Правительство Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего образования «Национальный исследовательский университет
«Высшая школа экономики»

Кафедра «Компьютерная безопасность»

ОТЧЕТ
К ЛАБОРАТОРНОЙ РАБОТЕ №2

по дисциплине

«Языки программирования»

Работу выполнил
студент группы СКБ-203 _____ К.А. Павкин
подпись, дата

Работу проверил _____ С.А. Булгаков
подпись, дата

Москва 2021

Содержание

«Контейнер»

Разработать программу на языке Си++ (ISO/IEC 14882:2014), демонстрирующую решение поставленной задачи.

Постановка задачи

Разработать класс ADT и унаследовать от него классы, разработанные в рамках лабораторной работы 1.

Разработать набор классов, объекты которых реализуют типы данных, указанные ниже. Для этих классов разработать необходимые конструкторы, деструктор, конструктор копирования. Разработать операции: добавления/удаления элемента (уточнено в задаче); получения количества элементов; доступа к элементу (перегрузить оператор []). При ошибках запускать исключение.

В главной функции разместить тесты, разработанные с использованием библиотеки GoogleTest.

Задачи

1. Динамический массив указателей на объекты ADT. Размерность массива указателей увеличивается в момент его переполнения. Начальная размерность задается как параметр конструктора, значение по умолчанию 0. Добавление/удаление элемента в произвольное место.

2. Стек, представленный динамическим массивом указателей на хранимые объекты ADT. Размерность стека увеличивается в момент его переполнения. Начальная размерность задается как параметр конструктора, значение по умолчанию 0. Добавление/удаление элемента в начало и в конец.

3. Односвязный список, содержащий указатели на объекты ADT. Добавление/удаление элемента в произвольное место.

4. Циклическая очередь, представленная динамическим массивом указателей на хранимые объекты ADT. Добавление/удаление элемента в произвольное место.

5. Двоичное дерево, содержащее указатели на объекты ADT. Добавление/удаление элемента в произвольное место.

1 Алгоритм решения задачи

В процессе анализа предметной области были выявлены основные алгоритмы, необходимые для реализации всеобъемлющей, асимптотически эффективной и отлаженной программы для решения поставленной задачи.

На рисунке 1 представлена диаграмма классов, демонстрирующая формат хранения данных и зависимости между классами. В целях сужения диаграммы без ущерба логике и смыслу были опущены закрытые служебные методы.

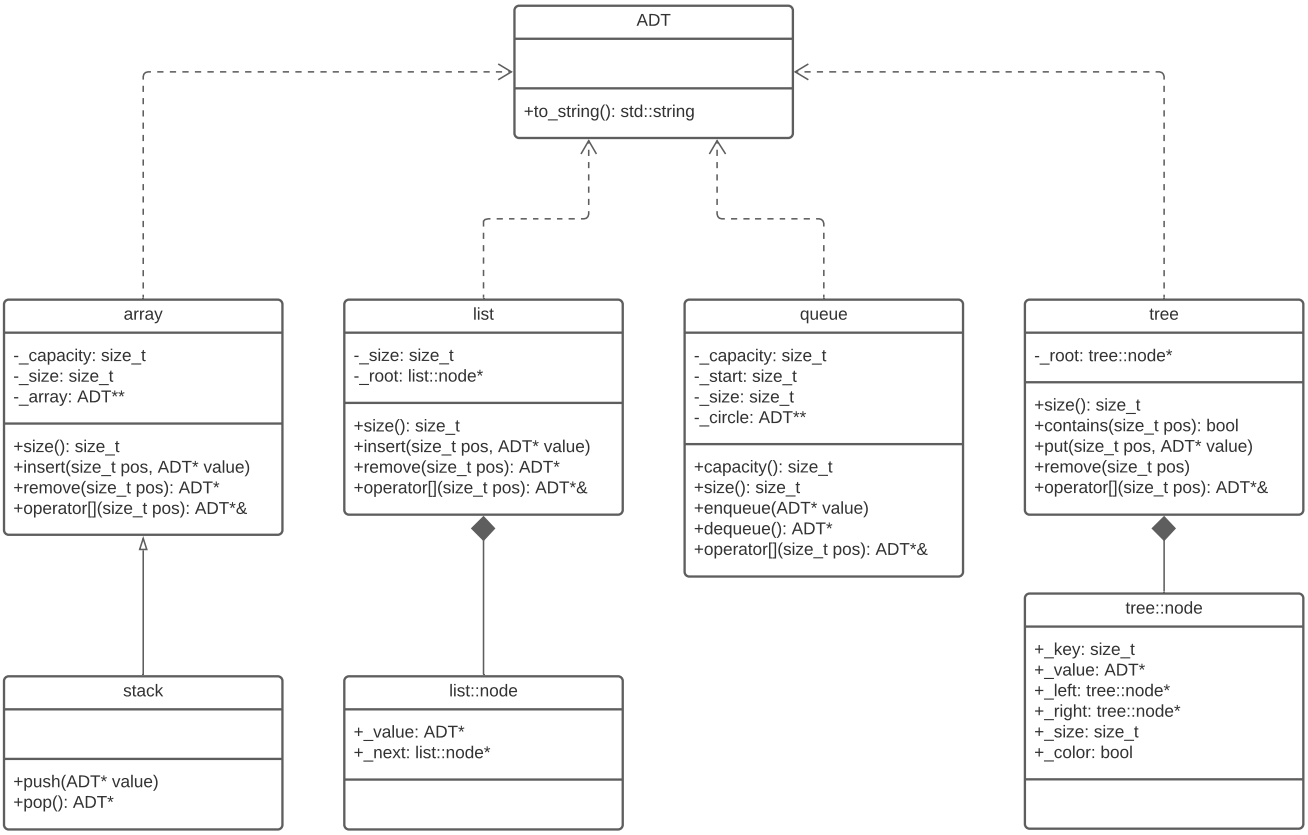

Рис. 1: Диаграмма классов

1.1 Динамический массив

В каноническом смысле, динамическим массивом называется контейнер, обладающий свойством расширяться и сжиматься в зависимости от количества заполненных элементов в нем. При этом размерность (*capacity*) — фактически занимаемая массивом память — не превышает $2 * size$, где $size$ — количество непустых элементов в контейнере. Такой подход позволяет добиться приемлимых затрат памяти при работе с неизвестным количеством входных данных. Основные методы для работы с динамическим массивом и их асимптотическая сложность приведены в таблице 1.

Таблица 1: Методы класса array

Название	Описание	Сложность
`insert`	Вставка элемента в произвольное место	$O(size)$
`remove`	Удаление элемента из произвольного места	$O(size)$
`operator[]`	Полулчение произвольного элемента	$O(1)$

Здесь время вставки занимает в худшем случае $size$ операций, т. е. пропорционально количеству непустых элементов. Однако, для случая вставки элемента в конец массива асимптотика будет несколько иная — $\Omega(1)$. Так происходит потому, что операция вставки, при достижении конца контейнера, сопровождается выполнением закрытой процедуры `resize` (листинг 1), которая увеличивает его размерность вдвое.

Листинг 1: Процедура `resize`

```
void array::resize(size_t capacity)
{
        ADT** array = new ADT*[capacity];

        std::copy(_array, _array + std::min(_capacity, capacity), array);

        delete[] _array;

        _array = new ADT*[capacity];

        std::copy(array, array + capacity, _array);

        delete[] array;

        _capacity = capacity;
}
```

Аналогичным образом, при удалении элемента может вызываться процедура `resize` в целях сужения массива, только уже втрое. Это позволяет исключить ситуации с пороговым значением, при котором попеременные вставка и удаление элементов будут давать наихудшую производительность.
Полный код заголовочных файлов и файлов реализации динамического массива приведены в приложении А.

1.1.1 Стек

Стек, будучи наследуюемым от динамического массива, имеет ту же асимптотическую сложность и те же тонкости работы с ним. Главное различие заключается в том, что операции вставки/удаления элементов стека ограничены лишь концом (по условию задачи и началом) контейнера. Отрывок из файла реализации стека приведен в листинге 2.

Листинг 2: Файл реализации стека

```
stack::stack(size_t capacity)
: array(capacity)
{
}

size_t stack::size() const
{
        return array::size();
}

void stack::push(ADT* value)
{
        array::insert(size(), value);
}

ADT* stack::pop()
{
        return array::remove(size() - 1);
}

ADT*& stack::operator[](size_t pos)
{
        return array::operator[](pos);
}
```

Полный код заголовочных файлов и файлов реализации стека приведены в приложении Б.

1.2 Односвязный список

В отличие от динамического массива, односвязный список занимает константное время для операций вставки/удаления элемента из начала или конца контейнера (в зависимости от реализации) при любых условиях. Это становится возможным благодаря устройству односвязного списка, при котором среди закрытых переменных хранится «корень» — первый или последний элемент контейнера, в котором, помимо значения, находится также ссылка на следующий (предыдущий) элемент. В листинге 3 показан отрывок заголовочного файла для описания односвязного списка.

Листинг 3: Отрывок заголовочного файла для описания односвязного списка

```
class list {
        struct node {
                ADT* _value;
                node* _next;

                node(ADT*, node* = nullptr);
        };

        size_t _size;
        node* _root;

        // ...
};
```

Вставка/удаление элемента из односвязного списка происходит несколько быстрее, нежели аналогичное действие в динамическом массиве, поскольку отсутствуют издержки для заполнения и копирования контейнера. Однако, асимптотическая сложность остается прежней — $O(size)$. При больших объемах данных односвязный список также выигрывает у динамического массива, поскольку первый занимает место, пропорциональное количеству элементов в нем.
Основным недостатком данной реализации является время, необходимое для доступа к элементу контейнера по индексу, ведь для этого необходимо пройти все узлы вплоть до получаемого элемента.
Полный код заголовочных файлов и файлов реализации односвязного списка приведены в приложении В.

1.3 Циклическая очередь

Будучи контейнером с константной размерностью, циклическая очередь не обеспечивает динамического расширения или сужения. Её основное преимущество заключается в том, что все операции, по определению свойственные этому типу данных, занимают константное время.
Для обеспечения подобной функциональности достаточно поддерживать обычный массив, а также индексы начала и конца отрезка хранящихся в нем элементов. В листинге 4 показан отрывок заголовочного файла для описания циклической очереди.

Листинг 4: Отрывок заголовочного файла для описания циклической очереди

```
class queue {
        size_t _capacity, _start, _size;
        ADT** _circle;

        // ...
};
```

Можно заметить, что в представленной реализации вместо индекса, служащего обозначением конца очереди, используется переменная *size*. Такой подход позволяет делать внутренние проверки без лишних вычислений. Для получения конечной позиции заполненного отрезка достаточно взять остаток от деления суммы стартовой позиции *start* и количества элементов контейнера *size*.
Полный код заголовочных файлов и файлов реализации односвязного списка приведены в приложении Г.

1.4 Бинарное дерево

Наиболее подходящей с точки зрения надежности и эффективности реализацией бинарного дерева является так называемое красно-черное дерево. Такое дерево идеально сбалансировано, и имеет асимтотическую сложность, не превышающую $O(\log_2 size)$ для всех необходимых операций (см. таблицу 2).

Таблица 2: Методы класса tree

Название	Описание	Сложность
`put`	Вставка элемента в произвольное место	$O(\log_2 size)$
`remove`	Удаление элемента из произвольного места	$O(\log_2 size)$
`operator[]`	Полулчение произвольного элемента	$O(\log_2 size)$

В отличие от классической реализации двоичного дерева поиска, красно-черное дерево может иметь помеченные (красные) ребра, которые говорят о том, что два узла, соединенные подобным ребром, можно рассматривать как один. Это необходимо для того, чтобы размещать элементы не только слева и справа, как элементы с меньшими и большими ключами соответственно, но и между родственными. Данное отличие как раз таки и позволяет достичь максимальной асимптотической эффективности для операции удаления элемента из произвольного места. Представленный вариант бинарного дерева повсеместно используется для таких структур данных как «множество» или «словарь».
В качестве хранимого значения выступает лишь одна переменная *root*, являющаяся экземпляром вложенного класса `node`. Этот класс содержит в себе: ключ, однозначно идентифицирующий положение элемента в дереве, значение, ссылки на левый и правый потомки, размер поддерева и цвет ребра, ведущего к родительскому узлу. В листинге 5 показан отрывок заголовочного файла для описания бинарного дерева.

Листинг 5: Отрывок заголовочного файла для описания бинарного дерева

```
class tree {
        struct node {
                K _key;
                ADT* _value;
                node* _left;
                node* _right;
                size_t _size;
                bool _color;

                node(K, ADT*, size_t, bool);
        };

        node* _root;

        // ...
};
```

Так, при вставке/удалении элемента может потребоваться смещение некоторых узлов дерева (балансировка) ради обеспечения эффективности последующих операций. Это обеспечивается, главным образом, благодаря функциям смещения поддерева влево или вправо (см. листинг 6).

Листинг 6: Функция смещения поддерева влево

```
template <typename K>
typename tree<K>::node* tree<K>::rotate_left(node* n)
{
        node* temp = n->_right;
        n->_right = temp->_left;
        temp->_left = n;
        temp->_color = temp->_left->_color;
        temp->_left->_color = RED;
        temp->_size = n->_size;
        n->_size = size(n->_left) + size(n->_right) + 1;

        return temp;
}
```

Полный код заголовочных файлов и файлов реализации бинарного дерева приведены в приложении Д.

2 Получение исполняемых модулей

Заголовочные файлы и их реализации были размещены в различных директориях. Для сборки проекта в каждую директорию были помещены файлы CMakeLists.txt, и, таким образом, каждая директория представляет из себя отдельную библиотеку, которая в последствии собирается в корневом каталоге. Файл сборки CMakeLists.txt из корневого каталога показан в листинге 7.

Листинг 7: Файл сборки CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)

project(laboratory-work-02-kapavkin)

set(CMAKE_CXX_STANDARD 14)

include(FetchContent)
FetchContent_Declare(
googletest
URL https://github.com/google/googletest/archive/609281088cfefc76f9d0
ce82e1ff6c30cc3591e5.zip
)

set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)

set(MAIN main.cpp)

enable_testing()

add_executable(main ${MAIN})

add_subdirectory(laboratory-work-01-kapavkin)
add_subdirectory(array)
add_subdirectory(list)
add_subdirectory(queue)
add_subdirectory(tree)

target_link_libraries(main
        laboratory-work-01-kapavkin
        array
        list
        queue
        tree
        gtest_main
)

include(GoogleTest)
gtest_discover_tests(main)
```

3 Тестирование

Файл `main.cpp` содержит юнит-тесты, целью которых является сравнение ожидаемых от функций значений с фактическими, и выдача оповещений при их несовпадении.

3.1 Прецеденты класса `array`

Тестовые случаи для проверки корректности работы методов класса `array`.

1. `constructor` — проверяет корректность конструкторов/деструкторов.
2. `insert` — проверяет корректность вставки элемента в произвольное место.
3. `remove` — проверяет корректность удаления элемента из произвольного места.

3.2 Прецеденты класса `stack`

Тестовые случаи для проверки корректности работы методов класса `stack`.

1. `constructor` — проверяет корректность конструкторов/деструкторов.
2. `push` — проверяет корректность вставки элемента в конец стека.
3. `pop` — проверяет корректность удаления элемента из конца стека.

3.3 Прецеденты класса `list`

Тестовые случаи для проверки корректности работы методов класса `list`.

1. `constructor` — проверяет корректность конструкторов/деструкторов.
2. `insert` — проверяет корректность вставки элемента в произвольное место.
3. `remove` — проверяет корректность удаления элемента из произвольного места.

3.4 Прецеденты класса `queue`

Тестовые случаи для проверки корректности работы методов класса `queue`.

1. `constructor` — проверяет корректность конструкторов/деструкторов.
2. `enqueue` — проверяет корректность вставки элемента в конец очереди.
3. `dequeue` — проверяет корректность удаления элемента из начала очереди.
4. `oversize` — проверяет корректность при переполнении очереди.

3.5 Прецеденты класса `tree`

Тестовые случаи для проверки корректности работы методов класса `tree`.

1. `constructor` — проверяет корректность конструкторов/деструкторов.
2. `put` — проверяет корректность вставки элемента в произвольное место.
3. `remove` — проверяет корректность удаления элемента из произвольного места.

Приложение А

А.1 Заголовочный файл класса array

```
#ifndef ARRAY_H
#define ARRAY_H

#include <algorithm>
#include <cmath>
#include <stdexcept>

#include "ADT.h"

class array {
    size_t _capacity, _size;
    ADT** _array;

    void resize(size_t);

public:
    array(size_t = 0);
    array(array&);
    virtual ~array();

    size_t size() const;
    bool is_empty() const;

    void insert(size_t, ADT*);
    ADT* remove(size_t);

    ADT*& operator[](size_t);
};

#endif
```

A.2 Файл реализации класса array

```
#include "array.h"

void array::resize(size_t capacity)
{
    ADT** array = new ADT*[capacity];

    std::copy(_array, _array + std::min(_capacity, capacity), array);

    delete[] _array;

    _array = new ADT*[capacity];

    std::copy(array, array + capacity, _array);

    delete[] array;

    _capacity = capacity;
}

array::array(size_t capacity)
{
    _capacity = capacity;
    _array = new ADT*[capacity];
    _size = 0;
}

array::array(array& other)
{
    _capacity = other._capacity;

    _array = new ADT*[_capacity];
    for (size_t i = 0; i < other._size; i++) {
        _array[i] = other._array[i];
    }

    _size = other._size;
}

array::~array()
{
    delete[] _array;
}

size_t array::size() const
{
    return _size;
}

bool array::is_empty() const
```

```
{
    return _size == 0;
}

void array::insert(size_t pos, ADT* value)
{
    if (pos > _size) {
        throw std::out_of_range("index out of range");
    }

    if (_size == _capacity) {
        if (_capacity > 0) {
            resize(2 * _capacity);
        } else {
            resize(1);
        }
    }

    ADT** array = new ADT*[_capacity];

    std::copy(_array, _array + pos, array);

    array[pos] = value;

    std::copy(_array + pos, _array + _capacity - 1, array + pos + 1);

    delete[] _array;

    _array = new ADT*[_capacity];

    std::copy(array, array + _capacity, _array);

    delete[] array;

    _size++;
}

ADT* array::remove(size_t pos)
{
    if (pos + 1 > _size) {
        throw std::out_of_range("index out of range");
    }

    ADT* value = _array[pos];

    ADT** array = new ADT*[_capacity];

    std::copy(_array, _array + pos, array);
    std::copy(_array + pos + 1, _array + _capacity, array + pos);
```

```
    delete[] _array;

    _array = new ADT*[_capacity];

    std::copy(array, array + _capacity, _array);

    delete[] array;

    _size--;

    if (_size <= _capacity / 2) {
        resize(_capacity / 2);
    }

    return value;
}

ADT*& array::operator[](size_t pos)
{
    if (pos + 1 > _size) {
        throw std::out_of_range("index out of range");
    }

    return _array[pos];
}
```

Приложение Б

Б.1 Заголовочный файл класса stack

```
#ifndef STACK_H
#define STACK_H

#include "array.h"

#include "ADT.h"

class stack : private array {
public:
    stack(size_t = 0);
    stack(stack&);

    size_t size() const;
    bool is_empty() const;

    void push(ADT*);
    ADT* pop();

    void push_front(ADT*);
    ADT* pop_front();

    ADT*& operator[](size_t);
};

#endif
```

Б.2 Файл реализации класса stack

```
#include "stack.h"

stack::stack(size_t capacity)
    : array(capacity)
{
}

stack::stack(stack& other)
    : array(other)
{
}

size_t stack::size() const
{
    return array::size();
}

bool stack::is_empty() const
{
    return array::is_empty();
}

void stack::push(ADT* value)
{
    array::insert(size(), value);
}

ADT* stack::pop()
{
    return array::remove(size() - 1);
}

void stack::push_front(ADT* value)
{
    array::insert(0, value);
}

ADT* stack::pop_front()
{
    return array::remove(0);
}

ADT*& stack::operator[](size_t pos)
{
    return array::operator[](pos);
}
```

Приложение B

B.1 Заголовочный файл класса list

```
#ifndef LIST_H
#define LIST_H

#include <stdexcept>

#include "ADT.h"

class list {
    struct node {
        ADT* _value;
        node* _next;

        node(ADT*, node* = nullptr);
    };

    size_t _size;
    node* _root;

public:
    list();
    list(list&);
    virtual ~list();

    size_t size() const;
    bool is_empty() const;

    void insert(size_t, ADT*);
    ADT* remove(size_t);

    ADT*& operator[](size_t);
};

#endif
```

B.2 Файл реализации класса list

```
#include "list.h"

list::node::node(ADT* value, node* next)
    : _value(value)
    , _next(next)
{
}

list::list()
    : _size(0)
    , _root(nullptr)
{
}

list::list(list& other)
{
    _size = other._size;
    _root = nullptr;

    for (size_t i = 0; i < other._size; i++) {
        insert(i, other[i]);
    }
}

list::~list()
{
    for (size_t i = 0; i < _size; i++) {
        remove(0);
    }
}

size_t list::size() const
{
    return _size;
}

bool list::is_empty() const
{
    return _root == nullptr;
}

void list::insert(size_t pos, ADT* value)
{
    if (pos > _size) {
        throw std::out_of_range("index out of range");
    }

    if (pos == 0) {
        if (_root == nullptr) {
```

```
            _root = new node(value);
        } else {
            _root = new node(value, _root);
        }

        _size++;

        return;
    }

    node* prev = _root;
    for (size_t i = 0; i < pos - 1; i++) {
        prev = prev->_next;
    }

    prev->_next = new node(value, prev->_next);

    _size++;
}

ADT* list::remove(size_t pos)
{
    if (pos + 1 > _size) {
        throw std::out_of_range("index out of range");
    }

    if (pos == 0) {
        ADT* value = _root->_value;
        node* next = _root->_next;

        delete _root;

        _root = next;

        _size--;

        return value;
    }

    node* prev = _root;
    for (size_t i = 0; i < pos - 1; i++) {
        prev = prev->_next;
    }

    ADT* value = prev->_next->_value;
    node* next = prev->_next->_next;

    delete prev->_next;

    prev->_next = next;
```

```
    _size--;

    return value;
}

ADT*& list::operator[](size_t pos)
{
    if (pos + 1 > _size) {
        throw std::out_of_range("index out of range");
    }

    node* node = _root;
    for (size_t i = 0; i < pos; i++) {
        node = node->_next;
    }

    return node->_value;
}
```

Приложение Г

Г.1 Заголовочный файл класса queue

```
#ifndef QUEUE_H
#define QUEUE_H

#include <stdexcept>

#include "ADT.h"

class queue {
    size_t _capacity, _start, _size;
    ADT** _circle;

public:
    queue(size_t);
    queue(queue&);
    virtual ~queue();

    size_t capacity() const;
    size_t size() const;
    bool is_empty() const;

    void enqueue(ADT*);
    ADT* dequeue();

    ADT*& operator[](size_t);
};

#endif
```

Г.2 Файл реализации класса queue

```
#include "queue.h"

queue::queue(size_t capacity)
{
    _capacity = capacity;
    _circle = new ADT*[capacity];
    _start = 0;
    _size = 0;
}

queue::queue(queue& other)
{
    _capacity = other._capacity;

    _circle = new ADT*[_capacity];
    for (size_t i = 0; i < _capacity; i++) {
        _circle[i] = other._circle[i];
    }

    _start = other._start;
    _size = other._size;
}

queue::~queue()
{
    delete[] _circle;
}

size_t queue::capacity() const
{
    return _capacity;
}

size_t queue::size() const
{
    return _size;
}

bool queue::is_empty() const
{
    return _size == 0;
}

void queue::enqueue(ADT* value)
{
    if (_size == _capacity) {
        throw std::out_of_range("index out of range");
    }
```

```
    _circle[(_start + _size++) % _capacity] = value;
}

ADT* queue::dequeue()
{
    if (_size == 0) {
        throw std::out_of_range("index out of range");
    }

    ADT* value = _circle[_start];

    _start = (_start + 1) % _capacity;
    _size--;

    return value;
}

ADT*& queue::operator[](size_t pos)
{
    if (pos + 1 > _size) {
        throw std::out_of_range("index out of range");
    }

    return _circle[(_start + pos) % _capacity];
}
```

Приложение Д

Д.1 Заголовочный файл класса tree

```
#ifndef TREE_H
#define TREE_H

#include <iostream>
#include <stdexcept>

#include "ADT.h"

template <typename K>
class tree {
    static const bool RED = true;
    static const bool BLACK = false;

    struct node {
        K _key;
        ADT* _value;
        node* _left;
        node* _right;
        size_t _size;
        bool _color;

        node(K, ADT*, size_t, bool);
    };

    node* _root;

    void copy(node*, node*);
    void clear(node*);

    bool is_red(node*) const;

    size_t size(node*) const;
    bool contains(node*, K) const;

    node* put(node*, K, ADT*);
    node* remove(node*, K);

    ADT*& get(node*, K);

    void flip_colors(node*);
    node* rotate_left(node*);
    node* rotate_right(node*);
    node* move_red_left(node*);
    node* move_red_right(node*);
```

```
    node* min(node*);
    node* max(node*);
    node* remove_min(node*);
    node* remove_max(node*);
    node* balance(node*);

public:
    tree();
    tree(tree&);
    virtual ~tree();

    size_t size() const;
    bool is_empty() const;
    bool contains(K) const;

    void put(K, ADT*);
    void remove(K);

    ADT*& operator[](K);
};

#endif
```

Д.2 Файл реализации класса tree

```
#include "tree.h"

template <typename K>
tree<K>::node::node(K key, ADT* value, size_t size, bool color)
    : _key(key)
    , _value(value)
    , _left(nullptr)
    , _right(nullptr)
    , _size(size)
    , _color(color)
{
}

template <typename K>
void tree<K>::copy(node* from, node* to)
{
    if (from == nullptr) {
        return;
    }

    to = new node(from->_key, from->_value, from->_size, from->_color);

    copy(from->_left, to->_left);
    copy(from->_right, to->_right);
}

template <typename K>
void tree<K>::clear(node* n)
{
    if (n == nullptr) {
        return;
    }

    clear(n->_left);
    clear(n->_right);

    delete n;
}

template <typename K>
bool tree<K>::is_red(node* n) const
{
    if (n == nullptr) {
        return BLACK;
    } else {
        return n->_color == RED;
    }
}
```

```
template <typename K>
size_t tree<K>::size(node* n) const
{
    if (n == nullptr) {
        return 0;
    } else {
        return n->_size;
    }
}

template <typename K>
bool tree<K>::contains(node* n, K key) const
{
    while (n != nullptr) {
        if (key < n->_key) {
            n = n->_left;
        } else if (key > n->_key) {
            n = n->_right;
        } else {
            return true;
        }
    }

    return false;
}

template <typename K>
typename tree<K>::node* tree<K>::put(node* n, K key, ADT* value)
{
    if (n == nullptr) {
        return new node(key, value, 1, RED);
    }

    if (key < n->_key) {
        n->_left = put(n->_left, key, value);
    } else if (key > n->_key) {
        n->_right = put(n->_right, key, value);
    } else {
        n->_value = value;
    }

    if (is_red(n->_right) && !is_red(n->_left)) {
        n = rotate_left(n);
    }

    if (is_red(n->_left) && is_red(n->_left->_left)) {
        n = rotate_right(n);
    }

    if (is_red(n->_left) && is_red(n->_right)) {
```

```
        flip_colors(n);
    }

    n->_size = size(n->_left) + size(n->_right) + 1;

    return n;
}

template <typename K>
typename tree<K>::node* tree<K>::remove(node* n, K key)
{
    if (key < n->_key) {
        if (!is_red(n->_left) && !is_red(n->_left->_left)) {
            n = move_red_left(n);
        }
        n->_left = remove(n->_left, key);
    } else {
        if (is_red(n->_left)) {
            n = rotate_right(n);
        }

        if (key == n->_key && n->_right == nullptr) {
            return nullptr;
        }

        if (!is_red(n->_right) && !is_red(n->_right->_left)) {
            n = move_red_right(n);
        }

        if (key == n->_key) {
            node* temp = min(n->_right);
            n->_key = temp->_key;
            n->_value = temp->_value;
            n->_right = remove_min(n->_right);
        } else {
            n->_right = remove(n->_right, key);
        }
    }

    return balance(n);
}

template <typename K>
ADT*& tree<K>::get(node* n, K key)
{
    while (n != nullptr) {
        if (key < n->_key) {
            n = n->_left;
        } else if (key > n->_key) {
            n = n->_right;
```

```
        } else {
            return n->_value;
        }
    }
}

template <typename K>
void tree<K>::flip_colors(node* n)
{
    n->_color = !n->_color;
    n->_left->_color = !n->_left->_color;
    n->_right->_color = !n->_right->_color;
}

template <typename K>
typename tree<K>::node* tree<K>::rotate_left(node* n)
{
    node* temp = n->_right;
    n->_right = temp->_left;
    temp->_left = n;
    temp->_color = temp->_left->_color;
    temp->_left->_color = RED;
    temp->_size = n->_size;
    n->_size = size(n->_left) + size(n->_right) + 1;

    return temp;
}

template <typename K>
typename tree<K>::node* tree<K>::rotate_right(node* n)
{
    node* temp = n->_left;
    n->_left = temp->_right;
    temp->_right = n;
    temp->_color = temp->_right->_color;
    temp->_right->_color = RED;
    temp->_size = n->_size;
    n->_size = size(n->_left) + size(n->_right) + 1;

    return temp;
}

template <typename K>
typename tree<K>::node* tree<K>::move_red_left(node* n)
{
    flip_colors(n);
    if (is_red(n->_right->_left)) {
        n->_right = rotate_right(n->_right);
        n = rotate_left(n);
        flip_colors(n);
```

```
    }

    return n;
}

template <typename K>
typename tree<K>::node* tree<K>::move_red_right(node* n)
{
    flip_colors(n);
    if (is_red(n->_left->_left)) {
        n = rotate_right(n);
        flip_colors(n);
    }

    return n;
}

template <typename K>
typename tree<K>::node* tree<K>::min(node* n)
{
    if (n->_left == nullptr) {
        return n;
    } else {
        return min(n->_left);
    }
}

template <typename K>
typename tree<K>::node* tree<K>::max(node* n)
{
    if (n->_right == nullptr) {
        return n;
    } else {
        return max(n->_right);
    }
}

template <typename K>
typename tree<K>::node* tree<K>::remove_min(node* n)
{
    if (n->_left == nullptr) {
        return nullptr;
    }

    if (!is_red(n->_left) && !is_red(n->_left->_left)) {
        n = move_red_left(n);
    }

    n->_left = remove_min(n->_left);
```

```
    return balance(n);
}

template <typename K>
typename tree<K>::node* tree<K>::remove_max(node* n)
{
    if (is_red(n->_left)) {
        n = rotate_right(n);
    }

    if (n->_right == nullptr) {
        return nullptr;
    }

    if (!is_red(n->_right) && !is_red(n->_right->_left)) {
        n = move_red_right(n);
    }

    n->_right = remove_max(n->_right);

    return balance(n);
}

template <typename K>
typename tree<K>::node* tree<K>::balance(node* n)
{
    if (is_red(n->_right) && !is_red(n->_left)) {
        n = rotate_left(n);
    }

    if (is_red(n->_left) && is_red(n->_left->_left)) {
        n = rotate_right(n);
    }

    if (is_red(n->_left) && is_red(n->_right)) {
        flip_colors(n);
    }

    n->_size = size(n->_left) + size(n->_right) + 1;

    return n;
}

template <typename K>
tree<K>::tree()
    : _root(nullptr)
{
}

template <typename K>
```

```
tree<K>::tree(tree& other)
    : _root(nullptr)
{
    copy(other._root, _root);
}

template <typename K>
tree<K>::~tree()
{
    clear(_root);
}

template <typename K>
size_t tree<K>::size() const
{
    return size(_root);
}

template <typename K>
bool tree<K>::is_empty() const
{
    return _root == nullptr;
}

template <typename K>
bool tree<K>::contains(K key) const
{
    return contains(_root, key);
}

template <typename K>
void tree<K>::put(K key, ADT* value)
{
    if (value == nullptr) {
        remove(key);
        return;
    }

    _root = put(_root, key, value);
    _root->_color = BLACK;
}

template <typename K>
void tree<K>::remove(K key)
{
    if (!contains(key)) {
        return;
    }

    if (!is_red(_root->_left) && !is_red(_root->_right)) {
```

```
        _root->_color = RED;
    }

    _root = remove(_root, key);

    if (!is_empty()) {
        _root->_color = BLACK;
    }
}

template <typename K>
ADT*& tree<K>::operator[](K key)
{
    if (!contains(key)) {
        throw std::invalid_argument("key is not exist");
    }

    return get(_root, key);
}

template class tree<size_t>;
```