**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе №3**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: Двоичные деревья

**Вариант 2**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Кнауб К.В. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

# Постановка задачи

Реализовать двоичную кучу и определенные методы и реализовать unit-тесты ко всем реализуемым методам., а также сделать итераторы, которые реализуют обход в глубину и обход в ширину.

# Описание реализуемого класса и методов

**Методы необходимые для реализации:**

|  |  |  |
| --- | --- | --- |
| **Название метода** | **Описание** | **Оценка временной сложности** |
| bool contains(int) | поиск элемента в дереве по ключу | О(n) |
| void insert(int) | добавление элемента в дерево по ключу | O(logn) |
| void remove(int) | удаление элемента дерева по ключу | O(n) |
| void check(int, int) | упорядочивание кучи | O(log(n)) |
| void top(int), | поднимает элемент к правильной позиции, | O(log(n)) |
| void bottom(int) | опускает к правильной позиции; | O(log(n)) |
| void out() | вывод элементов кучи в виде дерева | O(n) |

**Итераторы, которые представляют собой интерфейс, предоставляющий доступ к элементам двоичной кучи:**

|  |  |  |
| --- | --- | --- |
| **Название итератора** | **Описание** | **Оценка временной сложности** |
| Iterator create\_dft\_iterator() | итератор, реализующий метод прямого обхода в глубину (depth-first traverse); | O(1) |
| Iterator create\_bft\_iterator() | итератор, реализующий метод горизонтального обхода в ширину (breadth-first traverse) | O(1) |

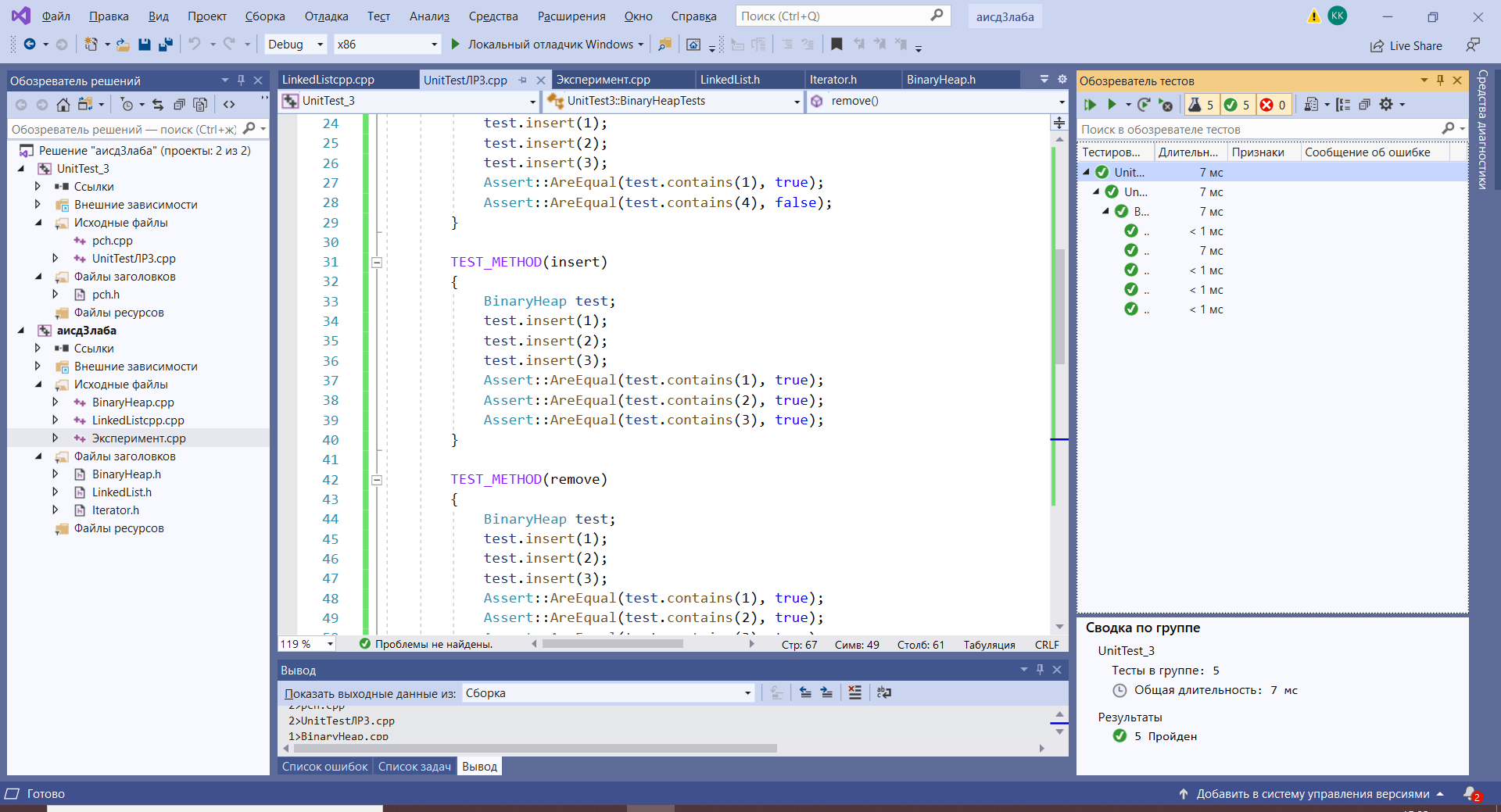
**BinaryHeap**

|  |  |
| --- | --- |
| **Классы** | **Поля** |
| dft\_iterator | int\* cur; int index; int size; |
| bft\_iterator | int\* cur; int index; int size; LinkedList stk; |

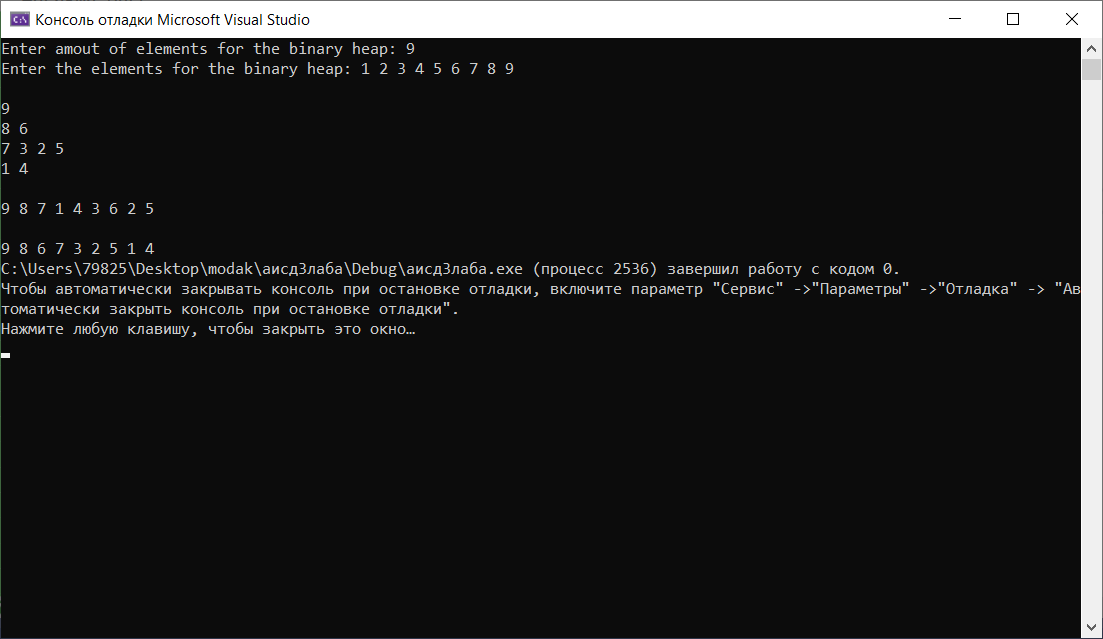
# Описание реализованных unit-тестов

Для проверки реализованных методов были написаны unit-тесты.

|  |  |
| --- | --- |
| **Название теста** | **Описание** |
| contains\_test | проверка наличия или отсутствия определенных элементов; |
| insert\_test | проверка наличия каждого элемента; |
| remove\_test | проверка, затем удаление одного элемента и повторная проверка; |
| dft\_iterator\_test | проверка каждого элемента из массива кучи, выводимого с помощью итератора и правильной кучи; |
| dft\_iterator\_test | проверка элементов массива и элементов кучи, выводимой с помощью итератора. |



# Пример работы



# Листинг

|  |
| --- |
| **Heap.h** |
| #pragma once  #include "List.h"  class BinaryHeap  {  int MaxSize = 10;  int\* root;  int HeapSize;  void check(int, int);  void top(int);  void bottom(int);  public:  BinaryHeap();  ~BinaryHeap();  bool contains(int);  void insert(int);  void remove(int);  void out();  Iterator\* create\_dft\_iterator();  class dft\_iterator : public Iterator  {  public:  dft\_iterator(int);  int next(BinaryHeap\*) override;  bool has\_next(BinaryHeap\*) override;  private:  int index;  LinkedList stk;  };  Iterator\* create\_bft\_iterator();  class bft\_iterator :public Iterator  {  public:  bft\_iterator(int);  int next(BinaryHeap\*) override;  bool has\_next(BinaryHeap\*) override;  private:  int index;  };  }; |
| **Heap.cpp** |
| #include "Heap.h"  #include <iostream>  using namespace std;  BinaryHeap::BinaryHeap()  {  HeapSize = 0;  root = new int[MaxSize];  }  BinaryHeap::~BinaryHeap()  {  delete[] root;  }  bool BinaryHeap::contains(int key)  {  for (int i = 0; i < HeapSize; i++)  {  if (key == root[i])  return true;  }  return false;  }  void BinaryHeap::top(int i)  {  int parent = (i - 1) / 2;  if (i > 0 && root[parent] < root[i])  {  int temp = root[i];  root[i] = root[parent];  root[parent] = temp;  top(parent);  }  }  void BinaryHeap::bottom(int i)  {  int left = 2 \* i + 1;  int right = 2 \* i + 2;  int greater;  if (right<HeapSize && root[right]>root[left])  greater = right;  else  greater = left;  if (greater<HeapSize && root[greater]>root[i])  {  int temp = root[i];  root[i] = root[greater];  root[greater] = temp;  bottom(greater);  }  }  void BinaryHeap::check(int key, int i)  {  int parent = (i - 1) / 2;  if (key >= root[parent])  top(i);  else  bottom(i);  }  void BinaryHeap::insert(int key)  {  if (HeapSize == MaxSize)  {  int\* transit = new int[MaxSize + 10];  for (int i = 0; i < HeapSize; i++)  transit[i] = root[i];  delete root;  root = transit;  MaxSize += 10;  }  root[HeapSize] = key;  HeapSize++;  check(key, HeapSize - 1);  }  void BinaryHeap::remove(int key)  {  if (this->HeapSize == 0)  throw "Heap is empty";  int i = 0;  while (root[i] != key)  i++;  if (i == HeapSize)  {  printf("\nThere is not such element\n");  return;  }  root[i] = root[HeapSize - 1];  HeapSize--;  check(key, i);  }  void BinaryHeap::out()  {  int i = 0;  int k = 1;  while (i < HeapSize)  {  while ((i < k) && (i < HeapSize))  {  cout << root[i] << ' ';  i++;  }  cout << endl;  k = k \* 2 + 1;  }  }  Iterator\* BinaryHeap::create\_dft\_iterator()  {  return new dft\_iterator(0);  }  BinaryHeap::dft\_iterator::dft\_iterator(int index = 0)  {  this->index = index;  }  bool BinaryHeap::dft\_iterator::has\_next(BinaryHeap\* binheap)  {  if (index == 0 && binheap->HeapSize != 0)  return true;  if (binheap->HeapSize == 0)  return false;  if (index == 0)  return true;  if (!stk.isEmpty() && index != 0)  return false;  else  return true;  }  int BinaryHeap::dft\_iterator::next(BinaryHeap\* binheap)  {  if (!has\_next(binheap))  throw "The last element in the heap";  int var = binheap->root[index];  int left = index \* 2 + 1, right = index \* 2 + 2;  if (index == 0)  stk.push\_back(0);  if (right < binheap->HeapSize)  {  stk.push\_back(right);  index = left;  }  else if (left < binheap->HeapSize)  index = left;  else  {  if (stk.at(stk.get\_size() - 1) == 0)  stk.pop\_back();  else  {  index = stk.at(stk.get\_size() - 1);  stk.pop\_back();  }  if (binheap->HeapSize == 1)  index = -1;  }  return var;  }  Iterator\* BinaryHeap::create\_bft\_iterator()  {  return new bft\_iterator(0);  }  BinaryHeap::bft\_iterator::bft\_iterator(int index = 0)  {  this->index = 0;  }  bool BinaryHeap::bft\_iterator::has\_next(BinaryHeap\* binheap)  {  return index != binheap->HeapSize;  }  int BinaryHeap::bft\_iterator::next(BinaryHeap\* binheap)  {  if (!has\_next(binheap))  throw "The last element in the heap";  this->index++;  return binheap->root[index - 1];  } |
| **List.h** |
| #pragma once  #pragma once  class LinkedList  {  public:  LinkedList();  ~LinkedList();  void push\_back(int newEl);  void push\_front(int newEl);  void pop\_back();  void pop\_front();  void insert(int newEl, size\_t index);  int at(size\_t index) const;  void remove(size\_t index);  size\_t get\_size();  void print\_to\_console();  void clear();  void set(size\_t index, int newData);  bool isEmpty();  size\_t find\_first(LinkedList \*list);  bool isPrinted();  private:  class Node  {  public:  Node(int data, Node\* next = nullptr, Node\* prev =nullptr)  {  this->data = data;  this->next = next;  this->prev = prev;  }  int data;  Node\* next;  Node\* prev;  };    Node\* head;  Node\* tail;  size\_t size;  };  class BinaryHeap;  class Iterator  {  public:  virtual int next(BinaryHeap\*) = 0;  virtual bool has\_next(BinaryHeap\*) = 0;  }; |
| **List.cpp** |
| #include "List.h"  #include <iostream>  #include <stdexcept>  #include <string>  using namespace std;  LinkedList::LinkedList()  {  head = nullptr;  tail = nullptr;  size = 0;  }  LinkedList::~LinkedList()  {  clear();  }  void LinkedList::insert(int newEl, size\_t index)  {  Node\* temp\_pointer = nullptr, \* cur;  Node\* temp = new Node(newEl);  cur = head;  int i = 0;  if (index >= size || index < 0)  throw out\_of\_range("Index is greater than list size");  while (cur != NULL && i<int(index))  {  cur = cur->next;  i++;  }  if (cur == NULL)  {  if (head == NULL)  {  cout << "List is empty\n";  return;  }  return;  }  if (cur != head)  {  temp\_pointer = cur->prev;  cur->prev = temp;  temp->next = cur;  temp->prev = temp\_pointer;  }  else  push\_front(newEl);  if (temp\_pointer != NULL)  temp\_pointer->next = temp;  size++;  }  void LinkedList::remove(size\_t index)  {  size\_t counter = 0;  Node\* cur = head;  if (index >= size || index < 0)  throw out\_of\_range("Index is greater than list size");  while (counter != index)  {  cur = cur->next;  counter++;  }  if (cur == head)  {  pop\_front();  return;  }  if (cur == tail)  {  pop\_back();  return;  }  Node\* prev, \* next;  prev = cur->prev;  next = cur->next;  if (prev != NULL)  prev->next = cur->next;  if (next != NULL)  next->prev = cur->prev;  delete cur;  size--;  }  void LinkedList::push\_back(int newEl)  {  Node\* temp;  if (size == 0)  {  head = new Node(newEl);  tail = head;  }  else  {  temp = tail;  tail->next = new Node(newEl);  tail = tail->next;  tail->prev = temp;  }  size++;  }  void LinkedList::push\_front(int newEl)  {  Node\* temp;  if (size == 0)  {  head = new Node(newEl);  tail = head;  }  else  {  temp = head;  head = new Node(newEl, head);  temp->prev = head;  }  size++;  }  void LinkedList::pop\_back()  {  if (size == 0)  return;  if (size == 1)  {  delete head;  head = nullptr;  tail = nullptr;  }  else  {  Node\* cur = tail;  cur = cur->prev;  cur->next = nullptr;  delete tail;  tail = cur;  cur->prev = tail->prev;  }  size--;  }  void LinkedList::pop\_front()  {  if (size == 0)  return;  if (size == 1)  {  delete head;  head = nullptr;  tail = nullptr;  }  else  {  Node\* cur = head;  cur = cur->next;  cur->prev = nullptr;  delete head;  head = cur;  }  size--;  }  int LinkedList::at(size\_t index) const  {  if (size == 0)  cout << "List is empty\n";  if (index >= size || index < 0) {  throw out\_of\_range("Index is greater than list size");  }  if((index<size)&&(size!=0))  {  size\_t counter = 0;  Node\* cur = head;  while (counter != index)  {  cur = cur->next;  counter++;  }  return cur->data;  }  }  size\_t LinkedList::get\_size()  {  return size;  }  void LinkedList::print\_to\_console()  {  Node\* cur = head;  cout << "List elements:\t";  while (cur != nullptr)  {  cout << cur->data << ' ';  cur=cur->next;  }  cout << '\n';  }  void LinkedList::clear()  {  while (head)  {  tail = head->next;  delete head;  head = tail;  size--;  }  }  void LinkedList::set(size\_t index, int newData)  {  Node\* cur = head;  if (index >= size || index < 0)  throw out\_of\_range("Index is greater than list size");  size\_t counter = 1;  while (counter <= index)  {  counter++;  cur = cur->next;  }  cout << "Element " << cur->data;  cur->data = newData;  cout << " replaced by " << cur->data<<'\n';  }  bool LinkedList::isEmpty()  {  if (head != NULL)  return true;  else  return false;  }  bool LinkedList::isPrinted()  {  Node\* cur = head;  while (cur != nullptr)  {  cout << cur->data << ' ';  cur = cur->next;  }  cout << '\n';  if (cur == nullptr)  return true;  else  return false;  }  size\_t LinkedList::find\_first(LinkedList\* list)  {  int counter = 0;  Node\* curL1, \* curL2;  curL1 = (\*list).head;  curL2 = head;  if (curL1==nullptr || curL2==nullptr)  return 0;  if ((\*list).get\_size() > size)  {  while (curL2->data != curL1->data && curL1->next != NULL)  {  curL1 = curL1->next;  counter++;  }  while (curL2->data == curL1->data && curL2->next != NULL && curL1->next != NULL)  {  curL2 = curL2->next;  curL1 = curL1->next;  }  if (curL2->next == NULL)  {  return counter;  }  else  return 0;  }  else  {  if ((\*list).get\_size() == size)  if (curL1->data == curL2->data)  goto analysis;  else  return 0;  while (curL1->data != curL2->data && curL2->next != NULL)  {  curL2 = curL2->next;  counter++;  }  analysis:  while (curL1->data == curL2->data && curL1->next != NULL && curL2->next != NULL)  {  curL1 = curL1->next;  curL2 = curL2->next;  }  if (curL1->next == NULL)  {  return counter;  }  else  return 0;  }  } |
| **Main.cpp** |
| #include "Heap.h"  #include <iostream>  using namespace std;  void main()  {  BinaryHeap binheap;  int input\_number, quantity;  cout << "Enter amout of elements for the binary heap: ";  cin >> quantity;  cout << "Enter the elements for the binary heap: ";  for (int i = 0; i < quantity; i++)  {  cin >> input\_number;  binheap.insert(input\_number);  }  cout << endl;  binheap.out();  cout << endl;  Iterator\* heap\_dft\_iterator = binheap.create\_dft\_iterator();  while (heap\_dft\_iterator->has\_next(&binheap))  cout << heap\_dft\_iterator->next(&binheap) << ' ';  cout << "\n\n";  Iterator\* heap\_bft\_iterator = binheap.create\_bft\_iterator();  while (heap\_bft\_iterator->has\_next(&binheap))  cout << heap\_bft\_iterator->next(&binheap) << ' ';  } |
| **UnitTest3.cpp** |
| #include "pch.h"  #include "CppUnitTest.h"  #include "C:\Users\79825\Desktop\modak\аисд3лаба\Heap.h"  #include "C:\Users\79825\Desktop\modak\аисд3лаба\List.h"  #include "C:\Users\79825\Desktop\modak\аисд3лаба\Heap.cpp"  #include "C:\Users\79825\Desktop\modak\аисд3лаба\List.cpp"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTest3  {  TEST\_CLASS(BinaryHeapTests)  {  public:  TEST\_METHOD(contains\_test)  {  BinaryHeap test;  test.insert(1);  test.insert(2);  test.insert(3);  Assert::AreEqual(test.contains(1), true);  Assert::AreEqual(test.contains(4), false);  }  TEST\_METHOD(insert\_test)  {  BinaryHeap test;  test.insert(1);  test.insert(2);  test.insert(3);  Assert::AreEqual(test.contains(1), true);  Assert::AreEqual(test.contains(2), true);  Assert::AreEqual(test.contains(3), true);  }  TEST\_METHOD(remove\_test)  {  BinaryHeap test;  test.insert(5);  test.insert(4);  test.insert(3);  test.insert(2);  test.insert(1);  Assert::AreEqual(test.contains(1), true);  Assert::AreEqual(test.contains(2), true);  Assert::AreEqual(test.contains(3), true);  Assert::AreEqual(test.contains(4), true);  Assert::AreEqual(test.contains(5), true);  test.remove(3);  test.remove(4);  Assert::AreEqual(test.contains(1), true);  Assert::AreEqual(test.contains(2), true);  Assert::AreEqual(test.contains(3), false);  Assert::AreEqual(test.contains(4), false);  Assert::AreEqual(test.contains(5), true);  try  {  test.remove(10);  }  catch (const char\* error)  {  Assert::AreEqual(error, "\nThere is no such element\n");  }    }  TEST\_METHOD(dft\_iterator\_test)  {  BinaryHeap test;  int array[5] = { 1,2,3,4,5 };  int check[5] = { 5,4,1,3,2 };  for (int i = 0; i < 5; i++)  test.insert(array[i]);  Iterator\* heap\_dft\_iterator = test.create\_dft\_iterator();  for (int i = 0; i < 5; i++)  Assert::IsTrue(heap\_dft\_iterator->next(&test) == check[i]);    }  TEST\_METHOD(bft\_iterator\_test)  {  BinaryHeap test;  int array[5] = { 1,2,3,4,5 };  int check[5] = { 5,4,2,1,3 };  for (int i = 0; i < 5; i++)  test.insert(array[i]);  Iterator\* heap\_bft\_iterator = test.create\_bft\_iterator();  for (int i = 0; i < 5; i++)  Assert::IsTrue(heap\_bft\_iterator->next(&test) == check[i]);  }  };  } |