**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 \_\_\_\_\_\_\_\_\_\_\_\_ Тарабурин А.П.

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)

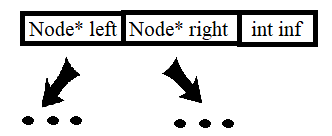
[5. Вывод 11](#_Toc58158479)

# Постановка задачи и описание реализуемых класса и алгоритмов

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

Используем для этой задачи хранение информации в узлах node, объединенных в двоичном дереве поиска bst.

Class node



Class BST



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

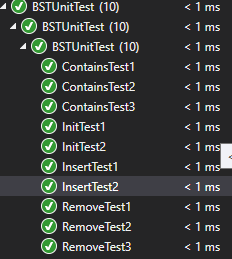
|  |  |
| --- | --- |
| Имя теста | Описание |
| InitTest1 | Создание итератора на непустом дереве |
| InitTest2 | Создание итератора на пустом дереве |
| InsertTest1 | Проверка работы итератора обхода в ширину |
| InsertTest2 | Проверка работы итератора обхода в глубину |
| ContainsTest1 | Проверка поиска существующего элемента в непустом дереве |
| ContainsTest2 | Проверка поиска несуществующего элемента в непустом дереве |
| ContainsTest3 | Проверка поиска элемента в пустом дереве |
| RemoveTest1 | Проверка удаления существующего элемента в непустом дереве |
| RemoveTest2 | Проверка удаления несуществующего элемента в непустом дереве |
| RemoveTest3 | Проверка удаления элемента в пустом дереве |

# Код программы

|  |
| --- |
| bst.h |
| #pragma once  #include "queue.h"  #include "stack.h"  #include "iterator.h"  #include "node.h"  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";  }  }  ~bft\_iterator() {  delete cur;  nodesQueue.clear();  }  int next() override {  if (nodesQueue.getSize() > 0) {  cur = nodesQueue.front();  nodesQueue.pop();  if (cur->left != nullptr) {  nodesQueue.push(cur->left);  }  if (cur->right != nullptr) {  nodesQueue.push(cur->right);  }  return cur->inf;  }  else cur = nullptr;  }  bool has\_next() override {  if (cur->left == nullptr && cur->right == nullptr) return false;  else return true;  }  };  class dft\_iterator : public Iterator {  private:  Node\* cur;  stack nodesStack;  bool flag = true;  public:  dft\_iterator(Node\* root) {  cur = root;  if (cur != nullptr) {  nodesStack.push(cur);  }  else {  throw "This tree is empty";  }  }  ~dft\_iterator() {  delete cur;  nodesStack.clear();  }  int next() override {  if (flag) {  flag = false;  Node\* temp = nodesStack.getTop();  nodesStack.pop();  return temp->inf;  }  if (cur->right != nullptr) nodesStack.push(cur->right);  if (cur->left != nullptr) cur = cur->left;  else  if (!nodesStack.isEmpty()) {  cur = nodesStack.getTop();  nodesStack.pop();  }  else cur = nullptr;  return cur->inf;  }  bool has\_next() override {  if (cur->left == nullptr && cur->right == nullptr) return false;  else return true;  }  };  class BST {  private:  Node\* root;  public:  BST() {  root = nullptr;  }  ~BST() {  clear(root);  }  bool contains(const int& value) {  if (root == nullptr) {  throw "This tree is empty";  return false;  }  Node\* temp = root;  while (value != temp->inf) {  if (value > temp->inf)  if (temp->right != nullptr) temp = temp->right;  else return false;  else  if (temp->left != nullptr) temp = temp->left;  else return false;    }  return true;  }  void insert(const int& value) {  if (root == nullptr) root = new Node(value, nullptr, nullptr);  else {  Node\* temp = root;  while (true) {  if (value > temp->inf)  if (temp->right != nullptr) temp = temp->right;  else {  temp->right = new Node(value, nullptr, nullptr);  break;  }  else  if (temp->left != nullptr) temp = temp->left;  else {  temp->left = new Node(value, nullptr, nullptr);  break;  }  }  }  }  void remove(const int& value) {  if (root == nullptr) throw "This tree is empty";  if (!contains(value)) throw "This element isn`t in tree";  else {  Node\* cur = root, \*prev = nullptr;  while (value != cur->inf) {  prev = cur;  if (value < cur->inf) cur = cur->left;  else cur = cur->right;  }  if (cur->left == nullptr && cur->right == nullptr) {  if (prev->left->inf = cur->inf) prev->left = nullptr;  else prev->right = nullptr;  delete cur;  }  else  if (cur->left != nullptr && cur->right != nullptr) {  Node\* temp;  if (cur->right->left != nullptr) {  temp = cur->right->left;  cur->inf = temp->inf;  cur->right->left = nullptr;  }  else {  temp = cur->right;  cur->inf = temp->inf;  Node\* new\_Node = new Node(temp->right->inf, temp->right->left, temp->right->right);  cur->right = new\_Node;  }  delete temp;  }  else {  if (cur->left != nullptr) {  cur->inf = cur->left->inf;  delete cur->left;  cur->left = nullptr;  }  else {  cur->inf = cur->right->inf;  delete cur->right;  cur->right = nullptr;  }  }  }  }  void clear(Node\* toDelete) {  if (toDelete != nullptr) {  clear(toDelete->left);  clear(toDelete->right);  delete toDelete;  }  }  Iterator\* create\_bft\_iterator() {  return new bft\_iterator(root);  }  Iterator\* create\_dft\_iterator() {  return new dft\_iterator(root);  }  }; |
| Main.cpp |
| #include <iostream>  #include "bst.h"  using namespace std;  int main() {  BST bst;  int arr[10] = { 7, 4, 9, 1, 6, 8, 0, 3, 5, 2};  for (int i = 0; i < 10; i++) bst.insert(arr[i]);  Iterator\* iter = bst.create\_dft\_iterator();  for (int i = 0; i < 10; i++) {  cout << iter->next();  //cout << bst.contains(arr[i]) << endl;  }  bst.remove(1);  cout << bst.contains(1);  return 0;  } |
| BSTUnitTest.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../laba3/bst.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace BSTUnitTest  {  TEST\_CLASS(BSTUnitTest)  {  public:  BST bst;  TEST\_METHOD(InitTest1)  {  bst.insert(1);  Iterator\* iter = bst.create\_dft\_iterator();  Assert::IsTrue(iter->next() == 1);  }  TEST\_METHOD(InitTest2)  {  try  {  bst.create\_dft\_iterator();  }  catch (const char\* warning)  {  Assert::AreEqual(warning, "This tree is empty");  }  }  TEST\_METHOD(InsertTest1)  {  int arr[10] = { 7, 4, 9, 1, 6, 8, 0, 3, 5, 2 };  for (int i = 0; i < 10; i++) bst.insert(arr[i]);  Iterator\* iter = bst.create\_bft\_iterator();  for (int i = 0; i < 10; i++) Assert::IsTrue(arr[i] == iter->next());  }  TEST\_METHOD(InsertTest2)  {  int arr[10] = { 7, 4, 1, 0, 3, 2, 6, 5, 9, 8 };  for (int i = 0; i < 10; i++) bst.insert(arr[i]);  Iterator\* iter = bst.create\_dft\_iterator();  for (int i = 0; i < 10; i++) Assert::IsTrue(arr[i] == iter->next());  }  TEST\_METHOD(ContainsTest1)  {  int arr[10] = { 7, 4, 9, 1, 6, 8, 0, 3, 5, 2 };  for (int i = 0; i < 10; i++) {  bst.insert(arr[i]);  Assert::IsTrue(bst.contains(arr[i]));  }  }  TEST\_METHOD(ContainsTest2)  {  int arr[10] = { 7, 4, 9, 1, 6, 8, 0, 3, 5, 2 };  for (int i = 0; i < 10; i++) bst.insert(arr[i]);  Assert::IsFalse(bst.contains(11));  }  TEST\_METHOD(ContainsTest3)  {  try  {  bst.contains(1);  }  catch (const char\* warning)  {  Assert::AreEqual(warning, "This tree is empty");  }  }  TEST\_METHOD(RemoveTest1)  {  int arr[10] = { 7, 4, 9, 1, 6, 8, 0, 3, 5, 2 };  for (int i = 0; i < 10; i++) bst.insert(arr[i]);  bst.remove(1);  Assert::IsFalse(bst.contains(1));  }  TEST\_METHOD(RemoveTest2)  {  int arr[10] = { 7, 4, 9, 1, 6, 8, 0, 3, 5, 2 };  for (int i = 0; i < 10; i++) bst.insert(arr[i]);  try  {  bst.remove(11);  }  catch (const char\* warning)  {  Assert::AreEqual(warning, "This element isn`t in tree");  }  }  TEST\_METHOD(RemoveTest3)  {  try  {  bst.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;  }  ~queue() {  clear();  }  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\* left, \* right;  int inf;  Node() {  inf = 0;  left = nullptr;  right = nullptr;  }  Node(const int& node\_inf, Node\* node\_left, Node\* node\_right) {  inf = node\_inf;  left = node\_left;  right = node\_right;  }  ~Node() {}  }; |
| 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;  }  ~Elem() {  delete value;  }  }; |
| Stack.h |
| #pragma once  #include "elem.h"  class stack {  private:  size\_t size;  Elem\* top;  public:  stack() {  size = 0;  top = nullptr;  }  ~stack() {  clear();  }  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