**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 \_\_\_\_\_\_\_\_\_\_\_\_ Горбанев А.В

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

Санкт-Петербург

2020

Оглавление

[1. Постановка задачи и описание реализуемых класса и алгоритмов 3](#_Toc58158475)

[2. Описание реализованных unit-тестов 3](#_Toc58158476)

[3. Код программы 4](#_Toc58158477)

[4. Пример работы 11](#_Toc58158478)

[5. Вывод 11](#_Toc58158479)

# Постановка задачи и описание реализуемых классов и алгоритмов

Реализовать алгоритмы поиска, добавления и удаления элементов в двоичном дереве поиска

Использую для реализации класс Node, который представляет из себя отдельное взятый элемент имеющий указатель на левый и правый элемент в двоичном дереве, а также на элемент от которого он идет, и класс Binary\_tree , который объединяет все элементы в полноценное двоичное дерево поиска

Для этого используем следующие функции:

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Оценка временной сложности** |
| bool contains(const int&) | Проверяет наличие элемента в дереве | O(logn) |
| void insert(const int&) | Вставляет элемент в дерево | O(logn) |
| void remove(const int&) | Удаляет элемент из деерева | 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) |

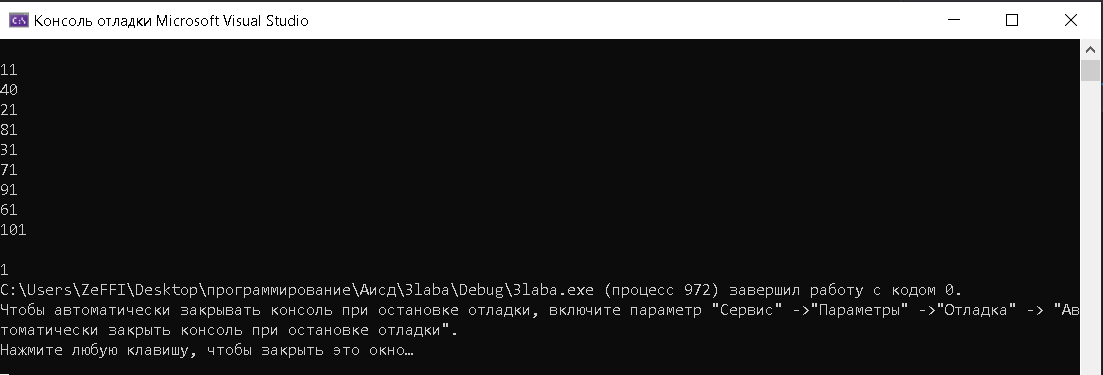
# Описание реализованных unit-тестов

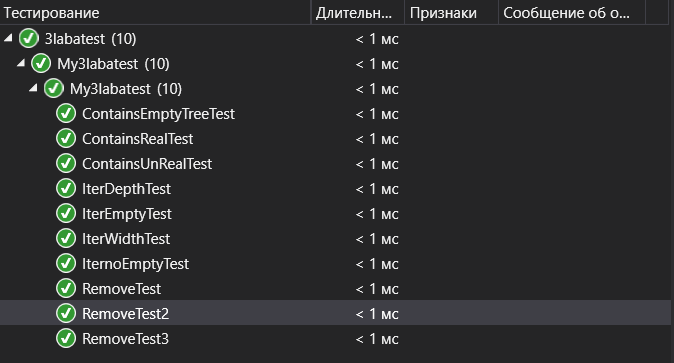
|  |  |
| --- | --- |
| Имя теста | Описание |
| IternoEmptyTest | Создание итератора на непустом дереве |
| IterEmptyTest | Создание итератора на пустом дереве |
| IterWidthTest | Проверка работы итератора обхода в ширину |
| IterDepthTest | Проверка работы итератора обхода в глубину |
| ContainsRealTest | Проверка поиска существующего элемента в непустом дереве |
| ContainsUnRealTest | Проверка поиска несуществующего элемента в непустом дереве |
| ContainsEmptyTreeTest | Проверка поиска элемента в пустом дереве |
| RemovenoEmptyTest | Проверка удаления существующего элемента в непустом дереве |
| RemoveunRealTest | Проверка удаления несуществующего элемента в непустом дереве |
| RemoveEmptyTest | Проверка удаления элемента в пустом дереве |

# Код программы

|  |
| --- |
| Tree.h |
| #pragma once  #include "queue.h"  #include "stack.h"  #include "iterator.h"  #include "node.h"  #include<iostream>  using namespace std;  class dft\_iterator : public Iterator {  private:  Node\* cur;  stack nodesStack;  bool bl = true;  public:  dft\_iterator(Node\* root) {  cur = root;  if (cur != nullptr) {  nodesStack.push(cur);  }  else {  throw "This tree is empty";  }  }  int next() override {  if (bl) {  bl = false;  Node\* temp = nodesStack.getTop();  nodesStack.pop();  return temp->data;  }  if (cur->pRight != nullptr) nodesStack.push(cur->pRight);  if (cur->pLeft != nullptr) cur = cur->pLeft;  else  if (!nodesStack.isEmpty()) {  cur = nodesStack.getTop();  nodesStack.pop();  }  else cur = nullptr;  return cur->data;  }  bool has\_next() override {  if (cur->pLeft == nullptr && cur->pLeft == nullptr) return false;  else return true;  }  };  class bft\_iterator : public Iterator {  private:  Node\* cur;  queue nodesQueue;  public:  bft\_iterator(Node\* root) {  cur = root;  if (cur != nullptr) {  nodesQueue.push(cur);  }  else {  throw "This tree is empty";  }  }  int next() override {  if (nodesQueue.getSize() > 0) {  cur = nodesQueue.front();  nodesQueue.pop();  if (cur->pLeft != nullptr) {  nodesQueue.push(cur->pLeft);  }  if (cur->pRight != nullptr) {  nodesQueue.push(cur->pRight);  }  return cur->data;;  }  else cur = nullptr;  }  bool has\_next() override {  if (cur->pLeft == nullptr && cur->pRight == nullptr) return false;  else return true;  }  };  class Binary\_tree {  private:  public:  Binary\_tree();  ~Binary\_tree() {  delete root;  }  void insert(int data);  void print\_tree(Node\* root);  bool contains(int data);  void remove(int data);  Node\* root;  Iterator\* create\_bft\_iterator() {  return new bft\_iterator(root);  }  Iterator\* create\_dft\_iterator() {  return new dft\_iterator(root);  }  };  Binary\_tree::Binary\_tree()  {  root = nullptr;  }  void Binary\_tree::insert(int data)  {  if (root == nullptr) {  root = new Node(data);  }  else {  Node\* current = this->root;  while (true) {  if (current->data < data) {  if (current->pRight == nullptr) {  current->pRight = new Node(data);  current->pRight->parents = current;  break;  }  if (current->pRight != nullptr) {  current = current->pRight;  continue;  }  }  if (current->data > data) {  if (current->pLeft == nullptr) {  current->pLeft = new Node(data);  current->pLeft->parents = current;  break;  }  if (current->pLeft != nullptr) {  current = current->pLeft;  continue;  }  }  if (current->data == data) break;  }  }  }  void Binary\_tree::print\_tree(Node\* root)  {  if (root == nullptr) return;  if (root->pLeft == nullptr && root->pRight == nullptr) cout << root->data << " ";  else {  cout << root->data << " ";  print\_tree(root->pRight);  print\_tree(root->pLeft);  }  }  bool Binary\_tree::contains(int data)  {  if (root == nullptr) throw "This tree is Empty!";  Node\* current = this->root;  if (current->data == data) return true;  while (current) {  if (current->data < data) current = current->pRight;  else if (current->data > data) current = current->pLeft;  else {  return true;  }  }  return false;  }  void Binary\_tree::remove(int data)  {  if (root == nullptr) throw "This tree is Empty!";  Node\* current = this->root;  while (current) {  if (current->data < data) current = current->pRight;  else if (current->data > data) current = current->pLeft;  else if (current->data == data) {  break;  }  }  if (current==nullptr) throw "This element isn`t in tree";  if (current->pLeft == nullptr && current->pRight == nullptr) {  Node\* cur\_par = current->parents;  if (current == cur\_par->pLeft) cur\_par->pLeft = nullptr;  if (current == cur\_par->pRight) cur\_par->pRight = nullptr;  delete current;  }  else if (current->pLeft == nullptr && current->pRight != nullptr) {  Node\* cur\_par = current->parents;  if (current == cur\_par->pLeft) cur\_par->pLeft = current->pRight;  if (current == cur\_par->pRight) cur\_par->pRight = current->pRight;  delete current;  }  else if (current->pRight == nullptr && current->pLeft != nullptr) {  Node\* cur\_par = current->parents;  if (current == cur\_par->pLeft) cur\_par->pLeft = current->pLeft;  if (current == cur\_par->pRight) cur\_par->pRight = current->pLeft;  delete current;  }  else if (current->pLeft != nullptr && current->pRight != nullptr) {  Node\* max\_elem;  Node\* cur\_par;  Node\* cur\_left = current->pLeft;  while (cur\_left->pRight != nullptr) {  cur\_left = cur\_left->pRight;  }  cur\_par = cur\_left->parents;  if (cur\_left == cur\_par->pLeft) cur\_par->pLeft = nullptr;  if (cur\_left == cur\_par->pRight) cur\_par->pRight = nullptr;  current->data = cur\_left->data;  delete cur\_left;  }  } |
| Source.cpp |
| #include<iostream>  #include "tree.h"  using namespace std;  int main() {  Binary\_tree tree;  cout << endl;  int arr[10] = {1,5,8,2,3,9,4,7,6,10 };  for (int i = 0; i < 10; i++) tree.insert(arr[i]);  tree.remove(5);  Iterator\* iter = tree.create\_bft\_iterator();  for (int i = 0; i < 9; i++) {  cout << iter->next();  cout << tree.contains(arr[i]) << endl;  }  cout << endl;    cout<<tree.contains(6);      } |
| 3labatest.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../3laba/tree.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace My3labatest  {  TEST\_CLASS(My3labatest)  {  public:    TEST\_METHOD(IternoEmptyTest)  {  Binary\_tree tree;  tree.insert(1);  Iterator\* iter = tree.create\_dft\_iterator();  Assert::IsTrue(iter->next() == 1);  }  TEST\_METHOD(IterEmptyTest)  {  Binary\_tree tree;  try  {  Iterator\* iter = tree.create\_dft\_iterator();  }  catch (const char\* warning)  {  Assert::AreEqual(warning, "This tree is empty");  }  }  TEST\_METHOD(IterWidthTest)  {  Binary\_tree tree;  int arr[10] = {7,4,9,1,6,8,0,3,5,2 };  for (int i = 0; i < 10; i++) tree.insert(arr[i]);  Iterator\* iter = tree.create\_bft\_iterator();  for (int i = 0; i < 10; i++) Assert::IsTrue(arr[i] == iter->next());  }  TEST\_METHOD(IterDepthTest)  {  Binary\_tree tree;  int arr[10] = { 7, 4, 1, 0, 3, 2, 6, 5, 9, 8 };  for (int i = 0; i < 10; i++) tree.insert(arr[i]);  Iterator\* iter = tree.create\_dft\_iterator();  for (int i = 0; i < 10; i++) Assert::IsTrue(arr[i] == iter->next());  }  TEST\_METHOD(ContainsRealTest)  {  Binary\_tree tree;  int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };  for (int i = 0; i < 10; i++) {  tree.insert(arr[i]);  Assert::IsTrue(tree.contains(arr[i]));  }  }  TEST\_METHOD(ContainsUnRealTest)  {  Binary\_tree tree;  int arr[4] = { 1,2,3,4 };  for (int i = 0; i < 4; i++) {  tree.insert(arr[i]);  }  Assert::IsFalse(tree.contains(6));  }  TEST\_METHOD(ContainsEmptyTreeTest)  {  Binary\_tree tree;  try {  tree.contains(1);  }  catch (const char\* warning)  {  Assert::AreEqual(warning, "This tree is Empty!");  }  }  TEST\_METHOD(RemovenoEmptyTest)  {  Binary\_tree tree;  int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };  for (int i = 0; i < 10; i++) tree.insert(arr[i]);  tree.remove(5);  Assert::IsFalse(tree.contains(5));  }  TEST\_METHOD(RemoveunRealTest)  {  Binary\_tree tree;  int arr[10] = {1,2,3,4,5,6,7,8,9,10 };  for (int i = 0; i < 10; i++) tree.insert(arr[i]);  try  {  tree.remove(11);  }  catch (const char\* warning)  {  Assert::AreEqual(warning, "This element isn`t in tree");  }  }  TEST\_METHOD(RemoveEmptyTest)  {  Binary\_tree tree;  try  {  tree.remove(1);  }  catch (const char\* warning)  {  Assert::AreEqual(warning, "This tree is Empty!");  }  }  };  } |
| Iterator.h |
| #pragma once  class Iterator {  public:  virtual int next() = 0;  virtual bool has\_next() = 0;  }; |
| Queue.h |
| #pragma once  #include "elem.h"  class queue {  private:  size\_t size;  Elem\* begin, \* end;  public:  queue() {  size = 0;  begin = nullptr;  end = nullptr;  }  Node\* front() {  return begin->value;  }  Node\* back() {  return end->value;  }  bool isEmpty() {  if (size == 0) return true;  else return false;  }  size\_t getSize() {  return size;  }  void push(Node\* value) {  Elem\* temp = new Elem(value, nullptr);  if (isEmpty()) begin = temp;  else end->next = temp;  end = temp;  size++;  }  void pop() {  Elem\* temp = begin;  begin = begin->next;  delete temp;  size--;  }  void clear() {  while (!isEmpty()) pop();  }  }; |
| Node.h |
| #pragma once  class Node {  public:  Node\* pLeft;  Node\* pRight;  Node\* parents;  int data;  Node(int data = NULL, Node\* pLeft = nullptr, Node\* pRight = nullptr, Node\* parents = nullptr) {  this->data = data;  this->pLeft = pLeft;  this->pRight = pRight;  this->parents = parents;  }  }; |
| Elem.h |
| #pragma once  #include "node.h"  class Elem {  public:  Elem\* next;  Node\* value;  Elem(Node\* elem\_value, Elem\* elem\_next) {  value = elem\_value;  next = elem\_next;  }  }; |
| Stack.h |
| #pragma once  #include "elem.h"  class stack {  private:  size\_t size;  Elem\* top;  public:  stack() {  size = 0;  top = nullptr;  }  Node\* getTop() {  return top->value;  }  void pop()  {  Elem\* temp = top;  top = top->next;  delete temp;  size--;  }  void push(Node\* value) {  top = new Elem(value, top);  size++;  }  size\_t getSize() {  return size;  }  bool isEmpty() {  if (size == 0) return true;  else return false;  }  void clear() {  while (!isEmpty()) pop();  }  }; |

# Пример работы





# Вывод

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