**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 2**

Студент гр. 9302 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Гелета А.И.

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

Санкт-Петербург

2020

# Постановка задачи

Реализовать класс бинарной кучи и различные алгоритмы для нее.

# Описание реализуемого класса и методов

class Iterator – класс итератора. Имеет два метода next() и has\_next().

class BinaryHeap

Класс кучи.

Поля:

int\* h – сама куча.

int size\_heap – размер кучи (количество элементов в ней)

int max\_size – максимально возможный размер кучи (можно задать при инициализации, в конструкторе)

Методы:

bool contains(int) -- поиск элемента в дереве по ключу.

void insert(int); -- добавление элемента в дерево по ключу.

void remove(int); -- удаление элемента дерева по ключу.

class bft\_iterator

Класс, наследуемый от Iterator, оператора обхода в ширину. Методы те же.

Поля:

int\* curr – куча.

int size – размер кучи.

int cur\_index – текущий индекс при обходе.

class dft\_iterator

Класс, наследуемый от Iterator, оператора обхода в глубину. Методы те же.

Поля:

int\* curr – сама куча.

int size – размер кучи.

int cur\_index – текущие индекс.

DuoList right\_stack – дополнительный «стек», необходимый для осуществления обхода.

# Оценка временной сложности каждого метода

|  |  |
| --- | --- |
| Метод | Сложность |
| insert | O(logn), в худшем – O(n) |
| contains | O(n) |
| remove | O(n) |

# Описание реализованных unit-тестов

TEST\_CLASS(Empty) – класс для проверки изначально пустой кучи.

error\_bft – проверяет выдаваемую ошибку при попытке обойти в ширину дальше, чем возможно.

error\_dft – проверяет выдаваемую ошибку при попытке обойти в глубину дальше, чем возможно.

hasnext\_bft – проверяет работу одноименного метода итератора обхода в ширину.

hasnext\_dft – проверяет работу одноименного метода итератора обхода в глубину.

insert\_bft – проверка вставки итератором обхода в ширину.

insert\_dft -- проверка вставки итератором обхода в глубину.

insert\_array\_rearrange – проверка вставки при превышении размера.

insert\_heapify\_correct – проверяет, восстанавливаются ли свойства кучи при вставке.

TEST\_CLASS(NotEmpty) – класс для проверки заранее инициализированной кучи.

Bft – проверяет правильность работы итератора обхода в ширину.

Dft – проверяет правильность работы итератора обхода в ширину.

Contains – проверка одноименного метода для присутствующего в куче элемента.

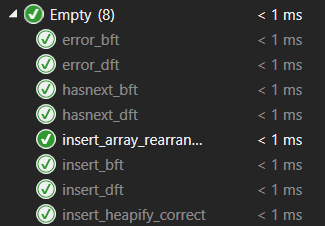
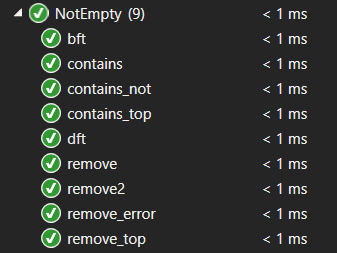
Contains\_not – аналогично, но для отсутствующего элемента.

Contains\_top – аналогично, для присутствующего верхнего (корневого элемента).

Remove – проверка удаления.

Remove2 – вторая проверка удаления.

Remove\_error – ошибка при удалении.

Remove\_top – удаление корневого элемента.

# Текст программы

|  |  |
| --- | --- |
|  | **Iterator.h** |
|  | #pragma once  class Iterator  {  public:  virtual int next() = 0;  virtual bool has\_next() = 0;  }; |
|  | **BinaryHeap.h** |
|  | #pragma once  #include "Iterator.h"  #include<iostream>  #include "DuoList.h"  class BinaryHeap  {  private:  int\* h;  int size\_heap;  int max\_size;  void siftDown(int);  void siftUp(int);  public:  BinaryHeap(int max\_size = 30) { h = new int[max\_size]; size\_heap = 0; this->max\_size = max\_size; };  ~BinaryHeap() { delete h; };  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  void out();  Iterator\* create\_bft\_iterator(); //breadth - first traverse)  Iterator\* create\_dft\_iterator(); //depth - first traverse  class bft\_iterator : public Iterator  {  public:  bft\_iterator(int\* cur, int size, int ind = 0) { this->curr = cur; this->size = size; this->cur\_index = ind; };  ~bft\_iterator() { delete curr; }  int next() override;  bool has\_next() override;  private:  int\* curr;  int size;  int cur\_index;  };  class dft\_iterator : public Iterator  {  public:  dft\_iterator(int\* cur, int size, int ind = 0) {  this->curr = cur; this->size = size;  //this->right\_stack.push(ind);  cur\_index = ind;  };  ~dft\_iterator() { delete curr; }  int next() override;  bool has\_next() override;  private:  int\* curr;  int size;  int cur\_index;  DuoList right\_stack;  };  }; |
|  | **BinaryHeap.cpp** |
|  | #include "BinaryHeap.h"  #include "DuoList.h"  void BinaryHeap::siftUp(int elem) {  if (elem != 0) {  int parent = (elem - 1) / 2;  if (h[parent] < h[elem]) {  std::swap(h[elem], h[parent]);  siftUp(parent);  }  }  }  void BinaryHeap::siftDown(int elem) {  int child1 = elem\*2 + 1, child2 = elem\*2 + 2;  if (child1 < size\_heap) {  if (child2 < size\_heap) {  int bigger;  if (h[child1] > h[child2]) bigger = child1;  else bigger = child2;  if (h[bigger] > h[elem]) {  std::swap(h[bigger], h[elem]);  siftDown(bigger);  }  }  else {  if (h[child1] > h[elem]) {  std::swap(h[child1], h[elem]);  siftDown(child1);  }  }  }  }  void BinaryHeap::insert(int info) {  if (size\_heap == max\_size) {  int\* h\_new = new int[max\_size \* 10];  for (int i = 0; i < size\_heap; i++) h\_new[i] = h[i];  delete h; h = h\_new;  max\_size \*= 10;  }  h[size\_heap] = info;  size\_heap++;  siftUp(size\_heap - 1);  }  bool BinaryHeap::contains(int info) {  for (int i = 0; i < size\_heap; i++) if (h[i] == info) return true;  return false;  }  void BinaryHeap::remove(int info) {  for (int i = 0; i < size\_heap; i++)  if (h[i] == info) {  std::swap(h[i], h[size\_heap - 1]);  size\_heap--;  siftDown(i);  return;  }  throw std::out\_of\_range("Element doesn't exist");  }  void BinaryHeap::out() {  std::cout << "\n";  for (int i = 0; i < size\_heap; i++) std::cout << h[i] << " ";  }  //breadth iterator  Iterator\* BinaryHeap::create\_bft\_iterator()  {  return new bft\_iterator(h, size\_heap, 0);  }  int BinaryHeap::bft\_iterator::next()  {  if (!has\_next()) {  throw std::out\_of\_range("No more elements");  }  int temp = curr[cur\_index];  cur\_index++;  return temp;  }  bool BinaryHeap::bft\_iterator::has\_next()  {  if (cur\_index == size) return false;  return true;  }  //deapth iterator  Iterator\* BinaryHeap::create\_dft\_iterator()  {  return new dft\_iterator(h, size\_heap, 0);  }  int BinaryHeap::dft\_iterator::next()  {  if (!has\_next()) {  throw std::out\_of\_range("No more elements");  }  int temp = curr[cur\_index];  int child\_right = cur\_index \* 2 + 2, child\_left = cur\_index \* 2 + 1;  if (cur\_index == 0) right\_stack.push\_back(0);  if (child\_right < size) {  right\_stack.push\_back(child\_right);  cur\_index = child\_left;  }  else if (child\_left < size) cur\_index = child\_left;  else {  if (right\_stack.at(right\_stack.get\_size() - 1) == 0) right\_stack.pop\_back();  else {  cur\_index = right\_stack.at(right\_stack.get\_size() - 1); //"stack" top  right\_stack.pop\_back();  }  if (size == 1) cur\_index = -1;  }  return temp;  }  bool BinaryHeap::dft\_iterator::has\_next()  {  if (size == 0) return false;  if (cur\_index == 0) return true;  if (right\_stack.IsEmpty() && cur\_index != 0) return false;  else return true;  } |
|  | **3semA3\_test.cpp** |
|  | #include "pch.h"  #include "CppUnitTest.h"  #include "../3semA3/BinaryHeap.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace My3semA3test  {  TEST\_CLASS(Empty)  {  public:  BinaryHeap heap;  Iterator\* test\_bft\_iterator;  Iterator\* test\_dft\_iterator;    TEST\_METHOD(hasnext\_bft)  {  test\_bft\_iterator = heap.create\_bft\_iterator();  Assert::AreEqual(test\_bft\_iterator->has\_next(), false);  }  TEST\_METHOD(hasnext\_dft)  {  test\_dft\_iterator = heap.create\_dft\_iterator();  Assert::AreEqual(test\_dft\_iterator->has\_next(), false);  }  TEST\_METHOD(insert\_bft)  {  heap.insert(20);  test\_bft\_iterator = heap.create\_bft\_iterator();  Assert::AreEqual(test\_bft\_iterator->next(), 20);  Assert::AreEqual(test\_bft\_iterator->has\_next(), false);  }    TEST\_METHOD(insert\_dft)  {  heap.insert(20);  test\_dft\_iterator = heap.create\_dft\_iterator();  Assert::AreEqual(test\_dft\_iterator->next(), 20);  Assert::AreEqual(test\_dft\_iterator->has\_next(), false);  }  TEST\_METHOD(error\_bft) {  heap.insert(20);  test\_bft\_iterator = heap.create\_bft\_iterator();  try {  test\_bft\_iterator->next();  test\_bft\_iterator->next();  }  catch (std::out\_of\_range e) {  Assert::AreEqual("No more elements", e.what());  }  }  TEST\_METHOD(error\_dft) {  heap.insert(20);  test\_dft\_iterator = heap.create\_dft\_iterator();  try {  test\_dft\_iterator->next();  test\_dft\_iterator->next();  }  catch (std::out\_of\_range e) {  Assert::AreEqual("No more elements", e.what());  }  }  TEST\_METHOD(insert\_heapify\_correct) {  for (int i = 0; i < 6; i++) heap.insert(i);  test\_bft\_iterator = heap.create\_bft\_iterator();  Assert::AreEqual(test\_bft\_iterator->next(), 5);  Assert::AreEqual(test\_bft\_iterator->next(), 3);  Assert::AreEqual(test\_bft\_iterator->next(), 4);  Assert::AreEqual(test\_bft\_iterator->next(), 0);  Assert::AreEqual(test\_bft\_iterator->next(), 2);  Assert::AreEqual(test\_bft\_iterator->next(), 1);  }  TEST\_METHOD(insert\_array\_rearrange) {  BinaryHeap h(1);  heap.insert(0);  heap.insert(1);  test\_bft\_iterator = heap.create\_bft\_iterator();  Assert::AreEqual(test\_bft\_iterator->next(), 1);  Assert::AreEqual(test\_bft\_iterator->next(), 0);  }  };  TEST\_CLASS(NotEmpty)  {  public:  BinaryHeap heap;  Iterator\* test\_bft\_iterator;  Iterator\* test\_dft\_iterator;  TEST\_METHOD\_INITIALIZE(setup) {  heap.insert(20);  heap.insert(2);  heap.insert(17);  heap.insert(16);  heap.insert(18);  }  TEST\_METHOD\_CLEANUP(cleanup) {  delete test\_bft\_iterator;  delete test\_dft\_iterator;  }  TEST\_METHOD(bft) {  test\_bft\_iterator = heap.create\_bft\_iterator();  Assert::AreEqual(test\_bft\_iterator->next(), 20);  Assert::AreEqual(test\_bft\_iterator->next(), 18);  Assert::AreEqual(test\_bft\_iterator->next(), 17);  Assert::AreEqual(test\_bft\_iterator->next(), 2);  Assert::AreEqual(test\_bft\_iterator->next(), 16);  Assert::AreEqual(test\_bft\_iterator->has\_next(), false);  }  TEST\_METHOD(dft) {  test\_dft\_iterator = heap.create\_dft\_iterator();  Assert::AreEqual(test\_dft\_iterator->next(), 20);  Assert::AreEqual(test\_dft\_iterator->next(), 18);  Assert::AreEqual(test\_dft\_iterator->next(), 2);  Assert::AreEqual(test\_dft\_iterator->next(), 16);  Assert::AreEqual(test\_dft\_iterator->next(), 17);  Assert::AreEqual(test\_dft\_iterator->has\_next(), false);  }  TEST\_METHOD(remove) {  heap.remove(18);  test\_bft\_iterator = heap.create\_bft\_iterator();  Assert::AreEqual(test\_bft\_iterator->next(), 20);  Assert::AreEqual(test\_bft\_iterator->next(), 16);  Assert::AreEqual(test\_bft\_iterator->next(), 17);  Assert::AreEqual(test\_bft\_iterator->next(), 2);  Assert::AreEqual(test\_bft\_iterator->has\_next(), false);  }  TEST\_METHOD(remove2) {  heap.remove(18);  heap.remove(2);  heap.remove(16);  heap.remove(17);  test\_bft\_iterator = heap.create\_bft\_iterator();  Assert::AreEqual(test\_bft\_iterator->next(), 20);  Assert::AreEqual(test\_bft\_iterator->has\_next(), false);  }  TEST\_METHOD(remove\_top) {  heap.remove(20);  test\_bft\_iterator = heap.create\_bft\_iterator();  Assert::AreEqual(test\_bft\_iterator->next(), 18);  Assert::AreEqual(test\_bft\_iterator->next(), 16);  Assert::AreEqual(test\_bft\_iterator->next(), 17);  Assert::AreEqual(test\_bft\_iterator->next(), 2);  Assert::AreEqual(test\_bft\_iterator->has\_next(), false);  }  TEST\_METHOD(contains\_top) {  Assert::AreEqual(heap.contains(20), true);  }  TEST\_METHOD(contains) {  Assert::IsTrue(heap.contains(2));  }  TEST\_METHOD(contains\_not) {  Assert::IsFalse(heap.contains(0));  }  TEST\_METHOD(remove\_error) {  try {  heap.remove(0);  }  catch (std::out\_of\_range e) {  Assert::AreEqual("Element doesn't exist", e.what());  }  }  };  } |