Правительство Российской Федерации

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

Кафедра «Компьютерная безопасность»

ОТЧЕТ
К ЛАБОРАТОРНОЙ РАБОТЕ №3

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

«Итератор»

Разработать программу на языке Си++ (ISO/IEC 14882:2014), демонстрирующую решение поставленной задачи. Весь код разбить на заголовочные файлы *.h и соответствующие им файлы реализации *.cpp (а также main.cpp, файл конфигурации системы сборки CMake и отчет о покрытии кода). По выполненной работе составить отчет согласно требованиям ГОСТ 7.32-2017, содержащий диаграмму классов согласно спецификации UML 2.0 и предоставить его копию в формате PDF. Все необходимые файлы разместить в git-репозитории (ссылка будет указана отдельно), взаимодействие выполнять с использованием платформы GitHub. Решения, предоставленные иными способами, не принимаются. Выполненные работы подлежат процедуре защиты. Необходимым условием является защита лабораторной работы 2. Защита проводится во время срока выполнения работы. Срок выполнения работы 19 июня 2021 года. После истечения срока работа считается не сданной и оценивается в 0 баллов. Все присланные работы проверяются на отсутствие ошибок сборки в автоматическом режиме, в случае отсутствия ошибок выполняется ручная проверка кода, в противном случае работа оценивается в 0 баллов. Ручная проверка работ выполняется преподавателем – оценивается полнота выполнения задания, а также качество решения. Оценка выставляется в 10 балльной шкале. Все работы проверяются в системе Антиплагиат. Компилятор и операционная система, используемые при проверке: GNU Compiler Collection 10.2.0 цель x86_64-slackware-linux система Slackware Linux current 2020-10-26.

Постановка задачи

Переработать классы, разработанные в рамках лабораторной работы 2.

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

Разработать два вида итераторов (обычный и константный) для указанных шаблонов классов.

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

При разработке тестов, добиться полного покрытия. Отчет о покрытии приложить к работе.

Задачи

1. Шаблон «динамический массив объектов». Размерность массива не изменяется в момент его переполнения. Начальная размерность задается как параметр конструктора, значение по умолчанию 0. Метод изменения размера. Добавление/удаление элемента в произвольное место.

2. Шаблон «стек» (внутреннее представление – динамический массив хранимых объектов). 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. Шаблон «односвязный список объектов». Добавление/удаление элемента в произвольное место.

4. Шаблон «циклическая очередь» (внутреннее представление – динамический массив хранимых объектов). Добавление/удаление элемента в произвольное место.

5. Шаблон «двоичное дерево объектов». 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.

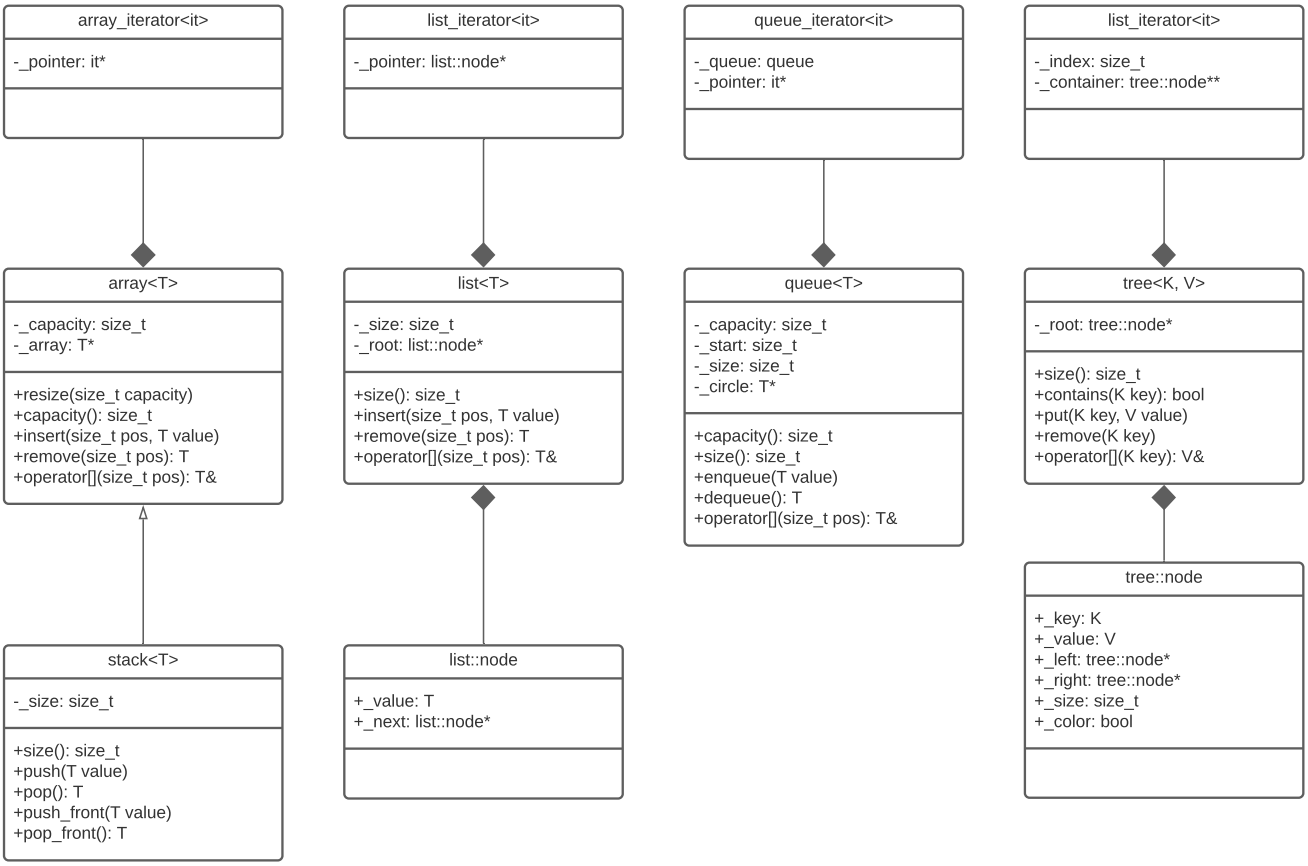

Рис. 1: Диаграмма классов

1.1 Динамический массив

Итератор для динамического массива предельно простой, поскольку внутри такой контейнер представлен как непрерывная последовательность ячеек памяти. Таким образом, для итерации по динамическому массиву достаточно воспользоваться стандартной арифметикой указателей. В связи с некоторыми отличиями текущего задания от предыдущего, класс для представления контейнера был несколько переработан. В частности, процедура resize была вынесена к публичным методам, и операции вставки/удаления больше не инициируют изменение размерности. За ненадобностью была удалена переменная size, ранее необходимая для хранения количества заполненных элементов динамического массива.
Полный код заголовочных файлов и файлов реализации динамического массива приведены в приложении А.

1.1.1 Стек

Стек, будучи наследуюемым от динамического массива, не имеет отличий при работе с итератором.
Однако, требования для стека отличаются от требований для динамического массива. Теперь стек содержит переменную size, а также инициирует изменение размерности контейнера в случаях его сужения/переполнения.
Полный код заголовочных файлов и файлов реализации стека приведены в приложении Б.

1.2 Односвязный список

В отличие от динамического массива, элементы односвязного списока не представлены в виде непрерывной последовательности ячеек в памяти. Для итерирования по контейнеру необходимо передавать указатель на `node` и при увеличении итератора заполнять его значение следующим — `node = node->next`. Концом контейнера считается значение `nullptr`, что является корректным решением, поскольку ни один заполненный узел, даже имея пустое значенение, не обращается в пустой указатель.
Полный код заголовочных файлов и файлов реализации односвязного списка приведены в приложении В.

1.3 Циклическая очередь

Хотя по представлению элементов в памяти циклическая очередь схожа с динамическим массивом, она имеет такую особенность, что указатель на конец контейнера может находиться в памяти левее, чем указатеель на его начало. Поэтому было решено дополнительно передавать в итератор указатель на очередь. При достижении итератором конца контейнера, значение указателя сбрасывается до начала очереди.
Полный код заголовочных файлов и файлов реализации односвязного списка приведены в приложении Г.

1.4 Бинарное дерево

Бинарное дерево является самым нетривиальным для итерирования контейнером. Дело в том, что при итерации существует необходимость сохранять порядок элементов, а стандартный поиск в ширину не удовлетворяет данному условию, поскольку в текущей реализации дерева в левых потомках содержатся только меньшие ключи, а в правых — только большие. Поиск в глубину тоже не решает задачу, ибо в текущей реализации бинарного дерева не хранятся указатели на родительский узел, а адаптация существующего интерфейса под нужды итератора — это кощунство по отношению к уже реализованным модулям.
Таким образом, было решено передавать в итератор указатель на корневой узел (точно так же, как это сделано в односвязном списке), и в конструкторе единожды заполнять временный контейнер, представленный динамическим списком, с помощью поиска в глубину. Для поддержания указателя на конец дерева, размерность временного контейнера на единицу больше количества элементов дерева, и последний элемент заполнен значением `nullptr`. Аналогично односвязному списку, значение `node` обращается в пустой указатель только в том случае, если был достигнут конец контейнера.
Функция `fill_container`, заполняющая временный контейнер исходя из значения указателя на корневой узел, представлена в листинге 1.

Листинг 1: Функция для заполнения временного контейнера

```
size_t fill_container(node* pointer, size_t index)
{
        if (pointer == nullptr) {
                return index;
        }

        index = fill_container(pointer->_left, index);

        _container[index] = pointer;

        return fill_container(pointer->_right, index + 1);
}
```

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 2.

Листинг 2: Файл сборки CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)

project(laboratory-work-03-kapavkin)

set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_FLAGS --coverage)

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(array)
add_subdirectory(list)
add_subdirectory(queue)
add_subdirectory(tree)

target_link_libraries(main
array
list
queue
tree
gtest_main
)

include(GoogleTest)
gtest_discover_tests(main)
```

3 Тестирование

В целях проверки корректности работы реализованных классов были проведены процедуры тестирования приложения, разработаны тест-кейсы с использованием библиотеки `Google Test`, составлен отчет о покрытии кода с помощью утилиты `gcov`.

3.1 Варианты использования

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

3.1.1 Класс `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.
4. `iterator` — проверяет корректность итерирования по динамическому массиву и совпадение полученного в результате итерирования количества элементов с размером контейнера.

3.1.2 Класс `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.
4. `iterator` — проверяет корректность итерирования постеку и совпадение полученного в результате итерирования количества элементов с размером контейнера.

3.1.3 Класс `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.
4. `iterator` — проверяет корректность итерирования по односвязному списку и совпадение полученного в результате итерирования количества элементов с размером контейнера.

3.1.4 Класс `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.
5. `iterator` — проверяет корректность итерирования по циклической очереди и совпадение полученного в результате итерирования количества элементов с размером контейнера.

3.1.5 Класс `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.
4. `iterator` — проверяет корректность итерирования по красно-черному дереву и совпадение полученного в результате итерирования количества элементов с размером контейнера.

3.2 Отчет о покрытии кода

На рисунке 2 показан отчет о покрытии кода тестами.

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level				
Test:	gtest_coverage.info	**Lines:**	992	1286	77.1 %
Date:	2021-06-17 01:56:48	**Functions:**	543	634	85.6 %

Directory	Line Coverage		Functions	
/Users/kirillpavkin/Downloads/laboratory-work-03-kapavkin	100.0 %	241 / 241	100.0 %	148 / 148
/Users/kirillpavkin/Downloads/laboratory-work-03-kapavkin/array	94.6 %	70 / 74	95.2 %	20 / 21
/Users/kirillpavkin/Downloads/laboratory-work-03-kapavkin/array/stack	94.7 %	36 / 38	92.3 %	12 / 13
/Users/kirillpavkin/Downloads/laboratory-work-03-kapavkin/build/_deps/googletest-src/googletest/include/gtest	26.6 %	21 / 79	30.8 %	16 / 52
/Users/kirillpavkin/Downloads/laboratory-work-03-kapavkin/build/_deps/googletest-src/googletest/include/gtest/internal	76.7 %	23 / 30	97.8 %	135 / 138
/Users/kirillpavkin/Downloads/laboratory-work-03-kapavkin/list	93.4 %	85 / 91	95.5 %	21 / 22
/Users/kirillpavkin/Downloads/laboratory-work-03-kapavkin/queue	88.5 %	54 / 61	95.0 %	19 / 20
/Users/kirillpavkin/Downloads/laboratory-work-03-kapavkin/tree	79.8 %	182 / 228	90.2 %	37 / 41
v1	63.1 %	280 / 444	75.4 %	135 / 179

Generated by: LCOV version 1.15

Рис. 2: Отчет о покрытии кода

Приложение А

А.1 Заголовочный файл класса array

```
#ifndef ARRAY_H
#define ARRAY_H

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

template <typename T>
class array {
protected:
    size_t _capacity;
    T* _array;

    template <typename it>
    class array_iterator : public std::iterator<std::input_iterator_tag, it> {
        it* _pointer;

    public:
        array_iterator(it* pointer = nullptr)
            : _pointer(pointer)
        {
        }

        array_iterator(const array_iterator& other)
            : _pointer(other._pointer)
        {
        }

        ~array_iterator()
        {
        }

        bool operator==(const array_iterator& other) const
        {
            return _pointer == other._pointer;
        }

        bool operator!=(const array_iterator& other) const
        {
            return _pointer != other._pointer;
        }

        it& operator*()
        {
```

```
            return *_pointer;
        }

        const it& operator*() const
        {
            return *_pointer;
        }

        array_iterator& operator++()
        {
            _pointer++;

            return *this;
        }

        array_iterator operator++(int)
        {
            array_iterator copy(*this);

            _pointer++;

            return copy;
        }
    };

public:
    typedef array_iterator<T> iterator;
    typedef array_iterator<const T> const_iterator;

    array(size_t capacity = 0)
    {
        _capacity = capacity;
        _array = new T[capacity];
    }

    array(array& other)
    {
        _capacity = other._capacity;
        _array = new T[_capacity];

        std::copy(other._array, other._array + _capacity, _array);
    }

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

    void resize(size_t capacity)
    {
```

```
    T* array = new T[capacity];

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

    delete[] _array;

    _array = new T[capacity];

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

    delete[] array;

    _capacity = capacity;
}

size_t capacity() const
{
    return _capacity;
}

void insert(size_t index, T value)
{
    if (index >= _capacity) {
        throw std::out_of_range("index out of range");
    }

    T* array = new T[_capacity];

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

    array[index] = value;

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

    delete[] _array;

    _array = new T[_capacity];

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

    delete[] array;
}

T remove(size_t index)
{
    if (index >= _capacity) {
        throw std::out_of_range("index out of range");
    }

    T value = _array[index];
```

```
        T* array = new T[_capacity];

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

        delete[] _array;

        _array = new T[_capacity];

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

        delete[] array;

        return value;
    }

    T& operator[](size_t index)
    {
        if (index >= _capacity) {
            throw std::out_of_range("index out of range");
        }

        return _array[index];
    }

    iterator begin()
    {
        return iterator(_array);
    }

    iterator end()
    {
        return iterator(_array + _capacity);
    }

    const_iterator begin() const
    {
        return const_iterator(_array);
    }

    const_iterator end() const
    {
        return const_iterator(_array + _capacity);
    }
};

#endif
```

A.2 Файл реализации класса array

```
#include "array.h"
```

Приложение Б

Б.1 Заголовочный файл класса stack

```
#ifndef STACK_H
#define STACK_H

#include "array.h"

template <typename T>
class stack : private array<T> {
    size_t _size;

public:
    typedef typename array<T>::template array_iterator<T> iterator;
    typedef typename array<T>::template array_iterator<const T> const_iterator;

    stack(size_t capacity = 0)
        : array<T>::array(capacity)
    {
        _size = 0;
    }

    stack(stack& other)
        : array<T>::array(other)
    {
        _size = other._size;
    }

    size_t size() const
    {
        return _size;
    }

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

    void push(T value)
    {
        if (_size == array<T>::_capacity) {
            if (array<T>::_capacity > 0) {
                array<T>::resize(2 * array<T>::_capacity);
            } else {
                array<T>::resize(1);
            }
        }
```

```
    array<T>::insert(size(), value);

    _size++;
}

T pop()
{
    T value = array<T>::remove(size() - 1);

    if (--_size <= array<T>::_capacity / 2) {
        array<T>::resize(array<T>::_capacity / 2);
    }

    return value;
}

void push_front(T value)
{
    if (_size == array<T>::_capacity) {
        if (array<T>::_capacity > 0) {
            array<T>::resize(2 * array<T>::_capacity);
        } else {
            array<T>::resize(1);
        }
    }

    array<T>::insert(0, value);

    _size++;
}

T pop_front()
{
    T value = array<T>::remove(0);

    if (--_size <= array<T>::_capacity / 2) {
        array<T>::resize(array<T>::_capacity / 2);
    }

    return value;
}

T& operator[](size_t index)
{
    if (index >= _size) {
        throw std::out_of_range("index out of range");
    }

    return array<T>::operator[](index);
```

```
    }

    iterator begin()
    {
        return array<T>::begin();
    }

    iterator end()
    {
        return iterator(array<T>::_array + _size);
    }

    const_iterator begin() const
    {
        return array<T>::begin();
    }

    const_iterator end() const
    {
        return const_iterator(array<T>::_array + _size);
    }
};

#endif
```

Б.2 Файл реализации класса stack

```
#include "stack.h"
```

Приложение B

B.1 Заголовочный файл класса list

```
#ifndef LIST_H
#define LIST_H

#include <iterator>
#include <stdexcept>

template <typename T>
class list {
    struct node {
        T _value;
        node* _next;

        node(T value, node* next = nullptr)
            : _value(value)
            , _next(next)
        {
        }
    };

    size_t _size;
    node* _root;

    template <typename it>
    class list_iterator : public std::iterator<std::input_iterator_tag, it> {
        node* _pointer;

    public:
        list_iterator(node* pointer = nullptr)
            : _pointer(pointer)
        {
        }

        list_iterator(const list_iterator& other)
            : _pointer(other._pointer)
        {
        }

        ~list_iterator()
        {
        }

        bool operator==(const list_iterator& other) const
        {
            return _pointer == other._pointer;
```

```
        }

        bool operator!=(const list_iterator& other) const
        {
            return _pointer != other._pointer;
        }

        it& operator*()
        {
            return _pointer->_value;
        }

        const it& operator*() const
        {
            return _pointer->_value;
        }

        list_iterator& operator++()
        {
            _pointer = _pointer->_next;

            return *this;
        }

        list_iterator operator++(int)
        {
            list_iterator copy(*this);

            _pointer = _pointer->_next;

            return copy;
        }
    };

public:
    typedef list_iterator<T> iterator;
    typedef list_iterator<const T> const_iterator;

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

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

        for (size_t i = 0; i < other._size; i++) {
```

```
        insert(i, other[i]);
    }
}

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

size_t size() const
{
    return _size;
}

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

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

    if (index == 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 < index - 1; i++) {
        prev = prev->_next;
    }

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

    _size++;
}

T remove(size_t index)
```

```
{
    if (index + 1 > _size) {
        throw std::out_of_range("index out of range");
    }

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

        delete _root;

        _root = next;

        _size--;

        return value;
    }

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

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

    delete prev->_next;

    prev->_next = next;

    _size--;

    return value;
}

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

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

    return node->_value;
}

iterator begin()
```

```
    {
        return iterator(_root);
    }

    iterator end()
    {
        return iterator();
    }

    const_iterator begin() const
    {
        return const_iterator(_root);
    }

    const_iterator end() const
    {
        return const_iterator();
    }
};

#endif
```

B.2 Файл реализации класса list

```
#include "list.h"
```

Приложение Г

Г.1 Заголовочный файл класса queue

```
#ifndef QUEUE_H
#define QUEUE_H

#include <algorithm>
#include <iterator>
#include <stdexcept>

template <typename T>
class queue {
    size_t _capacity, _start, _size;
    T* _circle;

    template <typename it>
    class queue_iterator : public std::iterator<std::input_iterator_tag, it> {
        queue _queue;
        it* _pointer;

    public:
        queue_iterator(const queue& queue, it* pointer = nullptr)
            : _queue(queue)
            , _pointer(pointer)
        {
        }

        queue_iterator(const queue_iterator& other)
            : _queue(other._queue)
            , _pointer(other._pointer)
        {
        }

        ~queue_iterator()
        {
        }

        bool operator==(const queue_iterator& other) const
        {
            return _pointer == other._pointer;
        }

        bool operator!=(const queue_iterator& other) const
        {
            return _pointer != other._pointer;
        }
```

```
        it& operator*()
        {
            return *_pointer;
        }

        const it& operator*() const
        {
            return *_pointer;
        }

        queue_iterator& operator++()
        {
            if (_pointer + 1 == _queue._circle + _queue._capacity) {
                _pointer = _queue._circle;
            } else {
                _pointer++;
            }

            return *this;
        }

        queue_iterator operator++(int)
        {
            queue_iterator copy(*this);

            if (_pointer + 1 == _queue._circle + _queue._capacity) {
                _pointer = _queue._circle;
            } else {
                _pointer++;
            }

            return copy;
        }
    };

public:
    typedef queue_iterator<T> iterator;
    typedef queue_iterator<const T> const_iterator;

    queue(size_t capacity)
        : _capacity(capacity + 1)
        , _circle(new T[capacity + 1])
        , _start(0)
        , _size(0)
    {
    }

    queue(const queue& other)
        : _capacity(other._capacity)
        , _circle(new T[_capacity])
```

```
    , _start(other._start)
    , _size(other._size)
{
    std::copy(other._circle, other._circle + _capacity, _circle);
}

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

size_t capacity() const
{
    return _capacity;
}

size_t size() const
{
    return _size;
}

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

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

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

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

    T value = _circle[_start];

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

    return value;
}

T& operator[](size_t index)
```

```
    {
        if (index >= _size) {
            throw std::out_of_range("index out of range");
        }

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

    iterator begin()
    {
        return iterator(*this, _circle + _start);
    }

    iterator end()
    {
        return iterator(*this, _circle + (_start + _size) % _capacity);
    }

    const_iterator begin() const
    {
        return const_iterator(*this, _circle + _start);
    }

    const_iterator end() const
    {
        return const_iterator(*this, _circle + (_start + _size) % _capacity);
    }
};

#endif
```

Г.2 Файл реализации класса queue

```
#include "queue.h"
```

Приложение Д

Д.1 Заголовочный файл класса tree

```
#ifndef TREE_H
#define TREE_H

#include <iterator>
#include <stdexcept>
#include <utility>

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

    struct node {
        std::pair<K, V> _cortege;
        node* _left;
        node* _right;
        size_t _size;
        bool _color;

        node(std::pair<K, V> cortege, size_t size, bool color)
            : _cortege(cortege)
            , _left(nullptr)
            , _right(nullptr)
            , _size(size)
            , _color(color)
        {
        }
    };

    node* _root;

    static void copy(node* from, node* to)
    {
        if (from == nullptr) {
            return;
        }

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

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

    static void clear(node* n)
```

```
{
    if (n == nullptr) {
        return;
    }

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

    delete n;
}

static bool is_red(node* n)
{
    if (n == nullptr) {
        return BLACK;
    } else {
        return n->_color == RED;
    }
}

static size_t size(node* n)
{
    if (n == nullptr) {
        return 0;
    } else {
        return n->_size;
    }
}

static bool contains(node* n, K key)
{
    while (n != nullptr) {
        if (key < n->_cortege.first) {
            n = n->_left;
        } else if (key > n->_cortege.first) {
            n = n->_right;
        } else {
            return true;
        }
    }

    return false;
}

static node* put(node* n, K key, V value)
{
    if (n == nullptr) {
        return new node(std::make_pair(key, value), 1, RED);
    }
```

```
    if (key < n->_cortege.first) {
        n->_left = put(n->_left, key, value);
    } else if (key > n->_cortege.first) {
        n->_right = put(n->_right, key, value);
    } else {
        n->_cortege.second = 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;
}

static node* remove(node* n, K key)
{
    if (key < n->_cortege.first) {
        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->_cortege.first && n->_right == nullptr) {
            return nullptr;
        }

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

        if (key == n->_cortege.first) {
            node* temp = min(n->_right);
            n->_cortege.first = temp->_cortege.first;
            n->_cortege.second = temp->_cortege.second;
            n->_right = remove_min(n->_right);
```

```
        } else {
            n->_right = remove(n->_right, key);
        }
    }

    return balance(n);
}

static V& get(node* n, K key)
{
    while (n != nullptr) {
        if (key < n->_cortege.first) {
            n = n->_left;
        } else if (key > n->_cortege.first) {
            n = n->_right;
        } else {
            return n->_cortege.second;
        }
    }
}

static void flip_colors(node* n)
{
    n->_color = !n->_color;
    n->_left->_color = !n->_left->_color;
    n->_right->_color = !n->_right->_color;
}

static node* 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;
}

static node* 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;
}

static node* 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;
}

static node* move_red_right(node* n)
{
    flip_colors(n);
    if (is_red(n->_left->_left)) {
        n = rotate_right(n);
        flip_colors(n);
    }

    return n;
}

static node* min(node* n)
{
    if (n->_left == nullptr) {
        return n;
    } else {
        return min(n->_left);
    }
}

static node* max(node* n)
{
    if (n->_right == nullptr) {
        return n;
    } else {
        return max(n->_right);
    }
}

static node* 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);
}

static node* 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);
}

static node* 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 it>
class tree_iterator : public std::iterator<std::input_iterator_tag, it> {
```

```
    size_t _index;
    node** _container;

    size_t fill_container(node* pointer, size_t index)
    {
        if (pointer == nullptr) {
            return index;
        }

        index = fill_container(pointer->_left, index);

        _container[index] = pointer;

        return fill_container(pointer->_right, index + 1);
    }

public:
    tree_iterator(node* pointer = nullptr)
        : _index(0)
        , _container(new node*[tree<K, V>::size(pointer) + 1])
    {
        fill_container(pointer, 0);
        _container[tree<K, V>::size(pointer)] = nullptr;
    }

    tree_iterator(const tree_iterator& other)
        : _index(other._index)
        , _container(new node*[sizeof(other._container) / sizeof(*(other._container)
    {
        std::copy(other._container, other._container + sizeof(other._container) / si
    }

    ~tree_iterator()
    {
        delete[] _container;
    }

    bool operator==(const tree_iterator& other) const
    {
        return _container[_index] == other._container[other._index];
    }

    bool operator!=(const tree_iterator& other) const
    {
        return _container[_index] != other._container[other._index];
    }

    it& operator*()
    {
        return _container[_index]->_cortege;
```

```
        }

        const it& operator*() const
        {
            return _container[_index]->_cortege;
        }

        tree_iterator& operator++()
        {
            _index++;

            return *this;
        }

        tree_iterator operator++(int)
        {
            tree_iterator copy(*this);

            _index++;

            return copy;
        }
    };

public:
    typedef tree_iterator<std::pair<K, V>> iterator;
    typedef tree_iterator<const std::pair<K, V>> const_iterator;

    tree()
        : _root(nullptr)
    {
    }

    tree(tree& other)
        : _root(nullptr)
    {
        copy(other._root, _root);
    }

    virtual ~tree()
    {
        clear(_root);
    }

    size_t size() const
    {
        return size(_root);
    }

    bool is_empty() const
```

```
{
    return _root == nullptr;
}

bool contains(K key) const
{
    return contains(_root, key);
}

void put(K key, V value)
{
    _root = put(_root, key, value);
    _root->_color = BLACK;
}

void 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;
    }
}

V& operator[](K key)
{
    if (!contains(key)) {
        throw std::invalid_argument("key is not exist");
    }

    return get(_root, key);
}

iterator begin()
{
    return iterator(_root);
}

iterator end()
{
    return iterator();
}
```

```
    const_iterator begin() const
    {
        return const_iterator(_root);
    }

    const_iterator end() const
    {
        return const_iterator();
    }
};

#endif
```

Д.2 Файл реализации класса tree

```
#include "tree.h"
```