

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

ОТЧЕТ
по лабораторной работе №2
по дисциплине «Алгоритмы и структуры данных»
Вариант 2
Тема: Двоичные деревья

Студент гр. 9302

Точилин А.Е.

Преподаватель

Тутуева А.В.

Санкт-Петербург

2020

Постановка задачи

Реализовать двоичную кучу с методами: добавления элемента в кучу, удаление элемента из кучи, проверки наличия элемента в куче. Так же было реализовано два вида итераторов, первый обходит кучу в глубину, второй обходит в ширину. Так же для реализации поиска были написаны собственная очередь и стек.

Описание реализуемых методов.

Оценка временной сложности методов.

Для реализации был набор функций и классов. Список функций каждого класса представлен в табл. 1.

Таблица 1 – Описание и оценка сложности функций поиска и сортировки.

Название метода	Описание	Оценка временной сложности
Class Heap		
void insert (int key);	Функция вставки элемента в кучу	$O(\log(n))$
bool contains (int key);	Функция проверки наличия элемента в куче	$O(n)$
std::string getHeapString ();	Возвращает массив, хранящий кучу в виде строки	$O(n)$
void remove(int key);	Функция сортировки пузырьком	$O(n)$
void heapify(int index);	Просеивание кучи	$O(\log(n))$
Iterator* create_dft_iterator();	Создает итератор поиска в глубину	$O(1)$
Iterator* create_bft_iterator();	Создает итератор поиска в ширину	$O(1)$
Class Iterator		
void next();	Переход к следующему элементу	$O(1)$
bool hasNext();	Проверка достижения последнего элемента	$O(1)$
int getCur();	Возвращает текущий элемент	$O(1)$
Class Stack/Class Queue		
void push(int elem);	Добавление элемента	$O(1)$
int pop();	Снятие элемента	$O(1)$
bool isEmpty();	Проверка на пустоту	$O(1)$

Код всех классов функций представлен в приложении А.

Описание реализованных unit-тестов

Для проверки работоспособности программы были реализованы unit-тесты. Для этого был создан новый проект в Visual Studio, в котором написан класс LR2tests, методы класса и их описание представлены в табл. 2, код в приложении В.

Таблица 2 – Описание методов класса LR2tests

Метод	Описание метода
TestInsert	Проверяется правильный порядок элементов в массиве куче после вставки
TestRemove	Проверяется правильный порядок элементов кучи после удаления и просеивания
TestContains	Проверяется функция проверки наличия элемента в куче
TestDFTIterator	Проверяется работоспособность итератора обхода в глубину
TestBFTIterator	Проверяется работоспособность итератора обхода в ширину

Пример работы программы представлен на рис 2., код функции main на рис. 1.

```

int main()
{
    Heap heap = Heap();
    heap.insert(5);
    heap.insert(15);
    heap.insert(34);
    heap.insert(77);
    heap.insert(100);
    heap.insert(50);
    heap.insert(1);
    heap.insert(75);
    heap.insert(350);

    std::cout << heap.getHeapString() << std::endl;

    heap.remove(350);
    heap.remove(50);

    std::cout << std::endl << heap.getHeapString() << std::endl;

    std::cout << std::endl;

    std::cout << heap.contains(100) << std::endl;
    std::cout << heap.contains(350) << std::endl;

    Iterator* dfs_iterator = heap.create_dft_iterator();
    Iterator* bfs_iterator = heap.create_bft_iterator();

    while (dfs_iterator->hasNext()) {
        dfs_iterator->next();
        std::cout << dfs_iterator->getCur() << " ";
    }
    std::cout << std::endl;
    std::cout << std::endl;
    std::cout << std::endl;
    std::cout << std::endl;

    while (bfs_iterator->hasNext()) {
        bfs_iterator->next();
        std::cout << bfs_iterator->getCur() << " ";
    }
    std::cout << std::endl;
}

```

Рисунок 1 – код функции main

```

350 100 50 77 34 15 1 5 75

100 77 15 75 34 5 1

1
0
100 77 75 34 15 5 1

100 77 15 75 34 5 1

```

Рисунок 2 – результат запуска программы

Приложение А

Листинг файла Heap.h

```
#pragma once
#include <algorithm>
#include <iostream>
#include <string>
#include "Iterator.h"
#include "Stack.h"
#include "Queue.h"

class Heap
{
private:
    void heapify(int index);
    int getIndex(int key);
    static const int MAX_SIZE = 500;
    int* h;
    int heapSize;
    //dft iterator extends iterator
    class DFT_Iterator : public Iterator {
    private:
        int curIndex;
        Heap* heap;
        Stack* stack;
    public:
        DFT_Iterator(Heap* heap);
        void next() override;
        bool hasNext() override;
        int getCur() override;
    };
    //bft iterator extends iterator
    class BFT_Iterator : public Iterator {
    private:
        int curIndex;
        Heap* heap;
        Queue* queue;
    public:
        BFT_Iterator(Heap* heap);
        void next() override;
        bool hasNext() override;
        int getCur() override;
    };
    //make iterators friend class to iterate through heap
    friend DFT_Iterator;
    friend BFT_Iterator;

public:
    Heap();
    void insert(int key);
    bool contains(int key);
    std::string getHeapString();
    void remove(int key);
    Iterator* create_dft_iterator();
    Iterator* create_bft_iterator();
};
```

Листинг файла Heap.cpp

```
#include "Heap.h"

Heap::Heap() //at create heap has 0 size
{
    h = new int[MAX_SIZE];
    heapSize = 0;
}

void Heap::insert(int key)
{
    int i, parent;
    i = this->heapSize;
    h[i] = key; //insert new element to the end of heap array
    parent = (i - 1) / 2; //start parent of new element
    while (parent >= 0 && i > 0) { //raise from bottom of heap to the top
        if (h[i] > h[parent]) { //raise element if parent lower
            std::swap(h[parent], h[i]);
        }
        i = parent;
        parent = (i - 1) / 2;
    }
    (this -> heapSize)++;
}

bool Heap::contains(int key)
{
    if (key > h[0]) { //if key higher than head no reason to search
        return false;
    }
    Iterator* dftIterator = create_dft_iterator(); //use dft iterator for searching
key
    while (dftIterator->hasNext()) {
        dftIterator->next();
        if (dftIterator->getCur() == key) {
            return true;
        }
    }
    return false;
}

std::string Heap::getHeapString() //function used for getting head as a string
{
    std::string result = "";
    for (int i = 0; i < this -> heapSize; i++) {
        result += std::to_string(h[i]);
        result += " ";
    }
    result.erase(result.length() - 1, 1);
    return result;
}

void Heap::remove(int key)
{
    int index = getIndex(key); //get index for removing element
    if (index == -1) {
        throw std::runtime_error("No such element in the heap");
    }
    h[index] = h[(this->heapSize)- 1]; //change deleting element to the last o heap
array
    (this->heapSize) --;
    heapify(index); //after deleting should make tree heap again
}
```

```

Iterator* Heap::create_dft_iterator()
{
    return new DFT_Iterator(this);
}

Iterator* Heap::create_bft_iterator()
{
    return new BFT_Iterator(this);
}

void Heap::heapify(int index) // function make tree heap again
{
    //start checking heap array from index send to function
    int indexLeft, indexRight;
    int temp;
    indexLeft = 2 * index + 1; //indexes of left and right roots of the current
    indexRight = 2 * index + 2;
    if (indexLeft < (this->heapSize)) { //if current elem has left root
        if (h[index] < h[indexLeft]) {
            std::swap(h[index], h[indexLeft]);
            heapify(indexLeft); //swap elems and check heap for left root
        }
    }
    if (indexRight < (this->heapSize)) { //if current elem has right root
        if (h[index] < h[indexRight]) {
            std::swap(h[index], h[indexRight]);
            heapify(indexRight);
        }
    }
}

int Heap::getIndex(int key) //check is elem in array and return int index in heap array
{
    for (int i = 0; i < this->heapSize; i++) {
        if (h[i] == key) {
            return i;
        }
    }
    return -1;
}

Heap::DFT_Iterator::DFT_Iterator(Heap* heap)
{
    this->curIndex = -1;
    this->heap = heap;
    this->stack = new Stack();
}

void Heap::DFT_Iterator::next() //move Iterator to the next elem
{
    if (curIndex == -1) {
        curIndex = 0;
        return;
    }
    int indexLeft = curIndex * 2 + 1;
    int indexRight = curIndex * 2 + 2;
    if (indexRight <= (this->heap->heapSize - 1)) { //if cur elem has left root push
to stack
        stack->push(indexRight);
        curIndex = indexLeft;
    }
    else if (indexLeft <= (this->heap->heapSize - 1)) { //if cur elem has only left
elem
        curIndex = indexLeft;
    }
}

```

```

        else if (stack -> isEmpty()) { //if stack is empty and no children
            std::cout << "No next" << std::endl;
        }
        else { //if no children pop root from stack and move to it
            curIndex = stack -> pop();
        }
    }

bool Heap::DFT_Iterator::hasNext()
{
    int indexLeft = curIndex * 2 + 1;
    int indexRight = curIndex * 2 + 2; //if root has childrens or stack isn't empty
    can move iterator
    if ((indexLeft <= (this->heap->heapSize - 1)) || (indexRight <= (this->heap->heapSize - 1)) || !stack->isEmpty()) {
        return true;
    }
    else {
        return false;
    }
}

int Heap::DFT_Iterator::getCur()
{
    if (curIndex < 0) {
        throw std::runtime_error("No such index in heap");
    }
    return this->heap->h[curIndex];
}

Heap::BFT_Iterator::BFT_Iterator(Heap* heap)
{
    curIndex = 0;
    this->heap = heap;
    this->queue = new Queue();
    queue -> push(curIndex);
}

void Heap::BFT_Iterator::next() //get current element from queue and push it's childrens
to queue
{
    curIndex = queue -> pop();
    int indexLeft = curIndex * 2 + 1;
    int indexRight = curIndex * 2 + 2;
    if (indexLeft <= (this->heap->heapSize - 1)) {
        queue -> push(indexLeft);
    }
    if (indexRight <= (this->heap->heapSize - 1)) {
        queue -> push(indexRight);
    }
}

bool Heap::BFT_Iterator::hasNext() //if wueue is't empty can move to the next
{
    if (queue -> isEmpty()) {
        return false;
    }
    else {
        return true;
    }
}

int Heap::BFT_Iterator::getCur()
{
    if (curIndex < 0) {

```



```

        throw std::runtime_error("No such index in heap");
    }
    return this->heap->h[curIndex];
}

```

Листинг файла Iterator.h

```

#pragma once

class Iterator {
public:
    virtual void next() = 0;

    virtual bool hasNext() = 0;

    virtual int getCur() = 0;
};

```

Листинг файла Stack.h

```

#pragma once
#include <stdexcept>

class Stack
{
private:
    static const int MAX_SIZE = 500;
    int* stack;
    int indexHead;
public:
    Stack() {
        this->indexHead = -1;
        this->stack = new int[MAX_SIZE];
    };

    void push(int elem) { //add elem to array and change index of head
        indexHead++;
        stack[indexHead] = elem;
    };

    int pop() { //if stack is't empty
        if (!isEmpty()) {
            indexHead--; //reduce index of head
            return stack[indexHead + 1]; //and return popped element
        }
        else {
            throw std::runtime_error("Stack is empty");
        }
    }

    bool isEmpty() {
        return indexHead == -1;
    }

    void printStack() {
        if (indexHead == -1) {
            std::cout << "empty" << std::endl;
        }
        else {
            for (int i = indexHead; i >= 0; i--)
            {
                std::cout << stack[i] << " ";
            }
        }
    }
};

```

```

    }
}
};

```

Листинг файла Queue.h

```

#pragma once
#pragma once
#include <stdexcept>
#include <iostream>

class Queue
{
private:
    static const int MAX_SIZE = 500;
    int* queue;
    int indexHead;
    int indexTail;

public:
    Queue() {
        this->indexHead = 0;
        this->indexTail = 0;
        this->queue = new int[MAX_SIZE];
    };

    void push(int elem) { //raise index of tail and push new elem to the end of queue
        queue[indexTail] = elem;
        indexTail++;
    };

    int pop() { //raise head and return head elem
        if (!isEmpty()) {
            indexHead++;
            return queue[indexHead - 1];
        }
        else {
            throw std::runtime_error("Stack is empty");
        }
    }

    bool isEmpty() { //tail and head equal
        return indexHead == indexTail;
    }

    void printQueue() {
        if (isEmpty()) {
            std::cout << "empty" << std::endl;
        }
        else {
            for (int i = indexTail - 1; i >= indexHead; i--)
            {
                std::cout << queue[i] << " ";
            }
        }
    }
};

```

Листинг файла testLR3.cpp

```
#include <string>
#include "pch.h"
#include "CppUnitTest.h"
#include "../LR3/Heap.h"
#include "../LR3/Stack.h"
#include "../LR3/Iterator.h"
#include "../LR3/Heap.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace testLR3
{
    TEST_CLASS(testLR3)
    {
    private:
        Heap heap = Heap();
    public:
        TEST_METHOD_INITIALIZE(init)
        {
            heap.insert(5);
            heap.insert(15);
            heap.insert(34);
            heap.insert(77);
            heap.insert(100);
            heap.insert(50);
            heap.insert(1);
            heap.insert(75);
            heap.insert(350);
        }
        TEST_METHOD(TestInsert)
        {
            std::string temp = heap.getHeapString();
            char tab2[1024];
            strcpy_s(tab2, temp.c_str());
            Assert::AreEqual("350 100 50 77 34 15 1 5 75", tab2);
        }
        TEST_METHOD(TestRemove)
        {
            heap.remove(350);
            heap.remove(1);
            std::string temp = heap.getHeapString();
            char tab2[1024];
            strcpy_s(tab2, temp.c_str());
            Assert::AreEqual("100 77 50 75 34 15 5", tab2);
        }
        TEST_METHOD(TestContains)
        {
            Assert::AreEqual(true, heap.contains(350));
            Assert::AreEqual(true, heap.contains(77));
            Assert::AreEqual(true, heap.contains(50));
            Assert::AreEqual(false, heap.contains(-10));
        }
        TEST_METHOD(TestDFTIterator)
        {
            Iterator* dft_iterator = heap.create_dft_iterator();
            dft_iterator->next();
            Assert::AreEqual(350, dft_iterator->getCur());
            dft_iterator->next();
            dft_iterator->next();
            dft_iterator->next();
            dft_iterator->next();
            dft_iterator->next();
        }
    }
}
```

```

        Assert::AreEqual(34, dft_iterator->getCur());
        dft_iterator->next();
        dft_iterator->next();
        dft_iterator->next();
        Assert::AreEqual(false, dft_iterator->hasNext());
    }
    TEST_METHOD(TestBFTIterator)
    {
        Iterator* bft_iterator = heap.create_bft_iterator();
        bft_iterator->next();
        Assert::AreEqual(350, bft_iterator->getCur());
        bft_iterator->next();
        bft_iterator->next();
        bft_iterator->next();
        bft_iterator->next();
        bft_iterator->next();
        Assert::AreEqual(15, bft_iterator->getCur());
        bft_iterator->next();
        bft_iterator->next();
        bft_iterator->next();
        Assert::AreEqual(false, bft_iterator->hasNext());
    }
};
}

```