**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**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9302 |  | Ленина М.Р. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

Оглавление

[1. Постановка задачи 3](#_Toc59130683)

[2. Описание реализуемого класса и методов 3](#_Toc59130684)

[3. Описание реализованных unit-тестов 3](#_Toc59130685)

[4. Пример работы 3](#_Toc59130686)

[5. Листинг 3](#_Toc59130687)

[6. Вывод 3](#_Toc59130688)

# Постановка задачи

Реализовать двоичное дерево поиска и методы к нему:

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

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

# Описание реализуемого класса и методов

Class Node

|  |  |
| --- | --- |
| Элемент | Описание |
| Node\* left | Указатель на следующий левый элемент дерева (меньше текущего элемента) |
| Node\* right | Указатель на следующий правый элемент дерева (больше текущего элемента) |
| Node\* parent | Указатель на предыдущий элемент дерева |
| int data | Целочисленные данные, хранящиеся в элементе дерева |

Class BinarySearchTree

|  |  |  |  |
| --- | --- | --- | --- |
| Элемент | Описание | | |
| Node\* root | Указатель на корень дерева | | |
| Метод | | 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() | | Создание итератора, реализующего методы обхода в ширину | O(1) |
| Iterator\* create\_dft\_iterator() | | Создание итератора, реализующего один из методов обхода в глубину | O(1) |

Class queue\_Node

|  |  |
| --- | --- |
| Элемент | Описание |
| Node\* element | Указатель на элемент дерева |
| queue\_Node\* next | Указатель на следующий элемент очереди |

Class Queue

|  |  |  |  |
| --- | --- | --- | --- |
| Элемент | Описание | | |
| queue\_Node\* first | Указатель на первый элемент очереди | | |
| queue\_Node\* last | Указатель на последний элемент очереди | | |
| Метод | | Описание | Оценка временной сложности |
| bool isEmpty() | | Проверка на пустоту очереди | O(1) |
| void push(Node\*) | | Добавление элемента в очередь | O(1) |
| Node\* pop() | | Удаление первого элемента очереди и возращение его значения | O(1) |
| Node\* front() | | Возвращает первый элемент очереди | O(1) |

Class bft\_Iterator : public Iterator

|  |  |  |  |
| --- | --- | --- | --- |
| Элемент | Описание | | |
| Queue\* queue | Очередь для реализации обхода в ширину | | |
| Метод | | Описание | Оценка временной сложности |
| bool has\_next() override | | Проверка на наличие следующего элемента | O(1) |
| int next() | | Переход к следующему элементу | O(1) |

Class stack\_Node

|  |  |
| --- | --- |
| Элемент | Описание |
| Node\* element | Указатель на элемент дерева |
| stack\_Node\* next | Указатель на следующий элемент стека |

Class Stack

|  |  |  |  |
| --- | --- | --- | --- |
| Элемент | Описание | | |
| stack\_Node\* top | Указатель на первый элемент стека | | |
| Метод | | Описание | Оценка временной сложности |
| bool isEmpty() | | Проверка на пустоту стека | O(1) |
| void push(Node\*) | | Добавление элемента в стек | O(1) |
| Node\* pop() | | Удаление первого элемента стека и возращение его значения | O(1) |
| Node\* getTop () | | Возвращает первый элемент стека | O(1) |

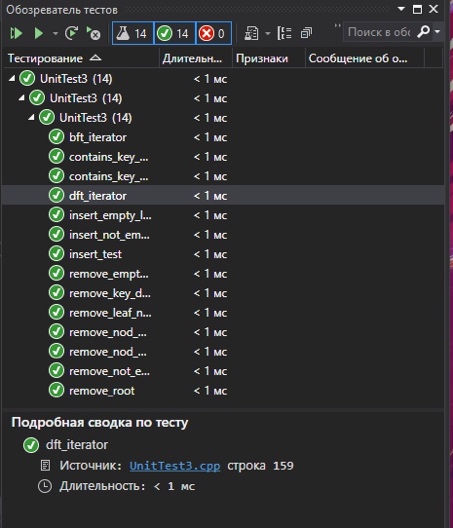
Class dft\_Iterator : public Iterator

|  |  |  |  |
| --- | --- | --- | --- |
| Элемент | Описание | | |
| Stack\* stack | Стек для реализации обхода в глубину | | |
| Метод | | Описание | Оценка временной сложности |
| bool has\_next() override | | Проверка на наличие следующего элемента | O(1) |
| int next() | | Переход к следующему элементу | O(1) |

# Описание реализованных unit-тестов

|  |  |
| --- | --- |
| contains\_key\_exist | Проверка метода contains, ключ содержится в дереве |
| contains\_key\_doesnt\_exist | Проверка метода contains, ключ не содержится в дереве |
| insert\_empty\_list | Проверка метода insert, вставка в пустое дерево |
| insert\_not\_empty\_tree | Проверка метода insert, вставка, дерево не пустой |
| insert\_test | Проверка метода insert |
| remove\_empty\_tree | Проверка метода remove, пустое дерево |
| remove\_key\_dosent\_exist | Проверка метода remove, ключ не существует |
| remove\_not\_empty\_tree | Проверка метода remove, не пустое дерево |
| remove\_root | Проверка метода remove, удаление корневого элемента |
| remove\_leaf\_nod | Проверка метода remove, удаление листа |
| remove\_nod\_have\_one\_child | Проверка метода remove, удаление элемента с одним потомком |
| remove\_nod\_have\_two\_children | Проверка метода remove, удаление элемента с двумя потомками |
| bft\_iterator | Проверка итератора обхода в ширину |
| dft\_iterator | Проверка итератора обхода в глубину |

# Пример работы



# Листинг

|  |
| --- |
| **BinarySearchTree.h** |
| #pragma once  #include "Iterator.h"  class Node {  public:  Node\* left;  Node\* right;  Node\* parent;  int data;  Node(int data, Node\* left = nullptr, Node\* right = nullptr, Node\* parent = nullptr) {  this->data = data;  this->left = left;  this->right = right;  this->parent = parent;  };  ~Node() {};  };  class BinarySearchTree  {  private:  Node\* root;  public:  BinarySearchTree();  ~BinarySearchTree();  bool contains(int);  void insert(int);  void remove(int);  class Queue {  private:  class queue\_Node {  public:  queue\_Node(Node\* element, queue\_Node\* next = nullptr) {  this->element = element;  this->next = next;  };  ~queue\_Node() {};  Node\* element;  queue\_Node\* next;  };  queue\_Node\* first;  queue\_Node\* last;  public:  Queue();  ~Queue();  bool isEmpty();  void push(Node\*);  Node\* pop();  Node\* front();  };  Iterator\* create\_bft\_iterator();    class bft\_Iterator : public Iterator {  public:  bft\_Iterator(Node\* start);  bool has\_next() override;  int next() override;  private:  Queue\* queue;  };  class Stack {  private:  class stack\_Node {  public:  stack\_Node(Node\* element, stack\_Node\* next = nullptr) {  this->element = element;  this->next = next;  };  ~stack\_Node() {};  Node\* element;  stack\_Node\* next;  };  stack\_Node\* top;  public:  Stack();  ~Stack();  void push(Node\*);  Node\* pop();  bool isEmpty();  Node\* getTop();  };  Iterator\* create\_dft\_iterator();  class dft\_Iterator: public Iterator{  public:  dft\_Iterator(Node\* start);  bool has\_next() override;  int next() override;  private:  Stack\* stack;  };  }; |
| Iterator.h |
| #pragma once  class Iterator {  public:  virtual int next() = 0;  virtual bool has\_next() = 0;  }; |
| **BinarySearchTree.cpp** |
| #include "BinarySearchTree.h"  #include <stdexcept>  using namespace std;  BinarySearchTree::BinarySearchTree() {  root = nullptr;  }  BinarySearchTree::~BinarySearchTree() {  while (root != nullptr) {  remove(root->data);  }  delete root;  }  // searching element in the tree by the key  bool BinarySearchTree::contains(int key) {  Node\* current = root;  while (current != nullptr) {  if (key > current->data)  current = current->right;  else  if (key < current->data)  current = current->left;  else  if (key == current->data)  return true;  }  return false;  }  //adding element into tree by key. it must takes O(logN)  void BinarySearchTree::insert(int key) {    if (root == nullptr)  root = new Node(key);  else {  Node\* current = root;  while ( !((key >= current->data) && (current->right==nullptr)) && !((key <= current->data) && (current->left == nullptr))) {  if (key >= current->data)  current = current->right;  else  if (key <= current->data)  current = current->left;  if (current == nullptr)  break;  }  if (current != nullptr) {  if ((key > current->data) && (current->right == nullptr))  current->right = new Node(key, nullptr, nullptr, current);  if ((key < current->data) && (current->left == nullptr))  current->left = new Node(key, nullptr, nullptr, current);  }  }  }  //deleting element of tree by the key  void BinarySearchTree::remove(int key) {  if (root == nullptr)  throw out\_of\_range("The tree is empty");  else  if (root->left == nullptr && root->right == nullptr && root->data == key) {  root = nullptr;  }  else  if (root->left == nullptr && root->right == nullptr && root->data != key) {  throw out\_of\_range("This key dosen't exist");  }  else {  Node\* current = root;  Node\* child\_elem;  while (current != nullptr) {  if (current->data == key) {  if (current->left == nullptr && current->right == nullptr) {  if ((current->parent)->left == current)  (current->parent)->left = nullptr;  else  (current->parent)->right = nullptr;  delete current;  }  else  if (current->left != nullptr && current->right == nullptr) {  child\_elem = current->left;  current->data = child\_elem->data;  current->left = child\_elem->left;  current->right = child\_elem->right;  if (current->left != nullptr)  current->left->parent = current;  if (current->right != nullptr)  current->right->parent = current;  delete child\_elem;  }  else  if (current->left == nullptr && current->right != nullptr) {  child\_elem = current->right;  current->data = child\_elem->data;  current->left = child\_elem->left;  current->right = child\_elem->right;  if (current->left != nullptr)  current->left->parent = current;  if (current->right != nullptr)  current->right->parent = current;  delete child\_elem;  }  else {  child\_elem = current->right;  if (child\_elem->left == nullptr) {  current->right = child\_elem->right;  if (child\_elem->right != nullptr)  (child\_elem->right)->parent = current;  }  else {  while (child\_elem->left != nullptr && child\_elem != nullptr)  child\_elem = child\_elem->left;  (child\_elem->parent)->left = child\_elem->right;  if (child\_elem->right != nullptr)  child\_elem->right->parent = child\_elem->parent;  }  current->data = child\_elem->data;  delete child\_elem;  }  return;  }  else  if (current->data > key)  current = current->left;  else  current = current->right;  }  throw out\_of\_range("This key dosen't exist");  }  } |
| **Iterator.cpp** |
| #include "BinarySearchTree.h"  #include <stdexcept>  using namespace std;  BinarySearchTree::Queue::Queue() {  first = nullptr;  last = nullptr;  }  BinarySearchTree::Queue::~Queue() {  while (first != nullptr) {  queue\_Node\* temp = first;  first = first->next;  delete temp;  }  }  bool BinarySearchTree::Queue::isEmpty() {  return (first == nullptr);  }  void BinarySearchTree::Queue::push(Node\* new\_elem) {  if (isEmpty()) {  first = new queue\_Node(new\_elem);  last = first;  } else {  last->next = new queue\_Node(new\_elem);  last = last->next;  }  }  Node\* BinarySearchTree::Queue::pop() {  Node\* temp = first->element;  first = first->next;  return temp;  }  Node\* BinarySearchTree::Queue::front() {  return first->element;  }  Iterator\* BinarySearchTree::create\_bft\_iterator() {  return new bft\_Iterator(root);  }  BinarySearchTree::bft\_Iterator::bft\_Iterator(Node\* start) {  queue = new Queue();  queue->push(start);  }  bool BinarySearchTree::bft\_Iterator::has\_next() {  return !(queue->isEmpty());  }  int BinarySearchTree::bft\_Iterator::next() {  Node\* current;  current = queue->pop();  if (current->left != nullptr)  queue->push(current->left);  if (current->right != nullptr)  queue->push(current->right);  return current->data;  }  BinarySearchTree::Stack::Stack() {  top = nullptr;  }  BinarySearchTree::Stack::~Stack() {  while (!isEmpty()) {  pop();  }  }  void BinarySearchTree::Stack::push(Node\* new\_elem) {  top = new stack\_Node(new\_elem, top);  }  Node\* BinarySearchTree::Stack::pop() {  Node\* temp = top->element;  top = top->next;  return temp;  }  bool BinarySearchTree::Stack::isEmpty() {  return (top == nullptr);  }  Node\* BinarySearchTree::Stack::getTop() {  return top->element;  }  Iterator\* BinarySearchTree::create\_dft\_iterator() {  return new dft\_Iterator(root);  }  BinarySearchTree::dft\_Iterator::dft\_Iterator(Node\* start) {  stack = new Stack();  stack->push(start);  }  bool BinarySearchTree::dft\_Iterator::has\_next() {  return !(stack->isEmpty());  }  int BinarySearchTree::dft\_Iterator::next() {  Node\* current;  current = stack->pop();  if (current->left != nullptr)  stack->push(current->left);  if (current->right != nullptr)  stack->push(current->right);  return current->data;  } |
| UnitTest3.cpp |
| #include "CppUnitTest.h"  #include "../astra\_l3/Iterator.h"  #include "../astra\_l3/BinarySearchTree.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTest3  {  TEST\_CLASS(UnitTest3)  {  public:    BinarySearchTree \*tree;    TEST\_METHOD\_INITIALIZE(setUp)  {  tree = new BinarySearchTree();  }    TEST\_METHOD\_CLEANUP(cleanUp)  {  delete tree;  }  TEST\_METHOD(contains\_key\_exist)  {    tree->insert(4);  tree->insert(1);  tree->insert(2);  tree->insert(3);  tree->insert(5);  tree->insert(6);  Assert::IsTrue(tree->contains(3));  }  TEST\_METHOD(contains\_key\_doesnt\_exist)  {  tree->insert(8);  tree->insert(6);  tree->insert(2);  tree->insert(9);  tree->insert(5);  tree->insert(1);  Assert::IsFalse(tree->contains(3));  }  TEST\_METHOD(insert\_empty\_list)  {  tree->insert(1);  Assert::IsTrue(tree->contains(1));  }  TEST\_METHOD(insert\_not\_empty\_tree)  {  tree->insert(8);  tree->insert(6);  tree->insert(1);  Assert::IsTrue(tree->contains(1));  }  TEST\_METHOD(insert\_test)  {  tree->insert(9);  tree->insert(10);  tree->insert(6);  tree->insert(8);  tree->insert(4);  tree->insert(5);  Assert::IsTrue(tree->contains(5));  }  TEST\_METHOD(remove\_empty\_tree)  {  try {  tree->remove(5);  }  catch (std::out\_of\_range e) {  Assert::AreEqual("The tree is empty", e.what());  }  }  TEST\_METHOD(remove\_key\_dosent\_exist)  {  tree->insert(4);  try {  tree->remove(5);  }  catch (std::out\_of\_range e) {  Assert::AreEqual("This key dosen't exist", e.what());  }  }  TEST\_METHOD(remove\_not\_empty\_tree)  {  tree->insert(9);  tree->insert(10);  tree->insert(6);  tree->insert(8);  tree->insert(4);  tree->insert(5);  tree->remove(6);  Assert::IsFalse(tree->contains(6));  }  TEST\_METHOD(remove\_root)  {  tree->insert(9);    tree->remove(9);  Assert::IsFalse(tree->contains(9));  }  TEST\_METHOD(remove\_leaf\_nod)  {  tree->insert(5);  tree->insert(2);  tree->insert(7);  tree->insert(6);  tree->insert(8);  tree->remove(6);  Assert::IsFalse(tree->contains(6));  }  TEST\_METHOD(remove\_nod\_have\_one\_child)  {  tree->insert(5);  tree->insert(2);  tree->insert(7);  tree->insert(6);  tree->remove(7);  Assert::IsFalse(tree->contains(7));  }  TEST\_METHOD(remove\_nod\_have\_two\_children)  {  tree->insert(5);  tree->insert(2);  tree->insert(7);  tree->insert(6);  tree->insert(8);  tree->remove(7);  Assert::IsFalse(tree->contains(7));  }  TEST\_METHOD(bft\_iterator)  {  int check\_array[] = {9, 6, 10, 4, 8, 5};  int index = 0;  Iterator\* bft\_iterator;  tree->insert(9);  tree->insert(10);  tree->insert(6);  tree->insert(8);  tree->insert(4);  tree->insert(5);  bft\_iterator = tree->create\_bft\_iterator();  while (bft\_iterator->has\_next()) {  Assert::AreEqual(check\_array[index],bft\_iterator->next());  index++;  }  }  TEST\_METHOD(dft\_iterator)  {  int check\_array[] = { 9, 10, 6, 8, 4, 5 };  int index = 0;  Iterator\* dft\_iterator;  tree->insert(9);  tree->insert(10);  tree->insert(6);  tree->insert(8);  tree->insert(4);  tree->insert(5);  dft\_iterator = tree->create\_dft\_iterator();  while (dft\_iterator->has\_next()) {  Assert::AreEqual(check\_array[index], dft\_iterator->next());  index++;  }  }  };  } |

# Вывод

Я изучила и освоила двоичные деревья.