**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 №1**

|  |  |  |
| --- | --- | --- |
| Студент гр. 0302 |  | Приезжих Т.А. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

**2021**

## Постановка задачи

Реализовать методы на основе двоичного дерева поиска

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 класса и методов

Node – узел

Queue – очередь для итератора обхода в ширину

Stack – стек для обхода в глубину

Iterator – для реализации обходов в глубину и ширину

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)

## Временная сложность

|  |  |
| --- | --- |
| Метод | Временная сложность |
| contains(int) | O(log n) |
| insert(int) | O(n) |
| remove(int) | O(n) |
| create\_dft\_iterator() | O(1) |
| create\_bft\_iterator() | O(1) |

## Unit-тесты

TestContains – вставляем элемент, проверяем его наличие

TestInsert - вставляем элемент, проверяем его наличие

TestRemove - вставляем элементы, некоторые удаляем, проверяем наличие

TestDftIterator – вставляем элементы, выполняем обход в глубину, сверяем полученные данные

TestBftIterator - вставляем элементы, выполняем обход в ширину, сверяем полученные данные

## Пример работы

## 

## Листинг

main.cpp

#include "binarySearchTree.h"

using namespace std;

int main()

{

try{

BinarySearchTree Tree;

Tree.insert(77);

Tree.insert(31);

Tree.insert(123);

Tree.insert(23);

Tree.insert(56);

Tree.insert(123);

Tree.insert(934);

cout << "Depth-first traverse:\n";

BinarySearchTree::Iterator\* D = Tree.create\_dft\_iterator();

while (D->hasNext()) {

cout<< D->next() << " ";

}

cout << "\n\nBreadth-first traverse:\n";

BinarySearchTree::Iterator\* B = Tree.create\_bft\_iterator();

while (B->hasNext2()){

cout << B->Next2() << " ";

}

}

catch (exception ex) { cout << endl << ex.what(); }

\_getch();

}

stack.h

#pragma once

#include <iostream>

using namespace std;

class Stack{

public:

Stack();

~Stack();

void push(int data);

int get\_head();

void pop\_front();

int get\_size() { return size; };

void clear();

friend std::ostream& operator<<(std::ostream& s, Stack& list){

unsigned count = 1;

Node\* cur = list.head;

while (count <= list.size) {

s << cur->data << '\t';

cur = cur->pNext;

count++;

}

return s;

}

private:

struct Node{

Node\* pNext;

int data;

Node(int data = int(), Node\* pNext = nullptr){

this->data = data;

this->pNext = pNext;

}

};

unsigned size;

Node\* head;

};

Stack::Stack(){

size = 0;

head = nullptr;

}

Stack::~Stack(){

clear();

}

void Stack::push(int data){

head = new Node(data, head);

size++;

}

int Stack::get\_head(){

int temp = head->data;

pop\_front();

return temp;

}

void Stack::pop\_front(){

if (this->size){

Node\* temp = head;

head = head->pNext;

delete temp;

size--;

}

}

void Stack::clear(){

while (size)

pop\_front();

}

queue.h

#pragma once

#include <iostream>

#include "node.h"

class Queue{

unsigned size;

Node\* head;

public:

Queue();

~Queue();

void push\_back(intNode\* data);

intNode\* get\_head(){

intNode\* temp = head->data;

pop\_front();

return temp;

}

void pop\_front();

int get\_size() { return size; };

void clear();

};

Queue::Queue(){

size = 0;

head = nullptr;

}

Queue::~Queue(){

clear();

}

void Queue::push\_back(intNode\* data){

if (head == nullptr)

head = new Node(data);

else{

Node\* cur = this->head;

while (cur->pNext != nullptr)

cur = cur->pNext;

cur->pNext = new Node(data);

}

size++;

}

void Queue::pop\_front(){

if (this->size){

Node\* temp = head;

head = head->pNext;

delete temp;

size--;

}

}

void Queue::clear(){

while (size)

pop\_front();

}

binarySearchTree.h

#pragma once

#include "stack.h"

#include "queue.h"

#include "conio.h"

using namespace std;

class BinarySearchTree{

public:

BinarySearchTree(){

root = nullptr;

}

~BinarySearchTree(){

while (root != nullptr)

remove(root->data);

}

bool contains(int data);

void insert(int data);

void remove(int data);

intNode\* root;

class Iterator{

public:

Iterator(intNode\* start){

cur = start;

if (start != nullptr)

Queue.push\_back(start);

}

bool hasNext();

bool hasNext2();

int next();

int Next2();

int getCurData() { return cur->data; };

private:

intNode\* cur;

Stack Stack;

Queue Queue;

};

Iterator\* create\_dft\_iterator(){

cout << root->data << " ";

return new Iterator(root);

}

Iterator\* create\_bft\_iterator(){

return new Iterator(root);

}

};

bool BinarySearchTree::contains(int data)

{

intNode\* cur = root;

while (cur != nullptr){

if (data > cur->data){

cur = cur->pRight;

continue;

}

if (data < cur->data){

cur = cur->pLeft;

continue;

}

if (data == cur->data)

return true;

}

return false;

}

void BinarySearchTree::insert(int data){

intNode\* cur = root;

if (cur == nullptr)

root = new intNode(nullptr, data);

else{

while ((cur->pLeft != nullptr) || (cur->pRight != nullptr)){

if ((data > cur->data) && (cur->pRight != nullptr))

cur = cur->pRight;

else if ((data > cur->data) && (cur->pRight == nullptr)) break;

if ((data <= cur->data) && (cur->pLeft != nullptr))

cur = cur->pLeft;

else if ((data <= cur->data) && (cur->pLeft == nullptr)) break;

}

if (data > cur->data)

cur->pRight = new intNode(cur, data);

else

cur->pLeft = new intNode(cur, data);

}

}

void BinarySearchTree::remove(int data){

if (!contains(data))

throw exception("You cannot remove a non-existent element!");

intNode\* cur = root;

while (data != cur->data){

if (data < cur->data)

cur = cur->pLeft;

else

cur = cur->pRight;

}

if (cur != nullptr){

if ((cur->pLeft == nullptr) && (cur->pRight == nullptr)){

intNode\* temp = cur;

if (cur == root)

root = nullptr;

else{

if (cur->data > cur->pPrev->data)

cur->pPrev->pRight = nullptr;

else cur->pPrev->pLeft = nullptr;

}

delete temp;

}

else if ((cur->pLeft != nullptr) ^ (cur->pRight != nullptr)){

intNode\* temp = cur;

if (cur == root)

if (cur->pLeft != nullptr)

root = root->pLeft;

else root = root->pRight;

else{

if (cur->data > cur->pPrev->data)

if (cur->pLeft != nullptr)

cur->pPrev->pRight = cur->pLeft;

else cur->pPrev->pRight = cur->pRight;

else

if (cur->pLeft != nullptr)

cur->pPrev->pLeft = cur->pLeft;

else cur->pPrev->pLeft = cur->pRight;

}

delete temp;

}

else if ((cur->pLeft != nullptr) && (cur->pRight != nullptr)){

intNode\* temp = cur;

cur = cur->pRight;

while (cur->pLeft != nullptr)

cur = cur->pLeft;

if (temp == root)

root->data = cur->data;

temp->data = cur->data;

intNode\* temp1 = temp;

temp = cur;

if (cur->pPrev == temp1)

cur->pPrev->pRight = nullptr;

else cur->pPrev->pLeft = nullptr;

delete temp;

}

}

}

bool BinarySearchTree::Iterator::hasNext(){

if ((Stack.get\_size() != 0) || (cur->pLeft != nullptr) || (cur->pRight != nullptr))

return true;

else return false;

}

bool BinarySearchTree::Iterator::hasNext2(){

if (Queue.get\_size())

return true;

else return false;

}

int BinarySearchTree::Iterator::next(){

if (cur->pRight != nullptr){

Stack.push(cur->pRight->data);

}

if (cur->pLeft != nullptr)

cur = cur->pLeft;

else{

if (Stack.get\_size()){

int temp = Stack.get\_head();

do

cur = cur->pPrev;

while ((cur->pRight == nullptr) || (cur->pRight->data != temp));

cur = cur->pRight;

}

}

return cur->data;

}

int BinarySearchTree::Iterator::Next2()

{

intNode\* tempNode = Queue.get\_head();

if (tempNode->pLeft != nullptr)

Queue.push\_back(tempNode->pLeft);

if (tempNode->pRight != nullptr)

Queue.push\_back(tempNode->pRight);

return tempNode->data;

}

node.h

#pragma once

struct intNode{

intNode\* pLeft;

intNode\* pRight;

intNode\* pPrev;

int data;

intNode(intNode\* pPrev = nullptr, int data = int(), intNode\* pLeft = nullptr, intNode\* pRight = nullptr){

this->pPrev = pPrev;

this->data = data;

this->pLeft = pLeft;

this->pRight = pRight;

}

};

struct Node{

Node\* pNext;

intNode\* data;

Node(intNode\* data, Node\* pNext = nullptr){

this->data = data;

this->pNext = pNext;

}

};

UnitTest3.cpp

#include "pch.h"

#include "CppUnitTest.h"

#include "..\Lab3\binarySearchTree.h"

#include "..\Lab3\node.h"

#include "..\Lab3\queue.h"

#include "..\Lab3\stack.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTestLab3

{

TEST\_CLASS(UnitTestLab3)

{

public:

TEST\_METHOD(TestContains)

{

BinarySearchTree Tree;

for (unsigned i = 50; i > 0; i--)

Tree.insert(i);

for (unsigned i = 50; i > 0; i--)

Assert::IsTrue(Tree.contains(i));

for (unsigned i = 100; i > 50; i--)

Assert::IsTrue(!Tree.contains(i));

}

TEST\_METHOD(TestInsert)

{

BinarySearchTree Tree;

Tree.insert(77);

Tree.insert(31);

Tree.insert(123);

Assert::IsTrue(Tree.contains(77));

Assert::IsTrue(Tree.contains(31));

Assert::IsTrue(Tree.contains(123));

Assert::IsTrue(!Tree.contains(1000));

}

TEST\_METHOD(TestRemove)

{

BinarySearchTree Tree;

Tree.insert(77);

Tree.insert(31);

Tree.insert(123);

Tree.insert(23);

Tree.insert(56);

Tree.remove(56);

Tree.remove(31);

Tree.remove(77);

Tree.remove(123);

Assert::IsTrue(!Tree.contains(77));

Assert::IsTrue(!Tree.contains(31));

Assert::IsTrue(!Tree.contains(123));

Assert::IsTrue(!Tree.contains(56));

Assert::IsTrue(Tree.contains(23));

}

TEST\_METHOD(TestDftIterator)

{

BinarySearchTree Tree;

Tree.insert(77);

Tree.insert(31);

Tree.insert(123);

Tree.insert(23);

Tree.insert(56);

Tree.insert(123);

Tree.insert(934);

BinarySearchTree::Iterator\* D = Tree.create\_dft\_iterator();

Assert::IsTrue(D->getCurData() == 77);

Assert::IsTrue(D->next() == 31);

Assert::IsTrue(D->next() == 23);

Assert::IsTrue(D->next() == 56);

Assert::IsTrue(D->next() == 123);

Assert::IsTrue(D->next() == 123);

Assert::IsTrue(D->next() == 934);

}

TEST\_METHOD(TestBftIterator)

{

BinarySearchTree Tree;

Tree.insert(77);

Tree.insert(31);

Tree.insert(123);

Tree.insert(23);

Tree.insert(56);

Tree.insert(123);

Tree.insert(934);

BinarySearchTree::Iterator\* B = Tree.create\_bft\_iterator();

Assert::IsTrue(B->Next2() == 77);

Assert::IsTrue(B->Next2() == 31);

Assert::IsTrue(B->Next2() == 123);

Assert::IsTrue(B->Next2() == 23);

Assert::IsTrue(B->Next2() == 56);

Assert::IsTrue(B->Next2() == 123);

Assert::IsTrue(B->Next2() == 934);

}

};

}