

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Кафедра «Компьютерная безопасность»

**ОТЧЕТ
К ЛАБОРАТОРНОЙ РАБОТЕ №2**

по дисциплине

«Языки программирования»

Работу выполнил
студент группы СКБ-201

подпись, дата

Г.П. Кашкин

Работу проверил

подпись, дата

С.А. Булгаков

Содержание

Постановка задачи	3
1 Алгоритм решения задачи	4
1.1 Задание 1	4
1.2 Задание 2	4
1.3 Задание 3	4
1.4 Задание 4	4
1.5 Задание 5	4
2 Выполнение задания	5
2.1 Задание 1	5
2.2 Задания 2 и 4	6
2.3 Задание 3	7
2.4 Задание 5	8
3 Получение исполняемых модулей	9
4 Тестирование	9
4.1 Задание 1	9
4.1.1 Конструкторы	9
4.1.2 Присвоение	9
4.1.3 Доступ к элементам	9
4.1.4 Итераторы	9
4.1.5 Вместимость	9
4.2 Задания 2 и 4	10
4.2.1 Добавление и удаление	10
4.2.2 Копирование и перемещение	10
4.2.3 Итераторы	10
4.2.4 Ручное изменение вместимости	10
4.3 Задание 3	10
4.3.1 Вставление с краев	10
4.3.2 Вставление и удаление с краев	10
4.3.3 Операторы	10
4.3.4 Копирование и перемещение	10
4.4 Задание 5	10
4.4.1 ObjPool	10
4.4.2 Итераторы	11
4.4.3 Вставка и удаление	11
4.4.4 Копирование и перемещение	11
Приложение А	12
Приложение Б	15
Приложение В	34
Приложение Г	52
Приложение Д	61

Постановка задачи

Разработать программу на языке Си++ (ISO/IEC 14882:2020), демонстрирующую решение поставленной задачи.

Общая часть

Разработать класс ADT и унаследовать от него классы, разработанные в рамках лабораторной работы 1. Разработать набор классов, объекты которых реализуют типы данных, указанные ниже. Для этих классов разработать необходимые конструкторы, деструктор, конструктор копирования. Разработать операции: добавления/удаления элемента (уточнено в задаче); получения количества элементов; доступа к элементу (перегрузить оператор []). При ошибках запускать исключение. В главной функции разместить тесты, разработанные с использованием библиотеки GoogleTest.

Задачи

- а) Динамический массив указателей на объекты ADT. Размерность массива указателей увеличивается в момент его переполнения. Начальная размерность задается как параметр конструктора, значение по умолчанию 0. Добавление/удаление элемента в произвольное место.
- б) Стек, представленный динамическим массивом указателей на хранимые объекты ADT. Размерность стека увеличивается в момент его переполнения. Начальная размерность задается как параметр конструктора, значение по умолчанию 0. Добавление/удаление элемента в начало и в конец.
- в) Односвязный список, содержащий указатели на объекты ADT. Добавление/удаление элемента в произвольное место.
- г) Циклическая очередь, представленная динамическим массивом указателей на хранимые объекты ADT. Добавление/удаление элемента в произвольное место.
- д) Двоичное дерево, содержащее указатели на объекты ADT. Добавление/удаление элемента в произвольное место.

1 Алгоритм решения задачи

1.1 Задание 1

Для решения задачи был реализован собственный класс с интерфейсом аналогичным интерфейсу `std::vector`. Многократно используются алгоритмы стандартной библиотеки для копирования перемещения и заполнения динамического массива.

1.2 Задание 2

Для решения задачи была написана структура дека (быстрое добавление и удаление объектов из обоих концов). Класс реализован на базе динамического массива, помимо него отслеживаются вместимость, текущий размер и позиции начала и конца значащих элементов. Таким образом, когда начало или конец находится на границе выделенной памяти, новый элемент добавляется с противоположной стороны, как и все последующие. Для большего удобства выбран инвариант вместимости контейнера как степени двойки, потому что тогда $(pos + n) \% capacity$ будет корректным с учетом переполнения для всех n . Помимо этого, `capacity` хранится в виде величины на единицу меньше вместимости, таким образом вместо операции деления с остатком можно использовать операцию побитового И (`size_t new_id = (id - n) & capacity` будет корректным в том числе когда $id < n$), что немного быстрее и изящнее. Получается, усредненная асимптотика аналогична асимптотике классического вектора и равна $O(1)$, как и доступ по индексу.

1.3 Задание 3

Для решения данной задачи был написан связный список, состоящий из внешнего класса и сервисной структуры узла (`node`). В узле хранится только `id` следующего значения и полезная нагрузка, в структуре хранится `ObjPool`, выполняющий функции простейшего аллокатора, `id` первого узла и размер списка. `ObjPool` в свою очередь действует как односвязный список на базе динамического массива. При выдаче узла из пула, возвращается значение из начала списка, при возвращении оно ставится в начало, таким образом это происходит с константной асимптотикой, но не требует вызова медленных системных инструкций выделения. Так как список односвязный по условию, реализован только `forward_iterator`, а добавление, удаление или модификация узла происходит с линейной сложностью, потому что до нее нужно добраться поштучно перебирая предков с начала.

1.4 Задание 4

Циклическая очередь реализована так же на базе дека. Так как добавление и удаление объекта в середину в любом случае имеет линейную сложность, оно реализовано в виде переаллоцирования в новый массив с выравниванием концов.

1.5 Задание 5

Для данного задания был взят уже написанный в предыдущей лабораторной класс декартового дерева. Для оптимизированной индексации была добавлена запланированная поддержка `random_access_iterator` базирующаяся на двухсвязности узлов дерева, предподсчета размера поддерева для каждого узла и базовом инварианте дерева поиска (все левые дети меньше, все правые больше). Таким образом, все требуемые операции совершаются за $O(\log n)$ - усредненную высоту дерева.

2 Выполнение задания

2.1 Задание 1

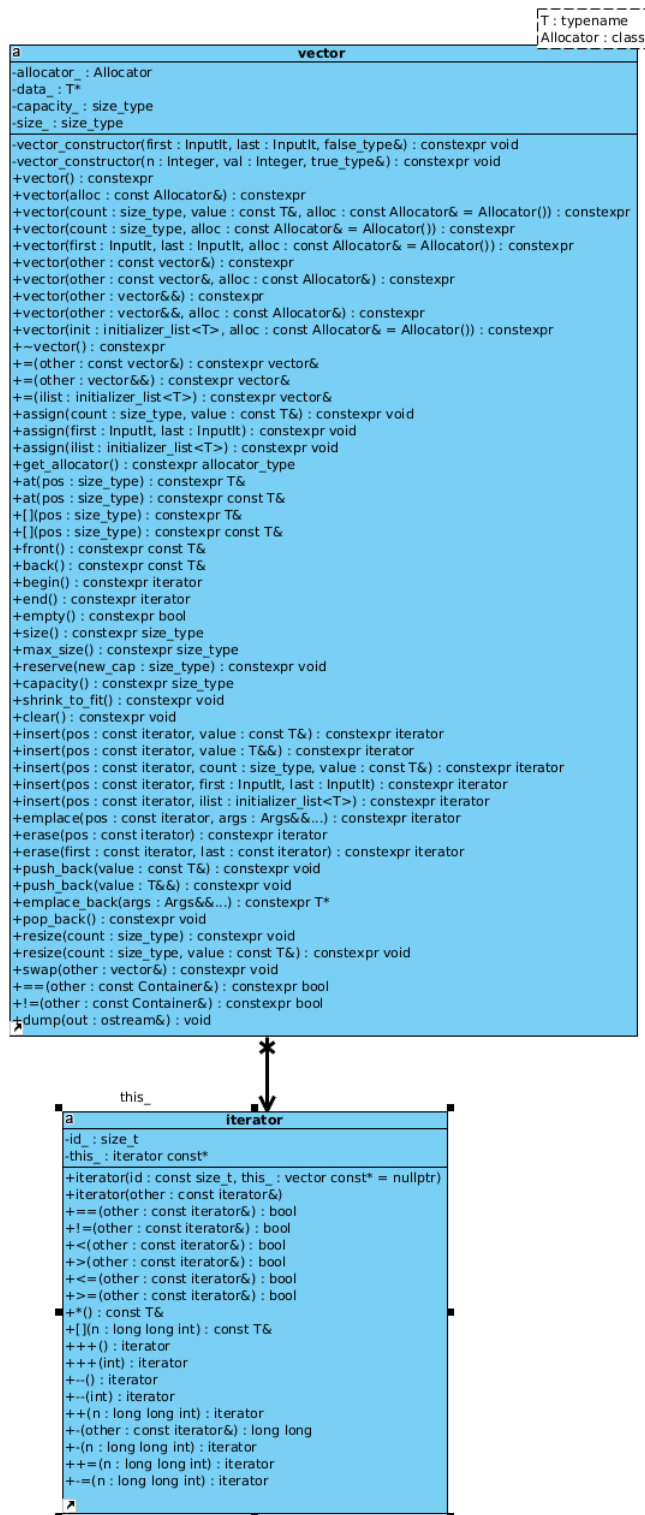


Рис. 1. UML 2.0 diagram for vector class

Для решения данной задачи разработан класс **vector**, хранящий в себе динамический массив объектов, текущий размер, вместимость и аллокатор для работы с системными вызовами. Аллокатор может предоставляться как параметр шаблона, по умолчанию используется `std::allocator` и вся работа с динамической памятью базируется на его интерфейсе. Интерфес класса намеренно копирует `std::vector` в стандарте c++20.

Одна из перегрузок конструктора принимает на вход пару произвольных итераторов и по ним создает новый вектор. Для реализации этой функции потребовалось создать сервисную функцию с двумя перегрузками. Первая для целочисленных переменных и по сути аналогична конструктору от длины и значения, вторая от итераторов, различаются они при помощи фиктивного значения одного из типов `std::false_type` или `std::true_type`, которые представляют только один из вариантов `bool` каждый. Выбирается тип при помощи `std::is_integral<InputIt>::type`.

Очевидно, что на такой простой линейной структуре легко реализовать `random_access` итераторы, что и было сделано. Для базовой безопасности при отладке расставлены ассерты, при проблемах с выделением памяти бросаются соответствующие ошибки.

2.2 Задания 2 и 4

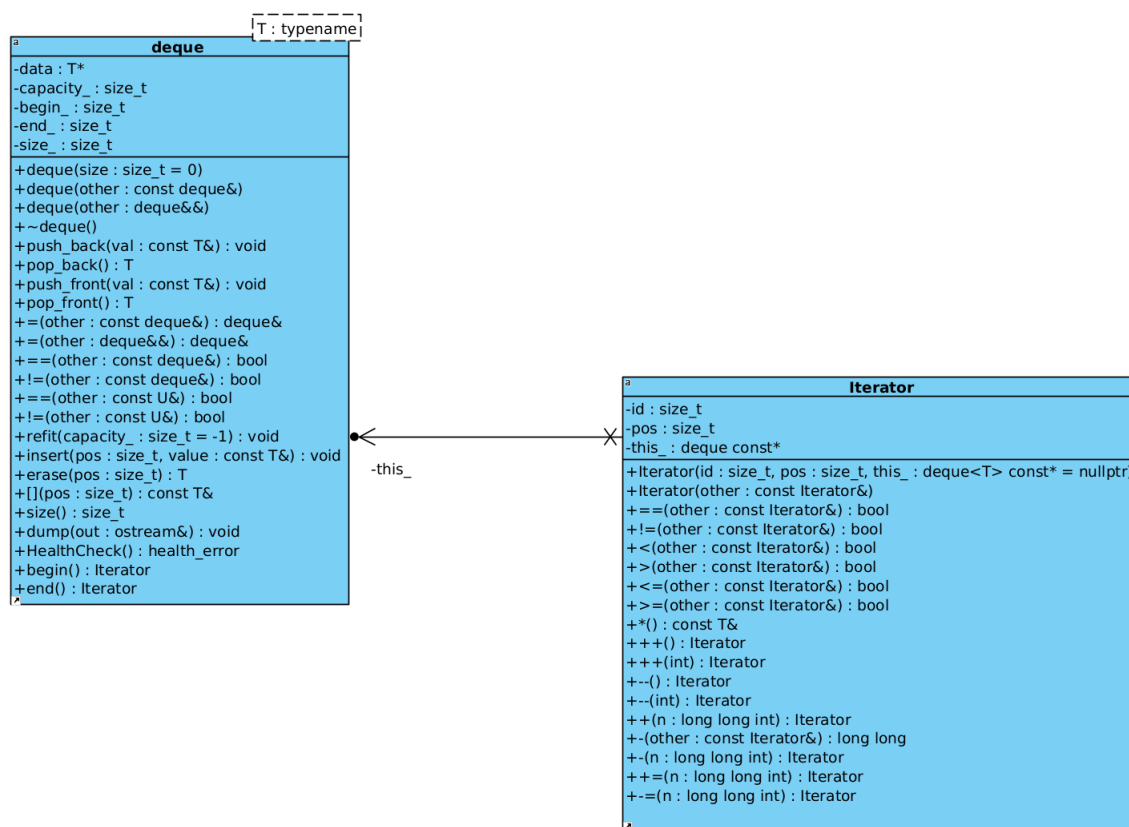


Рис. 2. UML 2.0 diagram for deque class

Для решения данной задачи разработан класс `deque`, основанный на массиве в динамической памяти. Для обработки выхода конца данных за край массива позиция постоянно делится с остатком на `capacity` (в данном случае `capacity = 2^n - 1`, так что используется эквивалентное и более оптимальное `size_t new_id = (id + n) & capacity`). Для упрощения итераторы хранят не только текущий `id`, но и позицию в деке. Помимо этого хранится указатель на структуру дека для упрощенного доступа к его полям. Так как хранится позиция, инвалидированный указатель (он же `end()`) это указатель с позицией равной длине дека, что позволяет узнавать длину структуры одним из каноничных способов, через `random_access` итераторы `end()` – `begin()`. Для комфортной отладки реализованы функции проверки состояния дека с макросом, возвращающим информацию о проблеме и делающем дамп структуры. Помимо проверки базовых инвариантов (`size <= capacity`) пустующие объекты заполняются специальным значением (`POISON`) и элементы массива проверяются на "отравленность" в соответствии со своим расположением относительно `begin` и `end`. Так как все эти функции значительно (в десятки раз) замедляют работу программы, их можно отключить определив макрос `NDEBUG`. Из-за особенностей алгоритма заполнения элементов `Poison` значением, невозможно использовать макросы отладки на типах шаблона меньше `int`, это учтено в тестах.

2.3 Задание 3

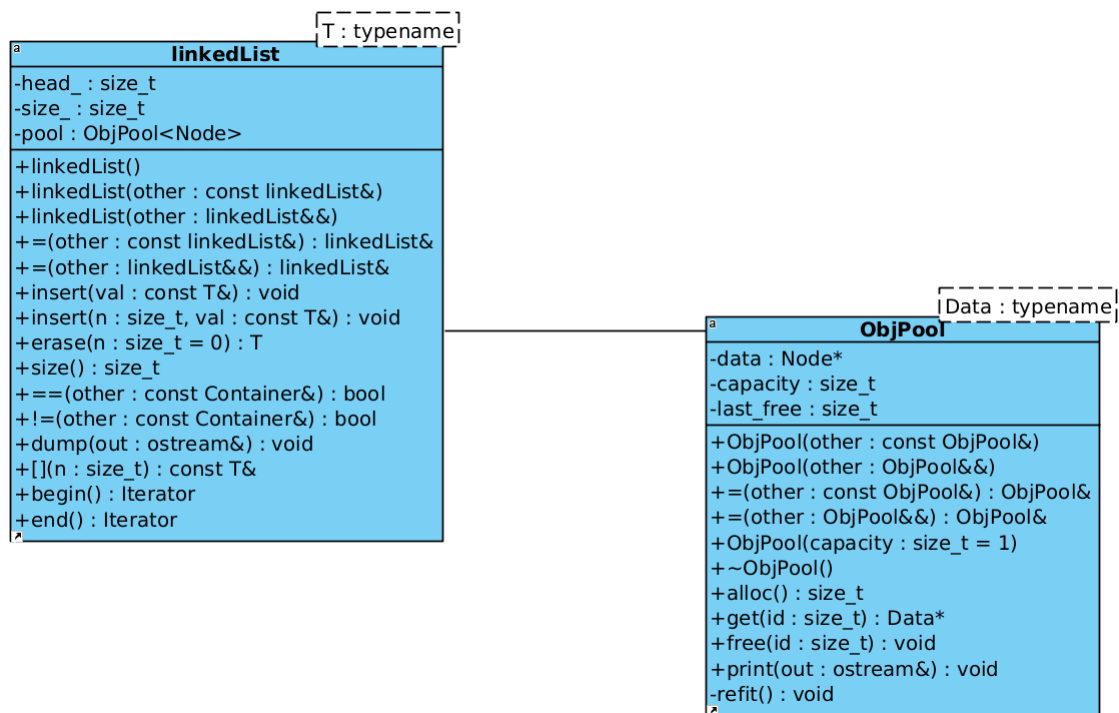


Рис. 3. UML 2.0 diagram for linkedlist class

Для решения данной задачи разработан класс `linkedList`, базирующийся на `ObjPool`. Таким образом хранится не указатель на узел, а `id` в `ObjPool`. В классе хранится `id` начала списка и его длина как сервисное значение. Добавление и удаление из произвольного места работает через серию переходов к следующему узлу от начальной и вставляет элемент до узла с заданной позицией. Сравнение списков осуществляется посимвольно через итераторы, которые из-за односвязности списка только **forward**. Конструкторы копирования и перемещения прото создают аналогичный пул и константы на вершины, после копирования новая структура будет полностью эквивалентна старой.

2.4 Задание 5

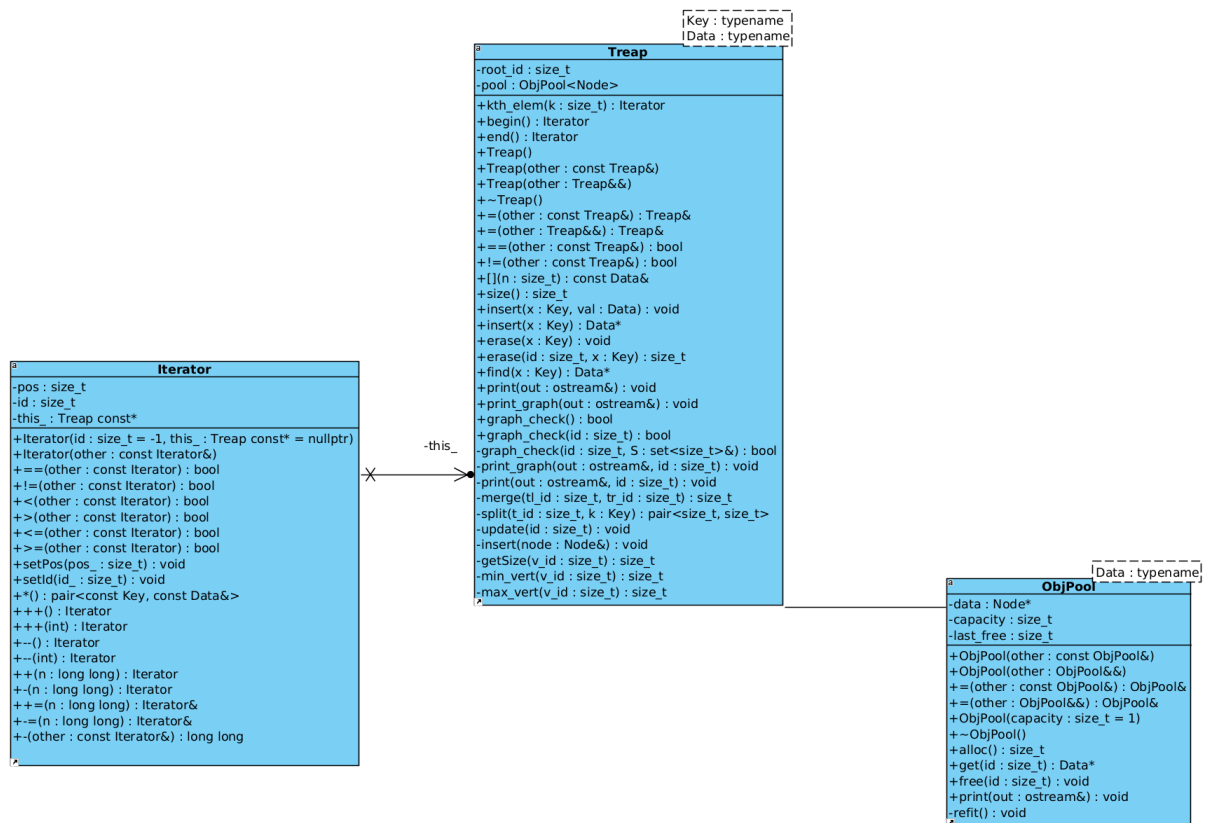


Рис. 4. UML 2.0 diagram for treap class

Для **treap** определена внутренняя структура узла, хранящая необходимые сервисные значения, то есть индекс предков и детей, случайный приоритет, размер поддерева данной вершины, ключ и полезную нагрузку. Для оптимизации выделения и освобождения памяти используется класс **ObjPool** (object pool), представляющий из себя связанный список доступных для выделения узлов **treap**. При аллоцировании занимается крайняя в списке узлов, если в **ObjPool** заканчиваются свободные узлы, он переаллоцируется целиком на память в два раза больше. Так как при переаллоцировании ломаются все указатели на выделенные узлы, приходится использовать индексы. На декартовом дереве реализована поддержка **random_access_access_iterator**, функции вставки, поиска и удаления элементов по ключу. Помимо этого есть несколько сервисных функций облегчающих отладку и небольшой макрос того же свойства. Хранение размеров поддеревьев в узлах **treap** реализовано для расширения итераторов в **random_access**, последнее было дописано и работает за $O(\log n)$. Разыменование итератора возвращает пару из ключа и его полезной нагрузки. **Treap** как и **ObjPool** базируется на механике шаблонов, таким образом и декартово дерево и пул можно использовать в других проектах, что и было сделано в данной лабораторной.

3 Получение исполняемых модулей

Для всего проекта использовалась система сборки `cmake`. В конфигурации системы сборки прописаны пять режимов компиляции: `basic`, `sanitizer`, `release`, `debug` и `coverage` с разными наборами флагов компиляции (использование посредством флага `-DCMAKE_BUILD_TYPE=*`), в каждом из них прописано использование требуемого стандарта `c++20`, а для проверки на покрытие тестов специфицируется компилятор `clang++`, из-за отсутствия универсального набора флагов у основных компиляторов. Помимо этого `cmake` автоматически скачивает с `github` и подключает библиотеку для `unit` тестов `GoogleTest`, которая используется для проверки корректности программ. Для каждого задания создан отдельный конфиг, который рекурсивно подключается в корневом `CMakeLists.txt`. Создан простой `python3` скрипт, парсящий результаты анализа покрытия и опускающий подробности касающиеся системных библиотек и `GoogleTest`. Так же в конфигурационный файл добавлена милотидная летающая капибара.

4 Тестирование

Тестирование производится при помощи библиотеки `GoogleTest`, везде, где позволяет логика используется многократный проход теста с генератором случайных значений и запуск шаблонной функции тестирования от нескольких различных базовых типов. Кроме этого, структуры на промежуточных стадиях сравниваются с эталонными из стандартной библиотеки. Анализ покрытия показывает не менее 97% задействованных строк кода на всех тестах, оставшиеся являются редкими случаями поддержания инвариантов на уровне ниже интерфейса.

4.1 Задание 1

4.1.1 Конструкторы

Тестируются на работоспособность и корректность все конструкторы класса. Для конструктора по итераторам используется собственный контейнер `deque`.

4.1.2 Присвоение

Тестируется корректность перегрузок функций `operator=` и `assign`.

4.1.3 Доступ к элементам

Проверяется доступ ко всем элементам случайно заполненного вектора (перегрузки функции `operator[]` и `at`).

4.1.4 Итераторы

Итераторы проверяются на эквивалентность `operator[]` и корректность с точки зрения задуманных инвариантов.

4.1.5 Вместимость

Проверяются функции относящиеся к модификации вместимости.

4.2 Задания 2 и 4

4.2.1 Добавление и удаление

Вначале создается дек большой случайной длины, путем последовательного добавления случайных элементов в начало и конец. После этого происходит удаление части элементов с обоих концов в случайной последовательности. В проверяется работа оператора доступа по индексу (`operator[]`).

4.2.2 Копирование и перемещение

Многократное копирование и перемещение случайно заполненных деков и проверка операций присваивания.

4.2.3 Итераторы

Итераторы проверяются на эквивалентность `operator[]` и корректность с точки зрения задуманных инвариантов, в том числе и с помощью собственных макросов.

4.2.4 Ручное изменение вместимости

Ручное изменение вместимости проверяется на корректность и отсутствие утечек. на случайно заполненных деках большой случайной длины.

4.3 Задание 3

4.3.1 Вставление с краев

Проверяется вставление элементов на крайних значениях концов списка.

4.3.2 Вставление и удаление с краев

Проверяется корректность вставления и удаления элементов на крайних значениях концов списка в случайном порядке.

4.3.3 Операторы

Немногочисленные операторы тестируются на повторяемость результатов операций, корректность копирования и стрессоустойчивость. Как и на прочих этапах производится сравнение с эталонным вектором.

4.3.4 Копирование и перемещение

Многократное копирование и перемещение случайно сгенерированных списков и проверка операций присваивания.

4.4 Задание 5

4.4.1 ObjPool

`ObjPool` тестировался вручную при помощи красивых функций вывода состояния и проверки встроеной инвариантов на крайних значениях в предыдущей лабораторной.

4.4.2 Итераторы

Итераторы проверяются на работоспособность и эквивалентность индексации вектора стандартной библиотеки.

4.4.3 Вставка и удаление

Вставка и удаление проверяются на эквивалентность аналогичным операциям вектора `std`.

4.4.4 Копирование и перемещение

Многократное копирование и перемещение случайно сгенерированных деревьев и проверка операций присваивания.

Приложение А

```
subsectionФайл CMakeLists.txt
1 cmake_minimum_required(VERSION 3.14)
2 project(Lab-3)
3
4 set(CMAKE_CXX_COMPILER "/usr/bin/clang++" CACHE string "clang compiler" FORCE)
   #TODO check if clang is present when COVERAGE needed
5
6
7 set(CMAKE_CXX_STANDARD 20)
8 set(CMAKE_CXX_STANDARD_REQUIRED ON)
9
10
11 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++20" CACHE STRING "Comment"
    FORCE)
12 set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS} -D NDEBUG" CACHE STRING "
    Comment" FORCE)
13 set(CMAKE_CXX_FLAGS_SANITIZE "${CMAKE_CXX_FLAGS} -Wpedantic -Wall -Wextra -
    Wformat=2 -fsanitize=address,undefined -g" CACHE STRING "Comment" FORCE)
14 set(CMAKE_CXX_FLAGS_COVERAGE "${CMAKE_CXX_FLAGS} -D NDEBUG -fprofile-instr-
    generate -fcoverage-mapping" CACHE STRING "Comment" FORCE)
15 set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS} -g -Werror -Wall -Wextra -Weffc
    ++ -Waggressive-loop-optimizations -Wc++0x-compat -Wc++11-compat -Wc++14-
    compat -Wcast-align -Wcast-qual -Wchar-subscripts -Wconditionally-supported
    -Wconversion -Wctor-dtor-privacy -Wempty-body -Wfloat-equal -Wformat-
    nonliteral -Wformat-security -Wformat-signedness -Wformat=2 -Winline -
    Wlarger-than=8192 -Wlogical-op -Wmissing-declarations -Wnon-virtual-dtor -
    Wopenmp-simd -Woverloaded-virtual -Wpacked -Wpointer-arith -Wredundant-
    decls -Wshadow -Wsign-conversion -Wsign-promo -Wstack-usage=8192 -Wstrict-
    null-sentinel -Wstrict-overflow=2 -Wsuggest-attribute=noreturn -Wsuggest-
    final-methods -Wsuggest-final-types -Wsuggest-override -Wswitch-default -
    Wswitch-enum -Wsync-nand -Wundef -Wunreachable-code -Wunused -Wuseless-cast
    -Wvariadic-macros -Wno-literal-suffix -Wno-missing-field-initializers -Wno
    -narrowing -Wno-old-style-cast -Wno-varargs -fcheck-new -fsized-
    deallocation -fstack-check -fstack-protector -fstrict-overflow -flto-odr-
    type-merging -fno-omit-frame-pointer -fPIE -fsanitize=address -fsanitize=
    alignment -fsanitize=bool -fsanitize=bounds -fsanitize=enum -fsanitize=
    float-cast-overflow -fsanitize=float-divide-by-zero -fsanitize=integer-
    divide-by-zero -fsanitize=leak -fsanitize=nonnull-attribute -fsanitize=null
    -fsanitize=object-size -fsanitize=return -fsanitize=returns-nonnull-
    attribute -fsanitize=shift -fsanitize=signed-integer-overflow -fsanitize=
    undefined -fsanitize=unreachable -fsanitize=vla-bound -fsanitize=vptr -lm -
    pie")
16
17 include(FetchContent)
18
19 FetchContent_Declare(
20     googletest
21     GIT_REPOSITORY https://github.com/google/googletest.git
22     GIT_TAG         release-1.8.0
23 )
24
```

```

25 FetchContent_GetProperties(googletest)
26 if(NOT googletest_POPULATED)
27   FetchContent_Populate(googletest)
28   add_subdirectory(${googletest_SOURCE_DIR} ${googletest_BINARY_DIR})
29 endif()
30
31
32 set_target_properties(gtest PROPERTIES FOLDER extern)
33 set_target_properties(gtest_main PROPERTIES FOLDER extern)
34 set_target_properties(gmock PROPERTIES FOLDER extern)
35 set_target_properties(gmock_main PROPERTIES FOLDER extern)
36
37 enable_testing()
38
39 add_subdirectory(./vector)
40 add_subdirectory(./treap)
41 add_subdirectory(./linkedList)
42 add_subdirectory(./deque)

```

A.1 Файл coverage-check.py

```

1 #!/bin/python3
2 import subprocess
3 import sys, getopt
4
5 def findnth(string, substring, n):
6     parts = string.split(substring, n + 1)
7     if len(parts) <= n + 1:
8         return -1
9     return len(string) - len(parts[-1]) - len(substring)
10
11 def main(argv):
12
13     toshow = False
14
15     try:
16         arg = argv[0]
17
18     except:
19         print("Bad argument.\ncoverage-check.py <dir> [--show]")
20         sys.exit(2)
21
22     for elem in argv[1:]:
23         if elem == "--show":
24             toshow = True
25         elif elem == "-h":
26             print("coverage-check.py <dir> [--show]")
27             sys.exit(2)
28         else:
29             print("Bad argument.\ncoverage-check.py <dir> [--show]")
30             sys.exit(2)
31
32     if (arg[-1] == '/'):

```

```

33     arg = arg[: -1]
34
35     name = arg[arg.rfind('/') + 1:]
36
37     #print("llvm-profdata merge -o out.profdata " + arg + "/default.profrac")
38     subprocess.check_output("llvm-profdata merge -o out.profdata " + arg + "/
default.profrac", shell=True, universal_newlines=True)
39     #print("llvm-cov report " + arg + '/' + name + " --instr-profile=out.
profdata")
40     report = subprocess.check_output("llvm-cov report " + arg + '/' + name + "
--instr-profile=out.profdata", shell=True, universal_newlines=True)
41
42     result = ""
43     for line in report.split('\n'):
44         if line.count("——") or line.count(name + '/' + name.lower() + '.h'
):
45             result += line + '\n'
46             if line.count("Filename") or line.count("TOTAL"):
47                 result += line + '\n'
48
49
50     if (toshow):
51         show = subprocess.check_output("llvm-cov show " + arg + '/' + name + "
--instr-profile=out.profdata", shell=True, universal_newlines=True)
52         result += '\n\n\n'
53         result += show[show.find('/') + name + '/' + name.lower() + ".h"):]
54
55     subprocess.check_output("rm ./out.profdata", shell=True,
universal_newlines=True)
56     print(result)
57
58 if __name__ == "__main__":
59     main(sys.argv[1:])

```

Приложение Б

Б.1 Файл `vector.hpp`

```
1 #ifndef VECTOR_HPP
2 #define VECTOR_HPP
3
4 #include <memory>
5 #include <utility>
6 #include <cassert>
7 #include <type_traits>
8 #include <algorithm>
9
10 #include <iostream>           //DEBUG
11
12
13
14 template< typename T, class Allocator = std::allocator<T> >
15 class vector {
16 private:
17     using value_type          = T;
18     using allocator_type      = Allocator;
19     using size_type           = std::size_t;
20     using difference_type     = std::ptrdiff_t;
21
22     template< class InputIt >
23     constexpr void vector_constructor( InputIt first, InputIt last, const std::
24     ::false_type& /*!isIntegral*/){
25         if (data_)
26             allocator_.deallocate(data_, capacity_);
27         capacity_ = 0;
28         size_ = 0;
29         data_ = nullptr;
30         for (; first != last; ++first)
31             push_back(*first);
32     }
33
34     template< class Integer >
35     constexpr void vector_constructor( Integer n, Integer val, const std::
36     true_type& /*!isIntegral*/){
37         if (data_)
38             allocator_.deallocate(data_, capacity_);
39         capacity_ = n;
40         size_ = n;
41         data_ = allocator_.allocate(capacity_);
42         if (!data_)
43             throw std::runtime_error("Failed to allocate memory");
44         std::fill(data_, data_ + capacity_, val);
45     }
46
47 public:
```

```

48 //=====
49 //  Member functions
50
51 constexpr vector() noexcept(noexcept(Allocator()));
52
53 constexpr explicit vector( const Allocator& alloc ) noexcept;
54
55 constexpr explicit vector( size_type count,
56                             const T& value,
57                             const Allocator& alloc = Allocator() );
58
59 constexpr explicit vector( size_type count,
60                             const Allocator& alloc = Allocator() );
61
62 template< class InputIt >
63 constexpr explicit vector( InputIt first, InputIt last,
64                             const Allocator& alloc = Allocator() ) { //->
65 vector<typename std::decay_t<decltype(*first)>> { // -> vector<typename std
66 ::iterator_traits<InputIt>::value_type> {
67     typedef typename std::is_integral<InputIt>::type _Integral;
68     allocator_ = alloc;
69     data_ = nullptr;
70     vector_constructor(first, last, _Integral());
71 }
72
73 constexpr vector( const vector& other );
74
75 constexpr vector( const vector& other, const Allocator& alloc );
76
77 constexpr vector( vector&& other );
78
79 constexpr vector( vector&& other, const Allocator& alloc );
80
81 constexpr vector( std::initializer_list<T> init, //TODO
82                 const Allocator& alloc = Allocator() );
83
84
85 constexpr ~vector() { if (data_) allocator_.deallocate(data_, capacity_);
86 }
87
88
89 constexpr vector& operator=( const vector& other );
90
91
92 constexpr vector& operator=( vector&& other ) noexcept ; //TODO fix memory
93 leak (God knows where)
94
95 constexpr vector& operator=( std::initializer_list<T> ilist ); //TODO
96
97
98 constexpr void assign( size_type count, const T& value );
99
100 template< class InputIt >
101 constexpr void assign( InputIt first, InputIt last );

```



```

97     constexpr void assign( std::initializer_list<T> ilist );           //TODO
98
99
100    constexpr allocator_type get_allocator() const noexcept { return
allocator_; }
101
102    //=====
103    // Element access
104
105    constexpr T& at( size_type pos );
106
107    constexpr const T& at( size_type pos ) const;
108
109
110    constexpr T& operator [] ( size_type pos );
111
112    constexpr const T& operator [] ( size_type pos ) const;
113
114
115    constexpr T& front()          { assert(size_ != 0); return data_[0]; }
116
117    constexpr const T& front() const { assert(size_ != 0); return data_[0]; }
118
119    constexpr T& back()           { assert(size_ != 0); return data_[size_
- 1]; }
120
121    constexpr const T& back() const { assert(size_ != 0); return data_[size_
- 1]; }
122
123    //=====
124    // Iterators
125
126    struct iterator {
127        using iterator_category = std::random_access_iterator_tag;
128        using difference_type   = std::ptrdiff_t;
129        using value_type        = T;
130        using pointer            = T*;
131        using reference          = T&;
132
133    public:
134        iterator( const size_t id, const vector* this_ = nullptr ) : id_(id),
this_(this_) {}
135        iterator( const iterator& other) = default;
136
137        bool operator==( const iterator &other ) const { return id_ == other.
id_; }
138        bool operator!=( const iterator &other ) const { return id_ != other.
id_; }
139
140        bool operator< ( const iterator &other ) const { return id_ < other.
id_; }
141        bool operator> ( const iterator &other ) const { return id_ > other.
id_; }

```

```

142     bool operator<=( const iterator &other ) const { return id_ <= other.
id_ ; }
143     bool operator>=( const iterator &other ) const { return id_ >= other.
id_ ; }
144
145     T& operator*() { assert(id_ < this_>size()); return this_>data_[id_
]; }
146     const T& operator*() const { assert(id_ < this_>size()); return this_
->data_[id_]; }
147
148     T& operator[]( long long int n ) { assert(id_ + n < this_>size());
return this_>data_[id_ + n]; }
149     const T& operator[]( long long int n ) const { assert(id_ + n < this_
->size()); return this_>data_[id_ + n]; }
150
151
152     iterator operator++() {
153         ++id_ ;
154         return *this ;
155     }
156
157     iterator operator++(int) {
158         iterator result(*this);
159         ++id_ ;
160         return result ;
161     }
162
163     iterator operator--() {
164         --id_ ;
165         return *this ;
166     }
167
168     iterator operator--(int) {
169         iterator result(*this);
170         --id_ ;
171         return result ;
172     }
173
174     iterator operator+(long long int n) const {
175         iterator result(*this);
176         result.id_ += n ;
177         return result ;
178     }
179
180     long long operator-(const iterator& other) const {
181         return id_ - other.id_ ;
182     }
183
184     iterator operator-(long long int n) const {
185         iterator result(*this);
186         result.id_ -= n ;
187         return result ;
188     }

```

```

189
190     iterator operator+=(long long int n){
191         id_ += n;
192         return (*this);
193     }
194
195     iterator operator--=(long long int n){
196         id_ -= n;
197         return (*this);
198     }
199
200
201
202
203     private:
204         size_t id_;
205         const vector* this_;
206     };
207
208     constexpr iterator begin() const noexcept { return iterator(0, this); }
209
210     constexpr iterator end() const noexcept { return iterator(this->size(),
211 this); }
212
213
214     //=====
215     // Capacity
216
217     [[nodiscard]] constexpr bool empty() const noexcept { return begin() ==
218 end(); }
219
220     constexpr size_type size() const noexcept { return size_; }
221
222     constexpr size_type max_size() const noexcept { return allocator_.max_size
223 (); } //TODO check if it is the right way to define max_size
224
225     constexpr void reserve( size_type new_cap );
226
227     constexpr size_type capacity() const noexcept { return capacity_; }
228
229     constexpr void shrink_to_fit();
230
231
232     //=====
233     // Modifiers
234
235     constexpr void clear() noexcept { if (data_) allocator_.deallocate(data_,
236 capacity_); size_ = 0; capacity_ = 0; data_ = nullptr; }
237
238     constexpr iterator insert( const iterator pos, const T& value ); //TODO
239 all insert(), emplace() and erase()
240
241     constexpr iterator insert( const iterator pos, T&& value );

```

```

237
238     constexpr iterator insert( const iterator pos, size_type count, const T&
value );
239
240     template<class InputIt>
241     constexpr iterator insert( const iterator pos, InputIt first, InputIt last
);
242
243     constexpr iterator insert( const iterator pos, std::initializer_list<T>
ilist);
244
245     template<class... Args>
246     constexpr iterator emplace( const iterator pos, Args&&... args );
247
248     constexpr iterator erase( const iterator pos );
249
250     constexpr iterator erase( const iterator first, const iterator last );
251
252     constexpr void push_back( const T& value );
253
254     constexpr void push_back( T&& value );
255
256     template<class... Args>
257     constexpr T* emplace_back( Args&&... args);          //TODO
258
259     constexpr void pop_back() noexcept { assert(size_ != 0); --size_; };
260
261     constexpr void resize( size_type count );
262
263     constexpr void resize( size_type count, const T& value );
264
265     constexpr void swap( vector& other ) noexcept { std::swap(data_, other.
data_); std::swap(capacity_, other.capacity_); std::swap(size_, other.size_
); }
266
267
268     //=====
269     // Comparing
270
271     template<typename Container>
272     constexpr bool operator==( const Container& other ) const;
273
274
275     template<typename Container>
276     constexpr bool operator!=( const Container& other ) const { return !(*this
== other); }
277
278
279 private:
280     Allocator allocator_;
281     T* data_;
282     size_type capacity_;
283     size_type size_;

```

```

284
285 #ifndef NDEBUG
286
287 public:
288
289     void dump(std::ostream& out) {
290         out << "capacity = " << capacity_ << "\n";
291         out << "size      = " << size_ << "\n";
292         for (auto elem : *this)
293             out << "{ " << elem << " } ";
294         out << '\n';
295     }
296
297 #endif
298 };
299
300
301
302
303 template< typename T, class Allocator >
304 constexpr vector<T, Allocator>::vector() noexcept(noexcept(Allocator())) :
    data_(nullptr), capacity_(0), size_(0) {}
305
306
307 template< typename T, class Allocator >
308 constexpr vector<T, Allocator>::vector( const Allocator& alloc ) noexcept :
    allocator_(alloc), data_(nullptr), capacity_(0), size_(0) {}
309
310
311 template< typename T, class Allocator >
312 constexpr vector<T, Allocator>::vector( size_type count, const T& value, const
    Allocator& alloc ) : allocator_(alloc), data_(nullptr), capacity_(count),
    size_(count) {
313     data_ = allocator_.allocate(capacity_);
314     if (!data_)
315         throw std::runtime_error("Failed to allocate memory");
316     std::fill(data_, data_ + capacity_, value);
317 }
318
319
320 template< typename T, class Allocator >
321 constexpr vector<T, Allocator>::vector( size_type count, const Allocator&
    alloc ) : allocator_(alloc), data_(nullptr), capacity_(count), size_(0) {
322     data_ = allocator_.allocate(capacity_);
323     if (!data_)
324         throw std::runtime_error("Failed to allocate memory");
325 }
326
327
328 template< typename T, class Allocator >
329 constexpr vector<T, Allocator>::vector( const vector& other ) : allocator_(
    other.allocator_), data_(nullptr), capacity_(other.capacity_), size_(other.
    size_) {

```

```

330     data_ = allocator_.allocate(capacity_);
331     if (!data_)
332         throw std::runtime_error("Failed to allocate memory");
333     std::copy(other.begin(), other.end(), data_);
334 }
335
336
337 template< typename T, class Allocator >
338 constexpr vector<T, Allocator>::vector( const vector& other, const Allocator&
    alloc ) : allocator_(alloc), capacity_(other.capacity_), size_(other.size_)
    {
339     data_ = allocator_.allocate(capacity_);
340     if (!data_)
341         throw std::runtime_error("Failed to allocate memory");
342     std::copy(other.begin(), other.end(), data_);
343 }
344
345
346 template< typename T, class Allocator >
347 constexpr vector<T, Allocator>::vector( vector&& other ) : allocator_(other.
    allocator_), data_(other.data_), capacity_(other.capacity_), size_(other.
    size_) {
348     other.data_ = nullptr;
349     other.size_ = 0;
350     other.capacity_ = 0;
351 }
352
353
354 template< typename T, class Allocator >
355 constexpr vector<T, Allocator>::vector( vector&& other, const Allocator& alloc
    ) : allocator_(alloc), data_(nullptr), capacity_(other.capacity_), size_(
    other.size_) {
356     if (alloc != other.allocator_) { //TODO check if
        this is the right way of element-wise move
357         data_ = allocator_.allocate(capacity_);
358         if (!data_)
359             throw std::runtime_error("Failed to allocate memory");
360
361         if (std::is_move_assignable<T>::value)
362             for (size_t i = 0; i < size_; ++i)
363                 data_[i] = std::move(other.data_[i]);
364         else
365             std::copy(other.begin(), other.end(), data_);
366     }
367     else {
368         data_ = other.data_;
369         other.size_ = 0;
370         other.capacity_ = 0;
371     }
372 }
373
374
375 template< typename T, class Allocator >

```

```

376 constexpr vector<T, Allocator>& vector<T, Allocator>::operator=( const vector&
    other ) {
377     size_ = other.size_ ;
378     if (data_)
379         allocator_.deallocate(data_ , capacity_);
380     capacity_ = other.capacity_ ;
381     data_ = allocator_.allocate(capacity_);
382     if (!data_)
383         throw std::runtime_error("Failed to allocate memory");
384     std::copy(other.begin(), other.end(), data_);
385     return *this;
386 }
387
388
389 template< typename T, class Allocator >
390 constexpr vector<T, Allocator>& vector<T, Allocator>::operator=( vector&&
    other ) noexcept {
391     allocator_ = other.allocator_ ;
392     capacity_ = std::exchange(other.capacity_ , 0);
393     size_ = std::exchange(other.size_ , 0);
394     data_ = std::exchange(other.data_ , nullptr);
395     return *this;
396 }
397
398
399 template< typename T, class Allocator >
400 constexpr void vector<T, Allocator>::assign( size_type count , const T& value)
    {
401     if (capacity_ < count){
402         allocator_.deallocate(data_ , capacity_);
403         capacity_ = count;
404         data_ = allocator_.allocate(capacity_);
405         if (!data_)
406             throw std::runtime_error("Failed to allocate memory");
407     }
408     size_ = count;
409     std::fill(data_ , data_ + size_ , value);
410 }
411
412
413 template< typename T, class Allocator >
414 template< class InputIt >
415 constexpr void vector<T, Allocator>::assign( InputIt first , InputIt last) {
416     typedef typename std::is_integral<InputIt>::type _Integral;
417     vector_constructor(first , last , _Integral());
418 }
419 /*
420     size_ = 0;
421     for (; first != last; ++first){
422         if (size_ == capacity_){
423             if (!data_){
424                 capacity_ = 2;
425                 data_ = allocator_.allocate(capacity_);

```

```

426         if (!data_)
427             throw std::runtime_error("Failed to allocate memory");
428     }
429     else {
430         size_t new_cap = capacity_ * 2;
431         T* tmp = allocator_.allocate(new_cap);
432         if (!tmp)
433             throw std::runtime_error("Failed to allocate memory");
434         std::copy(data_, data_ + capacity_, tmp); //TODO
435         think if it is more effitient (sometimes?) to make element-wise move
436         allocator_.deallocate(data_, capacity_);
437         data_ = tmp;
438         capacity_ = new_cap;
439     }
440     data_[size_] = *first;
441     ++size_;
442 }
443 }
444 */
445
446 template< typename T, class Allocator >
447 constexpr T& vector<T, Allocator>::at( size_type pos ) {
448     if (pos >= size())
449         throw std::out_of_range("Position given to at() is invalid");
450     return data_[pos];
451 }
452
453
454 template< typename T, class Allocator >
455 constexpr const T& vector<T, Allocator>::at( size_type pos ) const {
456     if (pos >= size())
457         throw std::out_of_range("Position given to at() is invalid");
458     return data_[pos];
459 }
460
461
462 template< typename T, class Allocator >
463 constexpr T& vector<T, Allocator>::operator[]( size_type pos ) {
464     assert(pos < size());
465     return data_[pos];
466 }
467
468
469 template< typename T, class Allocator >
470 constexpr const T& vector<T, Allocator>::operator[]( size_type pos ) const {
471     assert(pos < size());
472     return data_[pos];
473 }
474
475
476 template< typename T, class Allocator >
477 constexpr void vector<T, Allocator>::reserve( size_type new_cap ) {

```



```

478     if (new_cap > capacity_){
479         T* tmp = allocator_.allocate(new_cap);
480         if (!tmp)
481             throw std::runtime_error("Failed to allocate memory");
482
483         if (data_){
484             std::copy(data_, data_ + size_, tmp);
485             allocator_.deallocate(data_, capacity_);
486         }
487         data_ = tmp;
488         capacity_ = new_cap;
489         return;
490     }
491     assert(false);
492 }
493
494
495 template< typename T, class Allocator >
496 constexpr void vector<T, Allocator>::shrink_to_fit() {
497     if (size_ != capacity_){
498         T* tmp = allocator_.allocate(size_);
499         if (!tmp)
500             throw std::runtime_error("Failed to allocate memory");
501
502         std::copy(data_, data_ + size_, tmp);
503         assert(data_);
504         allocator_.deallocate(data_, capacity_);
505         data_ = tmp;
506         capacity_ = size_;
507     }
508 }
509
510
511 template< typename T, class Allocator > //TODO
512 constexpr typename vector<T, Allocator>::iterator vector<T, Allocator>::insert
513     ( const vector<T, Allocator>::iterator pos, const T& value ) {
514     assert(false); //DEBUG
515 }
516
517 template< typename T, class Allocator >
518 constexpr void vector<T, Allocator>::push_back( const T& value ) {
519     if ((size_ == capacity_) || !capacity_){
520         if (!capacity_)
521             capacity_ = 1;
522         reserve(capacity_ * 2);
523     }
524
525     data_[size_] = value;
526     ++size_;
527 }
528
529

```

```

530 template< typename T, class Allocator >
531 constexpr void vector<T, Allocator>::push_back( T&& value ) {
532     if (size_ == capacity_ || !capacity_){
533         if (!capacity_)
534             capacity_ = 1;
535         reserve(capacity_ * 2);
536     }
537
538     data_[size_] = std::move(value);
539     ++size_;
540 }
541
542
543 template< typename T, class Allocator >
544 template<class... Args>
545 constexpr T* vector<T, Allocator>::emplace_back( Args&&... args ) {
546     if (size_ == capacity_ || !capacity_){
547         if (!capacity_)
548             capacity_ = 1;
549         reserve(capacity_ * 2);
550     }
551
552     allocator_.construct(data_ + size_, args...);
553     ++size_;
554 }
555
556
557 template< typename T, class Allocator >
558 constexpr void vector<T, Allocator>::resize( size_type count ) {
559     if (count == size_)
560         return;
561     T* tmp = allocator_.allocate(count);
562     if (!tmp)
563         throw std::runtime_error("Failed to allocate memory");
564
565     if (count < size_)
566         std::copy(data_, data_ + count, tmp);
567
568     if (count > size_){
569         std::copy(data_, data_ + size_, tmp);
570     }
571
572     if (data_)
573         allocator_.deallocate(data_, capacity_);
574     data_ = tmp;
575     capacity_ = count;
576 }
577
578
579 template< typename T, class Allocator >
580 constexpr void vector<T, Allocator>::resize( size_type count, const T& value )
581 {
582     if (count == size_)

```

```

582         return;
583     T* tmp = allocator_.allocate(count);
584     if (!tmp)
585         throw std::runtime_error("Failed to allocate memory");
586
587     if (count < size_)
588         std::copy(data_, data_ + count, tmp);
589
590     if (count > size_){
591         std::copy(data_, data_ + size_, tmp);
592         std::fill(tmp + size_, tmp + count, value);
593     }
594
595     if (data_)
596         allocator_.deallocate(data_, capacity_);
597     data_ = tmp;
598     capacity_ = count;
599 }
600
601
602 template< typename T, class Allocator >
603 template<typename Container>
604 constexpr bool vector<T, Allocator>::operator==( const Container& other )
605     const {
606     if (size() != other.size())
607         return false;
608     auto iter_this = begin();
609     auto iter_other = other.begin();
610     while (iter_this != end()){
611         if (*iter_this != *iter_other)
612             return false;
613         ++iter_this; ++iter_other;
614     }
615     return true;
616 }
617 }
618
619 #endif

```

Б.2 Файл test-vector.hpp

```

1 #include <iostream>
2 #include <vector>
3 #include <random>
4 #include <deque>
5 #include "gtest/gtest.h"
6
7 #include "vector.hpp"
8 #include "../deque/deque.hpp"
9
10 std::mt19937 rnd(179);
11
12 TEST(Manual, Constructors)

```

```

13 {
14     deque<int> D;
15     for (int i = 0; i < 40; ++i){
16         D.push_back(rnd());
17         if (rnd() % 12)
18             D.push_front(rnd());
19     }
20
21     vector<long long> V1, V2(112), V3(1, 12345), V4(D.begin(), D.end());
22
23     std::vector<long long> STDV1, STDV2(112), STDV3(1, 12345), STDV4(D.begin(),
24     , D.end());
25
26     for (size_t i = 0; i < V1.size(); ++i)
27         EXPECT_EQ(V1[i], STDV1[i]);
28     for (size_t i = 0; i < V2.size(); ++i)
29         EXPECT_EQ(V2[i], STDV2[i]);
30     for (size_t i = 0; i < V3.size(); ++i)
31         EXPECT_EQ(V3[i], STDV3[i]);
32     for (size_t i = 0; i < V4.size(); ++i)
33         EXPECT_EQ(V4[i], STDV4[i]);
34
35     EXPECT_EQ(V4, STDV4);
36     EXPECT_NE(V1, V4);
37     vector<long long> V6(V4);
38     V6[2] = rnd();
39     EXPECT_NE(V6, V4);
40     vector<long long> V5(std::move(V4));
41     std::vector<long long> STDV5(std::move(STDV4));
42
43     for (size_t i = 0; i < V5.size(); ++i)
44         EXPECT_EQ(V5[i], STDV5[i]);
45 }
46 TEST(Manual, Assign)
47 {
48     deque<int> D;
49     for (int i = 0; i < 40; ++i){
50         D.push_back(rnd());
51         if (rnd() % 12)
52             D.push_front(rnd());
53     }
54
55     vector<long> V1, V2, V3;
56     std::vector<long> STDV1, STDV2, STDV3;
57     V1.assign(D.begin(), D.end());
58     STDV1.assign(D.begin(), D.end());
59     EXPECT_EQ(V1, STDV1);
60
61     int a = rnd() % 500;
62     int b = rnd();
63     V2.assign(a, b);
64     STDV2.assign(a, b);

```

```

65     EXPECT_EQ(V2, STDV2);
66
67     V3 = V1;
68     STDV3 = STDV1;
69     EXPECT_EQ(V1, STDV1);
70     EXPECT_EQ(V1, V3);
71     EXPECT_EQ(V3, STDV3);
72
73     V2.assign(V1.begin(), V1.end() - 1);
74     STDV2.assign(V1.begin(), V1.end() - 1);
75     EXPECT_NE(V2, V1);
76     EXPECT_EQ(V2, STDV2);
77     return; //TODO fix memory leak in move assign
78     V3 = std::move(V2);
79     STDV3 = std::move(STDV2);
80     EXPECT_EQ(V3, STDV3);
81 }
82
83 TEST(Manual, ElementAccess)
84 {
85     deque<int> D;
86     for (int i = 0; i < 40; ++i){
87         D.push_back(rnd());
88         if (rnd() % 12)
89             D.push_front(rnd());
90     }
91
92     vector<long> V1, V2, V3;
93     std::vector<long> STDV1, STDV2, STDV3;
94     V1.assign(D.begin(), D.end());
95     STDV1.assign(D.begin(), D.end());
96     EXPECT_EQ(V1, STDV1);
97
98     for (size_t i = 0; i < V1.size(); ++i){
99         EXPECT_EQ(V1[i], STDV1[i]);
100        EXPECT_EQ(V1.at(i), STDV1.at(i));
101    }
102    EXPECT_EQ(V1.front(), STDV1.front());
103    EXPECT_EQ(V1.back(), STDV1.back());
104
105    for (size_t i = 0; i < V1.size(); ++i){
106        +++V1[i];
107        +++STDV1[i];
108    }
109    for (size_t i = 0; i < V1.size(); ++i){
110        EXPECT_EQ(V1[i], STDV1[i]);
111        EXPECT_EQ(V1.at(i), STDV1.at(i));
112    }
113 }
114
115
116 TEST(Iterators, Forward)
117 {

```

```

118     deque<int> D;
119     for (int i = 0; i < 40; ++i){
120         D.push_back(rnd());
121         if (rnd() % 12)
122             D.push_front(rnd());
123     }
124
125     vector<long> V1, V2, V3;
126     std::vector<long> STDV1, STDV2, STDV3;
127     V1.assign(D.begin(), D.end());
128     STDV1.assign(D.begin(), D.end());
129     EXPECT_EQ(V1, STDV1);
130
131     auto iter_1 = V1.begin();
132     auto iter_2 = STDV1.begin();
133
134     while (iter_1 != V1.end()){
135         EXPECT_EQ(*(iter_1++), *(iter_2++));
136     }
137 }
138
139 TEST(Iterators, Backward)
140 {
141     deque<int> D;
142     for (int i = 0; i < 40; ++i){
143         D.push_back(rnd());
144         if (rnd() % 12)
145             D.push_front(rnd());
146     }
147
148     vector<long> V1, V2, V3;
149     std::vector<long> STDV1, STDV2, STDV3;
150     V1.assign(D.begin(), D.end());
151     STDV1.assign(D.begin(), D.end());
152     EXPECT_EQ(V1, STDV1);
153
154     auto iter_1 = V1.end();
155     auto iter_2 = STDV1.end();
156
157     while (iter_1 != V1.begin()){
158         EXPECT_EQ(*(--iter_1), *(--iter_2));
159     }
160 }
161
162
163
164 TEST(Iterators, RandomAccess)
165 {
166     deque<long> D;
167     for (int i = 0; i < 1000 + rnd() % 3000; ++i){
168         D.push_back(rnd());
169         if (rnd() % 2)
170             D.push_front(rnd());

```

```

171     }
172     vector<long> V1, V2, V3;
173     std::vector<long> STDV1, STDV2, STDV3;
174     V1.assign(D.begin(), D.end());
175     STDV1.assign(D.begin(), D.end());
176     EXPECT_EQ(V1, STDV1);
177
178
179     for (int i = 0; i < 100; ++i){
180         int a = rnd() % (V1.size() / 2 - 1) + 1;
181         int b = -(rnd() % (V1.size() / 2 - 1) + 1);
182         auto iter = V1.begin();
183         iter += a;
184         EXPECT_EQ(*(iter), V1[a]);
185         auto iter2 = V1.end();
186         EXPECT_EQ(*(iter2 - a), V1[V1.size() - a]);
187         iter2 -= b;
188     }
189 }
190
191
192 TEST(Manual, Capacity)
193 {
194     deque<long> D;
195     for (int i = 0; i < 1000 + rnd() % 3000; ++i){
196         D.push_back(rnd());
197         if (rnd() % 2)
198             D.push_front(rnd());
199     }
200     vector<long> V1, V2, V3;
201     std::vector<long> STDV1, STDV2, STDV3;
202     V1.assign(D.begin(), D.end());
203     STDV1.assign(D.begin(), D.end());
204     EXPECT_EQ(V1, STDV1);
205
206     EXPECT_EQ(V1.size(), STDV1.size());
207     //EXPECT_EQ(V1.max_size(), STDV1.max_size());
208     int a = rnd() % 50000;
209     V1.reserve(a);
210     EXPECT_EQ(V1.capacity(), a);
211     V1.shrink_to_fit();
212     STDV1.shrink_to_fit();
213
214     EXPECT_EQ(V1, STDV1);
215 }
216
217 TEST(Manual, Modifiers)
218 {
219     deque<long> D;
220     for (int i = 0; i < 1000 + rnd() % 3000; ++i){
221         D.push_back(rnd());
222         if (rnd() % 2)
223             D.push_front(rnd());

```

```

224     }
225     vector<long> V1, V2, V3;
226     std::vector<long> STDV1, STDV2, STDV3;
227     V1.assign(D.begin(), D.end());
228     STDV1.assign(D.begin(), D.end());
229     EXPECT_EQ(V1, STDV1);
230
231     V2 = V1;
232     STDV2 = STDV1;
233     V2.clear();
234     STDV2.clear();
235     EXPECT_EQ(V2, STDV2);
236
237     int a = rnd() % 200;
238     int b = rnd();
239     V2 = V1;
240     STDV2 = STDV1;
241
242     /*          //TODO write insert and tests for insert() (and emplace() and
erase())
243     V2.insert(V2.begin() + a, b);
244     STDV2.insert(STDV2.begin() + a, b);
245     EXPECT_EQ(V2, STDV2);
246     EXPECT_NE(V2, V1);
247     */
248
249     V2.push_back(b);
250     STDV2.push_back(b);
251
252     EXPECT_EQ(V2, STDV2);
253     EXPECT_NE(V2, V1);
254     V2.pop_back();
255     STDV2.pop_back();
256     EXPECT_EQ(V2, V1);
257
258     //V2.resize(V2.size() + a);          //TODO fix resize
259     //STDV2.resize(STDV2.size() + a);
260     EXPECT_EQ(V2, STDV2);
261
262     V2.resize(V2.size() + a, b);
263     STDV2.resize(STDV2.size() + a, b);
264     //EXPECT_EQ(V2, STDV2);
265
266     V2 = V1;
267     STDV2 = STDV1;
268
269     a = rnd() % 600;
270     V2.resize(V2.size() - a);
271     STDV2.resize(STDV2.size() - a);
272     // EXPECT_EQ(V2, STDV2);
273
274     V2 = V1;
275     STDV2 = STDV1;

```



```
276
277     V2[a] = b;
278     STDV2[a] = b;
279
280     swap(V1, V2);
281     std::swap(STDV1, STDV2);
282     EXPECT_EQ(V1, STDV1);
283     EXPECT_EQ(V2, STDV2);
284     EXPECT_NE(V1, V2);
285
286
287 }
```

Приложение В

В.1 Файл deque.hpp

```
1 #ifndef DEQUE_HPP
2 #define DEQUE_HPP
3
4 #include <iostream>
5 #include <cassert>
6 #include <algorithm>
7
8
9 #ifndef NDEBUG                                     //WARNING: debug features will
    fail with types smaller than int
10
11 using poison_t = int;
12 static constexpr int POISON = 0xDEADC0DE;
13 template<typename T>
14 void fillPoison(T* data) {
15     std::fill(reinterpret_cast<poison_t*>(data), reinterpret_cast<poison_t*>(
        data) + sizeof(*data) / sizeof(poison_t), POISON);
16 }
17
18 template<typename T>
19 bool isPoisoned(T* data) {
20     return (reinterpret_cast<poison_t*>(data)[0] == POISON);
21 }
22
23 enum class health_error { none, size, empty, nooverflow, overflow };
24
25 #define DEQUE_CHECK(v) { \
26     health_error error = HealthCheck(); \
27     if (error != health_error::none) \
28     { \
29         std::cerr << "Deque "; \
30         if (error == health_error::size) \
31             std::cerr << "size"; \
32         if (error == health_error::empty) \
33             std::cerr << "empty"; \
34         if (error == health_error::overflow) \
35             std::cerr << "overflow"; \
36         if (error == health_error::nooverflow) \
37             std::cerr << "nooverflow"; \
38         std::cerr << " error in " << __func__ << " at " << __LINE__ << "\n"; \
39         std::cerr.flush(); \
40         dump(std::cerr); \
41         std::exit(0); \
42     } }
43
44 #else
45
46 #define DEQUE_CHECK(v)
47
```

```

48 #endif
49
50
51
52
53 template<typename T>
54 class deque{
55 private:
56     T* data;
57     size_t capacity_;           // Capacity is a number power of two - 1; thus
    pos & capacity_ == pos % capacity_ <=> it takes into account overflow of a
    tip of deque
58     size_t begin_;             // id of a first element (can be smaller than
    end_)
59     size_t end_;               // id of the last element
60     size_t size_;             // independent counter of size of deque
61
62 public:
63
64     //=====
65     // Interface functions
66
67     deque( size_t size = 0 );
68     deque( const deque &other );
69     deque( deque &&other );
70
71     ~deque() { delete[] data; }
72
73     void push_back( const T& val );
74     T pop_back();
75     void push_front( const T& val );
76     T pop_front();
77     deque& operator=( const deque &other );
78     deque& operator=( deque &&other );
79
80     bool operator==( const deque &other ) const;
81     bool operator!=( const deque &other ) const { return !(*this == other); }
82
83     template<class U>
84     bool operator==( const U &other ) const; //
85     DEBUG
86     template<class U>
87     bool operator!=( const U &other ) const { return !(*this == other); }
88
89     void refit( size_t capacity_ = -1 );           //refit() fits data
    to the len OR reallocates memory
90
91     void insert( size_t pos, const T& value );
92     T erase( size_t pos );
93
94     T& operator[]( size_t pos ) { return data[(pos + begin_) &
    capacity_]; }
95     const T& operator[]( size_t pos ) const { return data[(pos + begin_) &

```

```

capacity_]; }

95
96     size_t size() const { return size_; }
97
98
99     #ifndef NDEBUG
100     //=====
101     // Debug funcs
102
103
104     void dump(std::ostream &out) {
105         bool flag = true;
106         out << "capacity = " << this->capacity_ << '\n';
107         out << "size = " << this->size() << '\n';
108         out << "begin = " << this->begin_ << '\n';
109         out << "end = " << this->end_ << '\n';
110         if (!data){
111             out << "EMPTY!\n";
112             return;
113         }
114         for (size_t i = 0; i <= capacity_; ++i){
115             if (!flag)
116                 out << ", ";
117             out << "{ ";
118             if (i == begin_)
119                 out << "|beg| ";
120             if (i == end_)
121                 out << "|end| ";
122             out << data[i] << " }";
123             flag = false;
124         }
125         out << '\n';
126     }
127
128     health_error HealthCheck() {
129         if (size_ > capacity_ + 1)
130             return health_error::size;
131
132         if (!size_){
133             for (size_t i = 0; i <= capacity_; ++i)
134                 if (!isPoisoned(&data[i]))
135                     return health_error::empty;
136             return health_error::none;
137         }
138
139         if (begin_ <= end_){
140             for (size_t i = 0; i <= capacity_; ++i){
141                 if (begin_ <= i && i <= end_){
142                     if (isPoisoned(&data[i]))
143                         return health_error::nooverflow;
144                 }
145                 else
146                     if (!isPoisoned(&data[i]))

```

```

147         return health_error::nooverflow;
148     }
149     else
150         for (size_t i = 0; i <= capacity_; ++i){
151             if (end_ < i && i < begin_){
152                 if (!isPoisoned(&data[i]))
153                     return health_error::overflow;
154             }
155             else
156                 if (isPoisoned(&data[i]))
157                     return health_error::overflow;
158         }
159     return health_error::none;
160 }
161 #endif
162
163
164
165
166 //=====
167 // Iterators
168
169 struct Iterator{
170
171     private:
172         size_t id;                // id in data_
173         size_t pos;              // pos in deque
174         const deque *this_;
175
176
177     public:
178         using iterator_category = std::random_access_iterator_tag;
179         using difference_type   = std::ptrdiff_t;
180         using value_type        = T;
181         using pointer            = T*;
182         using reference          = T&;
183
184         Iterator( size_t id, size_t pos, const deque<T>* this_ = nullptr ) :
185             id(id), pos(pos), this_(this_) {}
186         Iterator( const Iterator& other ) : id(other.id), pos(other.pos), this_
187             (other.this_) {}
188
189         bool operator==( const Iterator& other ) const { return pos == other.
190             pos; }
191         bool operator!=( const Iterator& other ) const { return pos != other.
192             pos; }
193
194         bool operator< ( const Iterator& other ) const { return pos < other.
195             pos; }
196         bool operator> ( const Iterator& other ) const { return pos > other.
197             pos; }
198         bool operator<= ( const Iterator& other ) const { return pos <= other.
199             pos; }

```

```

193     bool operator>=( const Iterator& other ) const { return pos >= other.
pos; }
194
195     T& operator*()                { assert(id <= this_>capacity_); return
this_>data[id]; }
196     const T& operator*() const { assert(id <= this_>capacity_); return
this_>data[id]; }
197
198
199     Iterator operator++(){
200         ++pos;
201         ++id;
202         id &= this_>capacity_ ;
203         return (*this);
204     }
205
206     Iterator operator++(int){
207         Iterator result(*this);
208         ++pos;
209         ++id;
210         id &= this_>capacity_ ;
211         return result;
212     }
213
214     Iterator operator--(){
215         --pos;
216         --id;
217         id &= this_>capacity_ ;
218         return (*this);
219     }
220
221     Iterator operator--(int){
222         Iterator result(*this);
223         --pos;
224         --id;
225         id &= this_>capacity_ ;
226         return result;
227     }
228
229     Iterator operator+(long long int n) const {
230         Iterator result(*this);
231         result.pos += n;
232         result.id += n;
233         result.id &= this_>capacity_ ;
234         return result;
235     }
236
237     long long operator-(const Iterator& other) const {
238         return pos - other.pos;
239     }
240
241     Iterator operator-(long long int n) const {
242         Iterator result(*this);

```

```

243         result.pos -= n;
244         result.id -= n;
245         result.id &= this_ ->capacity_;
246         return result;
247     }
248
249     Iterator operator+=(long long int n){
250         pos += n;
251         id += n;
252         id &= this_ ->capacity_;
253         return (*this);
254     }
255
256     Iterator operator-=(long long int n){
257         pos -= n;
258         id -= n;
259         id &= this_ ->capacity_;
260         return (*this);
261     }
262
263 };
264
265
266 Iterator begin() { return Iterator( begin_, 0, this); }
267 Iterator end()   { return Iterator( end_ + 1, size(), this); }
268
269 Iterator begin() const { return Iterator( begin_, 0, this); }
270 Iterator end()   const { return Iterator( end_ + 1, size(), this); }
271
272 };
273
274
275
276
277 template<typename T>
278 deque<T>::deque( const deque &other )
279     : data(nullptr), capacity_(other.capacity_), begin_(other.begin_), end_(
280         other.end_), size_(other.size_) {
281
282     if (capacity_){
283         data = new T[capacity_ + 1];
284         std::copy(other.data, other.data + capacity_ + 1, data);
285     }
286     DEQUE_CHECK(*this)
287 }
288
289 template<typename T>
290 deque<T>::deque( deque &&other )
291     : capacity_(other.capacity_), data(other.data), begin_(other.begin_), end_(
292         other.end_), size_(other.size_) {
293
294     other.data = nullptr;

```

```

294     other.capacity_ = 0;
295     other.begin_     = 0;
296     other.end_       = 0;
297     other.size_      = 0;
298     DEQUE_CHECK(*this)
299 }
300
301
302 template<typename T>
303 deque<T>::deque( size_t size)
304     : data(nullptr), capacity_(0), begin_(0), end_(0), size_(0) {
305
306     if (!size)
307         return;
308     size_t i = 2;
309     while (i < size)
310         i <<= 1;
311     capacity_ = i - 1;
312     data = new T[capacity_ + 1];
313
314     #ifndef NDEBUG
315     for (size_t j = 0; j <= capacity_; ++j)
316         fillPoison(&data[j]);
317     #endif
318     DEQUE_CHECK(*this)
319 }
320
321
322
323 template<typename T>
324 void deque<T>::push_back(const T &val) {
325     refit();
326     if (!size_){
327         data[0] = val;
328     }
329     else {
330         end_ = (end_ + 1) & capacity_;
331         data[end_] = val;
332     }
333     ++size_;
334     DEQUE_CHECK(*this)
335 }
336
337
338 template<typename T>
339 T deque<T>::pop_back() {
340     assert(size_);
341     --size_;
342     size_t return_pos = end_;
343     end_ = (end_ - 1) & capacity_;
344
345     #ifndef NDEBUG
346     T tmp = data[return_pos];

```



```

347     fillPoison(&data[return_pos]);
348     return tmp;
349 #else
350
351     return data[return_pos];
352 #endif
353
354     DEQUE_CHECK(*this)
355 }
356
357
358 template<typename T>
359 void deque<T>::push_front(const T &val) {
360     refit();
361     if (!size_){
362         data[0] = val;
363     }
364     else {
365         begin_ = (begin_ - 1) & capacity_;
366         data[begin_] = val;
367     }
368     ++size_;
369     DEQUE_CHECK(*this)
370 }
371
372
373
374 template<typename T>
375 T deque<T>::pop_front() {
376     assert(size_);
377     --size_;
378     size_t return_pos = begin_;
379     begin_ = (begin_ + 1) & capacity_;
380
381     #ifndef NDEBUG
382     T tmp = data[return_pos];
383     fillPoison(&data[return_pos]);
384     return tmp;
385 #else
386
387     return data[return_pos];
388 #endif
389     DEQUE_CHECK(*this)
390 }
391
392
393 template<typename T>
394 void deque<T>::insert(size_t pos, const T& value) {
395     size_t new_capacity = capacity_;
396     if (size_ == capacity_ + 1)
397         new_capacity = ((capacity_ + 1) << 1) - 1;
398     T* new_data = new T[new_capacity + 1];
399

```

```

400     #ifndef NDEBUG
401     for (size_t k = 0; k <= new_capacity; ++k)
402         fillPoison(&new_data[k]);
403     #endif
404
405     std::copy(begin(), begin() + pos, new_data);
406     new_data[pos] = value;
407     std::copy(begin() + pos, end(), new_data + pos + 1);
408
409     begin_ = 0;
410     end_ = size_;
411     ++size_;
412     capacity_ = new_capacity;
413     if (data)
414         delete[] data;
415     data = new_data;
416
417     DEQUE_CHECK(*this)
418 }
419
420
421 template<typename T>
422 T deque<T>::erase(size_t pos) {
423     T* new_data = new T[capacity_ + 1];
424     T result = data[pos];
425     #ifndef NDEBUG
426     for (size_t k = 0; k <= capacity_; ++k)
427         fillPoison(&new_data[k]);
428     #endif
429
430     std::copy(begin(), begin() + pos, new_data);
431     std::copy(begin() + pos + 1, end(), new_data + pos);
432
433     begin_ = 0;
434     end_ = size_ - 2;
435     --size_;
436
437     if (data)
438         delete[] data;
439     data = new_data;
440
441     DEQUE_CHECK(*this)
442     return result;
443 }
444
445
446 template<typename T>
447 void deque<T>::refit(size_t new_capacity) {
448     if (new_capacity == -1 && ((size_ == capacity_ + 1) || capacity_ == 0))
449         new_capacity = ((capacity_ + 1) << 1) - 1;
450     else if (new_capacity == -1)
451         return;
452     else {

```

```

453         size_t i = 2;
454         while (i < new_capacity)
455             i <<= 1;
456         new_capacity = i - 1;
457     }
458     if (new_capacity == capacity_)
459         return;
460
461     T* new_data = new T[new_capacity + 1];
462
463     #ifndef NDEBUG
464     for (size_t k = 0; k <= new_capacity; ++k)
465         fillPoisson(&new_data[k]);
466     #endif
467
468     if (new_capacity < size_){
469         std::copy(begin(), begin() + (new_capacity + 1), new_data);
470         size_ = new_capacity + 1;
471     }
472     else
473         std::copy(begin(), end(), new_data);
474
475     begin_ = 0;
476     if (size_)
477         end_ = size_ - 1;
478     else
479         end_ = 0;
480     capacity_ = new_capacity;
481     delete[] data;
482     data = new_data;
483
484
485
486     DEQUE_CHECK(*this)
487 }
488
489
490 template<typename T>
491 bool deque<T>::operator==( const deque &other ) const {
492     if (size_ != other.size_)
493         return false;
494     for (size_t i = 0; i < size_; ++i)
495         if ((*this)[i] != other[i])
496             return false;
497     return true;
498 }
499
500
501 template<typename T>
502 template<typename U>
503 bool deque<T>::operator==( const U &other ) const {
504     if (size_ != other.size())
505         return false;

```

```

506     for (size_t i = 0; i < size_; ++i)
507         if ((*this)[i] != other[i])
508             return false;
509     return true;
510 }
511
512
513 template<typename T>
514 deque<T>& deque<T>::operator=(const deque &other){
515     capacity_ = other.capacity_;
516     begin_ = other.begin_;
517     end_ = other.end_;
518     size_ = other.size_;
519     if (data)
520         delete [] data;
521     data = new T[capacity_ + 1];
522     std::copy(other.data, other.data + capacity_ + 1, data);
523     DEQUE_CHECK(*this)
524     return *this;
525 }
526
527
528 template<typename T>
529 deque<T>& deque<T>::operator=(deque &&other){
530     capacity_ = std::exchange(other.capacity_, 0);
531     begin_ = std::exchange(other.begin_, 0);
532     end_ = std::exchange(other.end_, 0);
533     size_ = std::exchange(other.size_, 0);
534     data = std::exchange(other.data, nullptr);
535     DEQUE_CHECK(*this)
536     return *this;
537 }
538
539
540 #endif

```

B.2 Файл test-deque.hpp

```

1 #include "deque.hpp"
2
3 #include <random>
4 #include <deque>
5 #include "gtest/gtest.h"
6
7
8 std::mt19937 rnd(179);
9
10 template<typename T>
11 void PushAndPopTest()
12 {
13     std::deque<T> STD1;
14     deque<T> D1;
15     for (int i = 0; i < rnd() % 30000 + 150; ++i){
16         int a = rnd(), b = rnd();

```

```

17         D1.push_back(a);
18         STD1.push_back(a);
19         D1.push_front(b);
20         STD1.push_front(b);
21     }
22     EXPECT_EQ(D1, STD1);
23     for (int i = 0; i < rnd() % 100 + 10; ++i){
24         if (i % 2){
25             D1.pop_front();
26             STD1.pop_front();
27         }
28         else{
29             D1.pop_back();
30             STD1.pop_back();
31         }
32     }
33     EXPECT_EQ(D1, STD1);
34     for (int i = 0; i < D1.size(); ++i)
35         EXPECT_EQ(D1[i], STD1[i]);
36 }
37
38
39
40 template<typename T>
41 void CopyAndMoveTest()
42 {
43     std::deque<T> STD1, STD2;
44     deque<T> D1, D2;
45     for (int i = 0; i < rnd() % 30000 + 150; ++i){
46         T a = rnd(), b = rnd();
47         D1.push_back(a);
48         STD1.push_back(a);
49         D1.push_front(b);
50         STD1.push_front(b);
51     }
52     EXPECT_EQ(D1, STD1);
53     D2 = D1;
54     STD2 = STD1;
55     for (int i = 0; i < rnd() % 100 + 10; ++i){
56         if (i % 2){
57             D2.pop_front();
58             STD2.pop_front();
59         }
60         else{
61             D2.pop_back();
62             STD2.pop_back();
63         }
64     }
65
66     EXPECT_EQ(D2, STD2);
67     EXPECT_EQ(D2, D2);
68     EXPECT_EQ(D1, D1);
69     int a = rnd();

```

```

70     int b = rnd();
71     D2.push_front(a);
72     D2.push_back(b);
73     STD2.push_front(a);
74     STD2.push_back(b);
75
76     deque<T> D3(D2);
77     std::deque<T> STD3(STD2);
78     EXPECT_EQ(D3, STD3);
79     EXPECT_EQ(D3, D2);
80
81     a = rnd();
82     D3[12] = a;
83     STD3[12] = a;
84     EXPECT_EQ(D3, STD3);
85     size_t i = 0;
86     for (auto elem : D3){
87         EXPECT_EQ(elem, STD3[i]);
88         ++i;
89     }
90
91     deque<T> D4(std::move(D2)), D5, D6;
92     std::deque<T> STD4(std::move(STD2)), STD5, STD6;
93
94     D5 = std::move(D3);
95     STD5 = std::move(STD3);
96
97     a = rnd();
98     D5[17] = a;
99     STD5[17] = a;
100    D6 = D5;
101    STD6 = STD5;
102
103    a = rnd();
104    D6.push_front(a);
105    STD6.push_front(a);
106
107
108
109    EXPECT_EQ(D4, STD4);
110    EXPECT_EQ(D5, STD5);
111    EXPECT_EQ(D6, STD6);
112    EXPECT_EQ(D4, D4);
113    EXPECT_NE(D4, D6);
114    deque<T> D7(rnd() % 4999), D8(rnd() % 5000);
115    EXPECT_NE(D4, D7);
116    EXPECT_NE(D4, D8);
117
118 }
119
120
121 template<typename T>
122 void ForwardIteratorTest()

```

```

123 {
124     deque<T> D1;
125     for (int i = 0; i < 1000 + rnd() % 3000; ++i){
126         D1.push_back(static_cast<T>(rnd()));
127         if (rnd() % 2)
128             D1.push_front(static_cast<T>(rnd()));
129     }
130
131     size_t i = 0;
132     for (auto elem : D1){
133         EXPECT_EQ(elem, D1[i]);
134         ++i;
135     }
136     i = 0;
137     for (auto elem : D1){
138         EXPECT_EQ(elem, D1[i]);
139         i++;
140     }
141 }
142
143
144
145 template<typename T>
146 void BackwardIteratorTest()
147 {
148     deque<T> D1;
149     for (int i = 0; i < 1000 + rnd() % 3000; ++i){
150         D1.push_back(static_cast<T>(rnd()));
151         if (rnd() % 2)
152             D1.push_front(static_cast<T>(rnd()));
153     }
154
155     auto iter = D1.end();
156     size_t i = D1.size();
157     while (i){
158         --iter;
159         --i;
160         EXPECT_EQ(*iter, D1[i]);
161     }
162     iter = D1.end();
163     i = D1.size();
164     while (i){
165         iter--;
166         --i;
167         EXPECT_EQ(*iter, D1[i]);
168     }
169 }
170 }
171
172
173 template<typename T>
174 void RandomAccessIteratorTest()
175 {

```

```

176     deque<T> D1;
177     for (int i = 0; i < 1000 + rnd() % 3000; ++i){
178         D1.push_back(static_cast<T>(rnd()));
179         if (rnd() % 2)
180             D1.push_front(static_cast<T>(rnd()));
181     }
182     for (int i = 0; i < 100; ++i){
183         int a = rnd() % (D1.size() / 2 - 1) + 1;
184         int b = -(rnd() % (D1.size() / 2 - 1) + 1);
185         auto iter = D1.begin();
186         iter += a;
187         EXPECT_EQ(*(iter + b), D1[a + b]);
188         auto iter2 = D1.end();
189         iter2 -= b;
190         EXPECT_EQ(*(iter2 - a), D1[D1.size() - a - b]);
191     }
192 }
193
194 template<typename T>
195 void InsertAndEraseTest()
196 {
197     std::deque<T> STD1;
198     deque<T> D1;
199     for (int i = 0; i < rnd() % 8000 + 500; ++i){
200         T a = rnd(), b = rnd();
201         D1.push_back(a);
202         STD1.push_back(a);
203         D1.push_front(b);
204         STD1.push_front(b);
205     }
206
207     EXPECT_EQ(D1, STD1);
208     for (int i = 0; i < 300; ++i){
209         int a = rnd() % (D1.size() / 2 - 1) + 1;
210         T b = rnd();
211         D1.insert(a, b);
212         STD1.insert(STD1.begin() + a, b);
213     }
214     EXPECT_EQ(D1, STD1);
215     for (int i = 0; i < 300; ++i){
216         int a = rnd() % (D1.size() / 2 - 1) + 1;
217         D1.erase(a);
218         STD1.erase(STD1.begin() + a);
219     }
220     EXPECT_EQ(D1, STD1);
221 }
222 }
223
224
225 template<typename T>
226 void RefitTest()
227 {
228     deque<T> D1;

```



```

229     for (int i = 0; i < 1000 + rnd() % 3000; ++i){
230         D1.push_back(rnd());
231         if (rnd() % 2)
232             D1.push_front(static_cast<T>(rnd()));
233     }
234
235     D1.refit(rnd() % 5000);
236     D1.refit(rnd() % 5000);
237     D1.refit(rnd() % 5000);
238     D1.refit(rnd() % 5000);
239 }
240
241
242
243 TEST(Basics, PushAndPop)
244 {
245     for (int p = 0; p < 50; ++p){
246         PushAndPopTest<int>();
247         PushAndPopTest<long>();
248         PushAndPopTest<unsigned long long>();
249         PushAndPopTest<double>();
250         PushAndPopTest<float>();
251
252         #ifdef NDEBUG
253         PushAndPopTest<short>();
254         PushAndPopTest<char>();
255         PushAndPopTest<bool>();
256         #endif
257     }
258 }
259
260 TEST(Basics, CopyAndMove){
261     for (int p = 0; p < 50; ++p){
262         CopyAndMoveTest<int>();
263         CopyAndMoveTest<long>();
264         CopyAndMoveTest<unsigned long long>();
265         CopyAndMoveTest<double>();
266         CopyAndMoveTest<float>();
267
268         #ifdef NDEBUG
269         CopyAndMoveTest<short>();
270         CopyAndMoveTest<char>();
271         CopyAndMoveTest<bool>();
272         #endif
273     }
274 }
275 }
276
277 TEST(Basics, Refit){
278     for (int p = 0; p < 50; ++p){
279         RefitTest<int>();
280         RefitTest<long>();
281         RefitTest<unsigned long long>();

```

```

282     RefitTest<double>();
283     RefitTest<float>();
284     #ifdef NDEBUG
285     RefitTest<short>();
286     RefitTest<char>();
287     RefitTest<bool>();
288     #endif
289 }
290 }
291
292 TEST(Basics , InsertAndErase) {
293     for (int p = 0; p < 20; ++p){
294         InsertAndEraseTest<int>();
295         InsertAndEraseTest<long>();
296         InsertAndEraseTest<unsigned long long>();
297         InsertAndEraseTest<double>();
298         InsertAndEraseTest<float>();
299
300         #ifdef NDEBUG
301         InsertAndEraseTest<short>();
302         InsertAndEraseTest<char>();
303         InsertAndEraseTest<bool>();
304         #endif
305
306     }
307 }
308 }
309
310 TEST(Iterators , ForwardIterator){
311     for (int p = 0; p < 20; ++p){
312         ForwardIteratorTest<int>();
313         ForwardIteratorTest<long>();
314         ForwardIteratorTest<unsigned long long>();
315         ForwardIteratorTest<double>();
316         ForwardIteratorTest<float>();
317
318         #ifdef NDEBUG
319         ForwardIteratorTest<short>();
320         ForwardIteratorTest<char>();
321         ForwardIteratorTest<bool>();
322         #endif
323     }
324 }
325
326 TEST(Iterators , BackwardIterator){
327     for (int p = 0; p < 20; ++p){
328         BackwardIteratorTest<int>();
329         BackwardIteratorTest<long>();
330         BackwardIteratorTest<unsigned long long>();
331         BackwardIteratorTest<double>();
332         BackwardIteratorTest<float>();
333
334         #ifdef NDEBUG

```

```

335         BackwardIteratorTest<short>();
336         BackwardIteratorTest<char>();
337         BackwardIteratorTest<bool>();
338     #endif
339 }
340 }
341
342 TEST(Iterators , RandomAccessIterator){
343     for (int p = 0; p < 20; ++p){
344         RandomAccessIteratorTest<int>();
345         RandomAccessIteratorTest<long>();
346         RandomAccessIteratorTest<unsigned long long>();
347         RandomAccessIteratorTest<double>();
348         RandomAccessIteratorTest<float>();
349     }
350 }
351 }

```

Приложение Г

Г.1 Файл linkedlist.hpp

```
1 #ifndef LINKEDLIST_HPP
2 #define LINKEDLIST_HPP
3
4 #include <iostream>
5 #include <cassert>
6
7
8 template<class T, class U = T>
9 T exchange(T& obj, U&& new_value)
10 {
11     T old_value = std::move(obj);
12     obj = std::forward<U>(new_value);
13     return old_value;
14 }
15
16 //=====
17 // Object pool
18
19 template<typename Data>
20 class ObjPool{
21 public:
22
23     ObjPool(const ObjPool &other)
24     {
25         capacity = other.capacity;
26         last_free = other.last_free;
27         data = new Node[capacity];
28         std::copy(other.data, other.data + other.capacity, data);
29     }
30
31     ObjPool(ObjPool &&other)
32     {
33         capacity = exchange(other.capacity, 0);
34         last_free = exchange(other.last_free, -1);
35         data = exchange(other.data, nullptr);
36     }
37
38     ObjPool& operator=(const ObjPool &other)
39     {
40         capacity = other.capacity;
41         last_free = other.last_free;
42         if (data)
43             delete [] data;
44         data = new Node[capacity];
45         std::copy(other.data, other.data + other.capacity, data);
46         return (*this);
47     }
48
49     ObjPool& operator=(ObjPool &&other)
```

```

50 {
51     capacity = exchange(other.capacity, 0);
52     last_free = exchange(other.last_free, -1);
53     if (data)
54         delete[] data;
55     data = exchange(other.data, nullptr);
56     return (*this);
57 }
58
59
60 ObjPool(size_t capacity=1) : capacity(capacity)
61 {
62     data = new Node [capacity];
63     for (size_t i=0; i<capacity - 1; ++i)
64         data[i].next = i + 1;
65     data[capacity - 1].next = -1;
66
67     last_free = 0;
68 }
69 ~ObjPool()
70 {
71     delete [] data;
72 }
73
74 size_t alloc()
75 {
76     refit();
77     size_t result = last_free;
78     last_free = data[last_free].next;
79     return result;
80 }
81
82 Data *get(size_t id) const
83 {
84     assert(id != -1);
85     assert(id < capacity);
86     return &data[id].val;
87 }
88
89 void free(size_t id)
90 {
91     data[id].next = last_free;
92     last_free = id;
93 }
94
95 void print(std::ostream& out)
96 {
97     for (size_t id = last_free; id != -1; id = data[id].next)
98     {
99         out << "(" << id << ") -> ";
100     }
101     out << '\n';
102 }

```

```

103
104
105 private:
106     struct Node{
107         size_t next;
108         Data val;
109     };
110
111     Node *data;
112     size_t capacity;
113     size_t last_free;
114
115     void refit()
116     {
117         if (last_free != -1)
118             return;
119         Node *nbuf = new Node[capacity * 2];
120         std::copy(data, data + capacity, nbuf);
121         assert(data);
122         delete [] data;
123         data = nbuf;
124         capacity *= 2;
125         for (size_t i = capacity / 2; i < capacity - 1; ++i)
126             data[i].next = i + 1;
127         data[capacity - 1].next = -1;
128         last_free = capacity / 2;
129     }
130 };
131
132 //=====
133 // LinkedList
134
135 template<typename T>
136 class linkedList {
137 public:
138     struct Node{
139         size_t next_;
140         T val_;
141     };
142
143 private:
144     size_t head_;
145     size_t size_;
146     ObjPool<Node> pool;
147
148 public:
149     //=====
150     // Interface functions
151
152     linkedList() : head_(-1), size_(0) {}
153     linkedList( const linkedList &other ) = default;
154     linkedList( linkedList &&other ) : head_(other.head_), size_(other.size_)
155     { pool = std::move(other.pool); other.head_ = -1; other.size_ = 0; }

```

```

155
156     linkedList& operator=( const linkedList &other ) { head_ = other.head_;
size_ = other.size_; pool = other.pool; return *this; }
157     linkedList& operator=( linkedList &&other ) { head_ = other.head_; size_
= other.size_; pool = std::move(other.pool); other.head_ = -1; other.size_
= 0; return *this; }
158
159
160     void insert( const T &val ) { insert(0, val); };
161     void insert( size_t n, const T &val );
162     T erase( size_t n = 0 );
163
164     size_t size() const { return size_; }
165
166     template<class Container>
167     bool operator==( const Container &other ) const;
168     template<class Container>
169     bool operator!=( const Container &other ) const { return !(*this == other)
; }
170
171     void dump(std::ostream &out) const;           //DEBUG
172
173     T& operator [] ( size_t n );
174     const T& operator [] ( size_t n ) const;
175
176     //=====
177     // Iterators
178
179     struct Iterator {
180         using iterator_category = std::forward_iterator_tag;
181         using difference_type    = std::ptrdiff_t;
182         using value_type         = Node;
183
184         Iterator( size_t id = -1, const ObjPool<Node> *pool = nullptr ) :
pool_(pool), id_(id) {};
185         Iterator( const Iterator &other ) = default;
186
187         bool operator==( const Iterator &other ) const { return id_ == other.
id_; }
188         bool operator!=( const Iterator &other ) const { return id_ != other.
id_; }
189
190         T operator*() { assert(id_ != -1); return pool_>get(id_)>val_; }
191         const T operator*() const { assert(id_ != -1); return pool_>get(id_
)>val_; }
192
193         Iterator operator++() {
194             id_ = pool_>get(id_)>next_;
195             return *this;
196         }
197
198         Iterator operator++(int) {
199             Iterator result(*this);

```

```

200         id_ = pool_>get(id_)>next_;
201         return result;
202     }
203
204
205     private:
206         const ObjPool<Node> *pool_;
207         size_t id_;
208
209     };
210
211     Iterator begin() const { return Iterator(head_, &pool); }
212     Iterator end()   const { return Iterator(-1, &pool); }
213 };
214
215 template<typename T>
216 void LinkedList<T>::insert(size_t n, const T &val) {
217     assert(n <= size_);
218     ++size_;
219     if (n == 0){
220         size_t id = pool.alloc();
221         pool.get(id)>next_ = head_;
222         pool.get(id)>val_ = val;
223         head_ = id;
224         return;
225     }
226     size_t id = head_;
227     for (size_t i = 0; i < n - 1; ++i)
228         id = pool.get(id)>next_;
229
230     size_t new_id = pool.alloc();
231     Node *v = pool.get(id), *new_ = pool.get(new_id);
232     new_>next_ = v->next_;
233     new_>val_ = val;
234     v->next_ = new_id;
235 }
236
237 template<typename T>
238 T LinkedList<T>::erase(size_t n) {
239     assert(n < size_);
240     --size_;
241     if (n == 0){
242         Node *v = pool.get(head_);
243         head_ = v->next_;
244         return v->val_;
245     }
246     size_t id = head_;
247     for (size_t i = 0; i < n - 1; ++i)
248         id = pool.get(id)>next_;
249     Node *v = pool.get(id);
250     size_t result_id = v->next_;
251     v->next_ = pool.get(result_id)>next_;
252     T result_val = pool.get(result_id)>val_;

```



```

253     pool.free(result_id);
254     return result_val;
255 }
256
257 template<typename T>
258 void linkedList<T>::dump(std::ostream &out) const {
259     out << "head = " << head_ << '\n';
260     out << "size = " << size_ << '\n';
261     for (auto elem : *this)
262         out << "( " << elem << " ) ";
263     out << '\n';
264 }
265
266 template<typename T>
267 template<class Container>
268 bool linkedList<T>::operator==(const Container &other) const {
269     if (size() != other.size()){
270         return false;
271     }
272
273     Iterator iter_1 = begin();
274     auto iter_2 = other.begin();
275     while (iter_1 != end()) {
276         if (*iter_1 != *iter_2)
277             return false;
278         ++iter_1; ++iter_2;
279     }
280     return true;
281 }
282
283
284 template<typename T>
285 T& linkedList<T>::operator[](size_t n) {
286     size_t id = head_;
287     for (size_t i = 0; i < n; ++i)
288         id = pool.get(id)->next_;
289     return pool.get(id)->val_;
290 }
291
292 template<typename T>
293 const T& linkedList<T>::operator[](size_t n) const {
294     size_t id = head_;
295     for (size_t i = 0; i < n; ++i)
296         id = pool.get(id)->next_;
297     return pool.get(id)->val_;
298 }
299
300 #endif

```

Г.2 Файл test-linkedlist.hpp

```

1 #include "linkedlist.hpp"
2 #include <vector>
3 #include <random>

```

```

4 #include "gtest/gtest.h"
5
6 std::mt19937 rnd(179);
7
8 TEST(Basics, insertOnTips) {
9     for (int k = 0; k < 1000; ++k){
10         linkedList<int> L1;
11         std::vector<int> V1;
12         for (int i = 0; i < rnd() % 100 + 10; ++i){
13             int a = rnd();
14             L1.insert(a);
15             V1.insert(V1.begin(), a);
16         }
17         EXPECT_EQ(L1, V1);
18
19         linkedList<int> L2;
20         std::vector<int> V2;
21         for (int i = 0; i < rnd() % 100 + 10; ++i){
22             int a = rnd();
23             L2.insert(i, a);
24             V2.insert(V2.begin() + i, a);
25         }
26         EXPECT_EQ(L2, V2);
27     }
28 }
29 TEST(Basics, insertAndEraseOnTips){
30     for (int k = 0; k < 1000; ++k){
31         linkedList<int> L1;
32         std::vector<int> V1;
33         size_t i;
34         for (i = 0; i < rnd() % 100 + 25; ++i){
35             int a = rnd();
36             L1.insert(a);
37             V1.insert(V1.begin(), a);
38         }
39         EXPECT_EQ(L1, V1);
40         size_t end = i;
41         for (; i < end + rnd() % 100 + 25; ++i){
42             int a = rnd();
43             L1.insert(i, a);
44             V1.insert(V1.begin() + i, a);
45         }
46         EXPECT_EQ(L1, V1);
47
48         for (int j = 0; j < 10; ++j){
49             int a = rnd() % 10 + 5;
50             L1.erase(a);
51             V1.erase(V1.begin() + a);
52         }
53         L1.erase(0);
54         V1.erase(V1.begin());
55         EXPECT_EQ(L1, V1);
56     }

```

```

57 }
58
59 TEST(Basics, operators){
60     for (int k = 0; k < 1000; ++k){
61         linkedList<int> L1;
62         std::vector<int> V1;
63         size_t i;
64         for (i = 0; i < rnd() % 100 + 25; ++i){
65             int a = rnd();
66             L1.insert(a);
67             V1.insert(V1.begin(), a);
68         }
69         EXPECT_EQ(L1, V1);
70         size_t end = i;
71         for (; i < end + rnd() % 100 + 25; ++i){
72             int a = rnd();
73             L1.insert(i, a);
74             V1.insert(V1.begin() + i, a);
75         }
76         EXPECT_EQ(L1, V1);
77
78         for (size_t j = 0; j < L1.size(); ++j)
79             EXPECT_EQ(L1[j], V1[j]);
80
81         for (size_t j = 0; j < L1.size(); ++j){
82             ++L1[j];
83             ++V1[j];
84             EXPECT_EQ(L1[j], V1[j]);
85         }
86
87         EXPECT_TRUE(L1 == V1);
88         EXPECT_FALSE(L1 != V1);
89     }
90 }
91
92 TEST(Basics, copyAndMove){
93     for (int k = 0; k < 1000; ++k){
94         linkedList<int> L1;
95         std::vector<int> V1;
96         size_t i;
97         for (i = 0; i < rnd() % 1000 + 25; ++i){
98             int a = rnd();
99             L1.insert(a);
100             V1.insert(V1.begin(), a);
101         }
102         EXPECT_EQ(L1, V1);
103         size_t end = i;
104         for (; i < end + rnd() % 1000 + 25; ++i){
105             int a = rnd();
106             L1.insert(i, a);
107             V1.insert(V1.begin() + i, a);
108         }
109         EXPECT_EQ(L1, V1);

```

```
110     linkedList<int> L2(L1), L3, L5;
111     linkedList<int> L4(std::move(L1));
112     L3 = L1;
113     EXPECT_EQ(L2, V1);
114     L4.insert(10, rnd());
115     L4.insert(10, rnd());
116     L3 = L4;
117     L5 = L4;
118     L5[10] = 10;
119     EXPECT_NE(L4, L2);
120     EXPECT_NE(L4, L5);
121 }
122 }
```

Приложение Д

Д.1 Файл treap.hpp

```
1 #ifndef TREAP_HPP
2 #define TREAP_HPP
3
4 #include <cstdint>
5 #include <cstddef>
6 #include <random>
7 #include <vector>
8 #include <iostream>
9 #include <cassert>
10 #include <set>
11
12 std::mt19937 rnd(179);
13
14 template<class T, class U = T>
15 T exchange(T& obj, U&& new_value)
16 {
17     T old_value = std::move(obj);
18     obj = std::forward<U>(new_value);
19     return old_value;
20 }
21
22 //=====
23 // Object pool
24
25 template<typename Data>
26 class ObjPool{
27 public:
28
29     ObjPool(const ObjPool &other)
30     {
31         capacity = other.capacity;
32         last_free = other.last_free;
33         data = new Node[capacity];
34         std::copy(other.data, other.data + other.capacity, data);
35     }
36
37     ObjPool(ObjPool &&other)
38     {
39         capacity = exchange(other.capacity, 0);
40         last_free = exchange(other.last_free, -1);
41         data = exchange(other.data, nullptr);
42     }
43
44     ObjPool& operator=(const ObjPool &other)
45     {
46         capacity = other.capacity;
47         last_free = other.last_free;
48         if (data)
49             delete [] data;
```

```

50     data = new Node[capacity];
51     std::copy(other.data, other.data + other.capacity, data);
52     return (*this);
53 }
54
55 ObjPool& operator=(ObjPool &&other)
56 {
57     capacity = exchange(other.capacity, 0);
58     last_free = exchange(other.last_free, -1);
59     if (data)
60         delete[] data;
61     data = exchange(other.data, nullptr);
62     return (*this);
63 }
64
65
66 ObjPool(size_t capacity=1) : capacity(capacity)
67 {
68     data = new Node [capacity];
69     for (size_t i=0; i<capacity - 1; ++i)
70         data[i].next = i + 1;
71     data[capacity - 1].next = -1;
72
73     last_free = 0;
74 }
75
76 ~ObjPool()
77 {
78     delete [] data;
79 }
80
81 size_t alloc()
82 {
83     refit();
84     size_t result = last_free;
85     last_free = data[last_free].next;
86     return result;
87 }
88
89 Data *get(size_t id) const
90 {
91     assert(id != -1);
92     assert(id < capacity);
93     return &data[id].val;
94 }
95
96 void free(size_t id)
97 {
98     data[id].next = last_free;
99     last_free = id;
100 }
101
102 void print(std::ostream& out)

```

```

103     {
104         for (size_t id = last_free; id != -1; id = data[id].next)
105         {
106             out << "(" << id << ") -> ";
107         }
108         out << '\n';
109     }
110
111
112 private:
113     struct Node{
114         size_t next;
115         Data val;
116         // ~Node() {};
117     };
118
119     Node *data;
120     size_t capacity;
121     size_t last_free;
122
123     void refit()
124     {
125         if (last_free != -1)
126             return;
127         Node *nbuf = new Node[capacity * 2];
128         std::copy(data, data + capacity, nbuf);
129         assert(data);
130         delete [] data;
131         data = nbuf;
132         capacity *= 2;
133         for (size_t i = capacity / 2; i < capacity - 1; ++i)
134             data[i].next = i + 1;
135         data[capacity - 1].next = -1;
136         last_free = capacity / 2;
137     }
138 };
139
140 //=====
141 // Treap
142
143 #ifndef NDEBUG
144 #define TREAP_CHECK(v) { \
145     if (!graph_check(v)) \
146     { \
147         std::cerr << "Treap error in " << __func__ << " at " << __LINE__ << "\n"; \
148         std::cerr.flush(); \
149         std::exit(0); \
150     } }
151 #else
152 #define TREAP_CHECK(v) {}
153 #endif
154

```

```

155 template<typename Key, typename Data>
156 class Treap
157 {
158 private:
159     struct Node
160     {
161         Key x;
162         size_t prior;
163         Data val = Data();
164         size_t parent;
165         size_t left, right;
166         size_t size;
167
168         Node() : prior(rnd()), parent(-1), left(-1), right(-1),
169 size(1) {}
170         Node(Key x, Data val) : x(x), prior(rnd()), val(val), parent(-1), left
171 (-1), right(-1), size(1) {}
172
173         ~Node() {};
174     };
175
176     size_t root_id;
177     ObjPool<Node> pool;
178
179 public:
180     struct Iterator
181     {
182         using iterator_category = std::random_access_iterator_tag;
183         using difference_type = std::ptrdiff_t;
184         using value_type = Node;
185
186         Iterator( size_t id=-1, const Treap* this_=nullptr ) : id(id), this_(
187 this_), pos(0) {} // WARNING: don't use for non-begin iterator, pos
188 will invalidate
189         Iterator( const Iterator &other ) = default;
190
191         bool operator==( const Iterator other ) const { return id == other.id;
192 }
193         bool operator!=( const Iterator other ) const { return id != other.id;
194 }
195
196         bool operator< ( const Iterator other ) const { return pos < other.
197 pos; }
198         bool operator> ( const Iterator other ) const { return pos > other.
199 pos; }
200         bool operator<=( const Iterator other ) const { return pos <= other.
201 pos; }
202         bool operator>=( const Iterator other ) const { return pos >= other.
203 pos; }
204
205         void setPos(size_t pos_) { pos = pos_; }
206         void setId (size_t id_) { id = id_; }
207

```



```

198         std::pair<Key, Data&> operator*() { Node* v = this_>pool.get(id); \
199                                         return {v->x, v->val}; }
200
201     const std::pair<const Key, const Data&> operator*() const { Node* v =
this_>pool.get(id); \
202                                         return std::make_pair(v->x, v->val
); }
203
204
205     Iterator operator++()
206     {
207         assert(id != -1);
208         ++pos;
209         Node* v = this_>pool.get(id);
210         if (v->right != -1)
211         {
212             id = v->right;
213             while ((v = this_>pool.get(id))->left != -1)
214                 id = v->left;
215             return (*this);
216         }
217         while ((v = this_>pool.get(id))->parent != -1)
218         {
219             if (this_>pool.get(v->parent)->left == id)
220             {
221                 id = v->parent;
222                 return (*this);
223             }
224             id = v->parent;
225         }
226         id = -1;
227         return (*this);
228     }
229
230     Iterator operator++(int)
231     {
232         assert(id != -1);
233         ++pos;
234         Node* v = this_>pool.get(id);
235         Iterator result(*this);
236         if (v->right != -1)
237         {
238             id = v->right;
239             while ((v = this_>pool.get(id))->left != -1)
240                 id = v->left;
241             return result;
242         }
243         while ((v = this_>pool.get(id))->parent != -1)
244         {
245             if (this_>pool.get(v->parent)->left == id)
246             {
247                 id = v->parent;
248                 return result;

```

```

249         }
250         id = v->parent;
251     }
252     id = -1;
253     return result;
254 }
255
256 Iterator operator--()
257 {
258     assert(id != -1);
259     --pos;
260     Node* v = this_>pool.get(id);
261     if (v->left != -1)
262     {
263         id = v->left;
264         while ((v = this_>pool.get(id))>right != -1)
265             id = v->right;
266         return (*this);
267     }
268     while ((v = this_>pool.get(id))>parent != -1)
269     {
270         if (this_>pool.get(v->parent)>right == id)
271         {
272             id = v->parent;
273             return (*this);
274         }
275         id = v->parent;
276     }
277     id = -1;
278     return (*this);
279 }
280
281 Iterator operator--(int)
282 {
283     assert(id != -1);
284     --pos;
285     Node* v = this_>pool.get(id);
286     Iterator result(*this);
287     if (v->left != -1)
288     {
289         id = v->left;
290         while ((v = this_>pool.get(id))>right != -1)
291             id = v->right;
292         return result;
293     }
294     while ((v = this_>pool.get(id))>parent != -1)
295     {
296         if (this_>pool.get(v->parent)>right == id)
297         {
298             id = v->parent;
299             return result;
300         }
301         id = v->parent;

```

```

302     }
303     id = -1;
304     return result;
305 }
306
307 Iterator operator+(long long n) const {
308     return this_ -> kth_elem(pos + n);
309 }
310
311 Iterator operator-(long long n) const {
312     return this_ -> kth_elem(pos - n);
313 }
314
315 Iterator& operator+=(long long n) {
316     *this = *this + n;
317     return *this;
318 }
319
320 Iterator& operator-=(long long n) {
321     *this = *this - n;
322     return *this;
323 }
324
325
326 long long operator-(const Iterator &other) const {
327     return pos - other.pos;
328 }
329
330 private:
331     size_t pos;
332     size_t id;
333     const Treap* this_;
334 };
335
336 Iterator kth_elem(size_t k) const {
337     Iterator result(0, this);
338     result.setPos(k);
339     size_t id = root_id;
340     assert(id != -1);
341     Node *v;
342     while (id != -1){
343         v = pool.get(id);
344         size_t i = getSize(v->left);
345         if (i == k){
346             result.setId(id);
347             return result;
348         }
349         if (k < i)
350             id = v->left;
351         else {
352             k -= i + 1;
353             id = v->right;
354         }

```

```

355     }
356     return end();
357 }
358
359
360
361 Iterator begin() const { return Iterator(min_vert(root_id), this); }
362 Iterator end()   const { return Iterator(-1, this); }
363
364
365
366 //=====
367 // TREAP interface functions
368
369 Treap() : root_id(-1) {}
370 Treap(const Treap &other) : root_id(other.root_id), pool(other.pool) {}
371 Treap(Treap &&other);
372 ~Treap() = default;
373
374
375 Treap& operator=( const Treap &other );
376 Treap& operator=( Treap &&other );
377
378 bool operator==( const Treap &other ) const;
379 bool operator!=( const Treap &other ) const { return !((*this) == other); }
380
381 Data& operator[]( size_t n ) { return (*(begin() + n)).second; }
382 const Data& operator[]( size_t n ) const { return *(begin() + n); }
383
384 size_t size() const { if (root_id == -1) return 0; return pool.get(root_id)
->size; }
385
386 void insert( Key x, Data val );
387 Data* insert( Key x );
388
389 void erase ( Key x ) { if (root_id != -1) root_id = erase(root_id, x); }
390 size_t erase ( size_t id, Key x );
391
392 Data* find( Key x ) const;
393
394 #ifndef NDEBUG
395 void print ( std::ostream &out ) const { print(out, root_id); out <<
'\n'; }
396 void print_graph( std::ostream &out ) const
397 {
398     static size_t dumpn = 0;
399     out << "digraph tree" << dumpn++ << "{\n"
400         "    node [shape=record];\n";
401     if (root_id != -1)
402         print_graph(out, root_id);
403     out << "};\n";

```

```

404     }
405
406     bool graph_check() const
407     {
408         if (root_id != -1 && pool.get(root_id)->parent != -1)
409             return false;
410         return graph_check(root_id);
411     }
412
413     bool graph_check(size_t id) const
414     {
415         std::set<size_t> S;
416         return graph_check(id, S);
417     }
418 #endif
419
420 private:
421
422     bool graph_check( size_t id, std::set<size_t> &S ) const;
423     void print_graph( std::ostream &out, size_t id ) const;
424     void print( std::ostream &out, size_t id ) const;
425
426     size_t merge( size_t tl_id, size_t tr_id );
427     std::pair<size_t, size_t> split( size_t t_id, Key k );
428
429     void update( size_t id );
430     void insert( Node &node); //TODO write it to
emplement faster 0 nodes removal
431
432     size_t getSize( size_t v_id ) const { if (v_id == -1) return 0; return
pool.get(v_id)->size; }
433
434     size_t min_vert( size_t v_id ) const;
435     size_t max_vert( size_t v_id ) const;
436 };
437
438
439 template<typename Key, typename Data>
440 Treap<Key, Data>::Treap(Treap &&other)
441 {
442     root_id = exchange(other.root_id, -1);
443     pool = std::move(other.pool);
444 }
445
446
447 template<typename Key, typename Data>
448 Treap<Key, Data>& Treap<Key, Data>::operator=(const Treap<Key, Data> &other)
449 {
450     root_id = other.root_id;
451     pool = other.pool;
452     return (*this);
453 }
454

```

```

455 template<typename Key, typename Data>
456 Treap<Key, Data>& Treap<Key, Data>::operator=(Treap<Key, Data> &&other)
457 {
458     root_id = exchange(other.root_id, -1);
459     pool = std::move(other.pool);
460     return (*this);
461 }
462
463
464 template<typename Key, typename Data>
465 bool Treap<Key, Data>::operator==(const Treap<Key, Data> &other) const {
466     if (root_id != other.root_id)
467         return false;
468
469     auto iter_this = this->begin(), iter_other = other.begin();
470     while (iter_this != this->end() && iter_other != other.end()){
471         if (*iter_this != *iter_other)
472             return false;
473         ++iter_this;
474         ++iter_other;
475     }
476     if (iter_this != this->end() || iter_other != other.end())
477         return false;
478     return true;
479 }
480
481
482 template<typename Key, typename Data>
483 void Treap<Key, Data>::insert(Key x, Data val)
484 {
485     Data* q = find(x);
486     if (q)
487     {
488         *q = val;
489         return;
490     }
491
492     auto [tl_id, tr_id] = split(root_id, x);
493     size_t tm_id = pool.alloc();
494     assert(tm_id != -1);
495     Node *v = pool.get(tm_id);
496     v->val = val;
497     v->x = x;
498     root_id = merge(merge(tl_id, tm_id), tr_id);
499     TREAP_CHECK(root_id);
500 }
501
502 template<typename Key, typename Data>
503 Data* Treap<Key, Data>::insert(Key x)
504 {
505     Data* q = find(x);
506     if (q)
507         return q;

```

```

508
509     auto [tl_id, tr_id] = split(root_id, x);
510     size_t tm_id = pool.alloc();
511     assert(tm_id != -1);
512     Node *v = pool.get(tm_id);
513     v->x = x;
514     root_id = merge(merge(tl_id, tm_id), tr_id);
515     TREAP_CHECK(root_id);
516     return &(v->val);
517 }
518
519
520 template<typename Key, typename Data>
521 size_t Treap<Key, Data>::erase(size_t id, Key x) //TODO find bug
522 {
523     if (id == -1)
524         return -1;
525     Node *v = pool.get(id);
526     if (v->x == x)
527     {
528         size_t tl_id = v->left;
529         size_t tr_id = v->right;
530
531         v->parent = -1;
532         if (tr_id != -1)
533             pool.get(tr_id)->parent = -1;
534         if (tl_id != -1)
535             pool.get(tl_id)->parent = -1;
536
537         pool.free(id);
538         return merge(tl_id, tr_id);
539     }
540
541     if (v->x < x)
542         v->right = erase(v->right, x);
543     else
544         v->left = erase(v->left, x);
545
546     update(id);
547     TREAP_CHECK(root_id);
548     return id;
549 }
550
551 template<typename Key, typename Data>
552 Data* Treap<Key, Data>::find(Key x) const
553 {
554     size_t cur_id = root_id;
555     Node *v;
556     while (cur_id != -1 && ((v = pool.get(cur_id))->x != x))
557     {
558         if (v->x > x)
559             cur_id = v->left;
560         else

```

```

561         cur_id = v->right;
562     }
563     if (cur_id != -1)
564         return &v->val;
565     return nullptr;
566 }
567
568 template<typename Key, typename Data>
569 bool Treap<Key, Data>::graph_check(size_t id, std::set<size_t> &S) const
570 {
571     if (id == -1)
572         return true;
573     if (S.find(id) != S.end())
574         return false;
575     S.insert(id);
576     Node *v = pool.get(id);
577     if (v->right != -1)
578         if (!graph_check(v->right, S) || pool.get(v->right)->parent != id)
579             return false;
580     if (v->left != -1)
581         if (!graph_check(v->left, S) || pool.get(v->left)->parent != id)
582             return false;
583     return true;
584 }
585
586 template<typename Key, typename Data>
587 void Treap<Key, Data>::print_graph(std::ostream &out, size_t id) const
588 {
589     assert(id != -1);
590     Node *v = pool.get(id);
591     out << "struct" << id << " [label=\"" << id << " | { key = " << v->x << "
592 | data = " << v->val
593 << " }\">";\n";
594     if (v->left != -1)
595     {
596         out << "struct" << id << " -> " << "struct" << v->left << ";\n";
597         print_graph(out, v->left);
598     }
599     if (v->right != -1)
600     {
601         out << "struct" << id << " -> " << "struct" << v->right << ";\n";
602         print_graph(out, v->right);
603     }
604 }
605
606 template<typename Key, typename Data>
607 size_t Treap<Key, Data>::merge(size_t tl_id, size_t tr_id)
608 {
609     TREAP_CHECK(tl_id);
610     TREAP_CHECK(tr_id);
611     if (tl_id == -1)
612         return tr_id;

```



```

613     if (tr_id == -1)
614         return tl_id;
615     Node *tl = pool.get(tl_id);
616     Node *tr = pool.get(tr_id);
617     if (tl->prior < tr->prior)
618     {
619         tl->right = merge(tl->right, tr_id);
620         update(tl_id);
621         TREAP_CHECK(tl_id);
622         return tl_id;
623     }
624     else
625     {
626         tr->left = merge(tl_id, tr->left);
627         update(tr_id);
628         TREAP_CHECK(tr_id);
629         return tr_id;
630     }
631 }
632
633 template<typename Key, typename Data>
634 std::pair<size_t, size_t> Treap<Key, Data>::split(size_t t_id, Key k)
635 {
636     if (t_id == -1)
637         return {-1, -1};
638     Node *t = pool.get(t_id);
639
640     if (t->x <= k)
641     {
642         auto [tl_id, tr_id] = split(t->right, k);
643         t->right = tl_id;
644         update(t_id);
645         if (tr_id != -1)
646             pool.get(tr_id)->parent = -1;
647         t->parent = -1;
648         TREAP_CHECK(tr_id);
649         TREAP_CHECK(t_id);
650         return {t_id, tr_id};
651     }
652     else
653     {
654         auto [tl_id, tr_id] = split(t->left, k);
655         t->left = tr_id;
656         update(t_id);
657         if (tl_id != -1)
658             pool.get(tl_id)->parent = -1;
659         t->parent = -1;
660         TREAP_CHECK(tl_id);
661         TREAP_CHECK(t_id);
662         return {tl_id, t_id};
663     }
664 }
665

```

```

666 template<typename Key, typename Data>
667 void Treap<Key, Data>::update(size_t id)
668 {
669     assert(id != -1);
670
671     Node* v = pool.get(id);
672     v->size = 1;
673     if (v->left != -1)
674     {
675         Node* tl = pool.get(v->left);
676         v->size += tl->size;
677         tl->parent = id;
678     }
679     if (v->right != -1)
680     {
681         Node* tr = pool.get(v->right);
682         v->size += tr->size;
683         tr->parent = id;
684     }
685 }
686
687 template<typename Key, typename Data>
688 size_t Treap<Key, Data>::min_vert(size_t v_id) const
689 {
690     if (v_id == -1)
691         return -1;
692     Node *v;
693     while ((v = pool.get(v_id))->left != -1)
694         v_id = v->left;
695     return v_id;
696 }
697
698 template<typename Key, typename Data>
699 void Treap<Key, Data>::print(std::ostream &out, size_t id) const
700 {
701     TREAP_CHECK(id);
702     if (id == -1) return;
703     Node *v = pool.get(id);
704     print(out, v->left);
705
706     out << '(' << v->x << ", " << v->val << ", " << v->size << ") ";
707
708     print(out, v->right);
709 }
710
711 template<typename Key, typename Data>
712 size_t Treap<Key, Data>::max_vert(size_t v_id) const
713 {
714     if (v_id == -1)
715         return -1;
716     Node *v;
717     while ((v = pool.get(v_id))->right != -1)
718         v_id = v->right;

```

```

719     return v_id;
720 }
721
722 #endif

```

Д.2 Файл test-treap.hpp

```

1 #include "treap.hpp"
2 #include <vector>
3 #include "gtest/gtest.h"
4
5 template<typename T>
6 void IteratorsTest(){
7     Treap<int, T> T1;
8     for (int i = 0; i < 500 + rnd() % 1500; ++i){
9         T1.insert(static_cast<int>(rnd()), static_cast<T>(rnd()));
10    }
11    auto iter = T1.begin();
12    for (size_t i = 0; i < T1.size() - 1; ++i){
13        EXPECT_EQ((*iter).second, T1[i]);
14        ++iter;
15    }
16    for (size_t i = T1.size() - 1; i > 0; --i){
17        EXPECT_EQ((*iter).second, T1[i]);
18        --iter;
19    }
20
21    T1.erase((*iter).first);
22    for (int i = 0; i < 100; ++i){
23        iter = T1.begin() + rnd() % 400;
24        T1.erase((*iter).first);
25    }
26    T *n = T1.insert(static_cast<int>(rnd()));
27    *n = 102;
28
29    iter = T1.begin();
30    for (size_t i = 0; i < T1.size() - 1; ++i){
31        EXPECT_EQ((*iter).second, T1[i]);
32        ++iter;
33    }
34    *n = 102;
35    iter = T1.begin();
36    for (size_t i = 0; i < T1.size(); ++i){
37        ++T1[i];
38        EXPECT_EQ((*iter).second, T1[i]);
39        ++iter;
40    }
41 }
42
43
44 TEST(Basics, Iterators)
45 {
46     for (int u = 0; u < 1; ++u){
47

```

```

48     IteratorsTest<int>();
49     //IteratorsTest<long>();
50     //IteratorsTest<unsigned long long>();
51     //IteratorsTest<short>();
52     //IteratorsTest<char>();
53 }
54
55 }
56
57 TEST(Basics , InsertEraseFind)
58 {
59     for (int p = 0; p < 5; ++p){
60         Treap<int , int> T1;
61         std::vector<int> V1;
62         for (int i = 0; i < 500 + rnd() % 1500; ++i){
63             int a = rnd();
64             T1.insert(a, a);
65             V1.push_back(a);
66         }
67         std::sort(V1.begin(), V1.end());
68         for (int i = 0; i < T1.size(); ++i)
69             EXPECT_EQ(T1[i], V1[i]);
70
71         for (int i = 0; i < 100; ++i){
72             int a = rnd() % 400;
73             auto iter = T1.begin() + a;
74             T1.erase((*iter).first);
75             V1.erase(V1.begin() + a);
76         }
77
78         for (int i = 0; i < T1.size(); ++i)
79             EXPECT_EQ(T1[i], V1[i]);
80
81         //continue;
82
83         for (int i = 0; i < 100; ++i){
84             int a = rnd() % 300, b = rnd();
85             int *q = T1.find((*(T1.begin() + a)).first);
86             *q = b;
87             V1[a] = b;
88         }
89         for (int i = 0; i < T1.size(); ++i)
90             EXPECT_EQ(T1[i], V1[i]);
91     }
92 }
93
94 TEST(Basics , CopyMove)
95 {
96     for (int u = 0; u < 6; ++u){
97         Treap<int , int> T1;
98         std::vector<int> V1;
99         for (int i = 0; i < 500 + rnd() % 1500; ++i){
100             int a = rnd();

```

```

101         T1.insert(a, a);
102         V1.push_back(a);
103     }
104     std::sort(V1.begin(), V1.end());
105     for (int i = 0; i < T1.size(); ++i)
106         EXPECT_EQ(T1[i], V1[i]);
107
108
109     Treap<int, int> T2(T1), T3, T4, T5(std::move(T1));
110     T3 = T2;
111     for (int i = 0; i < T2.size(); ++i)
112         EXPECT_EQ(T2[i], V1[i]);
113     for (int i = 0; i < T3.size(); ++i)
114         EXPECT_EQ(T3[i], V1[i]);
115     for (int i = 0; i < T5.size(); ++i)
116         EXPECT_EQ(T5[i], V1[i]);
117
118     T4 = std::move(T5);
119     for (int i = 0; i < T4.size(); ++i)
120         EXPECT_EQ(T4[i], V1[i]);
121 }
122 }

```