**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 |  | Бабкин И.А. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

**Постановка задачи**

Необходимо реализовать двоичную кучу и ее методы. Также необходимо реализовать итераторы для обхода кучи в ширину и глубину.

**Описание реализуемого класса и методов**

class Heap

|  |  |  |
| --- | --- | --- |
| Название метода | Описание метода | Временная сложность |
| bool contains(int number); | Поиск элемента в куче по ключу | O(n) |
| void insert(int number); | Добавление элемента в кучу по ключу | O(log(n)) |
| void remove(int number); | Удаление элемента в куче по ключу | O(n) |
| void heapify(int index); | Восстановление свойств кучи | O(log(n)) |
| void siftDown(int index); | Просеивание вниз | O(log(n)) |
| void siftUp(int index); | Просеивание вверх | O(log(n)) |
| void out(); | Вывод кучи | O(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

|  |  |
| --- | --- |
| Название метода | Описание метода |
| virtual int next() | Получения значения текущего элемента и переход к следующему |
| virtual bool has\_next() | Проверка наличия следующего элемента |

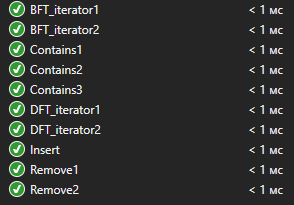
class bft\_iterator

|  |  |
| --- | --- |
| Поле | Описание |
| int\* current | Куча |
| int HeapSize | Размер кучи |
| int index | Индекс текущего элемента |

class dft\_iterator

|  |  |
| --- | --- |
| Поле | Описание |
| int\* current | Куча |
| int HeapSize | Размер кучи |
| int index | Индекс текущего элемента |
| LinkedList stack | Реализует стек |

**Описание реализованных unit-тестов**

Для проверки реализованных методов были написаны unit-тесты. В таблице ниже представлены их названия. Названия совпадают с названиями проверяемых методов.

|  |
| --- |
| Insert |
| Contains1, 2, 3 |
| Remove1, 2 |
| BFT\_iterator1, 2 |
| DFT\_iterator1, 2 |

**Листинг**

**LinkedList.h** и **LinkedList.cpp** Были взяты из лабораторной работы № 1

**Heap.h**

#pragma once

#include "Iterator.h"

#include "LinkedList.h"

class Heap

{

public:

const int size = 100;

int\* heap;

int HeapSize;

Heap();

~Heap();

bool contains(int number);

void insert(int number);

void remove(int number);

void heapify(int index);

void siftDown(int index);

void siftUp(int index);

void out();

Iterator\* create\_bft\_iterator();

Iterator\* create\_dft\_iterator();

class bft\_iterator : public Iterator

{

public:

bft\_iterator(int\* current, int HeapSize, int index);

~bft\_iterator();

int next() override;

bool has\_next() override;

private:

int\* current;

int HeapSize;

int index;

};

class dft\_iterator : public Iterator

{

public:

dft\_iterator(int\* current, int HeapSize, int index);

~dft\_iterator();

int next() override;

bool has\_next() override;

private:

int\* current;

int HeapSize;

int index;

LinkedList stack;

};

};

**Heap.cpp**

#include "Heap.h"

#include <iostream>

using namespace std;

Heap::Heap()

{

heap = new int[size];

HeapSize = 0;

}

Heap::~Heap()

{

delete heap;

HeapSize = 0;

}

void Heap::siftDown(int index) {

int left, right;

int j, temp;

while (2 \* index + 1 < HeapSize) {

left = 2 \* index + 1;

right = 2 \* index + 2;

j = left;

if (right < HeapSize && heap[right] < heap[left])

j = right;

if (heap[index] <= heap[j])

break;

temp = heap[index];

heap[index] = heap[j];

heap[j] = temp;

index = j;

}

}

void Heap::siftUp(int index) {

int temp;

while (heap[index] < heap[(index - 1) / 2]) {

temp = heap[index];

heap[index] = heap[(index - 1) / 2];

heap[(index - 1) / 2] = temp;

index = (index - 1) / 2;

}

}

void Heap::heapify(int index)

{

if (heap[index] > heap[2 \* index + 1] || heap[index] > heap[2 \* index + 2]) siftDown(index);

if (heap[index] < heap[(index - 1) / 2]) siftUp(index);

}

void Heap::insert(int number)

{

int index;

index = HeapSize;

heap[index] = number;

HeapSize++;

heapify(index);

}

bool Heap::contains(int number)

{

for (int i = 0; i < HeapSize; i++) {

if (heap[i] == number) return true;

}

return false;

}

void Heap::remove(int number)

{

int temp, index;

bool flag = false;

for (int i = 0; i < HeapSize; i++) {

if (heap[i] == number) {

flag = true;

index = i;

break;

}

}

if (flag == true) {

temp = heap[index];

heap[index] = heap[HeapSize - 1];

heap[HeapSize - 1] = temp;

HeapSize--;

heapify(index);

}

else throw("There is no such element");

}

void Heap::out()

{

int i = 0;

int k = 1;

while (i < HeapSize) {

while ((i < k) && (i < HeapSize)) {

cout << heap[i] << " ";

i++;

}

cout << endl;

k = k \* 2 + 1;

}

}

Iterator\* Heap::create\_bft\_iterator()

{

return new bft\_iterator(heap, size, 0);

}

Heap::bft\_iterator::bft\_iterator(int\* current, int HeapSize, int index = 0)

{

this->current = current;

this->HeapSize = HeapSize;

this->index = index;

}

Heap::bft\_iterator::~bft\_iterator()

{

delete current;

}

int Heap::bft\_iterator::next()

{

if (!has\_next())

throw ("No more elements");

index++;

return current[index - 1];

}

bool Heap::bft\_iterator::has\_next()

{

return index != this->HeapSize;

}

Iterator\* Heap::create\_dft\_iterator()

{

return new dft\_iterator(heap, HeapSize, 0);

}

Heap::dft\_iterator::dft\_iterator(int\* current, int HeapSize, int index = 0)

{

this->current = current;

this->HeapSize = HeapSize;

this->index = index;

}

Heap::dft\_iterator::~dft\_iterator()

{

delete current;

}

int Heap::dft\_iterator::next()

{

if (!has\_next())

throw ("No more elements");

int temp = current[index];

int left = index \* 2 + 1;

int right = index \* 2 + 2;

if (index == 0) stack.push\_back(0);

if (right < this->HeapSize) {

stack.push\_back(right);

index = left;

}

else if (left < this->HeapSize) index = left;

else {

if (stack.at(stack.get\_size() - 1) == 0)

stack.pop\_back();

else {

index = stack.at(stack.get\_size() - 1);

stack.pop\_back();

}

if (this->HeapSize == 1) index = -1;

}

return temp;

}

bool Heap::dft\_iterator::has\_next()

{

if (HeapSize == 0) return false;

if (index == 0) return true;

if (stack.isEmpty() && index != 0) return false;

else return true;

}

**Iterator.h**

class Iterator

{

public:

virtual int next() = 0;

virtual bool has\_next() = 0;

};