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 А.Р. Фахретдинов

подпись, дата

Работу проверил С.А. Булгаков

подпись, дата

Москва 2021

Содержание

Постановка задачи

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

Общая часть

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 количества 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. Разработать два вида итераторов (обычный и константный) для указанных шаблонов классов.
2. 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.
3. 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.
4. Шаблон «односвязный список объектов». Добавление/удаление элемента в произвольное место.
5. Шаблон «циклическая очередь» (внутреннее представление – динамический массив хранимых объектов). Добавление/удаление элемента в произвольное место.
6. Шаблон «двоичное дерево объектов». Добавление/удаление элемента в произвольное место.

1 Алгоритм решения задачи

1.1 Задача №1

Для решения задачи были разработаны классы Iterator и Const_iterator, унаследованые от шаблонного итератора из стандартной библиотеки Си++. Были реализованы базовые конрукторы и перегружены различные операторы. Отличие Const_iterator от iterator заключается в присутствие в методах первого модификаторов const, возвращающих константные ссылки на не изменяемые объекты.

1.2 Задача №2

Для решения задачи был разработан шаблонный класс типа «контейнер», хранящий в себе динамический массив объектов произвольного типа T. Был разработан конструктор по умолчанию, задающий размерность по умолчанию, равной 0. Так же были созданы методы для работы с объектами класса, в частности метод insert для добавления элемента в произвольное место массива и метод erase для удаления элемента из произвольного места массива. Для доступа к элементам массива был перегружен оператор квадратных скобок.

1.3 Задача №3

Для решения задачи был разработан шаблонный класс типа «контейнер», хранящий в себе динамический массив объектов произвольного типа T. Был разработан конструктор по умолчанию, задающий размерность по умолчанию, равной 0. По-мимо этого были созданы методы для работы с объектами класса, в частности методы push_back, push_front, pop_back, pop_front для добавления элемента в конец/начало массива и удаления элемента в конец/начало массива соответственно. Для доступа к элементам массива был перегружен оператор квадратных скобок.

1.4 Задача №4

Для решения задачи было разработано два класса: класс Node — узел, хранящий адрес следующего узла и значение объекта произвольного типа T и шаблонный класс типа «контейнер», хранящий в себе указатели на начальный и конечные узлы. Так же были созданы методы для работы с объектами класса, в частности метод insert для добавления элемента в произвольное место массива и метод erase для удаления элемента из произвольного места массива. Для доступа к указателям на объекты типа Node был перегружен оператор квадратных скобок.

1.5 Задача №5

Для решения задачи был разработан шаблонный класс типа «контейнер», хранящий в себе динамический массив объектов произвольного типа T и при переполнении, циклично перезаписывающий данные. Таким образом в массиве выполняется принцип FIFO — first in first out. Так же были созданы методы для работы с объектами класса — метод insert для добавления элемента в произвольное место массива и метод erase для удаления элемента из произвольного места массива. Для доступа к элементам массива был перегружен оператор квадратных скобок.

1.6 Задача №6

Для решения задачи было разработано два класса: класс Tnode — лист, хранящий указатели на левый и правый лист дерева, родитель и значение объекта произвольного типа T и шаблонный класс типа «контейнер», хранящий указатель tree на корень дерева типа Tnode. Так же были созданы методы для работы с объектами класса, в частности метод insert для добавления элемента в произвольное место дерева и метод erase для удаления элемента из произвольного места дерева. Для доступа к элементам массива был перегружен оператор квадратных скобок.

2 Выполнение задания

2.1 Задача №1

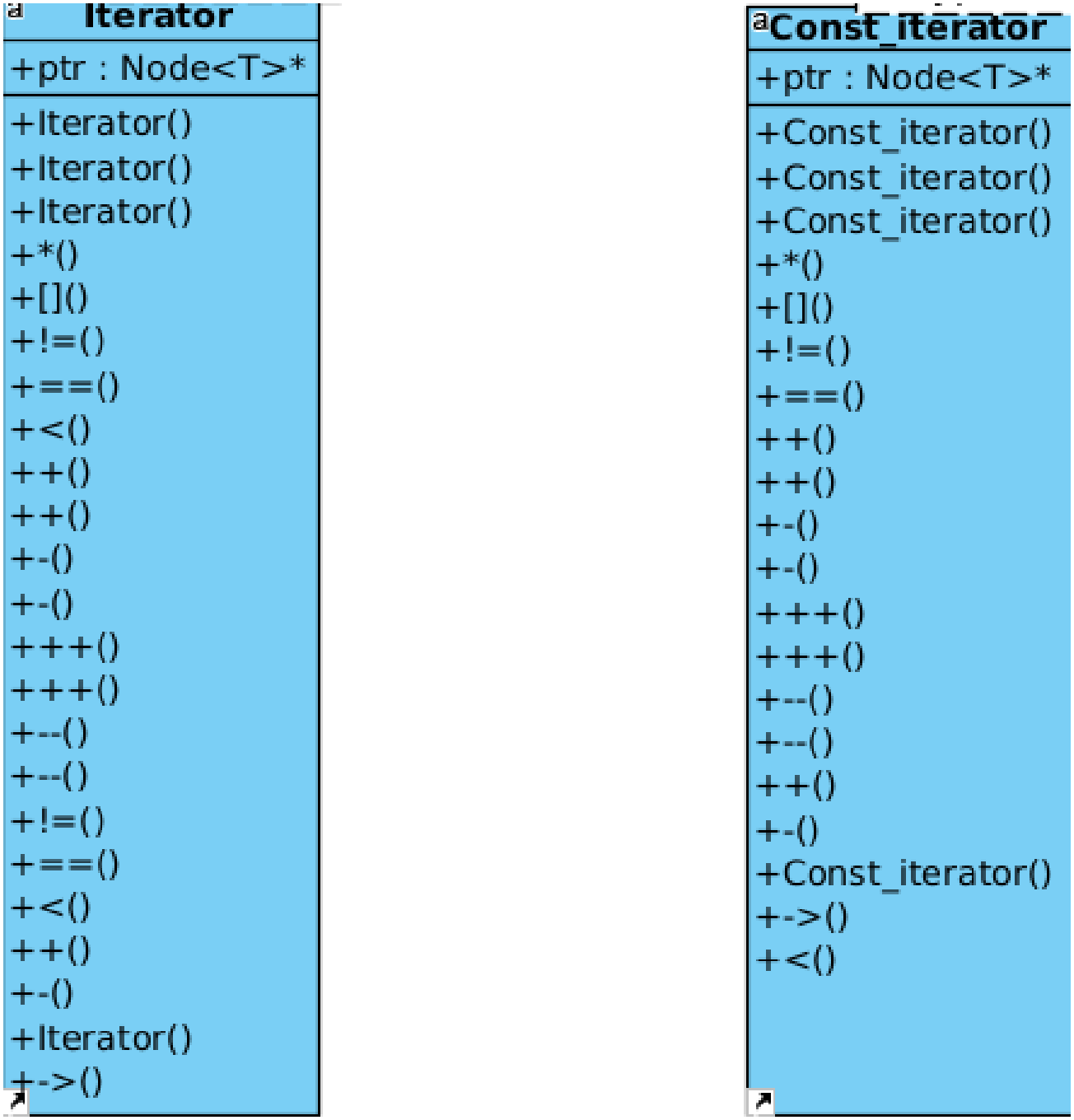

Рис.1. UML диаграмма классов Iterator и Const_iterator

Классы написаны на языке C++. Код классов размещается в публичной части классов шаблонных контейнеров, в файлах с их реализацией. Реализованы базовые конструкторы и перегржены необходимые для гибкой работы с объектами классов различные операторы, такие как оператор сравнения «<», оператор «++» и другие.

2.2 Задача №2

Класс написан на языке C++. Код класса и прототип размещаются в одной единице трансляции – в файле dycont_t.hpp. Тесты в виде проверки результатов и постусловий с помощью утверждений выполнены в единице трансляции main_dycont_t.cpp с функцией main, при этом, файл содержит защиту от повторного включения.

Реализация шаблонного класса включает в себя закрытые поля **_size**, **_capacity** типа size_t и **arr** — указатель на массив, содержащий значения объектов произвольного типа T, где **_size** – количество элементов в массиве, а **_capacity** — реальная выделенная память под массив, для оптимизации количества выделений памяти при работе с массивом. Конструктор по умолчанию задаёт значения **_size** и **_capacity** равными 0. Конструктор общего вида принимает число типа size_t — задающее, размер массива _size и выделяющее количество памяти равное (**_size** * 3) / 2 + 1 — **_capacity**.

Реализованы методы для взаимодействия с массивом — **push_back** и **pop_back**, для вставки и извлечения последнего элемента, **insert** — для добавления элемента в произвольное место массива, метод **erase** для удаления элемента из произвольного места массива, метод **empalce** для замены произвольного элмента. Для доступа к объектам произвольного типа T был перегружен оператор квадратных скобок, помимо этого он был перегружен для работы с разработанными итераторами. Для доступа к закрытым полям **_size** и **_capacity** были реализованы акцессоры. Методы **begin** и **end** перегружены и возвращают **Iterator/Const_iterator** итератор на начало и конец массива соответственно. Метод empty возвращает логическое значение соответствующие пустоте массива.

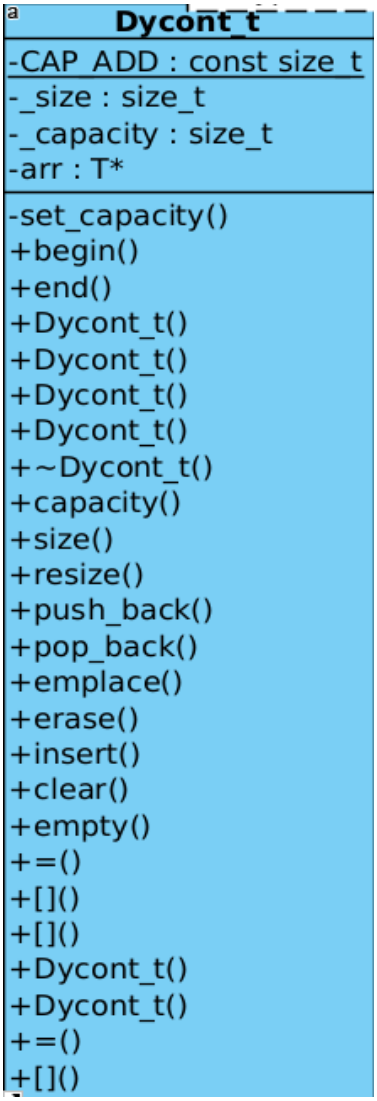

Рис.2 UML диаграмма класса Dycont_t

2.3 Задача №3

Класс написан на языке C++. Коод класса и прототип размещаются в одной единице трансляции – в файле stack_t.hpp. Тесты в виде проверки результатов и постусловий с помощью утверждений выполнены в единице трансляции main_stack_t.cpp с функцией main, при этом, файл содержит защиту от повторного включения.

Реализация шаблонного класса включает в себя закрытые поля **_size**, **_capacity** типа size_t и **arr** — указатель на массив, содержащий значения объектов произвольного типа T, где **_size** – количество элементов в массиве, а **_capacity** — реальная выделенная память под массив, для оптимизации количества выделений памяти при работе с массивом. Конструктор по умолчанию задаёт значения **_size** и **_capacity** равными 0. Конструктор общего вида принимает число типа size_t — задающее, размер массива **_size** и выделяющее количество памяти равное (**_size** * 3) / 2 + 1 — **_capacity**.

Реализованы методы для взаимодействия с массивом — **push_back, push_front, pop_back, pop_front** для добавления элемента в конец/начало массива и удаления элемента в конец/начало массива соответственно., **insert** — для добавления элемента в произвольное место массива, метод **erase** для удаления элемента из произвольного места массива, метод **empalce** для замены произвольного элмента. Для доступа к объектам произвольного типа T был перегружен оператор квадратных скобок, помимо этого он был перегружен для работы с разработанными итераторами. Для доступа к закрытым полям **_size** и **_capacity** были реализованы акцессоры. Методы **begin** и **end** возвращают **Iterator/Const_iterator** на начало и конец массива соответственно. Метод empty возвращает логическое значение соответствующие пустоте массива.

Рис.3 UML диаграмма класса Stack_t

2.4 Задача №4

Класс написан на языке C++. Код класса и прототип размещаются в одной единице трансляции – в файле list_t.hpp. Тесты в виде проверки результатов и постусловий с помощью утверждений выполнены в единице трансляции main_list_t.cpp с функцией main, при этом, файл содержит защиту от повторного включения.

Реализация шаблонного класса включает в себя закрытый класс Node — узел, хранящий адреса следующего узла — **next** и объект произвольного типа T — **data**. Класс List содержит закрытые поля **_size** типа size_t, **first** типа указатель на объект типа Tnode и last — типа указатель на объект типа Tnode, где **_size** – количество узлов в массиве, **fisrt** — указатель на начальный узел и **last** — указатель на конечный узел.

Присутствует конструктор по умолчанию задающий поля класса нулевыми значениями. Конструктор общего вида принимает число типа size_t — задающее, размер массива **_size** (количество узлов).

Реализованы методы для взаимодействия с массивом — **push_back, push_front, pop_back, pop_front** для добавления элемента в конец/начало массива и удаления элемента в конец/начало массива соответственно., **insert** — для добавления узла в произвольное место массива, метод **erase** для удаления узла из произвольного места массива, метод **empalce** для замены значения поля данных произвольного узла. Для доступа к закрытому полю **_size** был реализован соответствующий акцессор. Для доступа к указателям на объекты типа Node был перегружен оператор квадратных скобок, помимо этого, он был перегружен для работы с разработанными итераторами и возврата значения, хранящегося в узле. Метод **empty** возвращает логическое значение соответствующие пустоте массива. Методы **begin** и **end** возвращают **Iterator/Const_iterator** на начальный и конечный узел списка соответственно.

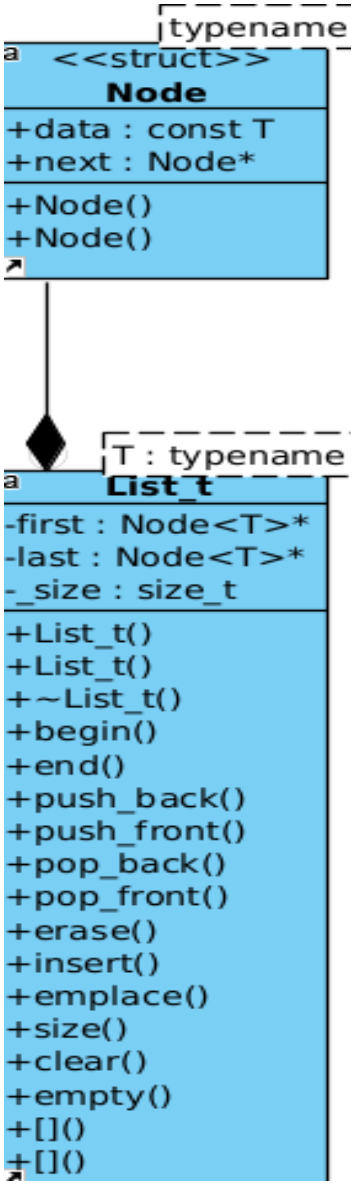

Рис.4 UML диаграмма класса List_t

2.5 Задача №5

Класс написан на языке C++. Код класса и прототип размещаются в одной единице трансляции – в файле queue_t.hpp. Тесты в виде проверки результатов и постусловий с помощью утверждений выполнены в единице трансляции main_queue_t.cpp с функцией main, при этом, файл содержит защиту от повторного включения.

Реализация шаблонного класса включает в себя закрытые поля **_size**, **_quantity** типа size_t и **arr** — указатель на массив, содержащий значения объектов произвольного типа T, где **_size** – количество выделенной памяти под массив, а **_quantity** — количество элементов в массиве. Конструктор по умолчанию отсутствует. Конструктор общего вида принимает число типа size_t — задающее, размер массива _size.

Реализованы методы для взаимодействия с массивом — **push_back** и **pop_back**, для вставки и извлечения последнего элемента, **insert** — для добавления элемента в произвольное место массива, метод **erase** для удаления элемента из произвольного места массива, метод **empalce** для замены произвольного элмента. Реализовано циклическое поведение массиво согласно принципу FIFO — first-in-first-out, при сдвиге элементов, последний элемент становится первым и продолжает сдвиг, элемент, стоящий на позиции до вставляемого уничтожается. Для доступа к объектам произвольного типа T был перегружен оператор квадратных скобок, помимо этого он был перегружен для работы с разработанными итераторами. Для доступа к закрытому полю **_size** был реализован акцессор. Методы **begin** и **end** возвращают **Iterator/Const_iterator** на начало и конец массива соответственно. Метод empty возвращает логическое значение соответствующие пустоте массива.

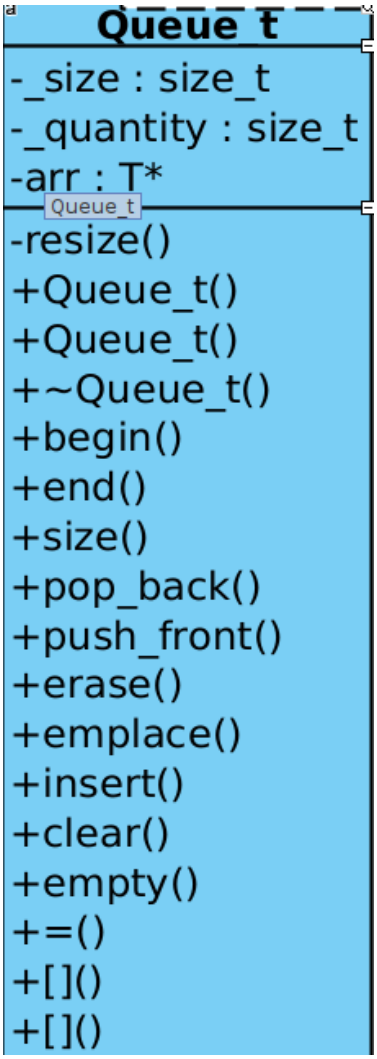

Рис.5 UML диаграмма класса Queue_t

2.6 Задача №6

Класс написан на языке C++. Код класса и прототип размещаются в одной единице трансляции – в файле tree_t.hpp. Тесты в виде проверки результатов и постусловий с помощью утверждений выполнены в единице трансляции main_tree_t.cpp с функцией main, при этом, файл содержит защиту от повторного включения.

Реализация шаблонного класса включает в себя закрытый класс Tnode — лист, хранящий указатели на левый — **left** и правый — **right** лист дерева, родитель — **parent** , объект произвольного типа T — **data** и **pos** — позиция элемента в дереве(ключ). По-мимо этого в класс входит поле tree — указатель на объект типа Tnode.

Контруктор по умолчанию задаёт поля класса нулевыми значениями. Присутствуют закрытые служебные методы, обеспечивающие работоспособность класса.

Для корректного освобождения памяти был создан закрытый метод **delete_tree**, рекурсивно вызывающий сам себя, для прохода по всем листам дерева. Аналогично работает метод **print** для отображения значений, хранящихся в листах дерева.

Для шаблонного класса Tree_t итераторы реализованы не были, в связи возможной неоднозначностью понимания поведения итераторов, при работе с деревом (различные пути обхода дерева).

Реализованы методы для взаимодействия с массивом — **insert** для добавления листа в произвольное место дерева и метод **erase** для удаления листа из произвольного места дерева. Для доступа к объектам произвольного типа T был перегружен оператор квадратных скобок. Для доступа к закрытому полю **_size** был реализован соответствующий акцессор. Для доступа к указателям на объекты типа Node был перегружен оператор квадратных скобок. Метод empty возвращает логическое значение соответствующие пустоте массива.

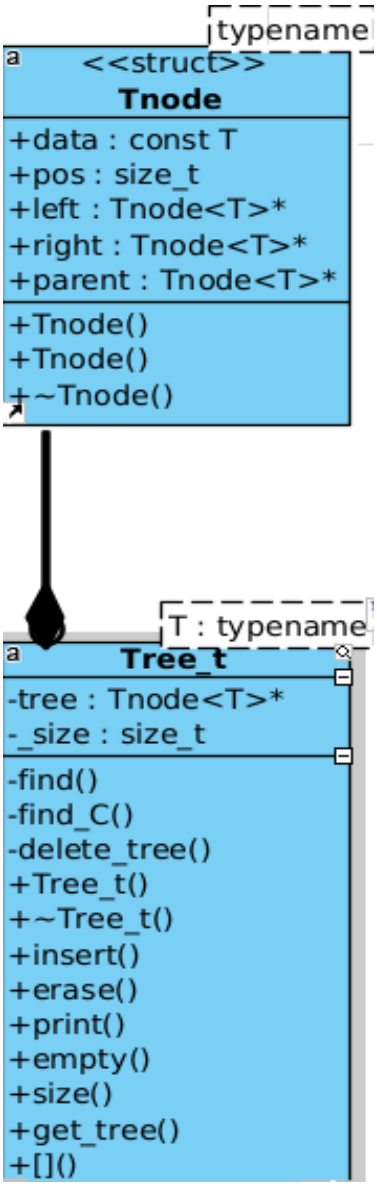

Рис.6 UML диаграмма класса Tree_t

3 Получение исполняемых модулей

Получение исполняемых модулей происходит с помощью системы сборки cmake. Задан стандарт языка C++14 и ключи компиляции -lgtest, для статического подключения библиотеки googletest, минимальная версия cmake 3.14.

Для подключения библеотеки для тестирования — googletest, в файле CmakeList.txt производится загрузка из оффициального источника распространения библиотеки.

Для упрощения тестирования всех исполняемых модулей, был разработан файл main_lab3, содержащий тестирования всех разработанных классов.

Был проведён анализ покрытия кода с помощью инструмента анализирования кода Lcov(gcov).

Листинг 1 – Файл CmakeList.txt

```
cmake_minimum_required(VERSION 3.14 FATAL_ERROR)
project(lab_03)
add_definitions(-lgtest)

set(CMAKE_CXX_STANDARD 14)

include(FetchContent)

FetchContent_Declare(
googletest
GIT_REPOSITORY https://github.com/google/googletest.git
GIT_TAG release-1.8.0
)

FetchContent_MakeAvailable(googletest)

add_executable(${PROJECT_NAME}  "dycont_t.hpp" "stack_t.hpp" list_t.hpp"
                                "queue_t.hpp" tree_t.hpp" "main_lab3.cpp"
                                )
target_link_libraries(${PROJECT_NAME} gtest)
```

Листинг 2 – Результат тестирования на покрытие кода

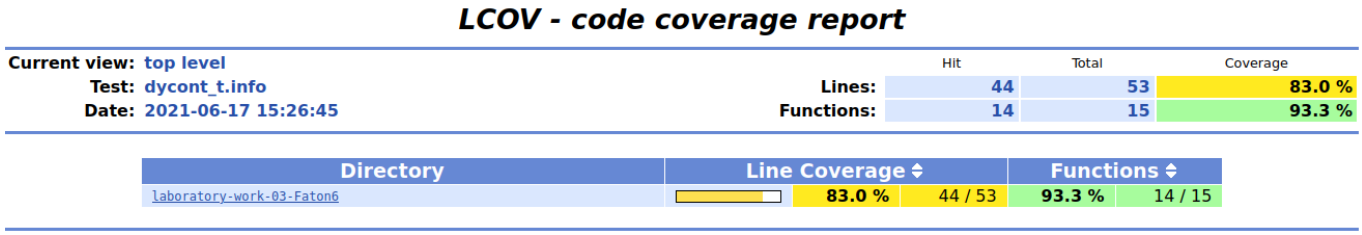

Рис.7 Изображение результата тестирования на покрытие кода

4 Тестирование

Тестирование классов проводилось с использованием макросов библиотеки googletest. Были задействованы макросы EXPECT_EQ — для сравнения арифметических значений, EXPECT_TRUE и EXPECT_FALSE — для логических сравнений.

4.1 Задача №1

Проведён базовый тест на функциональность классов: были созданы объекты класса и проведён перебор элементов различных контейнеров.

4.2 Задача №2

Проведён базовый тест на функциональность класса: был создан объект класса и переданы объекты базового типа int. Были проверены на работоспособность методы push_back, insert, erase и empty.

4.3 Задача №3

Проведён базовый тест на функциональность класса: был создан объект класса и переданы объекты базового типа int. Были проверены на работоспособность методы push_back, push_front, insert и empty.

4.4 Задача №4

Проведён базовый тест на функциональность класса: был создан объект класса и переданы объекты базового типа int. Были проверены на работоспособность методы push_back, push_front, insert и empty.

4.5 Задача №5

Проведён базовый тест на функциональность класса: был создан объект класса и переданы объекты базового типа int. Были проверены на работоспособность методы push_front, insert и empty.

4.6 Задача №6

Проведён базовый тест на функциональность класса: был создан объект класса и переданы объекты базового типа int. Были проверены на работоспособность методы insert и empty.

Приложение А

А – Файл main_lab2.cpp

```
#include <iostream>
#include <string>

#include "dycont_t.hpp"
#include "stack_t.hpp"
#include "list_t.hpp"
#include "queue_t.hpp"
#include "tree_t.hpp"
#include <gtest/gtest.h>

class TestClasses: public ::testing::Test
{
    protected:
        Dycont_t<int> qwa_dycont;
        Stack_t<int> qwa_stack;
        List_t<int> qwa_list;
        Queue_t<int> qwa_queue{6};
        Tree_t<int> qwa_tree;
};

TEST_F(TestClasses, push_back)
{
    qwa_dycont.push_back(111);
    EXPECT_EQ( 111 , qwa_dycont[0] );
    qwa_stack.push_back(111);
    EXPECT_EQ( 111 , qwa_stack[0] );
    qwa_list.push_back(111);
    EXPECT_EQ( 111 , qwa_list[0]->data );
    qwa_tree.insert(0, 111);
    EXPECT_EQ( 111 , qwa_tree[0] );
}
TEST_F(TestClasses, push_front)
{
    qwa_stack.push_back(333);
    EXPECT_EQ( 333 , qwa_stack[0] );

    qwa_list.push_front(333);
    EXPECT_EQ( 333 , qwa_list[0]->data );

    qwa_queue.push_front(333);
    EXPECT_EQ( 333 , qwa_queue[0] );
}
TEST_F(TestClasses, insert)
{
    qwa_dycont.insert(1, 222);
    EXPECT_EQ( 222 , qwa_dycont[0] );
    qwa_stack.insert(1, 222);
```

```
    EXPECT_EQ( 222 , qwa_stack[0] );
    qwa_list.insert(1, 222);
    EXPECT_EQ( 222 , qwa_list[1]->data );
    qwa_queue.insert(1, 222);
    EXPECT_EQ( 222 , qwa_queue[0] );
    qwa_tree.insert(1, 222);
    EXPECT_EQ( 222 , qwa_tree[1] );
}
TEST_F(TestClasses, empty)
{
    EXPECT_TRUE( qwa_dycont.empty() );
    EXPECT_TRUE( qwa_stack.empty() );
    EXPECT_TRUE( qwa_list.empty() );
    EXPECT_TRUE( qwa_queue.empty() );
    EXPECT_TRUE( qwa_tree.empty() );
}


int main(int argc, char **argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    Dycont_t<int> mass(5);
    mass[0] = 0;
    mass[1] = 1;
    mass[2] = 2;
    mass[3] = 3;
    mass[4] = 4;
    mass[5] = 5;
    mass.push_back(6);
    mass.erase(1);

    for (Dycont_t<int>::Iterator<int> it = mass.begin(); it < mass.end(); ++it)
        std::cout << mass[it] << '\n';
    return RUN_ALL_TESTS();
}
```

Приложение Б

Б – Файл dycont_t.hpp

```
#ifndef dycont_t_h
#define dycont_t_h

#include <iostream>
#include <algorithm>
#include <iterator>

#include "ADT.h"
```

```
template <typename T>
class Dycont_t
{


    private:

        const static size_t CAP_ADD = 5;

        size_t _size;
        size_t _capacity;
        T *arr = nullptr;

        void set_capacity(size_t new_size);

    public:
        template <typename>
        class Iterator;
        template <typename>
        class Const_iterator;
        //Dycont_t<T>::Iterator<T> Dycont_t<T>::
        //typename Dycont_t<T>::Iterator;
        Iterator<T> begin();
        Iterator<T> end();
        Const_iterator<T> begin() const;
        Const_iterator<T> end() const;


        Dycont_t();
        Dycont_t(size_t size);
        Dycont_t(std::initializer_list<T> values);
        Dycont_t(const Dycont_t<T> &right);
        ~Dycont_t();

        size_t capacity() const;
        size_t size() const;
        void resize(size_t new_size);
        void push_back(T right);
        void pop_back();
        void emplace(size_t pos, T value);
        void erase(size_t pos);
        void insert(size_t pos, T value);
        void clear();
        bool empty() const;

        const Dycont_t &operator= (const Dycont_t<T> &right);
        T &operator[](size_t pos);
        const T &operator[](Iterator<T> pos) const { return *pos.ptr; }

        template <typename >
        class Iterator : public std::iterator<std::input_iterator_tag, T>
        {
            public:
```

```
                T *ptr;

                Iterator() : ptr(nullptr) {}
                Iterator(T *ptr) : ptr(ptr) {}

                Iterator(const Iterator &right) : ptr(right.ptr) {}

                T &operator*() const { return *ptr; }
                T &operator[](size_t pos) const { return ptr[pos]; }
                //T *operator->() const { return ptr->data; }
                bool operator!=(const Iterator<T> &right) {return ptr != right.ptr; }
                bool operator==(const Iterator<T> &right) {return ptr == right.ptr; }
                bool operator<(const Iterator<T> &right)  {return ptr < right.ptr; }
                Iterator operator+(const Iterator<T> &right) const { return this->ptr
+ right.ptr; }
                Iterator operator+(int right) const { return Iterator(this->ptr +
right); }
                Iterator operator-(const Iterator<T> &right) const { return this->ptr
- right.ptr; }
                Iterator operator-(int right) const { return Iterator(this->ptr -
right); }

                Iterator &operator++() {++ptr; return *this; }
                Iterator operator++(int) {T *var_ptr = this->ptr; ++ptr; return
*var_ptr; }
                Iterator &operator--() {--ptr; return *this; }
                Iterator operator--(int) {T *var_ptr = this->ptr; --ptr; return
*var_ptr; }
        };

        template <typename>
        class Const_iterator : public std::iterator<std::input_iterator_tag, T>
        {
            public:
                T *ptr;

                Const_iterator() : ptr(nullptr) {}
                Const_iterator(T *ptr) : ptr(ptr) {}

                Const_iterator(const Const_iterator &right) : ptr(right.ptr) {}

                const T &operator*() const { return *ptr; }
                const T &operator[](size_t pos) const { return ptr[pos]; }
                //T *operator->() const { return ptr->data; }
                bool operator!=(const Const_iterator &right) const {return ptr !=
right.ptr;}
                bool operator==(const Const_iterator &right) const {return ptr ==
right.ptr;}
                Const_iterator operator+(const Const_iterator<T> &right) const
{ return this->ptr + right.ptr; }
                Const_iterator operator+(int right) const { return
Const_iterator(this->ptr + right); }
                Const_iterator operator-(const Const_iterator<T> &right) const
{ return this->ptr - right.ptr; }
                Const_iterator operator-(int right) const { return
Const_iterator(this->ptr - right); }
                Const_iterator &operator++() {++ptr; return *this; }
```

```
                Const_iterator operator++(int) {T *var_ptr = this->ptr; ++ptr; return
*var_ptr; }
                Const_iterator &operator--() {--ptr; return *this; }
                Const_iterator operator--(int) {T *var_ptr = this->ptr; --ptr; return
*var_ptr; }
        };

};


template <typename T>
void Dycont_t<T>::set_capacity(size_t new_size)
{
    if (new_size) this->_capacity = (new_size * 3) / 2 + 1;
    else this->_capacity = (_size * 3) / 2 + 1;
}

template <typename T>
Dycont_t<T>::Dycont_t() : _size(0), _capacity(0)  { arr = nullptr; }


template <typename T>
Dycont_t<T>::Dycont_t(std::initializer_list<T> values)
{
    _size = values.size();
    set_capacity(_size);
    arr = new T[_capacity];
    for (auto i = values.begin(); i < values.end();++i)
    {
        arr[i] = values[i];
    }
}

template <typename T>
Dycont_t<T>::Dycont_t(size_t size) : _size(size)
{
    if (_size < 0) std::runtime_error( "Error: length < 0");
    set_capacity(_size);
    arr = new T[_capacity];
}

template <typename T>
Dycont_t<T>::Dycont_t(const Dycont_t<T> &right)
: _size(right._size), _capacity(right._capacity)
{
    this->arr = new T[this->_capacity];
    for (size_t i = 0; i < _size; ++i)
    {
        arr[i] = right.arr[i];
    }
}
template <typename T>
Dycont_t<T>::~Dycont_t()
{
    if (arr != nullptr)
```

```
    {
        delete [] arr;
        arr = nullptr;
    }
}

template <typename T>
size_t Dycont_t<T>::capacity() const { return this->_capacity; }
template <typename T>
size_t Dycont_t<T>::size() const { return this->_size; }

template <typename T>
void Dycont_t<T>::resize(size_t new_size)
{
    //this->_size = new_size;
    set_capacity(new_size);
    T *new_arr = new T[this->_capacity];
    for (size_t i = 0; i < _size; ++i ) new_arr[i] = arr[i];
    delete [] arr;
    arr = new_arr;
}

template <typename T>
void Dycont_t<T>::push_back(T value)
{
    if (_capacity == _size) resize(_size + 1);
    if (_size == 0 ) { arr[0] = value; ++_size; }
    else arr[_size++] = value;
}

template <typename T>
void Dycont_t<T>::pop_back()
{
    if (arr != nullptr && _size > 0) --_size;
}
template <typename T>
void Dycont_t<T>::emplace(size_t pos, T value)
{
    arr[pos] = value;
}

template <typename T>
void Dycont_t<T>::insert(size_t pos, T value)
{

    ++_size;
    if (_capacity == _size)
    {
        set_capacity(_size);
        T *new_arr = new T[this->_capacity];
        for (size_t i = 0; i < pos; ++i ) new_arr[i] = arr[i];
        for (size_t i = pos+1; i < _size; ++i ) new_arr[i] = arr[i-1];
        new_arr[pos] = value;
        delete [] arr;
```

```
        arr = new_arr;
    }
    else if (pos == _size)
    {
        --_size;
        return this->push_back(value);
    }
    else
    {
        T var = arr[pos];
        arr[pos] = value;
        T qwa;
        for (size_t i = pos+1; i < _size; ++i )
        {
            if (i < _size) qwa = arr[i];
            arr[i] = var;
            var = qwa;
        }
    }
}

template <typename T>
void Dycont_t<T>::erase(size_t pos)
{
    if (_size >= pos)
    {
        --_size;
        for (size_t i = pos; i < _size; ++i) arr[i] = arr[i+1];
    }
    else return;
}

template <typename T>
void Dycont_t<T>::clear()
{
    delete [] arr;
    arr = nullptr;
    _size = 0;
    _capacity = 0;
}
template <typename T>
bool Dycont_t<T>::empty() const { return _size == 0; }

//const Dycont_t &Dycont_t::operator= (const ADT &right)
template <typename T>
const Dycont_t<T> &Dycont_t<T>::operator= (const Dycont_t<T>& right)
{
    if (_capacity != right._capacity)
    {
        this->_size = right._size;
        if (arr != nullptr)
        {
            delete [] arr;
            arr = nullptr;
```

```
        }
        this->_capacity = right._capacity;
        this->arr = new T[right._capacity];
    }
    for ( Iterator<T> it = begin(); it < right.end(); ++it )
        arr[it] = right.arr[it];
    return *this;
}

template <typename T>
T &Dycont_t<T>::operator[](size_t pos) { return arr[pos]; }

template <typename T>
Dycont_t<T>::Iterator<T> Dycont_t<T>::begin()
{
    return Iterator<T>(arr);
}
template <typename T>
Dycont_t<T>::Iterator<T> Dycont_t<T>::end()
{
    return Iterator<T>(arr + this->_size);
}
template <typename T>
Dycont_t<T>::Const_iterator<T> Dycont_t<T>::begin() const
{
    return Const_iterator<T>(arr[0]);
}
template <typename T>
Dycont_t<T>::Const_iterator<T> Dycont_t<T>::end() const
{
    return Const_iterator<T>(arr[this->_size]);
}
#endif // dycont_t_h
```

Приложение В

В – Файл stack_t.hpp

```
#ifndef stack_t_h
#define stack_t_h

#include <iostream>
#include <algorithm>
#include <iterator>

#include "ADT.h"



template <typename T>
class Stack_t
```

```
{

    private:

        const static size_t CAP_ADD = 5;

        size_t _size;
        size_t _capacity;
        T *arr = nullptr;

        void set_capacity(size_t new_size);

    public:

        template <typename>
        class Iterator;
        template <typename>
        class Const_iterator;

        Iterator<T> begin();
        Iterator<T> end();
        Const_iterator<T> begin() const;
        Const_iterator<T> end() const;


        Stack_t();
        Stack_t(size_t size);
        Stack_t(std::initializer_list<T> values);
        Stack_t(const Stack_t<T> &right);
        ~Stack_t();

        size_t capacity() const;
        size_t size() const;
        void resize(size_t new_size);
        void push_back(T right);
        void pop_back();
        void push_front(T right);
        void pop_front();
        void emplace(size_t pos, T value);
        void erase(size_t pos);
        void insert(size_t pos, T value);
        void clear();
        bool empty() const;

        const Stack_t &operator= (const Stack_t<T> &right);
        T &operator[](size_t pos);
        const T &operator[](Iterator<T> pos) const { return *pos.ptr; }

        template <typename>
        class Iterator : public std::iterator<std::input_iterator_tag, T>
        {
            public:
                T *ptr;
```

```
                Iterator() : ptr(nullptr) {}
                Iterator(T *ptr) : ptr(ptr) {}

                Iterator(const Iterator &right) : ptr(right.ptr) {}

                T &operator*() const { return *ptr; }
                T &operator[](size_t pos) const { return ptr[pos]; }
                bool operator!=(const Iterator<T> &right) {return ptr != right.ptr; }
                bool operator==(const Iterator<T> &right) {return ptr == right.ptr; }
                bool operator<(const Iterator<T> &right)  {return ptr < right.ptr; }
                Iterator operator+(const Iterator<T> &right) const { return this->ptr
+ right.ptr; }
                Iterator operator+(int right) const { return Iterator(this->ptr +
right); }
                Iterator operator-(const Iterator<T> &right) const { return this->ptr
- right.ptr; }
                Iterator operator-(int right) const { return Iterator(this->ptr -
right); }

                Iterator &operator++() {++ptr; return *this; }
                Iterator operator++(int) {T *var_ptr = this->ptr; ++ptr; return
*var_ptr; }
                Iterator &operator--() {--ptr; return *this; }
                Iterator operator--(int) {T *var_ptr = this->ptr; --ptr; return
*var_ptr; }
        };

        template <typename>
        class Const_iterator : public std::iterator<std::input_iterator_tag, T>
        {
            public:
                T *ptr;

                Const_iterator() : ptr(nullptr) {}
                Const_iterator(T *ptr) : ptr(ptr) {}

                Const_iterator(const Const_iterator &right) : ptr(right.ptr) {}

                const T &operator*() const { return *ptr; }
                const T &operator[](size_t pos) const { return ptr[pos]; }
                bool operator!=(const Const_iterator &right) const {return ptr !=
right.ptr;}
                bool operator==(const Const_iterator &right) const {return ptr ==
right.ptr;}
                Const_iterator operator+(const Const_iterator<T> &right) const
{ return this->ptr + right.ptr; }
                Const_iterator operator+(int right) const { return
Const_iterator(this->ptr + right); }
                Const_iterator operator-(const Const_iterator<T> &right) const
{ return this->ptr - right.ptr; }
                Const_iterator operator-(int right) const { return
Const_iterator(this->ptr - right); }
                Const_iterator &operator++() {++ptr; return *this; }
                Const_iterator operator++(int) {T *var_ptr = this->ptr; ++ptr; return
*var_ptr; }
                Const_iterator &operator--() {--ptr; return *this; }
```

```
                Const_iterator operator--(int) {T *var_ptr = this->ptr; --ptr; return
*var_ptr; }
        };
};


template <typename T>
void Stack_t<T>::set_capacity(size_t new_size)
{
    if (new_size) this->_capacity = (new_size * 3) / 2 + 1;
    else this->_capacity = (_size * 3) / 2 + 1;
}

template <typename T>
Stack_t<T>::Stack_t() : _size(0), _capacity(0)  { arr = nullptr; }


template <typename T>
Stack_t<T>::Stack_t(std::initializer_list<T> values)
{
    _size = values.size();
    set_capacity(_size);
    arr = new T[_capacity];
    for (auto i = values.begin(); i < values.end();++i)
    {
        arr[i] = values[i];
    }
}

template <typename T>
Stack_t<T>::Stack_t(size_t size) : _size(size)
{
    if (_size < 0) std::runtime_error( "Error: length < 0");
    set_capacity(_size);
    arr = new T[_capacity];
}

template <typename T>
Stack_t<T>::Stack_t(const Stack_t<T> &right)
: _size(right._size), _capacity(right._capacity)
{
    this->arr = new T[this->_capacity];
    for (size_t i = 0; i < _size; ++i)
    {
        arr[i] = right.arr[i];
    }
}

template <typename T>
Stack_t<T>::~Stack_t()
{
    if (arr != nullptr)
    {
        delete [] arr;
```

```
        arr = nullptr;
    }
}


template <typename T>
size_t Stack_t<T>::capacity() const { return this->_capacity; }
template <typename T>
size_t Stack_t<T>::size() const { return this->_size; }

template <typename T>
void Stack_t<T>::resize(size_t new_size)
{
    set_capacity(new_size);
    T *new_arr = new T[this->_capacity];
    for (size_t i = 0; i < _size; ++i ) new_arr[i] = arr[i];
    delete [] arr;
    arr = new_arr;
}

template <typename T>
void Stack_t<T>::push_back(T value)
{
    if (_capacity == _size) resize(_size + 1);
    if (_size == 0 ) { arr[0] = value; ++_size; }
    else arr[_size++] = value;
}

template <typename T>
void Stack_t<T>::pop_back()
{
    if (arr != nullptr && _size > 0) --_size;
}
template <typename T>
void Stack_t<T>::push_front(T right)
{
    this->insert(0, right);
}
template <typename T>
void Stack_t<T>::pop_front()
{
    this->erase(0);
}

template <typename T>
void Stack_t<T>::emplace(size_t pos, T value)
{
    arr[pos] = value;
}

template <typename T>
void Stack_t<T>::insert(size_t pos, T value)
{
    ++_size;
```

```
    if (_capacity == _size)
    {
        set_capacity(_size);
        T *new_arr = new T[this->_capacity];
        for (size_t i = 0; i < pos; ++i ) new_arr[i] = arr[i];
        for (size_t i = pos+1; i < _size; ++i ) new_arr[i] = arr[i-1];
        new_arr[pos] = value;
        delete [] arr;
        arr = new_arr;
    }
    else if (pos == _size)
    {
        --_size;
        return this->push_back(value);
    }
    else
    {
        T var(arr[pos]);
        arr[pos] = value;
        T qwa;
        for (size_t i = pos; i < _size; ++i )
        {
            if (i < _size) qwa = arr[i];
            arr[i] = var;
            var = qwa;
        }
    }
}

template <typename T>
void Stack_t<T>::erase(size_t pos)
{
    --_size;
    for (size_t i = pos; i < _size; ++i) arr[i] = arr[i+1];
}

template <typename T>
void Stack_t<T>::clear()
{
    delete [] arr;
    arr = nullptr;
    _size = 0;
    _capacity = 0;
}
template <typename T>
bool Stack_t<T>::empty() const { return _size == 0; }

template <typename T>
const Stack_t<T> &Stack_t<T>::operator= (const Stack_t<T>& right)
{
    if (_capacity != right._capacity)
    {
        this->_size = right._size;
        if (arr != nullptr)
```

```
        {
            delete [] arr;
            arr = nullptr;
        }
        this->_capacity = right._capacity;
        this->arr = new T[right._capacity];
    }
    for ( Iterator<T> it = begin(); it < right.end(); ++it )
        arr[it] = right.arr[it];
    return *this;
}

template <typename T>
T &Stack_t<T>::operator[](size_t pos) { return arr[pos]; }

template <typename T>
Stack_t<T>::Iterator<T> Stack_t<T>::begin()
{
    return Iterator<T>(arr);
}
template <typename T>
Stack_t<T>::Iterator<T> Stack_t<T>::end()
{
    return Iterator<T>(arr + this->_size);
}

template <typename T>
Stack_t<T>::Const_iterator<T> Stack_t<T>::begin() const
{
    return Const_iterator<T>(arr[0]);
}
template <typename T>
Stack_t<T>::Const_iterator<T> Stack_t<T>::end() const
{
    return Const_iterator<T>(arr[this->_size]);
}

#endif // stack_t_h
```

Приложение Г

Г – Файл list_t.hpp

```
#ifndef list_t_h
#define list_t_h

#include <iostream>
#include <algorithm>
#include <iterator>
```

```
#include "ADT.h"

template <typename T>
class List_t
{

    private:
        // Структура узла односвязного списка
        template <typename>
        struct Node
        {
            // Значение узла
            const T data;

            // Указатель на следующий узел
            Node *next;

            Node() : next(nullptr){}

            Node(const T obj ) : data(obj), next(nullptr) {}

        };

        Node<T> *first;
        Node<T> *last;
        size_t _size;

    public:

        template <typename>
        class Iterator;
        template <typename>
        class Const_iterator;

        List_t() : first(nullptr), last(nullptr), _size(0) {}
        List_t(size_t size); // наверное лучше убрать
        ~List_t();

        Iterator<T> begin();
        Iterator<T> end();
        Const_iterator<T> begin() const;
        Const_iterator<T> end() const;

        void push_back(const T right);
        void push_front(const T right);
        void pop_back();
        void pop_front();

        void erase(size_t pos);
        void insert(size_t pos, const T value);
        void emplace(size_t pos, const T obj);

        size_t size() const;
```

```
        void clear();
        bool empty() const;

        Node<T>* &operator[](size_t pos);
        const Node<T>* operator[](size_t pos) const;
        const T &operator[](Iterator<T> pos) const { return pos.ptr->data; }

        template <typename>
        class Iterator : public std::iterator<std::input_iterator_tag, T>
        {
            public:

                Node<T> *ptr;

                Iterator() : ptr(nullptr) {}
                Iterator(Node<T> *ptr) : ptr(ptr) {}

                Iterator(const Iterator &right) : ptr(right.ptr) {}

                T &operator*() const { return ptr->data; }
                T &operator[](size_t pos) const { return ptr[pos]->data; }
                T *operator->() const { return ptr->data; }
                bool operator!=(const Iterator<T> &right) {return ptr != right.ptr; }
                bool operator==(const Iterator<T> &right) {return ptr == right.ptr; }
                bool operator<(const Iterator<T> &right)  {return ptr < right.ptr; }

                Iterator &operator++() { if (ptr != nullptr) ptr = ptr->next; return
*this; }
                Iterator operator++(int) {T *var_ptr = this->ptr; ptr = ptr->next;
return *var_ptr; }
                Iterator &operator--() {--ptr; return *this; }
                Iterator operator--(int) {T *var_ptr = this->ptr; --ptr; return
*var_ptr; }
        };

        template <typename>
        class Const_iterator : public std::iterator<std::input_iterator_tag, T>
        {
            public:
                Node<T> *ptr;

                Const_iterator() : ptr(nullptr) {}
                Const_iterator(Node<T> *ptr) : ptr(ptr) {}

                Const_iterator(const Const_iterator &right) : ptr(right.ptr) {}

                const T &operator*() const { return ptr->data; }
                const T &operator[](size_t pos) const { return ptr[pos]->data; }
                const T *operator->() const { return ptr->data; }
                bool operator!=(const Const_iterator &right) const {return ptr !=
right.ptr;}
                bool operator==(const Const_iterator &right) const {return ptr ==
right.ptr;}
                bool operator<(const Iterator<T> &right)  {return ptr < right.ptr; }
```

```
                Const_iterator &operator++() {if (ptr != nullptr) ptr = ptr->next;
return *this; }
                Const_iterator operator++(int) {T *var_ptr = this->ptr; ++ptr; return
*var_ptr; }
                Const_iterator &operator--() {--ptr; return *this; }
                Const_iterator operator--(int) {T *var_ptr = this->ptr; --ptr; return
*var_ptr; }
        };

};

template <typename T>
List_t<T>::List_t(size_t size)
{
    Node<T> *var = new Node<T>;
    first = var;
    last = var;
    for (size_t i = 1; i < size-1; ++i)
    {
        Node<T> *next = new Node<T>;
        last->next = next;
        last = next;
    }
    _size = size;
}

template <typename T>
List_t<T>::~List_t()
{
    if (empty()) return;
    Node<T> *var = first;
    if (var) return;
    else {
        while (var->next != last)
        {
            delete var;
            var = var->next;
            if (!var) return;
        }
        delete var;
    }
}


template <typename T>
void List_t<T>::push_back(const T obj)
{
    Node<T> *var = new Node<T>(obj);
    if (empty())
    {
        first = var;
        last = var;
        ++_size;
        return;
    }
```

```
    last->next = var;
    last = var;
    ++_size;
}

template <typename T>
void List_t<T>::push_front(const T obj)
{
    Node<T> *var = new Node<T>(obj);
    var->next = first;
    first = var;
    ++_size;
}


template <typename T>
void List_t<T>::pop_front()
{
    if (empty()) return;
    --_size;
    Node<T> *val = first;
    first = first->next;
    delete val;
}

template <typename T>
void List_t<T>::pop_back()
{
    if (empty()) return;
    if (first == last)
    {
        pop_front();
        return;
    }
    Node<T>* val = first;
    while (val->next != last) val = val->next;
    val->next = nullptr;
    delete last;
    last = val;
    --_size;
}


template <typename T>
void List_t<T>::erase(size_t pos)
{
    if (pos == 1) return pop_front();
    if (empty()) return;
    --_size;
    Node<T> *var = first;
    for (size_t i = 1; i < pos-1; ++i)
    {
        var = var->next;
        if (!var) return;
```

```
    }

    Node<T> *temp = var->next;
    var->next = temp->next;

}

template <typename T>
void List_t<T>::insert(size_t pos, const T value)
{
    if (pos > _size) std::runtime_error("Position more size");
    if (empty()) return push_back(value);
    ++_size;
    Node<T> *var = first;
    for (size_t i = 1; i < pos-1; ++i)
    {
        var = var->next;
        if (!var) return;
    }
    Node<T> *temp = var->next;
    Node<T> *qwa = new Node<T>(value);
    var->next = qwa;
    qwa->next = temp;
}

template <typename T>
void List_t<T>::emplace(size_t pos, const T obj)
{
    if (empty()) return;
    if (pos > _size) std::runtime_error("Position more size");
    Node<T> *var = first;
    for (size_t i = 0; i < pos-1; ++i)
    {
        var = var->next;
        if (!var) return;
    }
    var->data = obj;
}


template <typename T>
size_t List_t<T>::size() const { return _size; }

template <typename T>
void List_t<T>::clear()
{
    if (empty()) return;
    Node<T> *var = first;
    if (var) return;
    else {
        while (var->next != last)
        {
            delete var;
            var = var->next;
```

```
            if (!var) return;
        }
        delete var;
    }
    _size = 0;
}

template <typename T>
bool List_t<T>::empty() const { return _size == 0; }


template <typename T>
List_t<T>::Node<T>* &List_t<T>::operator[](size_t pos)
{
    Node<T>* var = this->first;
    for (size_t i = 0; i < pos; ++i)
    {
        if (var->next != nullptr) var = var->next;
    }
    Node<T>* &ref = var;
    return ref;
}

template <typename T>
List_t<T>::Iterator<T> List_t<T>::begin()
{
    return Iterator<T>(first);
}
template <typename T>
List_t<T>::Iterator<T> List_t<T>::end()
{
    return Iterator<T>(last);
}

template <typename T>
List_t<T>::Const_iterator<T> List_t<T>::begin() const
{
    return Const_iterator<T>(first);
}
template <typename T>
List_t<T>::Const_iterator<T> List_t<T>::end() const
{
    return Const_iterator<T>(last);
}


#endif // list_t_h
```

Приложение Д

Д – Файл queue_t.hpp

```
#ifndef Queue_t_h
#define Queue_t_h

#include "ADT.h"


template <typename T>
class Queue_t
{
    private:

        size_t _size;    // размер очереди
        size_t _quantity;// количество элементов
        T* arr;

        void resize(size_t new_size);

    public:

        template <typename>
        class Iterator;
        template <typename>
        class Const_iterator;

        Queue_t() = delete;
        Queue_t(size_t size);
        Queue_t(const Queue_t &right);
        ~Queue_t();

        Iterator<T> begin();
        Iterator<T> end();
        Const_iterator<T> begin() const;
        Const_iterator<T> end() const;
        size_t size() const;
        T pop_back();
        void push_front(T value);
        void erase(size_t pos);
        void emplace(size_t, T value);
        void insert(size_t pos, T value);
        void clear();
        bool empty() const;

        const Queue_t &operator= (const Queue_t &right);
        T &operator[](size_t pos);
        const T &operator[](Iterator<T> pos) const { return *pos.ptr; }

        template <typename>
        class Iterator : public std::iterator<std::input_iterator_tag, T>
        {
            public:
```

```
                T *ptr;

                Iterator() : ptr(nullptr) {}
                Iterator(T *ptr) : ptr(ptr) {}

                Iterator(const Iterator &right) : ptr(right.ptr) {}

                T &operator*() const { return *ptr; }
                T &operator[](size_t pos) const { return ptr[pos]; }
                //T *operator->() const { return ptr->data; }
                bool operator!=(const Iterator<T> &right) {return ptr != right.ptr; }
                bool operator==(const Iterator<T> &right) {return ptr == right.ptr; }
                bool operator<(const Iterator<T> &right)  {return ptr < right.ptr; }
                Iterator operator+(const Iterator<T> &right) const { return this->ptr
+ right.ptr; }
                Iterator operator+(int right) const { return Iterator(this->ptr +
right); }
                Iterator operator-(const Iterator<T> &right) const { return this->ptr
- right.ptr; }
                Iterator operator-(int right) const { return Iterator(this->ptr -
right); }

                Iterator &operator++() {++ptr; return *this; }
                Iterator operator++(int) {T *var_ptr = this->ptr; ++ptr; return
*var_ptr; }
                Iterator &operator--() {--ptr; return *this; }
                Iterator operator--(int) {T *var_ptr = this->ptr; --ptr; return
*var_ptr; }
        };

        template <typename>
        class Const_iterator : public std::iterator<std::input_iterator_tag, T>
        {
            public:
                T *ptr;

                Const_iterator() : ptr(nullptr) {}
                Const_iterator(T *ptr) : ptr(ptr) {}

                Const_iterator(const Const_iterator &right) : ptr(right.ptr) {}

                const T &operator*() const { return *ptr; }
                const T &operator[](size_t pos) const { return ptr[pos]; }
                //T *operator->() const { return ptr->data; }
                bool operator!=(const Const_iterator &right) const {return ptr !=
right.ptr;}
                bool operator==(const Const_iterator &right) const {return ptr ==
right.ptr;}
                Const_iterator operator+(const Const_iterator<T> &right) const
{ return this->ptr + right.ptr; }
                Const_iterator operator+(int right) const { return
Const_iterator(this->ptr + right); }
                Const_iterator operator-(const Const_iterator<T> &right) const
{ return this->ptr - right.ptr; }
                Const_iterator operator-(int right) const { return
Const_iterator(this->ptr - right); }
                Const_iterator &operator++() {++ptr; return *this; }
```

```
                Const_iterator operator++(int) {T *var_ptr = this->ptr; ++ptr; return
*var_ptr; }
                Const_iterator &operator--() {--ptr; return *this; }
                Const_iterator operator--(int) {T *var_ptr = this->ptr; --ptr; return
*var_ptr; }
        };

};

template <typename T>
Queue_t<T>::Queue_t(size_t size) : _size(size)
{
    arr = new T[_size];
    _quantity = 0;
}

template <typename T>
Queue_t<T>::Queue_t(const Queue_t<T> &right)
{
    if (arr != nullptr)
    {
        delete [] arr;
        arr = nullptr;
    }

    this->_size = right._size;
    this->_quantity = right._quantity;
    this->arr = new T[this->_size];
    for (size_t i = 0; i < _quantity; ++i)
    {
        arr[i] = right.arr[i];
    }

}

template <typename T>
Queue_t<T>::~Queue_t()
{
    if (arr != nullptr)
    {
        delete [] arr;
        arr = nullptr;
    }

}

template <typename T>
size_t Queue_t<T>::size() const { return this->_quantity; }


template <typename T>
void Queue_t<T>::resize(size_t new_size)
{
    T *new_arr;
```

```
    new_arr = new T[new_size];
    for (size_t i = 0; i < this->_quantity; ++i )
    {
        new_arr[i] = arr[i];
    }

    delete [] arr;
    arr = new_arr;
    this->_size = new_size;

}


template <typename T>
T Queue_t<T>::pop_back()
{
    if (arr != nullptr && _quantity > 0)
    {
        T val = arr[_quantity-1];
        --_quantity;
        return val;
    }
    else return nullptr;
}

template <typename T>
void Queue_t<T>::push_front(T value)
{
    this->insert(0, value);
}

template <typename T>
void Queue_t<T>::emplace(size_t pos, T value)
{
    arr[pos] = value;
}

template <typename T>
void Queue_t<T>::insert(size_t pos, T value)
{
    if (empty())
    {
        arr[0] = value;
        ++_quantity;
    }
    else if (_quantity < _size)
    {
        ++_quantity;
        T var = arr[pos];
        arr[pos] = value;
        T qwa;
        for (size_t i = pos+1; i < _quantity; ++i)
        {
            qwa = arr[i];
```

```
            arr[i] = var;
            var = qwa;
        }
    }
    else if (_quantity == _size)
    {

        for (size_t i = 0; i < pos; ++i)
            arr[i+1] = arr[i];
        arr[0] = arr[_quantity-1];
        T var = arr[pos];
        arr[pos] = value;
        T qwa;
        for (size_t i = pos+1; i < _quantity; ++i)
        {
            qwa = arr[i];
            arr[i] = var;
            var = qwa;
        }
    }
}


template <typename T>
void Queue_t<T>::erase(size_t pos)
{
    --_quantity;
    for (size_t i = pos; i < _quantity; ++i)
        arr[i] = arr[i+1];
}

template <typename T>
void Queue_t<T>::clear()
{
    delete [] arr;
    arr = nullptr;
    _size = 0;
    _quantity = 0;
}

template <typename T>
bool Queue_t<T>::empty() const { return _quantity == 0; }

template <typename T>
const Queue_t<T> &Queue_t<T>::operator= (const Queue_t<T>& right)
{
    if (_size != right._size)
    {
        this->_quantity = right._quantity;
        if (arr != nullptr)
        {
            delete [] arr;
            arr = nullptr;
        }
```

```
        this->_size = right._size;
        this->arr = new T[right._size];
    }
    for ( size_t i = 0; i < right._quantity; ++i )
        arr[i] = right.arr[i];
    return *this;
}

template <typename T>
T &Queue_t<T>::operator[](size_t pos) { return arr[pos]; }


template <typename T>
Queue_t<T>::Iterator<T> Queue_t<T>::begin()
{
    return Iterator<T>(arr);
}
template <typename T>
Queue_t<T>::Iterator<T> Queue_t<T>::end()
{
    return Iterator<T>(arr + this->_size);
}

template <typename T>
Queue_t<T>::Const_iterator<T> Queue_t<T>::begin() const
{
    return Const_iterator<T>(arr[0]);
}
template <typename T>
Queue_t<T>::Const_iterator<T> Queue_t<T>::end() const
{
    return Const_iterator<T>(arr[this->_size]);
}



#endif // Queue_t_h
```

Приложение E

E – Файл tree_t.hpp

```
#ifndef tree_t_h
#define tree_t_h

#include "ADT.h"



template <typename T>
```

```
class Tree_t
{
    private:
    //public:
        template <typename>
        struct Tnode
        {
                const T data;    // поле данных
                size_t pos;         // префиксная позиция в дереве (иначе говоря
ключ)
                Tnode<T> *left;        // левый потомок
                Tnode<T> *right;       // правый потомок
                Tnode<T> *parent;      // родитель
                Tnode() : data(0), pos(0), left(nullptr), right(nullptr) {}
                Tnode(const T data, const size_t pos = 0)
                {
                    this->data = data;
                    this->pos = pos;
                    this->left = nullptr;
                    this->right = nullptr;
                }
                ~Tnode() = default;
        };

        Tnode<T> *find(Tnode<T> *tree, const size_t pos) const; // находит элемент, с
заданной позицией
        Tnode<T> *find_C(const size_t pos) const; // находит элемент с ключем,
следующим за данным числом
        void delete_tree(Tnode<T>* tree); // очистка памяти
        Tnode<T> *tree;
        size_t _size;

    public:

        Tree_t();
        ~Tree_t();
        void insert(const size_t pos, const T data);
        void erase(const size_t pos);
        void print(Tnode<T> *tree, std::ostream &out) const;
        const bool empty() const;
        const size_t size() const;
        Tnode<T> *get_tree();
        const T &operator[](const size_t pos) const;

};
template <typename T>
Tree_t<T>::Tree_t() : tree(0), _size(0) {}

template <typename T>
Tree_t<T>::~Tree_t()
{
    if (tree != nullptr) delete_tree(this->tree);
}

template <typename T>
```

```
void Tree_t<T>::insert(const size_t pos, const T data)
{
    Tnode<T> *var = new Tnode<T>(data, pos);
    Tnode<T> *p1;
    Tnode<T> *p2;
    p1 = tree;
    do
    {
        p2 = p1;
        if (p1 != nullptr && pos % 2 == 1) p1 = p1->left;
        else if (p1 != nullptr) p1 = p1->right;
    }while (p1 != nullptr);

    var->parent = p2;
    if (p2 == nullptr) tree = var;
    else
    {
        if ( pos %2==1  ) p2->left = var;
        else p2->right = var;
    }
    ++_size;
}

template <typename T>
void Tree_t<T>::erase(const size_t pos)
{
    Tnode<T> *p1;
    Tnode<T> *p2;
    Tnode<T> *var = this->find(this->tree, pos);
    if (var->left == nullptr || var->right == nullptr) p1 = var;
    else p1 = find_C(var->pos);

    if (p1->left != nullptr) p2 = p1->left;
    else p2 = p1->right;

    if (p2 != nullptr) p2->parent = p1->parent;

    if (p1->parent == nullptr) tree = p2;
    else
    {
        if (p1 == (p1->parent)->left) (p1->parent)->left = p2;
        else (p1->parent)->right = p2;
    }
    --_size;
}

template <typename T>
const bool Tree_t<T>::empty() const
{
    return this->_size == 0;
}

template <typename T>
const size_t Tree_t<T>::size() const
```

```
{
    return _size;
}

template <typename T>
Tree_t<T>::Tnode<T> *Tree_t<T>::get_tree()
{
    return tree;
}

template <typename T>
void Tree_t<T>::print(Tnode<T> *tree, std::ostream &out) const
{
    if (tree != nullptr)
    {
        print(tree->left, out);
        out << *(tree->data) << std::endl;
        print(tree->right, out);
    }
}

template <typename T>
const T &Tree_t<T>::operator[](const size_t pos) const
{
    Tnode<T> *var = find(this->tree, pos);
    if (var == nullptr)
    {
        std::runtime_error("Not real");
        exit(1);
    }
    return var->data;
}

template <typename T>
Tree_t<T>::Tnode<T> *Tree_t<T>::find(Tnode<T> *tree, const size_t pos) const
{
    if (tree == nullptr || pos == tree->pos) return tree;
    if (pos % 2 == 0) return find(tree->right, pos);
    else return find(tree->left, pos);
}

template <typename T>
Tree_t<T>::Tnode<T> *Tree_t<T>::find_C(const size_t pos) const
{
    Tnode<T> *p1 = find(tree, pos);
    Tnode<T> *p2;
    if ( p1 == nullptr ) return nullptr;
    if (p1->right != nullptr)
    {
        while (p1->left != nullptr)
        {
            p1 = p1->left;
        }
        return p1;
```

```
    }
    p2 = p1->parent;
    while (p2 != nullptr && p1 == p2->right)
    {
        p1 = p2;
        p2 = p2->parent;
    }
    return p2;

}

template <typename T>
void Tree_t<T>::delete_tree(Tnode<T>* tree)
{
    if (tree != nullptr)
    {
        delete_tree(tree->left);
        delete tree;
        delete_tree(tree->right);
    }
    tree = nullptr;
}


#endif // tree_t_h
```