL) {

iter->Lnext = new Node(key,iter);

break;

}

iter = iter->Lnext;

}

else{

if (iter->Rnext == NULL) {

iter->Rnext = new Node(key, iter);

break;

}

iter = iter->Rnext;

}

}

}

}

bool BinaryTree::contains(int key)

{

if (Root == NULL) {

return 0;

}

else {

Node\* iter;

iter = Root;

for (;;) {

if (key <= iter->key) {

if (iter->key == key) {

return 1;

}

if (iter->Lnext == NULL) {

return 0;

}

iter = iter->Lnext;

}

else {

if (iter->key == key) {

return 1;

}

if (iter->Rnext == NULL) {

return 0;

}

iter = iter->Rnext;

}

}

}

}

void BinaryTree::remove(int key)

{

if (!contains(key)) {

throw "No such element exists";

}

else {

Node\* iter;

iter = Root;

if (iter->key == key) {

if (iter->Lnext == NULL && iter->Rnext == NULL) {

delete iter;

return;

}

if (iter->Lnext != NULL && iter->Rnext == NULL) {

iter->Lnext->parent = iter->parent;

Root = iter->Lnext;

delete iter;

return;

}

if (iter->Lnext == NULL && iter->Rnext != NULL) {

iter->Rnext->parent = iter->parent;

Root = iter->Rnext;

delete iter;

return;

}

else {

Node\* iter2;

iter2 = iter->Rnext;

if (iter2->Lnext != NULL) {

while (iter2->Lnext != NULL) {

iter2 = iter2->Lnext;

}

iter2->parent->Lnext = NULL;

}

else {

iter2->parent->Rnext = iter2->Rnext;

}

iter->Lnext->parent = iter2;

iter2->Lnext = iter->Lnext;

iter2->parent = iter->parent;

while (iter2->Rnext!=NULL)

{

iter2 = iter2->Rnext;

}

iter->Rnext->parent = iter2;

iter2->Rnext = iter->Rnext;

return;

}

}

for (;;) {

if (key <= iter->key) {

iter = iter->Lnext;

if (iter->parent->Lnext->key == key) {

if (iter->Lnext == NULL && iter->Rnext == NULL) {

iter->parent->Lnext = NULL;

delete iter;

break;

}

if (iter->Lnext != NULL && iter->Rnext == NULL) {

iter->parent->Lnext = iter->Lnext;

iter->Lnext->parent = iter->parent;

delete iter;

break;

}

if (iter->Lnext == NULL && iter->Rnext != NULL) {

iter->parent->Lnext = iter->Rnext;

iter->Rnext->parent = iter->parent;

delete iter;

break;

}

else {

Node\* iter2;

iter2 = iter->Rnext;

if (iter2->Lnext != NULL) {

while (iter2->Lnext != NULL) {

iter2 = iter2->Lnext;

}

iter2->parent->Lnext = NULL;

}

else {

iter2->parent->Rnext = iter2->Rnext;

}

iter->parent->Lnext = iter2;

iter->Lnext->parent = iter2;

iter2->Lnext = iter->Lnext;

iter2->parent = iter->parent;

while (iter2->Rnext != NULL)

{

iter2 = iter2->Rnext;

}

iter->Rnext->parent = iter2;

iter2->Rnext = iter->Rnext;

delete iter;

break;

}

}

}

else {

iter = iter->Rnext;

if (iter->parent->Rnext->key == key) {

if (iter->Lnext == NULL && iter->Rnext == NULL) {

iter->parent->Rnext = NULL;

delete iter;

break;

}

if (iter->Lnext != NULL && iter->Rnext == NULL) {

iter->parent->Rnext = iter->Lnext;

iter->Lnext->parent = iter->parent;

delete iter;

break;

}

if (iter->Lnext == NULL && iter->Rnext != NULL) {

iter->parent->Rnext = iter->Rnext;

iter->Rnext->parent = iter->parent;

delete iter;

break;

}

else {

Node\* iter2;

iter2 = iter->Rnext;

if (iter2->Lnext != NULL) {

while (iter2->Lnext != NULL) {

iter2 = iter2->Lnext;

}

iter2->parent->Lnext = NULL;

}

else {

iter2->parent->Rnext = iter2->Rnext;

}

iter->parent->Rnext = iter2;

iter->Lnext->parent = iter2;

iter2->Lnext = iter->Lnext;

iter2->parent = iter->parent;

while (iter2->Rnext != NULL)

{

iter2 = iter2->Rnext;

}

iter2->Rnext = iter->Rnext;

iter->Rnext->parent = iter2;

delete iter;

break;

}

}

}

}

}

}

BinaryTree::dft\_iterator::dft\_iterator(Node\* root) {

cur = root;

if (cur != nullptr) {

Stack.push(cur);

}

else {

throw "This tree is empty";

}

}

BinaryTree::dft\_iterator::~dft\_iterator() {

}

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

int temp = cur->key;

if (!has\_next()) {

throw "This tree is empty";

}

if (cur->Rnext != nullptr) {

if (flag) {

Stack.pop\_front();

flag = false;

}

Stack.push(cur->Rnext);

}

if (cur->Lnext != nullptr) {

cur = cur->Lnext;

}

else

{

if (flag) {

Stack.pop\_front();

}

if (has\_next()) {

cur = Stack.head->date;

}

flag = true;

}

return temp;

}

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

if (cur->Lnext == nullptr && cur->Rnext == nullptr && Stack.size == 0) return false;

else return true;

}

BinaryTree::bft\_iterator::bft\_iterator(Node\* root) {

cur = root;

if (cur != nullptr) {

Queue.push(cur);

}

else {

throw "This tree is empty";

}

}

BinaryTree::bft\_iterator::~bft\_iterator() {

delete cur;

Queue.clear();

}

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

if (!has\_next()) {

throw "This tree is empty";

}

cur = Queue.head->date;

Queue.pop();

if (cur->Lnext != nullptr) {

Queue.push(cur->Lnext);

}

if (cur->Rnext != nullptr) {

Queue.push(cur->Rnext);

}

return cur->key;

}

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

if (cur->Lnext == nullptr && cur->Rnext == nullptr && Queue.size == 0) return false;

else return true;

}

void BinaryTree::preOrder(Node\* node) {

if (node == nullptr) return;

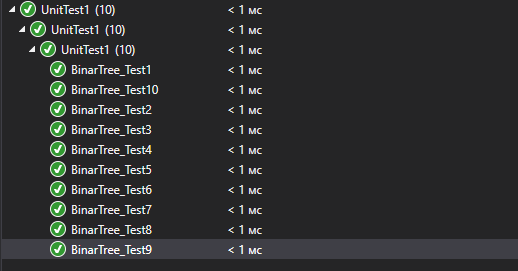
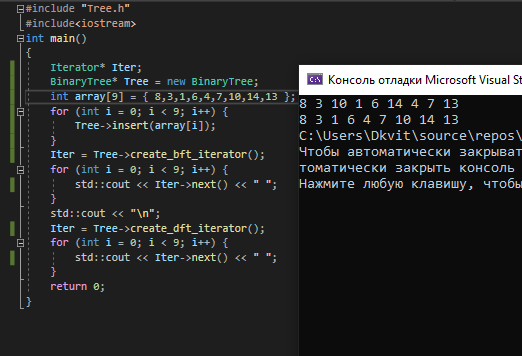
preOrder(node->Lnext);

preOrder(node->Rnext);

delete node;

}

# Пример работы



# Вывод

При выполнении данной лабораторной работы я познакомился с реализацией различных алгоритмов для двоичного дерева поиска