**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

ОТЧЁТ

**по лабораторной работе №3**

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

**Тема: «Двоичные деревья»**

**Вариант 2.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Новокрещенов Д.К. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

## Постановка задачи и описание реализуемого класса и методов.

Для решения поставленной задачи необходимо реализовать двоичную кучу и итераторы обхода в глубину и ширину. Для этого мне понадобились классы «BinaryHeap», «Iterator», «bft\_iterator», «dft\_iterator».

**class BinaryHeap**

|  |  |  |
| --- | --- | --- |
| **Компонент** | **Описание** | |
| BiList Heap; | Список элементов кучи | |
| size\_t size; | Размер кучи | |
| size\_t max\_size; | Максимальный размер кучи | |
| class bft\_iterator | Итератор обхода в ширину | |
| class dft\_iterator | Итератор обхода в глубину | |
| **Метод** | **Описание** | **Оценка временной сложности** |
| void heapify(size\_t); | Восстановление кучи по правилам | О(n) |
| void insert\_ready\_heap(int\*, size\_t); | Вставка готовой двоичной кучи | О(n) |
| bool contains(int); | Поиск элемента | О(n) |
| void insert(int); | Вставка элемента в кучу | О(logn) |
| void remove(int); | Удаление элемента по ключу | О(n) |
| void resize\_heap(); | Изменение размера кучи | O(n) |
| void print\_heap(); | Вывод двоичного дерева | О(n) |
| bool isHeap(size\_t); | Проверка на соответствие основным правилам бинарной кучи | O(n) |
| Iterator\* create\_dft\_iterator(); | Создание итератора обхода в глубину | O(1) |
| Iterator\* create\_bft\_iterator(); | Создание итератора обхода в ширину | O(1) |

**class Iterator**

|  |  |
| --- | --- |
| **Компонент** | **Описание** |
| virtual int next(); | Получения значения следующего элемента |
| virtual bool has\_next(); | Проверка на наличие следующего элемента |

**class bft\_iterator**

|  |  |  |
| --- | --- | --- |
| **Компонент** | **Описание** | |
| BinaryHeap\* data; | Указатель на элемент кучи | |
| size\_t index; | Переменная индекса элемента | |
| **Метод** | **Описание** | **Оценка временной сложности** |
| int next() override; | Получения значения следующего элемента | O(1) |
| bool has\_next() override; | Проверка на наличие следующего элемента | O(1) |

**class dft\_iterator**

|  |  |  |
| --- | --- | --- |
| **Компонент** | **Описание** | |
| BinaryHeap\* data; | Указатель на кучу | |
| size\_t index; | Переменная индекса элемента | |
| BiList stack\_right; | Аналог стека для реализации обхода в глубину | |
| **Метод** | **Описание** | **Оценка временной сложности** |
| int next() override; | Получения значения следующего элемента | O(1) |
| bool has\_next() override; | Проверка на наличие следующего элемента | O(1) |

## Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| **Наименование теста** | **Описание** |
| correct\_heap\_after\_insert | Проверка на свойства кучи после вставки одного элемента |
| correct\_heap\_after\_remove | Проверка на свойства кучи после удаления элемента |
| correct\_insert | Проверка на корректную вставку элемента |
| correct\_insert\_ready\_heap | Проверка на корректную вставку готовой кучи |
| correct\_remove | Проверка на корректное удаление элемента, который встречается один раз в куче |
| correct\_remove\_2 | Проверка на корректное удаление элемента, который встречается несколько раз в куче |
| isHeap\_test | Проверка метода isHeap |
| test\_bft\_iterator | Проверка корректной работы итератора обхода в ширину |
| test\_dft\_iterator | Проверка корректной работы итератора обхода в глубину |

## Листинги программы и тестов

|  |
| --- |
| BinaryHeap.h |
| #pragma once  #include "List.h"  #include "Iterator.h"  class BinaryHeap  {  private:  int\* Heap;  size\_t size;  size\_t max\_size;  void heapify(size\_t);  public:  BinaryHeap();  ~BinaryHeap();  void insert\_ready\_heap(int\*, size\_t);  bool contains(int);  void insert(int);  void remove(int);  void print\_heap();  void resize\_heap();  bool isHeap(size\_t);  Iterator\* create\_dft\_iterator();  Iterator\* create\_bft\_iterator();  class bft\_iterator : public Iterator  {  public:  bft\_iterator(BinaryHeap\*, size\_t);  int next() override;  bool has\_next() override;  private:  BinaryHeap\* data;  size\_t index;  };  class dft\_iterator : public Iterator  {  public:  dft\_iterator(BinaryHeap\*, size\_t);  int next() override;  bool has\_next() override;  private:  BinaryHeap\* data;  size\_t index;  BiList stack\_right;  };  }; |
| BinaryHeap.cpp |
| #pragma once  #include "List.h"  #include "BinaryHeap.h"  #include <iostream>  bool BinaryHeap::isHeap(size\_t index=0)  {  size\_t left\_child = 2 \* index + 1, right\_child = 2 \* index + 2;  if (left\_child >= size)  return true;  if (right\_child >= size && Heap[index] >= Heap[left\_child])  return true;  if (Heap[index] >= Heap[left\_child] && Heap[index] >= Heap[right\_child])  return (isHeap(left\_child) && isHeap(right\_child));  return false;  }  void BinaryHeap::heapify(size\_t index=0)  {  size\_t root = index;  size\_t left = 2 \* root + 1, right = left + 1;  if (left<size && Heap[left]>Heap[root])  root = left;  if (right<size && Heap[right]>Heap[root])  root = right;  if (root != index)  {  std::swap(Heap[root], Heap[index]);  heapify(root);  }  }  BinaryHeap::BinaryHeap()  {  this->size = 0;  this->max\_size = 250;  Heap = new int[max\_size];  }  BinaryHeap::~BinaryHeap()  {  delete[]Heap;  }  void BinaryHeap::insert\_ready\_heap(int\* ready\_heap, size\_t size)  {  while (size > max\_size)  max\_size += 250;  delete[]Heap;  Heap = new int[max\_size];  this->size = size;  for (size\_t i = 0; i < size; i++)  Heap[i] = ready\_heap[i];  for (size\_t i = 0; i < this->size; i += 2 \* i + 1)  heapify();  }  bool BinaryHeap::contains(int key)  {  if (size == 0)  return false;  for (size\_t i = 0; i < size; i++)  if (Heap[i] == key)  return true;  return false;  }  void BinaryHeap::insert(int value)  {  size\_t size\_heap = size;  int parent = (size\_heap - 1) / 2;  if (size+1 > max\_size)  resize\_heap();  Heap[size] = value;  while (parent >= 0 && size\_heap > 0)  {  if (Heap[size\_heap] > Heap[parent])  std::swap(Heap[size\_heap], Heap[parent]);  size\_heap = parent;  parent = (size\_heap - 1) / 2;  }  size++;  }  void BinaryHeap::remove(int key)  {  int control = 0;  for (size\_t i = 0; i < size-control; i++)  {  if (Heap[i] == key)  control++;  Heap[i] = Heap[i + control];  }  size -= control;  for (size\_t i = 0; i < size; i += 2 \* i + 1)  heapify();  }  void BinaryHeap::print\_heap()  {  size\_t i = 0, k = 1;  while (i < size) {  while ((i < k) && (i < size)) {  std::cout << Heap[i] << " ";  i++;  }  std::cout << std::endl;  k = k \* 2 + 1;  }  }  void BinaryHeap::resize\_heap()  {  max\_size += 250;  int\* temp = new int[max\_size];  for (size\_t i = 0; i < size; i++)  temp[i] = Heap[i];  delete []Heap;  Heap = temp;  }  BinaryHeap::bft\_iterator::bft\_iterator(BinaryHeap\* cur, size\_t index=0)  {  data = cur;  this->index = index;  }  int BinaryHeap::bft\_iterator::next()  {  if (!has\_next())  throw "No more elements in heap";  this->index++;  return data->Heap[index-1];  }  bool BinaryHeap::bft\_iterator::has\_next()  {  if (data->size != index) return true;  return false;  }  BinaryHeap::dft\_iterator::dft\_iterator(BinaryHeap\* cur, size\_t index=0)  {  data = cur;  this->index = index;  }  int BinaryHeap::dft\_iterator::next()  {  if (!has\_next())  throw "No more elements in heap";  int temp = data->Heap[index];  size\_t right\_child = index \* 2 + 2, left\_child = index \* 2 + 1;  if (index == 0)  stack\_right.push\_back(0);  if (right\_child < data->size)  {  stack\_right.push\_back(right\_child);  index = left\_child;  }  else  {  if (left\_child < data->size)  index = left\_child;  else  {  if (stack\_right.at(stack\_right.get\_size() - 1) == 0)  stack\_right.pop\_back();  else  {  index = stack\_right.at(stack\_right.get\_size() - 1);  stack\_right.pop\_back();  }  if (data->size == 1)  index--;  }  }  return temp;  }  bool BinaryHeap::dft\_iterator::has\_next()  {  if (data->size == 0)  return false;  if(index == 0)  return true;  if (stack\_right.isEmpty() && index != 0)  return false;  else  return true;  }  Iterator\* BinaryHeap::create\_dft\_iterator()  {  return new dft\_iterator(this);  }  Iterator\* BinaryHeap::create\_bft\_iterator()  {  return new bft\_iterator(this);  } |
| UnitTest.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../Lab3/BinaryHeap.cpp"  #include "../Lab3/List.cpp"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace SuperTestMachine  {  TEST\_CLASS(SuperTestMachine)  {  public:  BinaryHeap Test;  TEST\_METHOD(isHeap\_test)  {  int test[5] = { 10,5,4,3,2 };  Test.insert\_ready\_heap(test, 5);  Assert::AreEqual(Test.isHeap(), true);  }  TEST\_METHOD(correct\_insert)  {  for (size\_t i = 0; i < 5; i++)  Test.insert(i);  for (size\_t i = 0; i < 5; i++)  Assert::AreEqual(Test.contains(i), true);  }  TEST\_METHOD(correct\_heap\_after\_insert)  {  Test.insert(123);  Assert::AreEqual(Test.isHeap(), true);  }  TEST\_METHOD(correct\_insert\_ready\_heap)  {  int test[5] = { 10,5,4,3,2 };  Test.insert\_ready\_heap(test, 5);  for (size\_t i = 0; i < 5; i++)  Assert::AreEqual(Test.contains(test[i]), true);  }  TEST\_METHOD(correct\_remove)  {  int test[5] = { 10,5,4,3,2 };  Test.insert\_ready\_heap(test, 5);  Test.remove(10);  Assert::AreEqual(Test.contains(10), false);  }  TEST\_METHOD(correct\_heap\_after\_remove)  {  int test[5] = { 10,5,4,3,2 };  Test.insert\_ready\_heap(test, 5);  Test.remove(10);  Assert::AreEqual(Test.isHeap(), true);  }  TEST\_METHOD(test\_bft\_iterator)  {  int test[5] = { 10,5,4,3,2 };  Test.insert\_ready\_heap(test, 5);  Iterator\* Iter = Test.create\_bft\_iterator();  for (size\_t i = 0; i < 5; i++)  Assert::AreEqual(Iter->next(), test[i]);  Assert::AreEqual(Iter->has\_next(), false);  }  TEST\_METHOD(test\_dft\_iterator)  {  int test[5] = { 10,5,4,3,2 };  Test.insert\_ready\_heap(test, 5);  int result[5] = {10,5,3,2,4};  Iterator\* Iter = Test.create\_dft\_iterator();  for (size\_t i = 0; i < 5; i++)  Assert::AreEqual(Iter->next(), result[i]);  Assert::AreEqual(Iter->has\_next(), false);  }  TEST\_METHOD(test\_resize\_heap)  {  int test[5] = { 10,5,4,3,2 };  Test.insert\_ready\_heap(test, 5);  Test.resize\_heap();  try {  for (size\_t i = 0; i < 450; i++)  Test.insert(i);  }  catch(const char\*)  {  Assert::AreEqual(21, 1); //если исключение, то будет false  }  Assert::AreEqual(1, 1); //если все операции прошли успешно  }  };  } |
| Iterator.h |
| #pragma once  class Iterator {  public:  virtual int next() = 0;  virtual bool has\_next() = 0;  }; |

## Вывод

В ходе выполнения работы я научился реализовывать двоичную кучу, итераторы обхода в глубину и ширину.