**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) |

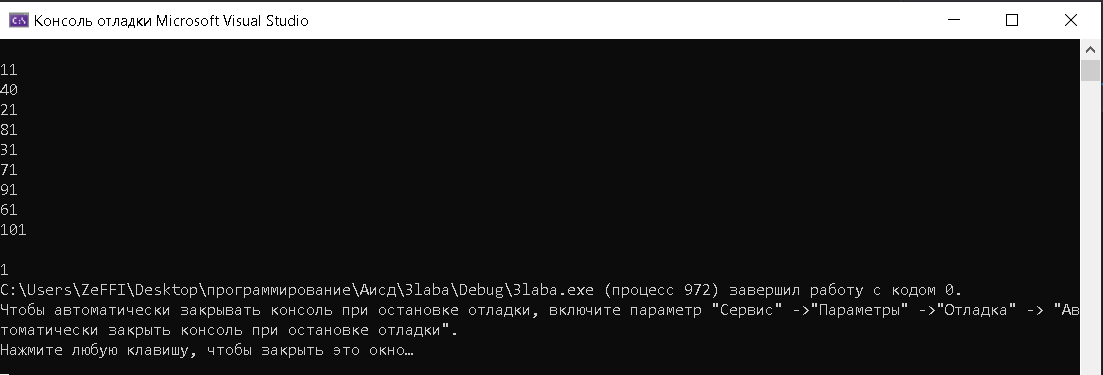
# Описание реализованных unit-тестов

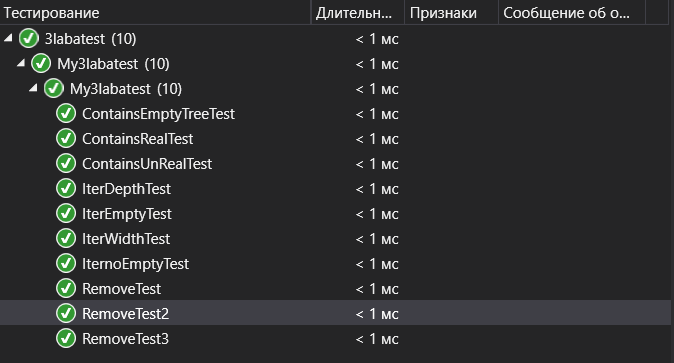
|  |  |
| --- | --- |
| Имя теста | Описание |
| IternoEmptyTest | Создание итератора на непустом дереве |
| IterEmptyTest | Создание итератора на пустом дереве |
| IterWidthTest | Проверка работы итератора обхода в ширину |
| IterDepthTest | Проверка работы итератора обхода в глубину |
| ContainsRealTest | 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 { ///depth-first iterator  private:  Node\* cur;  stack nodesStack;  bool label = true; //label showing we are working with the last element of the stack or not  public:  dft\_iterator(Node\* root) {  cur = root;  if (cur != nullptr) {  nodesStack.push(cur);  }  else {  throw "This tree is empty";  }  }  int next() override {  if (label) {  label = false;  Node\* temp = nodesStack.getTop(); // assign temp element from the stack  nodesStack.pop(); // immediately remove the assigned value  return temp->data; // return value  }  if (cur->pRight != nullptr) nodesStack.push(cur->pRight); // if the element has a right element add to the stack  if (cur->pLeft != nullptr) cur = cur->pLeft; // same with the left  else  if (!nodesStack.isEmpty()) { // if the stack is not empty  cur = nodesStack.getTop(); // assign the variable an item from the stack  nodesStack.pop(); // immediately remove the assigned value  }  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 { //breadth - first 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) { //if heap size is greater than 0  cur = nodesQueue.front(); //take an element from the heap from the beginning  nodesQueue.pop(); //remove the element from the beginning  if (cur->pLeft != nullptr) { //if element has a left element  nodesQueue.push(cur->pLeft); //insert the left element at the end of the heap  }  if (cur->pRight != nullptr) { //the same with the right  nodesQueue.push(cur->pRight);  }  return cur->data;; //return the value of the element  }  else cur = nullptr; //otherwise the element does not exist  }  bool has\_next() override {  if (cur->pLeft == nullptr && cur->pRight == nullptr) return false;  else return true;  }  };  class Binary\_tree {  private:  public:  Binary\_tree();    void insert(int data);  void print\_tree(Node\* root);  bool contains(int data);  void remove(int data);  Node\* root;  Iterator\* create\_bft\_iterator() { //creating a breadth - first iterator  return new bft\_iterator(root);  }  Iterator\* create\_dft\_iterator() { //creating a depth-first iterator  return new dft\_iterator(root);  }  };  Binary\_tree::Binary\_tree()  {  root = nullptr; //initially the root of the tree does not exist  }  void Binary\_tree::insert(int data)  {  if (root == nullptr) { //if the tree is empty, create the root of the tree and insert the value there  root = new Node(data);  }  else {  Node\* current = this->root; //assign the root of the tree to the temporary pointer  while (true) { //until the element falls into place  if (current->data < data) { //if the element that needs to be inserted is larger than the element in this cell of the tree,  //then we insert it to the right of this cell if it is free there, otherwise we go to the right cell  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) { //the same as with the right cell only on the left, if the element to be inserted is smaller  //than the element in the given cell  if (current->pLeft == nullptr) {  current->pLeft = new Node(data);  current->pLeft->parents = current;  break;  }  if (current->pLeft != nullptr) {  current = current->pLeft;  continue;  }  }  //if such an element is already in the tree, exit the loop and do not insert it  if (current->data == data) break;  }  }  }  void Binary\_tree::print\_tree(Node\* root) //a function that displays all the elements of the tree  {  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!"; //if there is no tree root, then the tree is empty  Node\* current = this->root;//assign the root of the tree to the temporary pointer  if (current->data == data) return true; // if the tree is the desired value, we immediately return the truth  while (current) { //loop until we reach a nonexistent element  if (current->data < data) current = current->pRight;  else if (current->data > data) current = current->pLeft;  else {  return true;  }  }  return false; //if nothing was found, then the element is not in the tree  }  void Binary\_tree::remove(int data)  {  if (root == nullptr) throw "This tree is Empty!"; //if there is no tree root, then the tree is empty  Node\* current = this->root;//assign the root of the tree to the temporary pointer  while (current) { //see if there is a given value in the tree at all  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 the pointer has reached a non-existent value,  //then the element is not in the tree  if (current->pLeft == nullptr && current->pRight == nullptr) { //if the element does not have a right and left element,  //then simply delete it and update the information about which element  //its parent points to  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) { //if the element does not have a left element but has a right one,  //then put the right element in its place, and delete the element itself  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) { //similar to the left  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) { //if there is both right and left element, find the largest element by value on  //the left and put it in place  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; //go to the next email, return the current email  virtual bool has\_next() = 0; //checking the existence of the next email  }; |
| 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() { //returns the element from the head of the queue without deleting. O (1)  return begin->value;  }  Node\* back() { //returns an item from the end of the queue without deleting it. O (1)  return end->value;  }  bool isEmpty() { //checks if the queue is empty. O (1)  if (size == 0) return true;  else return false;  }  size\_t getSize() {  return size;  }  void push(Node\* value) { //adds an item to the end of the queue. O (1)  Elem\* temp = new Elem(value, nullptr);  if (isEmpty()) begin = temp;  else end->next = temp;  end = temp;  size++;  }  void pop() { //removes the element from the head of the queue. O (1)  Elem\* temp = begin;  begin = begin->next;  delete temp;  size--;  }  void clear() {  while (!isEmpty()) pop();  }  }; |
| Node.h |
| #pragma once  class Node {  public:  Node\* pLeft; //the element is smaller than the current one and is on the left  Node\* pRight; //element larger than the current one and standing on the right  Node\* parents; //parent of element  int data; //the value stored in this element  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; //pointer to next element  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() { //returns an item from the stack without removing. O (1)  return top->value;  }  void pop() //removes an item from the stack. O (1)  {  Elem\* temp = top;  top = top->next;  delete temp;  size--;  }  void push(Node\* value) { //adds an item to the stack. O (1)  top = new Elem(value, top);  size++;  }  size\_t getSize() {  return size;  }  bool isEmpty() { //checks if the stack is empty. O (1)  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