**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: «Программирование двунаправленных связных списков»**

**Вариант 2.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Давтян С.Д. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

## Постановка задачи и описание реализуемого класса и методов.

Задание: Реализовать двоичную кучу и её методы. Также создать итераторы, реализующие методы обхода в ширину и глубину

Для этого мне понадобились классы «BinaryHeap», «List», «Iterator», «bft\_iterator», «dft\_iterator».

class BinaryHeap

|  |  |  |
| --- | --- | --- |
| Поля | Описание | |
| int\* heap | Куча | |
| size\_t size | Размер кучи | |
| size\_t capacity | Ёмкость кучи | |
| Методы | Описание | Оценка временной сложности |
| bool contains(int) | Поиск элемента в дереве по ключу | О(n) |
| void insert(int) | Добавление элемента в кучу по ключу | О(log(n)) |
| void remove(int) | Удаление кучи по ключу | О(n) |
| void heapify(size\_t) | Восстановление кучи | О(n) |
| void input\_heap(int\*, size\_t); | Преобразование массива в двоичную кучу | О(n) |
| void print\_heap() | Вывод двоичной кучи | О(n) |
| void sift\_up(int) | Просеивание вверх | О(log(n)) |
| Iterator\* create\_bft\_iterator() | Создание итератора обхода в ширину | O(1) |
| Iterator\* create\_dft\_iterator() | Создание итератора обхода в глубину | O(1) |

class Iterator

|  |  |
| --- | --- |
| Методы | Описание |
| virtual int next() | Получения значения текущего элемента, переход к следующему |
| virtual bool has\_next() | Проверка на наличие следующего элемента |

class bft\_iterator

|  |  |
| --- | --- |
| Поля | Описание |
| int\* current | Куча |
| size\_t size | Размер |
| size\_t index | Индекс текущего элемента |

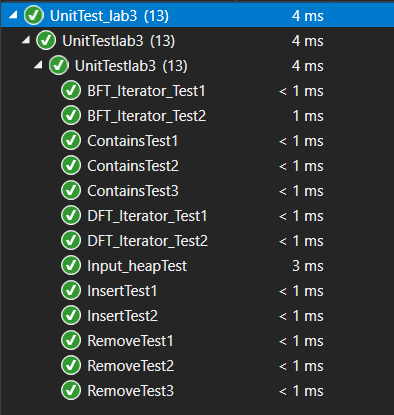
class dft\_iterator

|  |  |
| --- | --- |
| Поля | Описание |
| int\* current | Куча |
| size\_t size | Размер |
| size\_t index | Индекс текущего элемента |
| List stack\_right | «Стек» для обхода в глубину |

## Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| Название теста | Что проверяет |
| Input\_heapTest | Проверка чтения двоичной кучи из массива |
| ContainsTest1 | Проверка метода «contains» на куче, в которой есть искомый элемент |
| ContainsTest2 | Проверка метода «contains» на куче, в которой нет искомого элемента |
| ContainsTest3 | Проверка метода «contains» на пустой куче |
| InsertTest1 | Проверка метода «insert» на непустой куче |
| InsertTest2 | Проверка метода «insert» на пустой куче |
| RemoveTest1 | Проверка метода «remove» на непустой куче |
| RemoveTest2 | Проверка метода «insert» на пустой куче |
| BFT\_Iterator\_Test1 | Проверка метода «next» итератора обхода в ширину в непустой куче |
| BFT\_Iterator\_Test2 | Проверка метода «next» итератора обхода в ширину в пустой куче |
| DFT\_Iterator\_Test1 | Проверка метода «next» итератора обхода в ширину в непустой куче |
| DFT\_Iterator\_Test2 | Проверка метода «next» итератора обхода в ширину в пустой куче |

## Результат выполнения всех unit-тестов



## Листинг

|  |
| --- |
| List.h |
| #pragma once  #include <iostream>  class List  {  public:  List();  List(const List &lst);  ~List();  void push\_back(int data);  void push\_front(int data);  void pop\_back();  void pop\_front();  void insert(size\_t index, int data);  void remove(size\_t index);  void clear();  void print\_to\_console();  size\_t GetSize();  int at(const int index);  void set(size\_t index, int data);  bool isEmpty();  void push\_front(List lst);  private:  class Node  {  public:  Node\* next;  Node\* prev;  int data;  Node(int data = 0, Node\* next = nullptr, Node\* prev =nullptr)  {  this->data = data;  this->next = next;  this->prev = prev;  }  };  Node\* head;  Node\* tail;  size\_t size;  }; |
| List.cpp |
| #include "List.h"  // constructor  List::List()  {  size = 0;  head = nullptr;  tail = nullptr;  }  // copy constructor  List::List(const List &lst)  {  size = 0;  head = nullptr;  tail = nullptr;  Node\* temp = lst.head;  for (int i = 0; i < lst.size; i++)  {  push\_back(temp->data);  temp = temp->next;  }  }  // destructor  List::~List()  {  clear();  }  // inserting an element at the end of the list  void List::push\_back(int data)  {  if (head == nullptr)  {  this->head = this->tail = new Node(data);  }  else  {  Node\* temp = new Node(data, nullptr, tail);  tail->next = temp;  tail = temp;  }  size++;  }  // inserting an element at the beginning of the list  void List::push\_front(int data)  {  if (!(isEmpty()))  {  Node\* lasthead = head;  head = new Node(data, head);  lasthead->prev = head;  }  else  {  head = new Node(data, head);  }  size++;  }  // remove the last element  void List::pop\_back()  {  if (isEmpty())  throw "Error! Linked list is Empty.";  else  {  Node\* todelete = tail;  tail = tail->prev;  delete todelete;  size--;  }  }  // remove the first element  void List::pop\_front()  {  if (head != nullptr) {  Node\* temp = this->head;  head = head->next;  delete temp;  size--;  }  else throw "Error! Linked list is Empty.";  }  // insert into an arbitrary place in the list by index  void List::insert(size\_t index, int data)  {  if (!isEmpty())  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  }  else  {  if (index != size)  throw "Error! Incorrect input.";  }  if (index == 0)  {  push\_front(data);  }  else  {  if (index == size)  push\_back(data);  else {  if (index < (size - 1) / 2)  {  Node\* previous = this->head;  for (size\_t i = 0; i < index - 1; i++)  {  previous = previous->next;  }  previous->next = new Node(data, previous->next, previous);  }  else  {  Node\* previous = this->tail;  for (size\_t i = size - 1; i > index - 1; i--)  {  previous = previous->prev;  }  previous->next = new Node(data, previous->next, previous);  }  size++;  }  }  }  // deleting an element by index  void List::remove(size\_t index)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  if (index == 0)  pop\_front();  else  {  if (index < (size - 1) / 2)  {  Node\* previous = this->head;  for (size\_t i = 0; i < index - 1; i++)  {  previous = previous->next;  }  Node\* todelete = previous->next;  previous->next = todelete->next;  delete todelete;  }  else  {  Node\* previous = this->tail;  for (size\_t i = size - 1; i >= index; i--)  {  previous = previous->prev;  }  Node\* todelete = previous->next;  previous->next = todelete->next;  delete todelete;  }  size--;  }  }  // clear list  void List::clear()  {  while (size) {  pop\_front();  }  }  // output the list to the console  void List::print\_to\_console()  {  if (isEmpty())  throw "Linked List is empty.\n";  else  {  Node\* current = head;  for (int i = 0; i < GetSize()-1; i++)  {  std::cout << current->data << " - ";  current = current->next;  }  std::cout << current->data;  }  }  // getting list size  size\_t List::GetSize()  {  return size;  }  // getting an element by index  int List::at(const int index)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  int counter = 0;  Node\* current = head;  while (current != nullptr)  {  if (counter == index)  return current->data;  current = current->next;  counter++;  }  }  // replacing the element by index with the passed element  void List::set(size\_t index, int data)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  if (index == 0)  {  pop\_front();  push\_front(data);  }  else  {  if (index < (size - 1) / 2)  {  Node\* previous = this->head;  for (size\_t i = 0; i < index - 1; i++)  {  previous = previous->next;  }  Node\* todelete = previous->next;  previous->next = todelete->next;  delete todelete;  previous->next = new Node(data, previous->next, previous);  }  else  {  Node\* previous = this->tail;  for (size\_t i = size - 1; i >= index; i--)  {  previous = previous->prev;  }  Node\* todelete = previous->next;  previous->next = todelete->next;  delete todelete;  previous->next = new Node(data, previous->next, previous);  }    }  }  // checking the list for emptiness  bool List::isEmpty()  {  if (head == nullptr)  return true;  else return false;  }  // insert at the beginning of the list of another list  void List::push\_front(List lst)  {  if (!isEmpty() && lst.isEmpty())  throw "Error! Inserting an empty list at the beginning is not possible.";  Node\* temp = lst.tail;  for (size\_t i = 0; i < lst.GetSize(); i++) {  this->push\_front(temp->data);  temp = temp->prev;  }  } |
| Iterator.h |
| #pragma once  class Iterator {  public:  virtual int next() = 0;  virtual bool has\_next() = 0;  }; |
| BinaryHeap.h |
| #pragma once  #include "Iterator.h"  #include "List.h"  class BinaryHeap  {  private:  int\* heap;  size\_t size;  size\_t capacity;  public:  BinaryHeap();  ~BinaryHeap();  bool contains(int);  void insert(int);  void remove(int);  void heapify(size\_t);  void input\_heap(int\*, size\_t);  void print\_heap();  Iterator\* create\_dft\_iterator();  Iterator\* create\_bft\_iterator();  class bft\_iterator : public Iterator  {  public:  bft\_iterator(int\*, size\_t, size\_t);  ~bft\_iterator();  int next() override;  bool has\_next() override;  private:  int\* current;  size\_t index;  size\_t size;  };  class dft\_iterator : public Iterator  {  public:  dft\_iterator(int\*, size\_t, size\_t);  ~dft\_iterator();  int next() override;  bool has\_next() override;  private:  int\* current;  size\_t index;  size\_t size;  List stack\_right;  };  }; |
| BinaryHeap.cpp |
| bool BinaryHeap::contains(int key)  {  if (this->size == 0)  return false;  for (size\_t i = 0; i < this->size; i++)  if (heap[i] == key)  return true;  return false;  }  /\* adding an element to the tree by key \*/  void BinaryHeap::insert(int data)  {  if (size == capacity) {  capacity \*= 100;  int\* temp = new int[capacity];  for (int i = 0; i < size; i++)  temp[i] = heap[i];  delete[]heap;  heap = temp;  }  heap[size] = data;  sift\_up(this->size);  this->size++;  }  /\* the removal of the tree element by key \*/  void BinaryHeap::remove(int key)  {  if (this->size==0)  throw "Error! Heap is empty.";  int\* new\_heap = new int[this->size];  int count = 0, new\_heap\_size = 0;  for (size\_t i = 0; i < size; i++)  {  if (heap[i] != key)  {  new\_heap[new\_heap\_size] = heap[i];  new\_heap\_size++;  }  else  {  count++;  }  }  if (count != 0)  {  delete[] heap;  heap = new\_heap;  this->size -= count;  }  else delete[] new\_heap;  for (size\_t i = 0; i < this->size; i = 2 \* i + 1)  heapify();  }  void BinaryHeap::input\_heap(int\* heap, size\_t size)  {  for (size\_t i = 0; i < size; i++)  this->heap[i] = heap[i];  this->size = size;  for (size\_t i = 0; i < size; i = 2 \* i + 1)  heapify();  }  void BinaryHeap::print\_heap()  {  int i = 0;  int k = 1;  while (i < this->size) {  while ((i < k) && (i < this->size)) {  std::cout << heap[i] << " ";  i++;  }  std::cout << std::endl;  k = k \* 2 + 1;  }  }  void BinaryHeap::sift\_up(int index)  {  while (heap[(index - 1) / 2] < heap[index])  {  std::swap(heap[index], heap[(index - 1) / 2]);  index = (index - 1) / 2;  }  }  Iterator\* BinaryHeap::create\_dft\_iterator()  {  return new dft\_iterator(heap, size, 0);  }  Iterator\* BinaryHeap::create\_bft\_iterator()  {  return new bft\_iterator(heap, size, 0);  }  BinaryHeap::bft\_iterator::bft\_iterator(int\* cur,size\_t size, size\_t index = 0)  {  current = cur;  this->size = size;  this->index = index;  }  BinaryHeap::bft\_iterator::~bft\_iterator()  {  delete current;  }  int BinaryHeap::bft\_iterator::next()  {  if (!has\_next())  throw "No more elements in heap";  this->index++;  return current[index - 1];  }  bool BinaryHeap::bft\_iterator::has\_next()  {  return index != this->size;  }  BinaryHeap::dft\_iterator::dft\_iterator(int\* cur, size\_t size, size\_t index=0)  {  current = cur;  this->size = size;  this->index = index;  }  BinaryHeap::dft\_iterator::~dft\_iterator()  {  delete current;  }  int BinaryHeap::dft\_iterator::next()  {  if (!has\_next()) {  throw "No more elements in heap";  }  int temp = current[index];  int child\_right = index \* 2 + 2, child\_left = index \* 2 + 1;  if (index == 0) stack\_right.push\_back(0);  if (child\_right < size) {  stack\_right.push\_back(child\_right);  index = child\_left;  }  else if (child\_left < size) index = child\_left;  else {  if (stack\_right.at(stack\_right.GetSize() - 1) == 0) stack\_right.pop\_back();  else {  index = stack\_right.at(stack\_right.GetSize() - 1);  stack\_right.pop\_back();  }  if (size == 1) index = -1;  }  return temp;  }  bool BinaryHeap::dft\_iterator::has\_next()  {  if (size == 0) return false;  if (index == 0) return true;  if (stack\_right.isEmpty() && index != 0) return false;  else return true;  } |
| UnitTest\_lab3.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "..\Program3\List.cpp"  #include "..\Program3\BinaryHeap.cpp"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTestlab3  {  TEST\_CLASS(UnitTestlab3)  {  public:  TEST\_METHOD(Input\_heapTest)  {  BinaryHeap heap;  int arr[5] = { 1,2,3,4,5 };  heap.input\_heap(arr, 5);  for (size\_t i = 0; i < 5; i++)  Assert::AreEqual(heap.contains(arr[i]), true);  }  TEST\_METHOD(ContainsTest1)  {  int arr[5] = { 1,2,3,4,5 };  BinaryHeap heap;  heap.input\_heap(arr, 5);  Assert::IsTrue(heap.contains(3));  }  TEST\_METHOD(ContainsTest2)  {  int arr[5] = { 1,2,3,4,5 };  BinaryHeap heap;  heap.input\_heap(arr, 5);  Assert::IsTrue(heap.contains(21) == false);  }  TEST\_METHOD(ContainsTest3)  {  BinaryHeap heap;  Assert::IsTrue(heap.contains(5) == false);  }  TEST\_METHOD(InsertTest1)  {  int arr[5] = { 1,2,3,5,6 };  BinaryHeap heap;  heap.input\_heap(arr, 5);  heap.insert(4);  Assert::IsTrue(heap.contains(4));  }  TEST\_METHOD(InsertTest2)  {  BinaryHeap heap;  heap.insert(1);  Assert::IsTrue(heap.contains(1));  }  TEST\_METHOD(RemoveTest1)  {  int arr[5] = { 1,2,3,4,5 };  BinaryHeap heap;  heap.input\_heap(arr, 5);  heap.remove(4);  Assert::IsTrue(heap.contains(4)==false);  }  TEST\_METHOD(RemoveTest2)  {  BinaryHeap heap;  try  {  heap.remove(4);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Heap is empty.");  }  }  TEST\_METHOD(BFT\_Iterator\_Test1)  {  BinaryHeap heap;  int arr[7] = { 7,4,6,1,3,2,5 };  heap.input\_heap(arr, 7);  Iterator\* Iter = heap.create\_bft\_iterator();  for (size\_t i = 0; i < 7; i++)  Assert::IsTrue(Iter->next() == arr[i]);  }  TEST\_METHOD(BFT\_Iterator\_Test2)  {  BinaryHeap heap;  Iterator\* Iter = heap.create\_bft\_iterator();  try  {  Iter->next();  }  catch (const char\* error)  {  Assert::AreEqual(error, "No more elements in heap");  }  }  TEST\_METHOD(DFT\_Iterator\_Test1)  {  BinaryHeap heap;  int arr[7] = { 7,4,6,1,3,2,5 };  int control[7] = { 7,4,1,3,6,2,5 };  heap.input\_heap(arr, 7);  Iterator\* Iter = heap.create\_dft\_iterator();  for (size\_t i = 0; i < 7; i++)  Assert::IsTrue(Iter->next() == control[i]);  }  TEST\_METHOD(DFT\_Iterator\_Test2)  {  BinaryHeap heap;  Iterator\* Iter = heap.create\_dft\_iterator();  try  {  Iter->next();  }  catch (const char\* error)  {  Assert::AreEqual(error, "No more elements in heap");  }  }  };  } |
| main.cpp |
| #include "BinaryHeap.h"  #include <iostream>  using namespace std;  int main()  {  BinaryHeap HEAP;    int arr[7] = { 1,2,3,4,5,6,7 };  HEAP.input\_heap(arr, 7);  HEAP.print\_heap();  HEAP.insert(10);  HEAP.print\_heap();  return 0;  } |