Правительство Российской Федерации

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

Кафедра «Компьютерная безопасность»

ОТЧЕТ
К ЛАБОРАТОРНОЙ РАБОТЕ №3

по дисциплине

«Языки программирования»

Работу выполнил
студент группы СКБ-201 _____ П.Е. Зильберштейн
подпись, дата

Работу проверил _____ С.А. Булгаков
подпись, дата

Москва 2021

Содержание

Постановка задачи

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

Общая часть

Переработать классы, разработанные в рамках лабораторной работы 2. Разработать шаблоны классов, объекты которых реализуют типы данных, указанные ниже. Для этих шаблонов классов разработать необходимые конструкторы, деструктор, конструктор копирования. Разработать операции: добавления/удаления элемента (уточнено в задаче); получения количества элементов; доступа к элементу (перегрузить оператор []). При ошибках запускать исключение.Разработать два вида итераторов (обычный и константный) для указанных шаблонов классов. В главной функции разместить тесты, разработанные с использованием библиотеки GoogleTest. При разработке тестов, добиться полного покрытия. Отчет о покрытии приложить к работе.

Задачи

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

б) Шаблон «стек» (внутреннее представление динамический массив хранимых объектов). Размерность стека увеличивается в момент его переполнения.Начальная размерность задается как параметр конструктора, значение по умолчанию 0. Добавление/удаление элемента в начало и в конец.

в) Шаблон «односвязный список объектов».Добавление/удаление элемента в произвольное место.

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

д) Шаблон «двоичное дерево объектов».Добавление/удаление элемента в произвольное место.

1 Алгоритм решения задачи

1.1 Задача 1

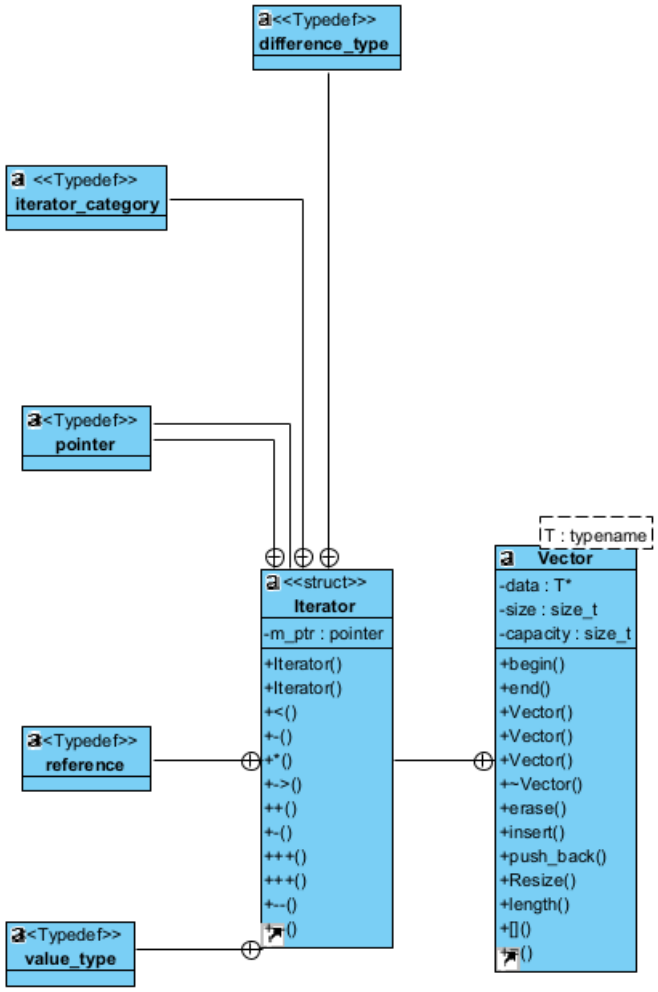

Рис. 1. UML-диаграмма класса Vector.

Для решения данной задачи был разработан шаблон класса Vector, UML диаграмма которого приведена на рис. 1, содержащий закрытые поля size, capacity типа `std::size_t` и data шаблонного типа `T*`, требуемые по заданию. Первое поле отвечает за хранение размера объекта, второе - за хранение потенциального размера (размера выделенной памяти), третье - за хранение данных, занесенных в объект.
Также класс содержит:

- конструктор по умолчанию;
- конструктор с параметром, создающий объект на основе целого числа - размера объекта;
- конструктор копирования;
- декструктор;

Помимо этого в классе имеются:

– Функция begin, возвращающая объект структуры Iterator, построенный от нулевого элемента поля data;

– Функция end, возвращающая объект структуры Iterator, построенный от последнего элемента поля data;

– функция erase, принимающая индекс элемента в массиве и удаляющая его;

– функция insert, принимающая индекс элемента в массиве и объект типа T, которая вставляет в это место переданный указатель;

– функция push_back, принимающая объект типа T, которая добавляет в конец массива переданный объект;

– функция Resize, принимающая целое неотрицательное число и изменяющая размер массива;

– функция length, возвращающая размер массива;

– перегрузка операции `[ ]` для двух случаев, когда необходим элемент массива в качестве lvalue, то есть изменяемого значения и когда необходим объект в качестве rvalue;

– перегрузка операции присваивания ('=') для случая, когда объекту класса присваивается другой объект класса;

1.2 Задача 2

Для решения данной задачи был разработан шаблон класса Stack, UML диаграмма которого приведена на рис. 2, содержащий закрытые поля size, capacity типа `std::size_t` и data типа `T*`, требуемые по заданию. Первое поле отвечает за хранение размера объекта, второе - за хранение потенциального размера (размера выделенной памяти), третье - за хранение данных, занесенных в объект.
Также класс содержит:

– Функция begin, возвращающая объект структуры Iterator, построенный от нулевого элемента поля data;

– Функция end, возвращающая объект структуры Iterator, построенный от последнего элемента поля data;

– конструктор по умолчанию;

– конструктор с параметром, создающий объект на основе целого числа - размера объекта;

– конструктор копирования;

– декструктор;

Помимо этого в классе имеются:

– функция pop_back, удаляющая последний элемент из массива;

– функция push_back, принимающая объект типа T, которая добавляет в конец массива переданный объект;

– функция pop_up, удаляющая первый элемент из массива;

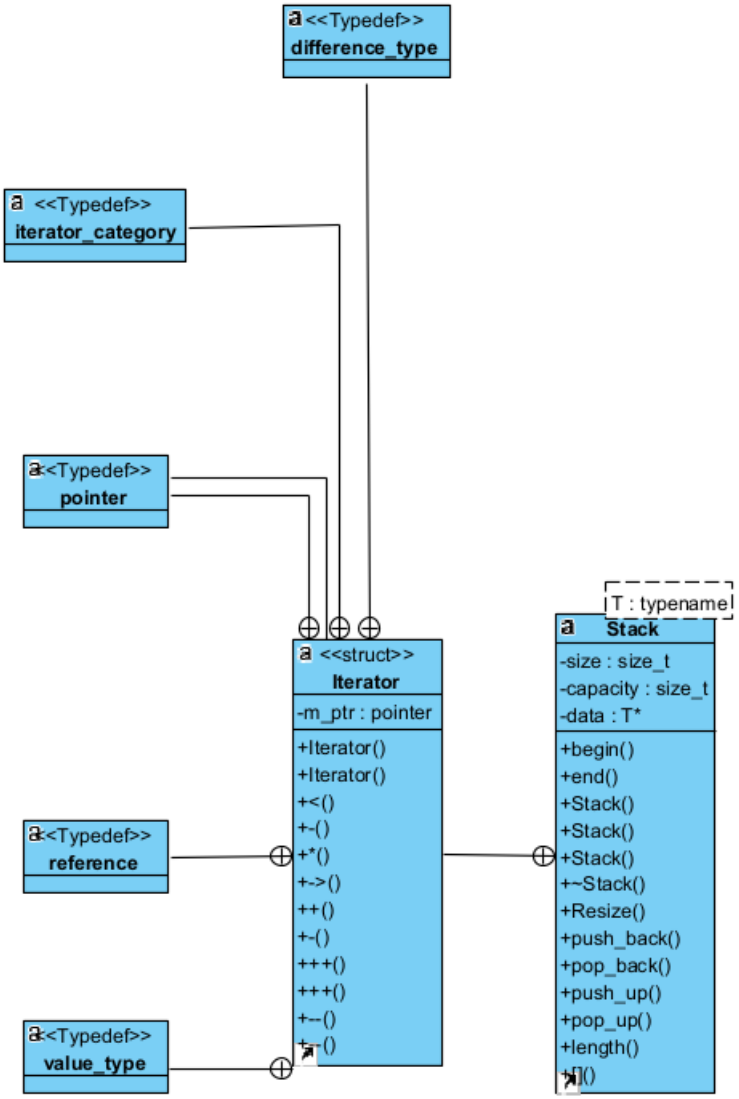

Рис. 2. UML-диаграмма класса Stack.

– функция push_up, принимающая объект типа T, которая добавляет в начало массива переданный объект;

– функция Resize, принимающая целое неотрицательное число и изменяющая размер массива;

– функция length, возвращающая размер массива;

– перегрузка операции `[ ]` (`operator[ ]`) для случая, когда необходим элемент массива в качестве rvalue;

1.3 Задача 3

Для решения данной задачи был разработан класс List, UML диаграмма которого приведена на рис. 3, содержащий закрытое поле head типа `Node*`, которое отвечает за хранение адреса на головной элемент списка. Объекты структуры Node содержат указатель на следующий элемент (по умолчанию nullptr) и переменную шаблонного типа T
Также класс содержит:

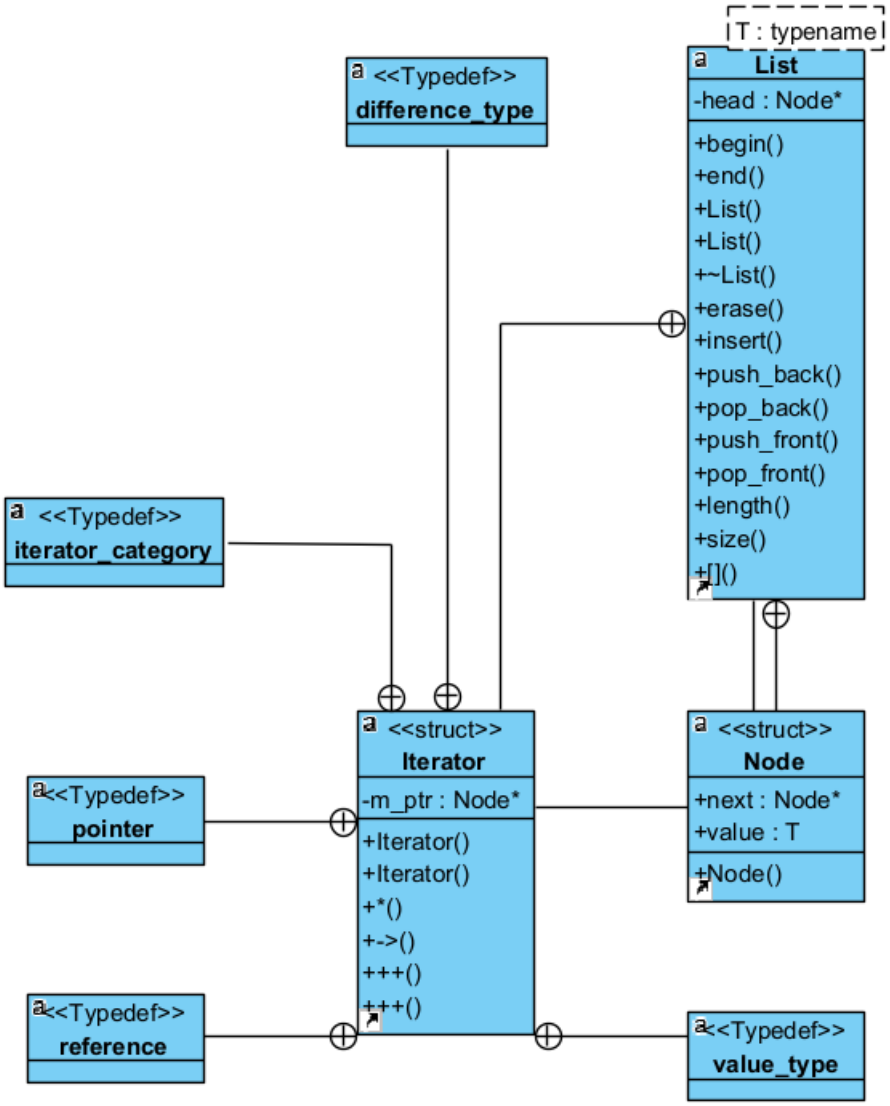

Рис. 3. UML-диаграмма класса List.

– конструктор по умолчанию;

– конструктор копирования;

– декструктор;

Помимо этого в классе имеются:

– функция begin, возвращающая итератор на начало списка; end, возвращающая итератор от нулевого указателя;

– функция erase, принимающая индекс элемента в массиве и удаляющая его;

– функция insert, принимающая индекс элемента в массиве и объект шаблонного типа T, которая вставляет в это место переданный объект;

– функция push_back, принимающая объект шаблонного типа T, которая добавляет в конец массива переданный объект;

– функция pop_back, удаляющая последний элемент из массива;

– функция pop_front, удаляющая первый элемент из массива;

– функция push_front, принимающая объект шаблонного типа T, которая добавляет в начало массива переданный элемент;

– функция length, возвращающая размер массива;

– перегрузка операции `[ ]` для двух случаев, когда необходим элемент массива в качестве lvalue, то есть изменяемого значения и когда необходим объект в качестве rvalue;

1.4 Задача 4

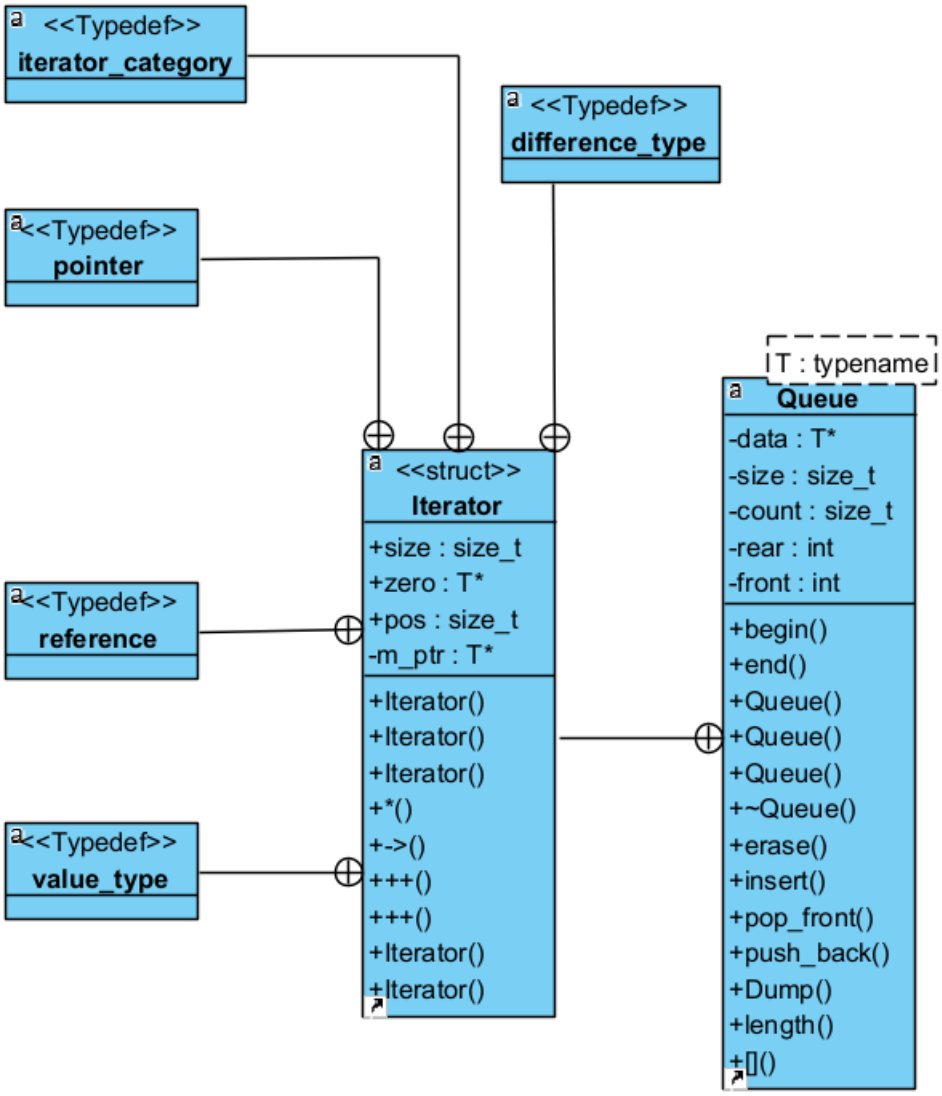

Рис. 4. UML-диаграмма класса Queue.

Для решения данной задачи был разработан класс Queue, UML диаграмма которого приведена на рис. 4, содержащий закрытые поля data, являющееся указателем на шаблонный тип (T*), отвечающее за хранение данных; size и count типа `std::size_t`, отвечающие за размер очереди и количество заполненных "ячеек"соответсвенно; rear и front типа int, отвечающие за хранение индекс элемента, являющегося последним и первым соответсвенно.
Также класс содержит:

– конструктор по умолчанию;

– конструктор копирования;

– конструктор с параметром - целое неотрицательное число;

– декструктор;

Помимо этого в классе имеются:

– функция begin, возвращающая итератор на элмент с индексом front;

– функция end, возвращающая итератор на элмент с индексом rear+1;

– функция erase, принимающая индекс элемента в очереди и удаляющая его;

– функция insert, принимающая индекс элемента в очереди и объект шаблонного типа T, которая вставляет в это место переданный объект;

– функция push_back, принимающая объект шаблонного типа T, которая добавляет в конец очереди переданный объект;

– функция pop_front, удаляющая передний элемент из очереди;

– функция length, возвращающая размер массива;

– перегрузка операции `[ ]` (`operator[ ]`) для двух случаев, когда необходим элемент массива в качестве lvalue, то есть изменяемого значения и когда необходим объект в качестве rvalue;

1.5 Задача 5

Для решения данной задачи был разработан шаблон класса Tree, UML диаграмма которого приведена на рис. 5, содержащий закрытые поля head типа Node*, отвечающее за хранение корня дерева; size и steps типа `std::size_t`, отвечающие за количество элементов в дереве и количество ступеней.
Также класс содержит:

– конструктор по умолчанию;

– конструктор копирования;

– конструктор с параметром - целое неотрицательное число;

– декструктор;

Помимо этого в классе имеются:

– функция begin, возвращающая итератор на первый элемент;

– функция end, возвращающая итератор на последний элемент;

– функция erase, принимающая индекс элемента в дереве и удаляющая его и его потмков;

– функция insert, принимающая индекс элемента в очереди и объект типа T, которая вставляет в это место переданный объект;

– функция length, возвращающая размер дереве - количество элементов в нем;

– перегрузка операции `[ ]` (`operator[ ]`) для двух случаев, когда необходим элемент массива в качестве lvalue, то есть изменяемого значения и когда необходим объект в качестве rvalue;

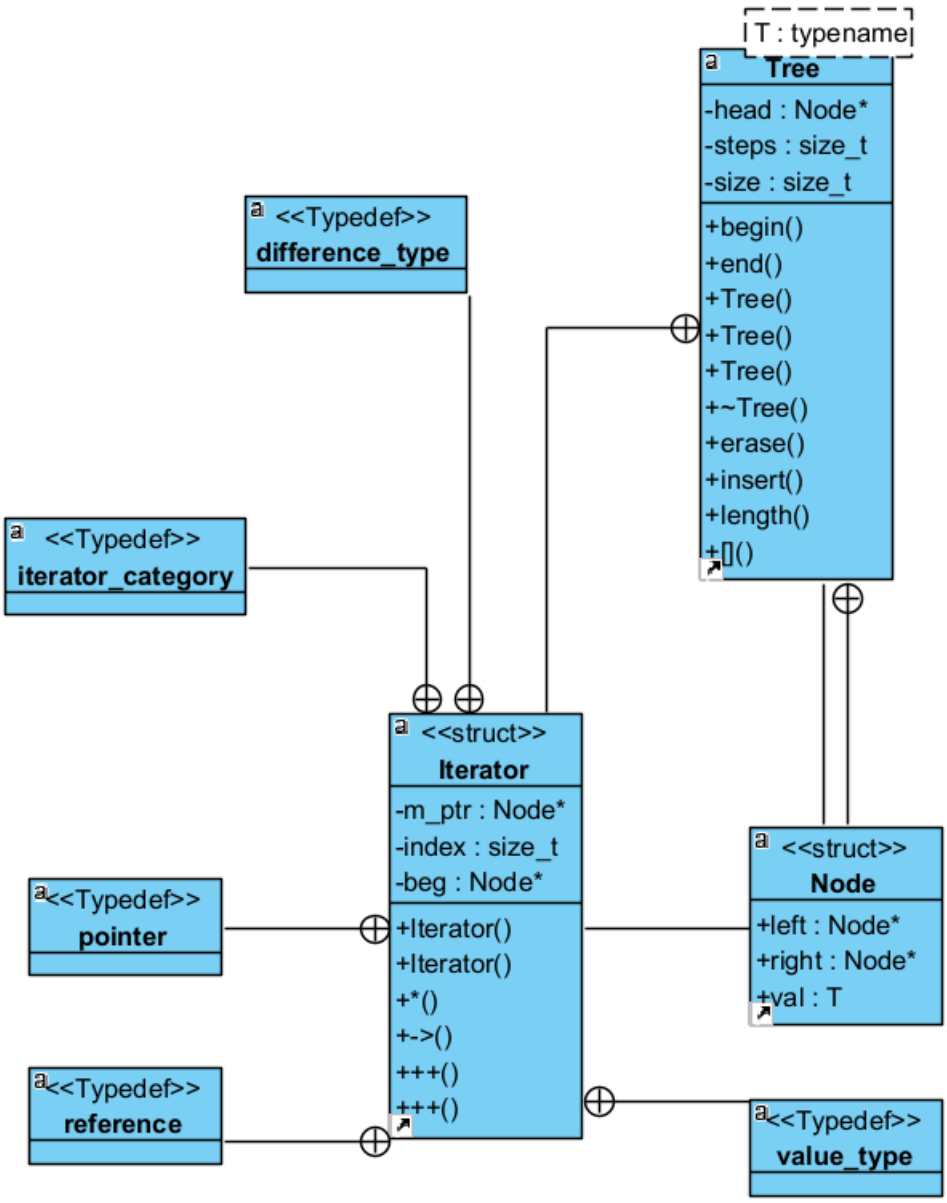

Рис. 5. UML-диаграмма класса Tree.

2 Выполнение задания

2.1 Задача 1

2.1.1 Конструкторы и деструктор

Конструктор по умолчанию: полям size и capacity присваиваются нулевые значения, а полю data — nullptr.

Конуструктор копирования: полям нового объекта соотвественно сопоставляются поля уже имеющегося объекта.

Конструктор с параметром от одного целого числа: полю size присваивается значенние параметра, полю capacity — удвоенное значение параметра, а для data выделяется память размером capacity.

Деструктор: сначала проверяется, не является ли data нулевым указателем, затем с помощью векторной формы delete[] очищается память и полю data присваивается значение nullptr.

2.1.2 Остальные функции

Функция удаления элемента из произвольного места (erase): сначала проверяется, есть ли элемент с переданным индексом в массиве: если нет, то бросается исключение. Затем, в цикле происходит сдвиг на один элемент влево, начиная с элемента с индексом-параметром. В конце размер массива уменьшается на 1.

Функция вставки элемента в произвольное место (insert): сначала проверяется, есть ли элемент с переданным индексом в массиве: если нет, то бросается исключение. Затем, проверяется, превзойдет ли размер массива после вставки элемента количество выделенной памяти. Если да, то создается временный объект класса - полная копия исходного. Потом функцией Resize изменяется размер исходного объекта (увеличивается на 1) и в цикле данные исходного объекта возвраоаются на свои места. Если же не превзойдет, то просто увеличивается размер массива на 1. Далее, в цикле осуществляется сдвиг элементов массива вправо, начиная с элемента, индекс которого былл передан в качестве параметра. В конце элементу с этим индексом присваивается указатель-параметр.

Функция вставки элемента в конец (push_back!: сначала проверяется, превзойдет ли размер массива после вставки элемента количество выделенной памяти. Если да, то создается временный объект класса - полная копия исходного. Потом функцией Resize изменяется размер исходного объекта (увеличивается на 1) и в цикле данные исходного объекта возвраоаются на свои места. Если же не превзойдет, то просто увеличивается размер массива на 1. В конце последнему элементу присваивается указатель-параметр.

Функция изменения размера массива (Resize): для поля data выделяется память, равная удвоенному значению параметра, затем это же значение присваивается полю capacity. В поле size записывается значение параметра.

Функция получения размера массива (length): возвращает значение поля size.

Перегрузка '=' для случая, когда объекту класса присваивается другой объект класса: сначала проверяется случай самоприсваивания. Затем память выделенная под data очищается, полю size присваивается поле size другого объекта, аналогично с полем capacity. Затем в цикле в массив вносятся значения из массива другого объекта. В конце функция возвращает исходный измененный объект.

Перегрузка '[ ]' для обоих случаев: сначала проверяется, есть ли элемент с переданным индексом в массиве: если нет, то бросается исключение. В конце функция возвращает элемент массива data с индексом, переданным в качестве параметра.

2.2 Задача 2

2.2.1 Конструкторы и деструктор

Конструктор по умолчанию: полям size и capacity присваиваются нулевые значения, а полю data - nullptr.

Конуструктор копирования:полям нового объекта соотвественно сопоставляются поля уже имеющегося объекта.

Конструктор с параметром от одного целого числа: полю size присваивается значенние параметра, полю capacity - удвоенное значение параметра, а для data выделяется память размером capacity.

Деструктор: сначала проверяется, не является ли data нулевым указателем, затем с помощью векторной формы delete[] очищается память и полю data присваивается значение nullptr.

2.2.2 Остальные функции

Функция изменения размера массива (Resize): для поля data выделяется память, равная удвоенному значению параметра, затем это же значение присваивается полю capacity. В поле size записывается значение параметра.

Функция вставки элемента в конец (push_back!): сначала проверяется, превзойдет ли размер массива после вставки элемента количество выделенной памяти. Если да, то создается временный объект класса - полная копия исходного. Потом функцией Resize изменяется размер исходного объекта (увеличивается на 1) и в цикле данные исходного объекта возвраоаются на свои места. Если же не превзойдет, то просто увеличивается размер массива на 1. В конце последнему элементу присваивается параметр.

Функция вставки элемента в начало (push_up): сначала проверяется, превзойдет ли размер массива после вставки элемента количество выделенной памяти. Если да, то создается временный объект класса - полная копия исходного. Потом функцией Resize изменяется размер исходного объекта (увеличивается на 1) и в цикле элементы массива временного объекта присваюваются исходному со сдвигом на 1 вправо. Если же не превзойдет, то увеличивается размер массива на 1, происходит сдвиг элементов массива вправо на 1. В конце первому элементу присваивается параметр.

Функция удаления последнего элемента (pop_back): сначала проверяется размер массива: если он нулевой, то бросается исключение. Затем размер массива уменьшается на 1.

Функция удаления первого элемента (pop_up): сначала проверяется размер массива: если он нулевой, то бросается исключение. Затем происходит сдвиг элементов массива на 1 влево. В конце размер массива уменьшается на 1.

Функция получения размера массива (length): возвращает значение поля size.

Перегрузка '[ ]': сначала проверяется, есть ли элемент с переданным индексом в массиве: если нет, то бросается исключение. В конце функция возвращает элемент массива data с индексом, переданным в качестве параметра.

2.3 Задача 3

2.3.1 Конструкторы и деструктор

Конструктор по умолчанию: полю head присваивается значение nullptr.

Конуструктор копирования: реализация доверена компилятору (с помощью конструкции =default).

Деструктор: реализация доверена компилятору (с помощью конструкции =default).

2.3.2 Остальные функции

Функция удаления элемента из произвольного места (erase): сначала проверяется, есть ли элемент с переданным индексом в массиве: если нет, то бросается исключение. Также проверяется количество элементов в списке: если остался только один элемент, то бросается исключение с фразой о том, что происходит удаление головного элемента, которое запрещено. Затем, если переданный индекс указывает на последний элемент списка, то вызывается функция удаления последнего элемента pop_back, если индекс ссылается на головной элемент, то вызывается функция удаления первого элемента (pop_front). Иначе полю next присваивается адрес следующего за удаляемым элемента.

Функция вставки элемента в произвольное место (insert): сначала проверяется, есть ли элемент с переданным индексом в массиве: если нет, то бросается исключение. Если индекс указывает на головной элемент, то вызывается функция вставки первого элемента (push_front),

иначе создается объект типа Node*, для которого укащатель на следующий элемент становится указатель на элемент с переданным индексом, следующим элементом для предыдущего становится новый временный.

Функция вставки элемента в конец (push_back): Сначала создается объект tmp типа Node*, затем проверяется головной элемент: если он отсутсвует, то головным элементом становится tmp, иначе создается объект current типа Node*, которому присваивается поле head. Затем current изменяется, пока не дойдет по списку до последнего. Как только current имеет значение nullptr, цикл заканчиваеь, а следующим элементом для current становится tmp.

Функция удаления последнего элемента (pop_back): сначала проверяется размер массива: если он нулевой или единичный, то бросается исключение. Затем для предпоследнего элемента следующий становится nullptr.

Функция вставки элемента в начало (push_front): сначала создается объект tmp типа Node*, следующим для него элементом становится головной, а tmp теперь сам становится головным.

Функция удаления первого элемента (pop_front): сначала проверяется размер массива: если он нулевой или единичный, то бросается исключение. Затем головным элементом становится второй.

Функция получения размера массива (length): если значение поля head равно nullptr, то функция возвращает 0. Создается переменная size типа `std::size_t` со значением 1, которая будет хранить размер. Также создается переменная current типа Node*, затем в цикле пока следующий элемент для текущего не nullptr, увеличивается size на 1. В конце функция возвращает значение переменной size.

Перегрузка '`[ ]`' для обоих случаев: сначала проверяется, есть ли элемент с переданным индексом (index) в массиве: если нет, то бросается исключение. Затем создается объект current типа Node*, которому присваивается поле head. Потом в цикле current присваивается следующий элемент index раз. Функция возвращает current.

2.4 Задача 4

2.4.1 Конструкторы и деструктор

Конструктор по умолчанию: полю data присваивается значение nullptr, size — 1, count и rear — 0, front — -1.

Конуструктор копирования: полям конструируемого объекта соответсвенно присваются поля копируемого объекта.

Конструктор с параметром: полю size присваивается параметр, count - 0, rear - 0, front -1. Затем для поля data выделяется память, и каждому указатель из data присваивается значение 0 с помощью `std::fill`.

Деструктор: если data не равно nullptr, то с помощью векторной формы delete память очищается, а data присваивается nullptr

2.4.2 Остальные функции

Функция удаления элемента из произвольного места (erase): сначала проверяется, есть ли элемент с переданным индексом в массиве: если нет, то бросается исключение. Затем, если index и front это один и тот же элемент, то вызывается функция pop_front. Иначе (при условии, что index не равен rear, так как удаление первого элемента очереди - нелогично) происходит циклический сдвиг массива и его размер уменьшается на 1. Последний элемент обнуляется.

Функция вставки элемента в произвольное место (insert): сначала проверяется, есть ли элемент с переданным индексом в массиве: если нет, то бросается исключение. Если индекс равен rear+1, то вызывается функция добавления в конец (push_back). Если элемент с переданным

индексом равен nullptr, то он просто перезаписывается, иначе осуществляется циклический сдвиг и элемент вписывается.

Функция вставки элемента в конец (push_back): проверяется несколько условий: если при увеличении rear на 1 оно становится равным front, то front сдвигается циклично на 1 вправо; если количество элементов равно нулю, то front увеличивается на если количество элементов меньше выделенного размера, то count увеличивается на 1. Затем элемент вписывается в конец очереди.

Функция удаления первого элемента (pop_front): сначала проверяется размер массива: если он нулевой, то бросается исключение. Затем осуществляется циклический сдвиг на 1 вправо и первый элемент удаляется.

Функция получения размера массива (length): функция возвращает значение поля size.

Перегрузка '[ ]' для обоих случаев: сначала проверяется, есть ли элемент с переданным индексом (index) в массиве: если нет, то бросается исключение. Затем функция возвращает data[index].

2.5 Задача 5

2.5.1 Конструкторы и деструктор

Конструктор по умолчанию: полю head присваивается значение new Node.

Конуструктор копирования: полям конструируемого объекта соответсвенно присваются поля копируемого объекта.

Конструктор с параметром: полю steps присваивается параметр, под head выделяется память с помощью new Node, size присваивается значение `2^{steps}-1`. Затем в цикле от второго ряда до предпоследнего i-ому элементу выделяется память, а его полю val присваивается 0.

Деструктор: реализация доверена компилятору с помощью конструкции =default.

2.5.2 Остальные функции

Функция удаления элемента из произвольного места (erase): сначала проверяется, есть ли элемент с переданным индексом в дереве: если нет, то бросается исключение. Затем, полям элемента дерева с таким индексом присваивается значение 0.

Функция вставки элемента в произвольное место (insert): сначала проверяется, есть ли элемент с переданным индексом в массиве: если нет, то бросается исключение. Затем исследуется, сколько рядов необходимо добавить в дерево при вставке в него нового узла. На основе этого меняются поля steps и size. Затем создается объект cur типа Node* и выделяется память. Его полю left присваивается элемент дерева с индексом, right - nullptr, val - второй параметр функции. Потом с помощью тернарной операции определяется к какой ветке высшего порядка прикрепляется cur.

Функция получения размера массива (length): функция возвращает значение поля size.

Перегрузка '[ ]' для обоих случаев: сначала проверяется, есть ли элемент с переданным индексом (index) в массиве: если нет, то бросается исключение. Затем строится двоичное представление индекса, от которого отбрасывается первая цифра и получается путь от корня дерева к необходимому потомку (цифра 0 - переход в левую ветку, 1 - в правую). В версии для rvalue сделано еще две дополнительные проверки: если head имеет значение nullptr, то дерево пусто, и если полученное значение current равно nullptr, то такого элемента в дереве нет.

3 Получение исполняемых модулей

Для получения исполняемых модулей main0, main1, main2 была использована система сборки cmake - написан файл `CMakeLists.txt`. Минимальная требуемая версия системы cmake - 3.12.

Флаги компиляции:

- Wall (вывод всех предупреждений);
- pedantic-errors (проверяет соответствие кода стандарту ISO C++, сообщает об использовании запрещённых расширений, считает все предупреждения ошибками);
- fsanitize=undefined (санитайзер для неопределенного поведения);
- std=c++20 (устанавливает стандарт языка);
- lgtest (необходим для работы тестов GoogleTest);

Файлы из которых собирается исполняемый модуль:

- main0 составляется из файлов Test.cpp, Vector.h
- main1 составляется из файлов Test1.cpp, Stack.h;
- main2 составляется из файлов Test2.cpp, List.hpp;
- main3 составляется из файлов Test3.cpp, Queue.hpp;
- main4 составляется из файлов Test4.cpp, Tree.hpp;

Листинг-1. Файл `CMakeLists.txt`

```
   identifierstyle
cmake_minimum_required(VERSION 3.12)

SET(CMAKE_C_COMPILER clang)
SET(CMAKE_CXX_COMPILER clang++)

set(CMAKE_CXX_FLAGS
"${CMAKE_CXX_FLAGS} -Wall -pedantic-errors -fsanitize=undefined -std=c++20 -lgtest")

#target_link_libraries(${CMAKE_PROJECT_NAME} "-lgtest")

#set(CMAKE_LD_FLAGS
#"${CMAKE_LD_FLAGS} -lgtest")

project(Laba3)

add_executable(main0 Test.cpp Vector.h)

add_executable(main1 Test1.cpp Stack.h)

add_executable(main2 Test2.cpp List.hpp)

add_executable(main3 Test3.cpp Queue.hpp)

add_executable(main4 Test4.cpp Tree.hpp)
```

4 Тестирование

4.1 Тест №1

Процент покрытия - 91.57%

4.1.1 Проверка работоспособности конструкторов и функции получения размера

Создается три объекта: один с помощью конструктора по умолчанию, второй - с помощью конструктора с параметрами от значения 4, а третий - с помощью конструктора копирования. Затем с помощью макроса `EXPECT_EQ` из `GoogleTest` проверяются на равенства: размер первого вектора и нуля, второго вектора и третьего вектора и четырех.

4.1.2 Проверка работоспособности функция Resize и push_back

Создается два объекта: один с помощью конструктора по умолчанию, второй - с помощью конструктора с параметрами от значения 4. Потом размер обоих векторов изменяется с помощью функции Resize на значение 2. Затем в цикле range-based for элементам вектора A присваиваются значения, равные порядковому номеру в массиве, демонстрируя работу итераторов. Потом к векторам добавляется в конец с помощью функции push_back по одному элементу. В конце с помощью макроса `EXPECT_EQ` из библиотеки `GoogleTest` проверяются на равенства: размер первого вектора и трех, второго вектора и трех, а также элементы векторов.

4.1.3 Проверка работоспособности остальных функций

Создается два объекта: вектор A, хранящий объекты типа `std::string` с помощью конструктора от числа 2, и вектор B, хранящий объекты типа `double` с помощью конструктора от числа 4. Затем к обоим векторам в конец добавляетсся по одному объекту, в вектор A на нулевую позицию вставляется объект типа `double`. Потом задаются значения первым четырем элементам вектора. Далее в вектор A сначала вставляется объект на позицию 2, потом - 1, а позднее из него удаояется элемент с индексом 2. Потом с помощью функции `std::swap` значениями обмениваются два элемента вектора. В конце с помощью макросов `EXPECT_DOUBLE_EQ и EXPECT_EQ` из GoogleTest проверяются на равенства задуманным значениям все элемент векторов и их длины.

4.2 Тест №2

Процент покрытия - 98.16%

4.2.1 Проверка работоспособности конструкторов, функции получения размера и его изменения

Создается три объекта: один с помощью конструктора по умолчанию, второй - с помощью конструктора с параметрами от значения 4, а третий - с помощью конструктора копирования. Затем размер векторов изменяется с помощью функции Resize на значение 10, 5 и 0 соответственно. Потом с помощью макроса `EXPECT_EQ` из GoogleTest проверяются на равенства: размер первого стека и 10, второго - и 5, третьего - и 0.

4.2.2 Проверка работоспособности итераторов

Создается три объекта, хранящих объекты типа `std::string`: стек A с помощью конструктора по умолчанию, стек B - с помощью конструктора с параметрами от значения 4, а стек C - с помощью конструктора копирования. Затем размер векторов изменяется с помощью функции Resize на значение 10, 5 и 0 соответственно. Потом с помощью `std::fill` происходит заполнение вектора A строками «OMG». Далее с помощью range-based for элементы стека A помещаются в строковый поток. Потом с помощью макроса `EXPECT_EQ` из GoogleTest проверяются на равенства: содержимое потока и 10 копиями строки «OMG », размер B - и 5, C - и 0.

4.2.3 Проверка работоспособности остальных функций

Создается один объект класса, хранящий `long` с помощью конструктора по умолчанию. Затем в конец стека добавляются две сущности, потом 2 в начало стека. Далее это объект копируются в новый, из которого удаляется последний элемент, первый и снова последний. Далее вычисляется сумма элемента стека A с помощью функции `std::accumulate`. В конце с помощью макроса `EXPECT_EQ` из GoogleTest проверяется состав стеков (с помощью операции индексации) их размерность, а также значение суммы.

4.3 Тест №3

Процент покрытия - 93.89%

4.4 Проверка работоспособности интерфейса

Создается объект A класса List, хранящй объекты типа double, над которым совершаются различные преобразования:
добавление в конец элемента, добавление в начало, затем опять в конец, затем вставка элемента на позицию 1 (2 раза), вставка элемента на позицию 4 (2 раза), потом удаление с позиции 6 (= с конца), два раза с позиции 1, удаоение с начала, удаление с конца и удаление с нулевой позиции. На каждом этапе изменений с помощью операции `[ ]` и макроса `EXPECT_EQ` из GoogleTest проверялся весь список целиком.

4.5 Проверка работоспособности итераторов (часть 1)

Создается и заполняется список, хранящий объекты типа int. Затем каждый элемент проверяется с помощью макроса `EXPECT_EQ` и цикла range-based for.

4.6 Проверка работоспособности итераторов (часть 2)

Создается и заполняется список K, хранящий объекты типа int. Затем создается еще один список, в который помещаются элементы из K функцией `std::swap`. В конце каждый элемент проверяется с помощью макроса `EXPECT_EQ` и цикла range-based for.

4.7 Тест №4

Процент покрытия - 91.38%

4.7.1 Проверка работоспособности интерфейса

Создается объект A класса Queue, содержащий объекты типа int, над которым совершаются различные преобразования:
добавление в конец элемента, добавление в позицию 1 место, два добавления в конец, еще два добавления в конец, удаление из начала, вставка в позицию 1 место, удаление первого элемента с помощью erase и два добавления в конец. На каждом этапе изменений с помощью операции `[ ]` и макроса `EXPECT_EQ` из `GoogleTest` проверялся весь список целиком.

4.7.2 Проверка работоспособности итераторов

Создается и заполняется псевдослучайными значениями очередь P на 5 элементов, хранящая объекты типа `unsigned int`. Из нее удаляется первый элемент, затем с помощью цикла range-based for ее элементы помещаются в строковый поток. Затем содержимое потока с помощью макроса `EXPECT_EQ` провереется и сравнивается с эталонным.

4.8 Тест №5

Процент покрытия - 96.76%

4.8.1 Проверка работоспособности интерфейса

Создаются два объекта tree1 и tree2 класса Tree, содержащие объекты типа int. tree1 заполняется псевдослучайными значениями, затем вставляется элемент с помощью функции insert. На каждом этапе значения в дереве проверяются с помощью макроса `EXPECT_EQ`. Потом с помощью конструктора копирования создается объект tree3 - полная копия tree1. Проверяется правильность копирования, далее из tree3 удаляется элемент с помощью erase. В конце опять осуществляется проверка элементов дерева.

4.8.2 Проверка работоспособности итераторов

Создается и заполняется псевдослучайными значениями дерево tree1 на 3 ступени, хранящее объекты типа int. Также создается дерево tree2, в которое помещаются значения из tree1 с помощью `std::swap`. Затем с помощью цикла range-based for и макроса `EXPECT_EQ` проверяется содержимое дерева tree2 и сравнивается с эталонным.

Приложение А

А.1 Файл Vector.h

```
identifierstyle
#ifndef Vector_H
#define Vector_H

#include <cstddef>
#include <iterator>
#include <algorithm>

template <typename T>
class Vector
{
private:
        T* data;
        std::size_t size, capacity;

public:
        struct Iterator
        {
                using iterator_category = std::forward_iterator_tag;
                using difference_type = std::ptrdiff_t;
                using value_type = T;
                using pointer = value_type*;
                using reference = value_type&;

                Iterator(pointer ptr) : m_ptr(ptr) {}
                Iterator(const Iterator& mit) : m_ptr(mit.m_ptr) {}

                bool operator< (const Iterator& tmp) const
        { return m_ptr < tmp.m_ptr; }

                difference_type operator-(const Iterator& rhs) const
        { return m_ptr - rhs.m_ptr; }

                reference operator*() const { return *m_ptr; }
                pointer operator->() { return m_ptr; }

                Iterator operator+(std::size_t x) const
        { return Iterator(m_ptr + x); }
                Iterator operator-(std::size_t x) const
        { return Iterator(m_ptr - x); }

                Iterator& operator++() { m_ptr++; return *this; }
                Iterator operator++(int)
        { Iterator tmp = *this; ++(*this); return tmp; }
                Iterator& operator--()  { --m_ptr;      return *this; }
                Iterator operator--(int)
        { Iterator tmp(*this); --m_ptr; return tmp; }

                friend bool operator==
            (const Iterator& a, const Iterator& b)
        { return a.m_ptr == b.m_ptr; };
                friend bool operator!=
            (const Iterator& a, const Iterator& b)
        { return a.m_ptr != b.m_ptr; };

        private:
```

```
56                  pointer m_ptr;
57          };
58
59          Iterator begin() { return Iterator(&data[0]); }
60          Iterator end() { return Iterator(&data[size]); }
61
62                  Vector();
63                  Vector(std::size_t s);
64                  Vector(const Vector & other);
65                  ~Vector();
66
67                  void erase(std::size_t pos);
68                  void insert(std::size_t pos, T val);
69                  void push_back(T value);
70                  void Resize(std::size_t NewSize); // with losing data
71                  std::size_t length() const;
72
73                  T& operator[](std::size_t index);
74                  T& operator[](std::size_t index) const;
75                  Vector& operator=(const Vector & other);
76  };
77
78
79  template <typename T>
80  Vector<T>::Vector()
81      :data{ nullptr }, size{ 0 }, capacity{ 0 }
82  {}
83
84  template <typename T>
85  Vector<T>::Vector(std::size_t s)
86      : size{ s }, capacity{ 2 * s }
87  {
88      data = new T[2 * s];
89  }
90
91  template <typename T>
92  Vector<T>::Vector(const Vector<T>& other)
93      : size{ other.size }, capacity{ other.capacity }
94  {
95      data = new T[size];
96
97      std::copy(other.data, other.data + size, data);
98  }
99
100 template<typename T>
101 Vector<T>::~Vector()
102 {
103     if (data != nullptr)
104     {
105         delete[] data;
106         data = nullptr;
107     }
108 }
109
110 template<typename T>
111 void Vector<T>::erase(std::size_t pos)
112 {
113     if (pos >= size)
114         throw std::runtime_error
115         ("Index is outside the bounds of the vector");
```

```
116
117     for (std::size_t i{ pos }; i < size - 1; ++i)
118     {
119         (*this)[i] = (*this)[i + 1];
120     }
121
122     --size;
123 }
124
125 template<typename T>
126 void Vector<T>::insert(std::size_t pos, T val)
127 {
128     if (pos >= size)
129         throw std::runtime_error
130         ("Index is outside the bounds of the vector");
131
132     if (size + 1 > capacity)
133     {
134         Vector temp(*this);
135         this->Resize(size + 1);
136         for (std::size_t i{ 0 }; i < size - 1; ++i)
137         {
138             (*this)[i] = temp[i];
139         }
140     }
141     else
142     {
143         ++size;
144     }
145
146
147     for (std::size_t i{ size - 1 }; i > pos; --i)
148     {
149         T temp = (*this)[i];
150         (*this)[i] = (*this)[i - 1];
151         (*this)[i - 1] = temp;
152     }
153
154     (*this)[pos] = val;
155
156 }
157
158 template<typename T>
159 void Vector<T>::push_back(T value)
160 {
161     if (size + 1 > capacity)
162     {
163         Vector res(*this);
164
165         Resize(size + 1);
166
167         for (std::size_t i{ 0 }; i < size; ++i)
168             (*this)[i] = res[i];
169     }
170     else
171     {
172         ++size;
173     }
174
175     (*this)[size - 1] = value;
```

```
176 }
177
178 template<typename T>
179 void Vector<T>::Resize(std::size_t NewSize)
180 {
181     data = new T[2 * NewSize];
182
183     capacity = 2 * NewSize;
184     size = NewSize;
185 }
186
187 template<typename T>
188 std::size_t Vector<T>::length() const
189 {
190     return size;
191 }
192
193 template<typename T>
194 T& Vector<T>::operator[](std::size_t index)
195 {
196     if (index >= size)
197         throw std::runtime_error
198         ("Index is outside the bounds of the vector");
199
200     return data[index];
201 }
202
203 template<typename T>
204 T& Vector<T>::operator[](std::size_t index) const
205 {
206     if (index >= size)
207         throw std::runtime_error
208         ("Index is outside the bounds of the vector");
209
210     return data[index];
211 }
212
213 template<typename T>
214 Vector<T>& Vector<T>::operator=(const Vector<T>& other)
215 {
216     if (this != &other)
217     {
218         if (data != nullptr)
219         {
220             delete[] data;
221             data = nullptr;
222         }
223
224         size = other.size;
225         capacity = other.capacity;
226         data = new T[capacity];
227
228         for (std::size_t i{ 0 }; i < size; ++i)
229             data[i] = other.data[i];
230     }
231
232     return *this;
233 }
234
235 #endif //Vector_H
```

A.2 Файл Test.cpp

```
identifierstyle
#include "gtest/gtest.h"
#include "Vector.h"
#include <algorithm>
#include <string>

TEST(vector, ConstructorsAndLength) {
        Vector<int> A, B(4), C(B);
        EXPECT_EQ(A.length(), 0);
        EXPECT_EQ(B.length(), 4);
        EXPECT_EQ(C.length(), 4);
}

TEST(vector, ResizeAndPushBack) {
        Vector<int> A, B(4);
        A.Resize(2); B.Resize(2);

        std::size_t i{ 0 };

        for (auto& elem : A)
        {
                ++i;
                elem = i;
        }

        B.push_back(5);
        A.push_back(1234567);

        EXPECT_EQ(A.length(), 3);
        EXPECT_EQ(B.length(), 3);
        EXPECT_EQ(B[2], 5);
        EXPECT_EQ(A[0], 1);
        EXPECT_EQ(A[1], 2);
        EXPECT_EQ(A[2], 1234567);
}

TEST(vector, EraseInsertOperator) {
        Vector<std::string> A(2);
        Vector<double> B(4);

        double tmp1 = 5040.5;
        std::string tmp2 = "Hello", tmp3 = "World!";

        B.push_back(tmp1);
        A.push_back(tmp2);
        A.insert(0, tmp3);
        A.erase(1);

        B[0] = 24.2;
        B[1] = 120.3;
        B[2] = 6.1;
        B[3] = 720.4;

        std::string tmp5 = "OMG", tmp6("WTF");

        A.insert(2, tmp5);
        A.insert(1, tmp6);
        A.erase(2);
```

```
58
59          std::swap(A[0], A[3]);
60          //std::sort(B.begin(), B.end());
61
62          EXPECT_DOUBLE_EQ(B[2], 6.1);
63          EXPECT_DOUBLE_EQ(B[0], 24.2);
64          EXPECT_DOUBLE_EQ(B[1], 120.3);
65          EXPECT_DOUBLE_EQ(B[3], 720.4);
66          EXPECT_DOUBLE_EQ(B[4], 5040.5);
67          EXPECT_EQ(A.length(), 4);
68          EXPECT_EQ(B.length(), 5);
69          EXPECT_EQ(A[0], std::string("Hello"));
70          EXPECT_EQ(A[1], std::string("WTF"));
71          EXPECT_EQ(A[2], std::string("OMG"));
72          EXPECT_EQ(A[3], std::string("World!"));
73  }
74
75  int main(int argc, char** argv)
76  {
77          ::testing::InitGoogleTest(&argc, argv);
78          return RUN_ALL_TESTS();
79  }
```

Приложение Б

Б.1 Файл Stack.h

identifierstyle

```
#ifndef STACK_H
#define STACK_H

#include <iostream>

template<typename T>
class Stack
{
private:
        std::size_t size, capacity;
        T* data;

public:
        struct Iterator
        {
                using iterator_category = std::forward_iterator_tag;
                using difference_type = std::ptrdiff_t;
                using value_type = T;
                using pointer = T*;
                using reference = T&;

                Iterator(pointer ptr)
                        : m_ptr(ptr)
                {}

                Iterator(const Iterator& mit)
                        : m_ptr(mit.m_ptr)
                {}

                bool operator< (const Iterator& tmp) const
                { return m_ptr < tmp.m_ptr; }
                difference_type operator-(const Iterator& rhs) const
                { return m_ptr - rhs.m_ptr; }

                reference operator*() const
                {
                        return *m_ptr;
                }

                pointer operator->() {
                        return m_ptr;
                }

                Iterator operator+(std::size_t x) const
                {
                        return Iterator(m_ptr + x);
                }

                Iterator operator-(std::size_t x) const
                {
                        return Iterator(m_ptr - x);
                }

                Iterator& operator++()
                {
```

```
56                          m_ptr++;
57                          return *this;
58                  }
59
60                  Iterator operator++(int)
61                  {
62                          Iterator tmp = *this;
63                          ++(*this);
64                          return tmp;
65                  }
66
67                  Iterator& operator--()
68                  {
69                          --m_ptr;
70                          return *this;
71                  }
72
73                  Iterator operator--(int)
74                  {
75                          Iterator tmp(*this);
76                          --m_ptr;
77                          return tmp;
78                  }
79
80                  friend bool operator==
81                          (const Iterator& a, const Iterator& b)
82                  {
83                          return a.m_ptr == b.m_ptr;
84                  }
85
86                  friend bool operator!=
87                          (const Iterator& a, const Iterator& b)
88                  {
89                          return a.m_ptr != b.m_ptr;
90                  }
91
92          private:
93                  pointer m_ptr;
94          };
95
96          Iterator begin() { return Iterator(&data[0]); }
97          Iterator end() { return Iterator(&data[size]); }
98
99          Stack();
100         Stack(const Stack& other);
101         Stack(std::size_t s);
102         ~Stack();
103
104         void Resize(std::size_t NewSize); //with losing data
105         void push_back(T val);
106         void pop_back();
107         void push_up(T val);
108         void pop_up();
109         std::size_t length() const;
110
111         T& operator[](std::size_t index) const;
112 };
113
114 template<typename T>
115 Stack<T>::Stack()
```

```
116         :size{ 0 }, capacity{ 0 }, data{ nullptr }
117 {}
118
119 template<typename T>
120 Stack<T>::Stack(const Stack<T>& other)
121         : size{ other.size }, capacity{ other.capacity }
122 {
123         data = new T[capacity];
124
125         std::copy(other.data, other.data + size, data);
126 }
127
128 template<typename T>
129 Stack<T>::Stack(std::size_t s)
130         : size{ s }, capacity{ 2 * s }
131 {
132         data = new T[2 * s];
133 }
134
135 template<typename T>
136 Stack<T>::~Stack()
137 {
138         if (data != nullptr)
139         {
140                 delete[] data;
141                 data = nullptr;
142         }
143 }
144
145 template<typename T>
146 void Stack<T>::Resize(std::size_t NewSize)
147 {
148         data = new T[2 * NewSize];
149
150         capacity = 2 * NewSize;
151         size = NewSize;
152 }
153
154 template<typename T>
155 void Stack<T>::push_back(T val)
156 {
157         if (size + 1 > capacity)
158         {
159                 Stack res(*this);
160
161                 this->Resize(size + 1);
162
163                 for (std::size_t i{ 0 }; i < size - 1; ++i)
164                         (*this)[i] = res[i];
165         }
166         else
167         {
168                 ++size;
169         }
170
171         (*this)[size - 1] = val;
172 }
173
174 template<typename T>
175 void Stack<T>::pop_back()
```

```
{
        if (size == 0) throw std::runtime_error("Stack is empty");

        --size;
}

template<typename T>
void Stack<T>::push_up(T val)
{
        if (size + 1 > capacity)
        {
                Stack temp(*this);
                this->Resize(size + 1);
                for (std::size_t i{ 1 }; i < size; ++i)
                {
                        (*this)[i] = temp[i - 1];
                }
        }
        else
        {
                ++size;

                for (std::size_t i{ size - 1 }; i > 0; --i)
                {
                        T temp = (*this)[i];
                        (*this)[i] = (*this)[i - 1];
                        (*this)[i - 1] = temp;
                }
        }

        (*this)[0] = val;
}

template<typename T>
void Stack<T>::pop_up()
{
        if (size == 0) throw std::runtime_error("Stack is empty");

        for (std::size_t i{ 0 }; i < size - 1; ++i)
                (*this)[i] = (*this)[i + 1];

        --size;
}

template<typename T>
std::size_t Stack<T>::length() const
{
        return size;
}

template<typename T>
T& Stack<T>::operator[](std::size_t index) const
{
        if (index >= size)
                throw std::runtime_error
                ("Index is outside the bounds of the vector");

        return data[index];
}

```

```
236 #endif //STACK_H
```

Б.2 Файл Test1.cpp

```
identifierstyle
1  #include "gtest/gtest.h"
2  #include "Stack.h"
3  #include <string>
4  #include <sstream>
5  #include <numeric>
6  #include <iostream>
7
8  TEST(stack, ConstructorsLengthResize) {
9          Stack<std::string> A, B(4), C(B);
10         A.Resize(10);
11         B.Resize(5);
12         C.Resize(0);
13
14         EXPECT_EQ(A.length(), 10);
15         EXPECT_EQ(B.length(), 5);
16         EXPECT_EQ(C.length(), 0);
17 }
18
19 TEST(stack, Iter) {
20         Stack<std::string> A, B(4), C(B);
21         A.Resize(10);
22         B.Resize(5);
23         C.Resize(0);
24
25         std::fill(A.begin(), A.end(), "OMG");
26
27         std::stringstream ss;
28
29         for (const auto& elem : A)
30                 ss << elem << ' ';
31
32         std::string stroka;
33         std::getline(ss, stroka);
34
35         EXPECT_EQ(stroka, "OMG OMG OMG OMG OMG OMG OMG OMG OMG OMG ");
36         EXPECT_EQ(B.length(), 5);
37         EXPECT_EQ(C.length(), 0);
38 }
39
40 TEST(stack, Other) {
41         Stack<long> A;
42
43         long tmp1 = 987654321;
44         long tmp2 = 123456789;
45         long tmp3 = 0;
46
47         A.push_back(tmp1); // {tmp1}
48         A.push_back(tmp2); // {tmp1, tmp2}
49         A.push_up(tmp3);   // {tmp3, tmp1, tmp2}
50         A.push_up(tmp2);   // {tmp2, tmp3, tmp1, tmp2}
51
52         Stack<long> B(A);
53
54         B.pop_back();
```

```
55          B.pop_up();
56          B.pop_back();
57
58          long sum = std::accumulate(A.begin(), A.end(), 0);
59
60          EXPECT_EQ(A[0], tmp2);
61          EXPECT_EQ(A[1], tmp3);
62          EXPECT_EQ(A[2], tmp1);
63          EXPECT_EQ(A[3], tmp2);
64          EXPECT_EQ(A.length(), 4);
65          EXPECT_EQ(B.length(), 1);
66          EXPECT_EQ(B[0], tmp3);
67          EXPECT_EQ(sum, 1234567899);
68  }
69
70  int main(int argc, char** argv)
71  {
72          ::testing::InitGoogleTest(&argc, argv);
73          return RUN_ALL_TESTS();
74  }
```

Приложение В

В.1 Файл List.hpp

```
identifierstyle
#ifndef LIST_HPP
#define LIST_HPP

#include <iostream>
#include <algorithm>

template <typename T>
class List
{
private:
    struct Node
    {
        Node* next = nullptr;
        T value;

        Node(T val)
            :next{ nullptr }, value{ val }
        {}
    };

    Node* head;

public:
    struct Iterator
    {
        using iterator_category = std::forward_iterator_tag;
        using difference_type = std::ptrdiff_t;
        using value_type = T;
        using pointer = value_type*;
        using reference = value_type&;

        Iterator(Node* ptr) : m_ptr(ptr) {}
        Iterator(const Iterator& mit) : m_ptr(mit.m_ptr) {}

        const T& operator*() const { return m_ptr->value; }
        T& operator*() { return m_ptr->value; }
        pointer operator->() { return &m_ptr; }


        Iterator& operator++() {
            m_ptr = m_ptr->next;
            return *this;
        }
        Iterator operator++(int)
        { Iterator tmp = *this; ++(*this); return tmp; }

        friend bool operator==
            (const Iterator& a, const Iterator& b)
        { return a.m_ptr == b.m_ptr; };
        friend bool operator!=
            (const Iterator& a, const Iterator& b)
        { return a.m_ptr != b.m_ptr; };

    private:
        Node* m_ptr;
```

```
56      };
57
58      Iterator begin() const { return Iterator(head); }
59      Iterator end()   const { return Iterator(nullptr); }
60
61      List();
62      List(const List& other) = default;
63      ~List() = default;
64
65      void erase(std::size_t index);
66      void insert(std::size_t index, T val);
67      void push_back(T val);
68      void pop_back();
69      void push_front(T val);
70      void pop_front();
71
72      std::size_t length() const;
73      size_t size() const { return std::distance(begin(), end()); }
74
75      Node*& operator[](std::size_t index);
76      Node*& operator[](std::size_t index) const;
77  };
78
79  template <typename T>
80  List<T>::List()
81      :head{ nullptr }
82  {}
83
84  template <typename T>
85  void List<T>::erase(std::size_t index)
86  {
87      if (index >= length())
88          throw std::runtime_error
89          ("index was outside the bounds of the list");
90      if (length() == 1)
91          throw std::runtime_error
92          ("Attempt to delete the head of the list");
93
94      if (index == length() - 1) pop_back();
95      else if (index == 0) pop_front();
96      else (*this)[index - 1]->next = (*this)[index]->next;
97  }
98
99  template <typename T>
100 void List<T>::insert(std::size_t index, T val)
101 {
102     if (index >= length())
103         throw std::runtime_error
104         ("index was outside the bounds of the list");
105
106     if (index == 0) push_front(val);
107     else {
108         Node* tmp = new Node(val);
109
110         tmp->next = (*this)[index];
111         (*this)[index - 1]->next = tmp;
112     }
113 }
114
115 template <typename T>
```

```
116 void List<T>::push_back(T val)
117 {
118     Node* tmp = new Node(val);
119     if (head == nullptr)
120         head = tmp;
121     else {
122         Node* current = head;
123 
124         while (current->next != nullptr)
125             current = current->next;
126 
127         current->next = tmp;
128     }
129 }
130 
131 template <typename T>
132 void List<T>::pop_back()
133 {
134     if (length() == 0)
135         throw std::runtime_error
136         ("List is empty");
137 
138     if (length() == 1)
139         throw std::runtime_error
140         ("Attempt to delete the head of the list");
141 
142     (*this)[length() - 2]->next = nullptr;
143 }
144 
145 template <typename T>
146 void List<T>::push_front(T val)
147 {
148     Node* tmp = new Node(val);
149     tmp->next = head;
150     head = tmp;
151 }
152 
153 template <typename T>
154 void List<T>::pop_front()
155 {
156     if (length() == 0)
157         throw std::runtime_error
158         ("List is empty");
159 
160     if (length() == 1)
161         throw std::runtime_error
162         ("Attempt to delete the head of the list");
163 
164     head = (*this)[1];
165 }
166 
167 template <typename T>
168 std::size_t List<T>::length() const
169 {
170     if (head == nullptr) return 0;
171 
172     std::size_t size{ 1 };
173     Node* current = head;
174 
175     while (current->next != nullptr)
```

```
176     {
177         current = current->next;
178         ++size;
179     }
180 
181     return size;
182 }
183 
184 template <typename T>
185 typename List<T>::Node*& List<T>::operator[](std::size_t index)
186 {
187     if (index > length())
188         throw std::runtime_error
189         ("index was outside the bounds of the list");
190 
191     Node* current = head;
192 
193     for (std::size_t pos{ 0 }; pos < index; ++pos)
194     {
195         current = current->next;
196     }
197 
198     return current;
199 }
200 
201 template <typename T>
202 typename List<T>::Node*& List<T>::operator[](std::size_t index) const
203 {
204     if (index >= length())
205         throw std::runtime_error
206         ("index was outside the bounds of the list");
207 
208     Node current = head;
209 
210     for (std::size_t pos{ 0 }; pos < index; ++pos)
211     {
212         current = current.next;
213     }
214 
215     return current;
216 }
217 #endif //LIST_HPP
```

B.2 Файл Test2.cpp

```
identifierstyle
1  #include "gtest/gtest.h"
2  #include "List.hpp"
3  #include <vector>
4 
5  TEST(list, All) {
6          for (std::size_t j{ 0 }; j < 1000; ++j)
7          {
8                  List<double> A;
9                  std::size_t i{ 0 };
10                 std::vector<double> tmp(6);
11 
12                 for (double elem : tmp)
13                 {
```

```
14                  tmp.push_back((10 * i + i) / 10);
15                  ++i;
16              }
17
18              A.push_back(tmp[0]); // {tmp1}
19
20              EXPECT_EQ(A[0]->value, tmp[0]);
21
22              A.push_front(tmp[1]); // {tmp2, tmp1}
23
24              EXPECT_EQ(A[0]->value, tmp[1]);
25              EXPECT_EQ(A[1]->value, tmp[0]);
26
27              A.push_back(tmp[2]); // {tmp2, tmp1, tmp3}
28
29              EXPECT_EQ(A[0]->value, tmp[1]);
30              EXPECT_EQ(A[1]->value, tmp[0]);
31              EXPECT_EQ(A[2]->value, tmp[2]);
32
33              A.insert(1, tmp[3]); //{tmp2, tmp4, tmp1, tmp3}
34
35              EXPECT_EQ(A[0]->value, tmp[1]);
36              EXPECT_EQ(A[1]->value, tmp[3]);
37              EXPECT_EQ(A[2]->value, tmp[0]);
38              EXPECT_EQ(A[3]->value, tmp[2]);
39
40              A.insert(1, tmp[4]);
41              // {tmp2, tmp5, tmp4, tmp1, tmp3}
42
43              EXPECT_EQ(A[0]->value, tmp[1]);
44              EXPECT_EQ(A[1]->value, tmp[4]);
45              EXPECT_EQ(A[2]->value, tmp[3]);
46              EXPECT_EQ(A[3]->value, tmp[0]);
47              EXPECT_EQ(A[4]->value, tmp[2]);
48
49              A.insert(4, tmp[5]);
50              // {tmp2, tmp5, tmp4, tmp1,  tmp6, tmp3}
51
52              EXPECT_EQ(A[0]->value, tmp[1]);
53              EXPECT_EQ(A[1]->value, tmp[4]);
54              EXPECT_EQ(A[2]->value, tmp[3]);
55              EXPECT_EQ(A[3]->value, tmp[0]);
56              EXPECT_EQ(A[4]->value, tmp[5]);
57              EXPECT_EQ(A[5]->value, tmp[2]);
58
59              A.insert(4, tmp[0]);
60              // {tmp2, tmp5, tmp4, tmp1, tmp1, tmp6, tmp3}
61
62              EXPECT_EQ(A[0]->value, tmp[1]);
63              EXPECT_EQ(A[1]->value, tmp[4]);
64              EXPECT_EQ(A[2]->value, tmp[3]);
65              EXPECT_EQ(A[3]->value, tmp[2]);
66              EXPECT_EQ(A[4]->value, tmp[0]);
67              EXPECT_EQ(A[5]->value, tmp[5]);
68              EXPECT_EQ(A[6]->value, tmp[2]);
69
70              A.erase(6);
71              // {tmp2, tmp5, tmp4, tmp1, tmp1, tmp6}
72
73              EXPECT_EQ(A[0]->value, tmp[1]);
```

```
74                  EXPECT_EQ(A[1]->value, tmp[4]);
75                  EXPECT_EQ(A[2]->value, tmp[3]);
76                  EXPECT_EQ(A[3]->value, tmp[0]);
77                  EXPECT_EQ(A[4]->value, tmp[0]);
78                  EXPECT_EQ(A[5]->value, tmp[5]);
79
80                  A.erase(1);
81                  // {tmp2, tmp4, tmp1, tmp1, tmp6}
82
83                  EXPECT_EQ(A[0]->value, tmp[1]);
84                  EXPECT_EQ(A[1]->value, tmp[3]);
85                  EXPECT_EQ(A[2]->value, tmp[0]);
86                  EXPECT_EQ(A[3]->value, tmp[0]);
87                  EXPECT_EQ(A[4]->value, tmp[5]);
88
89                  A.erase(1);
90                  // {tmp2, tmp1, tmp1, tmp6}
91
92                  EXPECT_EQ(A[0]->value, tmp[1]);
93                  EXPECT_EQ(A[1]->value, tmp[0]);
94                  EXPECT_EQ(A[2]->value, tmp[0]);
95                  EXPECT_EQ(A[3]->value, tmp[5]);
96
97                  List<double> B(A);
98
99                  EXPECT_EQ(B[0]->value, tmp[1]);
100                 EXPECT_EQ(B[1]->value, tmp[0]);
101                 EXPECT_EQ(B[2]->value, tmp[0]);
102                 EXPECT_EQ(B[3]->value, tmp[5]);
103
104                 A.pop_front(); // {tmp1, tmp1, tmp6}
105
106                 EXPECT_EQ(A[0]->value, tmp[0]);
107                 EXPECT_EQ(A[1]->value, tmp[0]);
108                 EXPECT_EQ(A[2]->value, tmp[5]);
109
110                 A.pop_back(); // {tmp1, tmp1}
111
112                 EXPECT_EQ(A[0]->value, tmp[0]);
113                 EXPECT_EQ(A[1]->value, tmp[0]);
114
115                 A.erase(0); // {tmp1}
116
117                 EXPECT_EQ(A[0]->value, tmp[0]);
118         }
119 }
120
121 TEST(list, Iterator)
122 {
123         for (std::size_t j{ 0 }; j < 1000; ++j)
124         {
125                 List<int> K;
126                 std::vector<int> tmp = { 4 };
127                 K.push_back(4);
128                 for (int i{ 1 }; i < rand() % 5000; ++i)
129                 {
130                         tmp.push_back(rand());
131                         K.push_back(tmp[i]);
132                 }
133
```

```
134                 std::size_t i{ 0 };
135
136                 for (auto elem : K)
137                 {
138                         EXPECT_EQ(elem, tmp[i]);
139                         ++i;
140                 }
141         }
142 }
143
144 TEST(list, Iterator2)
145 {
146         for (std::size_t j{ 0 }; j < 1000; ++j)
147         {
148                 List<int> K;
149                 std::vector<int> tmp = { 4 };
150                 K.push_back(4);
151                 for (int i{ 1 }; i < rand() % 5000; ++i)
152                 {
153                         tmp.push_back(rand());
154                         K.push_back(tmp[i]);
155                 }
156
157                 List<int> L;
158                 std::swap(K, L);
159                 std::size_t i{ 0 };
160
161                 for (auto elem : L)
162                 {
163                         EXPECT_EQ(elem, tmp[i]);
164                         ++i;
165                 }
166
167         }
168 }
169
170 int main(int argc, char** argv)
171 {
172         ::testing::InitGoogleTest(&argc, argv);
173         return RUN_ALL_TESTS();
174 }
```

Приложение Г

Г.1 Файл Queue.hpp

```
identifierstyle
#ifndef QUEUE_HPP
#define QUEUE_HPP

#include <iostream>

template <typename T>
class Queue
{
private:
        T* data;
        std::size_t size, count;
        int rear, front;

public:
        struct Iterator
        {
                using iterator_category = std::forward_iterator_tag;
                using difference_type = std::ptrdiff_t;
                using value_type = T;
                using pointer = value_type*;
                using reference = value_type&;

                Iterator(pointer ptr) : m_ptr(ptr) {}

                Iterator(pointer ptr, std::size_t size,
                        T* zero, std::size_t pos)
                        : m_ptr(ptr), size(size),
                        zero(zero), pos(pos) {}

                Iterator(const Iterator& mit)
                        : m_ptr(mit.m_ptr), size(mit.size),
                        zero(mit.zero), pos(mit.pos) {}

                reference operator*() const { return *m_ptr; }
                pointer operator->() { return m_ptr; }

                Iterator& operator++() {
                        ++pos;

                        if (m_ptr - zero == size - 1)
                                m_ptr = zero;
                        else
                                m_ptr++;

                        return *this;
                }

                Iterator operator++(int)
                {
                        Iterator tmp = *this;
                        ++(*this);
                        return tmp;
                }

                friend bool operator==
```

```
56                      (const Iterator& a, const Iterator& b)
57              { return a.pos == b.pos; }
58              friend bool operator!=
59                      (const Iterator& a, const Iterator& b)
60              { return a.pos != b.pos; }
61
62              std::size_t size;
63              T* zero;
64              std::size_t pos;
65      private:
66              T* m_ptr;
67      };
68
69      Iterator begin() const
70      {
71              Iterator i(&data[front], size, &data[0], 0);
72              return i;
73      }
74
75      Iterator end() const
76      {
77              Iterator i(&data[rear] + 1, size, &data[0], size);
78              return i;
79      }
80
81
82      Queue();
83      Queue(const Queue& other);
84      Queue(std::size_t size);
85      ~Queue();
86
87      void erase(std::size_t index);
88      void insert(std::size_t index, T val);
89      void pop_front();
90      void push_back(T val);
91
92      void Dump() const;
93
94      std::size_t length() const;
95
96      T& operator[](std::size_t index);
97      T& operator[](std::size_t index) const;
98  };
99
100 template <typename T>
101 Queue<T>::Queue()
102         :data{ nullptr }, size{1}, count{0}, rear{0}, front{-1}
103 {}
104
105 template <typename T>
106 Queue<T>::Queue(const Queue& other)
107         : size{ other.size }, count{ other.count }, rear{ other.rear },
108         front{ other.front }
109 {
110         data = new T[size];
111
112         std::copy(other.data, other.data + size, data);
113 }
114
115 template <typename T>
```

```
116 Queue<T>::Queue(std::size_t size)
117         : size{ size }, count{ 0 }, rear{ 0 }, front{ -1 }
118 {
119         data = new T[size];
120 
121         std::fill(data, data + size, 0);
122 }
123 
124 template <typename T>
125 Queue<T>::~Queue()
126 {
127         if (data != nullptr)
128         {
129                 delete[] data;
130                 data = nullptr;
131         }
132 }
133 
134 template <typename T>
135 void Queue<T>::erase(std::size_t index)
136 {
137         if (index >= size)
138                 throw std::runtime_error
139                 ("Index is outside the bounds of the queue");
140 
141         if (index == front) pop_front();
142         else if (index != rear)
143         {
144                 int ind = index;
145                 std::size_t end = (ind > rear) ? (rear + size) : rear;
146 
147                 for (std::size_t i{ index + 1 }; i < end; ++i)
148                 {
149                         T temp = (*this)[i % size];
150                         (*this)[i % size] = (*this)[(i - 1) % size];
151                         (*this)[(i - 1) % size] = temp;
152                 }
153                 int a = rear - 1;
154                 a = (a >= 0) ? a : (a + size);
155                 std::swap((*this)[rear], (*this)[a]);
156 
157                 (*this)[rear] = 0;
158 
159                 --count;
160         }
161 }
162 
163 template <typename T>
164 void Queue<T>::insert(std::size_t index, T val)
165 {
166         if (index >= size)
167                 throw std::runtime_error
168                 ("Index is outside the bounds of the queue");
169 
170         if (index == rear + 1) push_back(val);
171 
172         if ((*this)[index] == 0/*nullptr*/)
173                 (*this)[index] = val;
174         else {
175                 if (index != front)
```

```
176                 {
177                         std::size_t end = (index - rear > 0) ?
178                                 rear + size : rear;
179                         for (std::size_t i{ index + 1 }; i < end; ++i)
180                         {
181                                 T temp = (*this)[i % size];
182                                 (*this)[i % size] =
183                                         (*this)[(i - 1) % size];
184                                 (*this)[(i - 1) % size] = temp;
185                         }
186                         std::swap((*this)[index], (*this)[rear]);
187                         (*this)[index] = val;
188                 }
189         }
190 }
191
192 template <typename T>
193 void Queue<T>::pop_front()
194 {
195         if (count == 0)
196                 throw std::runtime_error
197                 ("Queue is empty");
198
199         (*this)[front] = 0/*nullptr*/;
200
201         for (std::size_t i = front; i < size + front; ++i)
202         {
203                 T temp = (*this)[i % size];
204                 (*this)[i % size] = (*this)[(i - 1) % size];
205                 (*this)[(i - 1) % size] = temp;
206         }
207
208         rear = (rear - 1) % size;
209
210         rear = (rear >= 0) ? rear : rear + size;
211
212         int a = (rear) % size;
213         int b = (front - 1) % size;
214         b = (b >= 0) ? b : b + size;
215
216         std::swap((*this)[a], (*this)[b]);
217
218         --count;
219 }
220
221 template <typename T>
222 void Queue<T>::push_back(T val)
223 {
224         if ((rear + 1) % size == front)
225                 front = (front + 1) % size;
226
227         if (count == 0)
228                 ++front;
229
230         if (count < size)
231                 ++count;
232
233         if ((*this)[rear] != 0/*nullptr*/)
234                 rear = (rear + 1) % size;
235
```

```
236         (*this)[rear] = val;
237 }
238
239 template <typename T>
240 std::size_t Queue<T>::length() const
241 {
242         return size;
243 }
244
245 template <typename T>
246 T& Queue<T>::operator[](std::size_t index)
247 {
248         if (index >= size)
249                 throw std::runtime_error
250                 ("Index is outside the bounds of the queue");
251
252         return data[index];
253 }
254
255 template <typename T>
256 T& Queue<T>::operator[](std::size_t index) const
257 {
258         if (index >= size)
259                 throw std::runtime_error
260                 ("Index is outside the bounds of the queue");
261
262         return data[index];
263 }
264
265 template <typename T>
266 void Queue<T>::Dump() const
267 {
268         for (std::size_t i{ 0 }; i < length(); ++i)
269                 std::cout << (*this)[i] << ' ';
270
271         std::cout << '\n';
272
273         std::cout << front << ' ' << rear << '\n';
274 }
275 #endif //QUEUE_HPP
```

Г.2 Файл Test3.cpp

```
identifierstyle
1  #include "gtest/gtest.h"
2  #include "Queue.hpp"
3  #include <vector>
4  #include <random>
5  #include <sstream>
6
7  TEST(queue, All) {
8          for (std::size_t j{ 0 }; j < 1000; ++j)
9          {
10                 Queue<int> A(4);
11
12                 std::vector<int> tmp(9);
13
14                 for (std::size_t i{ 0 }; i < 9; ++i)
15                         tmp[i] = rand() + 1;
```

```
16
17                  A.push_back(tmp[0]); // {tmp1, - , - , - }
18
19                  EXPECT_EQ(A[0], tmp[0]);
20
21                  A.insert(1, tmp[1]); // {tmp1, tmp2 , - , - }
22
23                  EXPECT_EQ(A[0], tmp[0]);
24                  EXPECT_EQ(A[1], tmp[1]);
25
26                  A.push_back(tmp[2]); A.push_back(tmp[3]);
27                  // {tmp1, tmp2, tmp3, tmp4}
28
29                  EXPECT_EQ(A[0], tmp[0]);
30                  EXPECT_EQ(A[1], tmp[1]);
31                  EXPECT_EQ(A[2], tmp[2]);
32                  EXPECT_EQ(A[3], tmp[3]);
33
34                  A.push_back(tmp[4]);    A.push_back(tmp[5]);
35                  // {tmp5, tmp6 tmp3, tmp4}
36
37                  EXPECT_EQ(A[0], tmp[4]);
38                  EXPECT_EQ(A[1], tmp[5]);
39                  EXPECT_EQ(A[2], tmp[2]);
40                  EXPECT_EQ(A[3], tmp[3]);
41
42                  A.pop_front(); // {tmp6 , - , tmp4, tmp5}
43
44                  EXPECT_EQ(A[0], tmp[5]);
45                  EXPECT_EQ(A[1], 0/*nullptr*/);
46                  EXPECT_EQ(A[2], tmp[3]);
47                  EXPECT_EQ(A[3], tmp[4]);
48
49                  A.insert(1, tmp[6]); // {tmp6 , tmp7 , tmp4, tmp5}
50
51                  EXPECT_EQ(A[0], tmp[5]);
52                  EXPECT_EQ(A[1], tmp[6]);
53                  EXPECT_EQ(A[2], tmp[3]);
54                  EXPECT_EQ(A[3], tmp[4]);
55
56                  A.erase(0); // {tmp7, - , tmp4, tmp5}
57
58                  EXPECT_EQ(A[0], tmp[6]);
59                  EXPECT_EQ(A[1], 0/*nullptr*/);
60                  EXPECT_EQ(A[2], tmp[3]);
61                  EXPECT_EQ(A[3], tmp[4]);
62
63                  A.push_back(tmp[7]);    A.push_back(tmp[8]);
64                  // {tmp7, tmp8, tmp9, tmp5}
65
66                  EXPECT_EQ(A[0], tmp[6]);
67                  EXPECT_EQ(A[1], tmp[7]);
68                  EXPECT_EQ(A[2], tmp[8]);
69                  EXPECT_EQ(A[3], tmp[4]);
70          }
71  }
72
73  TEST(queue, Iterator)
74  {
75          for (std::size_t j{ 0 }; j < 1000; ++j)
```

```
76          {
77                  std::random_device rd;
78                  std::mt19937 mersenne(rd());
79                  std::stringstream ss;
80                  Queue<unsigned> P(5);
81
82                  std::vector<unsigned> tmp(8);
83
84                  for (std::size_t i{ 0 }; i < 8; ++i)
85                  {
86                          tmp[i] = mersenne();
87                          P.push_back(tmp[i]);
88                  }
89
90                  P.pop_front();
91
92                  for (auto elem : P)
93                          ss << elem << ' ';
94
95                  std::string str, ans,
96                          tmp4 = std::to_string(tmp[4]),
97                          tmp5 = std::to_string(tmp[5]),
98                          tmp6 = std::to_string(tmp[6]),
99                          tmp7 = std::to_string(tmp[7]);
100                 getline(ss, str);
101                 ans = tmp4 + ' ' + tmp5 + ' ' + tmp6 + ' ' + tmp7
102                         + " 0 ";
103
104                 EXPECT_EQ(str, ans);
105         }
106 }
107
108 int main(int argc, char** argv)
109 {
110         ::testing::InitGoogleTest(&argc, argv);
111         return RUN_ALL_TESTS();
112 }
```

Приложение Д

Д.1 Файл Tree.hpp

```
identifierstyle
#ifndef TREE_HPP
#define TREE_HPP

#include <iostream>
#include <cmath>
#include <vector>

// Numeration of tree starts from 1
// Not from 0! Head element has index 1
template <typename T>
class Tree
{
private:
        struct Node {
                Node* left = nullptr;
                Node* right = nullptr;
                T val;
        };

        Node* head;
        std::size_t steps = 0, size = 0;

public:
    struct Iterator
    {
        using iterator_category = std::forward_iterator_tag;
        using difference_type = std::ptrdiff_t;
        using value_type = T;
        using pointer = value_type*;
        using reference = value_type&;

        Iterator(Node* ptr, std::size_t ind, Node* beg)
            : m_ptr(ptr), index(ind), beg (beg) {}
        Iterator(const Iterator& mit)
            : m_ptr(mit.m_ptr) {}

        const T& operator*() const { return m_ptr->val; }
        T& operator*() { return m_ptr->val; }
        pointer operator->() { return &m_ptr; }


        Iterator& operator++() {
            ++index;
            std::vector<bool> bits = Dec2Bin(index);
            bits.erase(bits.begin());

            Node* current = beg;

            for (bool elem : bits)
                if (elem)
                    current = current->right;
                else
                    current = current->left;

            m_ptr = current;
```

```
56              return *this;
57          }
58
59
60          Iterator operator++(int)
61          {
62              Iterator tmp = *this; ++(*this); return tmp;
63          }
64
65          friend bool operator==
66              (const Iterator& a, const Iterator& b)
67          {
68              return a.m_ptr == b.m_ptr;
69          };
70          friend bool operator!=
71              (const Iterator& a, const Iterator& b)
72          {
73              return a.m_ptr != b.m_ptr;
74          };
75
76      private:
77          Node* m_ptr;
78          std::size_t index;
79          Node* beg;
80      };
81
82      Iterator begin() const { return Iterator(head, 1, head); }
83      Iterator end()   const { return Iterator(nullptr, size+1, head); }
84
85
86          Tree();
87          Tree(const Tree& other);
88          Tree(std::size_t steps);
89          ~Tree() = default;
90
91          //void Resize(std::size_t NewSteps);
92
93          void erase(std::size_t index);
94          //Deleting all children starting from index
95
96          void insert(std::size_t index, T val);
97
98          std::size_t length() const;
99
100         Node*& operator[](std::size_t index);
101         Node*& operator[](std::size_t index) const;
102
103     static
104     std::vector<bool> Dec2Bin(std::size_t param);
105 };
106
107
108 template<typename T>
109 std::vector<bool> Tree<T>::Dec2Bin(std::size_t param)
110 {
111     std::vector<bool> bits;
112
113     while (param)
114     {
115         bits.push_back(param % 2);
```

```
116             param >>= 1;
117         }
118
119         std::reverse(bits.begin(), bits.end());
120
121         return bits;
122     }
123
124     template <typename T>
125     Tree<T>::Tree()
126         :head{ nullptr }
127     {}
128
129     template <typename T>
130     Tree<T>::Tree(const Tree<T>& other)
131         : steps{ other.steps }, size{ other.size }
132     {
133         head = new Node;
134         head = other.head;
135     }
136
137     template <typename T>
138     Tree<T>::Tree(std::size_t steps)
139         : steps{ steps }
140     {
141         head = new Node;
142         size = std::pow(2, steps) - 1;
143
144         for (std::size_t i{ 2 }; i <= size / 2; ++i)
145         {
146             (*this)[i] = new Node;
147             (*this)[i]->val = 0;
148         }
149     }
150
151     template <typename T>
152     void Tree<T>::erase(std::size_t index)
153     {
154         if (index > size || index == 0)
155             throw std::runtime_error
156             ("Index is outside the bounds of the vector");
157
158         (*this)[index]->val = 0;
159         (*this)[index]->left = nullptr;
160         (*this)[index]->right = nullptr;
161     }
162
163     template <typename T>
164     void Tree<T>::insert(std::size_t index, T val)
165     {
166         if (index > size || index == 0)
167             throw std::runtime_error
168             ("Index is outside the bounds of the vector");
169
170         double step = std::log2(index);
171         std::size_t delta = steps - static_cast<int>(step) - 1;
172
173         steps += delta;
174
175         size = std::pow(2, steps) - 1;
```

```
176
177         Node* cur = new Node;
178
179         cur->left = (*this)[index];
180         cur->right = nullptr;
181         cur->val = val;
182
183         (index % 2) ? ((*this)[index / 2]->right = cur)
184             : ((*this)[index / 2]->left = cur);
185     }
186
187     template <typename T>
188     std::size_t Tree<T>::length() const
189     {
190         return size;
191     }
192
193     template <typename T>
194     typename Tree<T>::Node*& Tree<T>::operator[](std::size_t index)
195     {
196         /*if (head == nullptr)
197             throw std::runtime_error
198                 ("Tree is empty");*/
199
200         if (index > size || index == 0)
201             throw std::runtime_error
202             ("Index is outside the bounds of the vector");
203
204         if (index == 1) return head;
205
206         std::vector<bool> bits = Dec2Bin(index);
207         bits.erase(bits.begin());
208
209         Node* current = head;
210
211         for (bool elem : bits)
212             if (elem)
213                 current = current->right;
214             else
215                 current = current->left;
216
217         if (current == nullptr)
218             current = new Node;
219
220         (index % 2) ? ((*this)[index / 2]->right = current)
221             : ((*this)[index / 2]->left = current);
222
223         return current;
224     }
225
226     template <typename T>
227     typename Tree<T>::Node*& Tree<T>::operator[](std::size_t index) const
228     {
229         if (head == nullptr)
230             throw std::runtime_error
231             ("Tree is empty");
232
233         if (index >= size || index == 0)
234             throw std::runtime_error
235             ("Index is outside the bounds of the vector");
```

```
236
237     std::vector<bool> bits = Dec2Bin(index);
238     bits.erase(bits.begin());
239
240     Node* current = head;
241
242     for (bool elem : bits)
243         if (elem)
244             current = current->right;
245         else
246             current = current->left;
247
248     if (current == nullptr)
249         throw std::runtime_error
250         ("Element doesn't exist");
251
252     return current;
253 }
254
255 #endif //TREE_HPP
```

Д.2 Файл Test4.cpp

```
identifierstyle
1  #include "gtest/gtest.h"
2  #include "Tree.hpp"
3  #include <random>
4
5  TEST(tree, All) {
6          for (std::size_t j{ 0 }; j < 500; ++j)
7          {
8                  Tree<int> tree1(3), tree2(4);
9                  std::vector<int>  tmp(8);
10
11                 std::random_device rd;
12                 std::mt19937 mersenne(rd());
13
14                 for (std::size_t i{ 0 }; i < 8; ++i)
15                 {
16                         tmp[i] = mersenne();
17                 }
18
19                 EXPECT_EQ(tree1.length(), 7);
20                 EXPECT_EQ(tree2.length(), 15);
21
22                 tree1[1]->val = tmp[0];
23                 tree1[2]->val = tmp[1];
24                 tree1[3]->val = tmp[2];
25                 tree1[4]->val = tmp[3];
26                 tree1[5]->val = tmp[4];
27                 tree1[6]->val = tmp[5];
28                 tree1[7]->val = tmp[6];
29
30                 EXPECT_EQ(tree1[1]->val, tmp[0]);
31                 EXPECT_EQ(tree1[2]->val, tmp[1]);
32                 EXPECT_EQ(tree1[3]->val, tmp[2]);
33                 EXPECT_EQ(tree1[4]->val, tmp[3]);
34                 EXPECT_EQ(tree1[5]->val, tmp[4]);
35                 EXPECT_EQ(tree1[6]->val, tmp[5]);
```

```
36                  EXPECT_EQ(tree1[7]->val, tmp[6]);
37
38                  tree1.insert(3, tmp[7]);
39
40                  EXPECT_EQ(tree1.length(), 15);
41                  EXPECT_EQ(tree1[1]->val, tmp[0]);
42                  EXPECT_EQ(tree1[2]->val, tmp[1]);
43                  EXPECT_EQ(tree1[3]->val, tmp[7]);
44                  EXPECT_EQ(tree1[4]->val, tmp[3]);
45                  EXPECT_EQ(tree1[5]->val, tmp[4]);
46                  EXPECT_EQ(tree1[6]->val, tmp[2]);
47                  //EXPECT_EQ(tree1[7]->val, nullptr);
48                  //EXPECT_EQ(tree1[8]->val, nullptr);
49                  //EXPECT_EQ(tree1[9]->val, nullptr);
50                  //EXPECT_EQ(tree1[10]->val, nullptr);
51                  //EXPECT_EQ(tree1[11]->val, nullptr);
52                  EXPECT_EQ(tree1[12]->val, tmp[5]);
53                  EXPECT_EQ(tree1[13]->val, tmp[6]);
54                  //EXPECT_EQ(tree1[14]->val, nullptr);
55                  //EXPECT_EQ(tree1[15]->val, nullptr);
56
57                  Tree<int> tree3(tree1);
58
59                  EXPECT_EQ(tree3.length(), 15);
60                  EXPECT_EQ(tree3[1]->val, tmp[0]);
61                  EXPECT_EQ(tree3[2]->val, tmp[1]);
62                  EXPECT_EQ(tree3[3]->val, tmp[7]);
63                  EXPECT_EQ(tree3[4]->val, tmp[3]);
64                  EXPECT_EQ(tree3[5]->val, tmp[4]);
65                  EXPECT_EQ(tree3[6]->val, tmp[2]);
66                  //EXPECT_EQ(tree3[7]->val, nullptr);
67                  //EXPECT_EQ(tree3[8]->val, nullptr);
68                  //EXPECT_EQ(tree3[9]->val, nullptr);
69                  //EXPECT_EQ(tree3[10]->val, nullptr);
70                  //EXPECT_EQ(tree3[11]->val, nullptr);
71                  EXPECT_EQ(tree3[12]->val, tmp[5]);
72                  EXPECT_EQ(tree3[13]->val, tmp[6]);
73                  //EXPECT_EQ(tree3[14]->val, nullptr);
74                  //EXPECT_EQ(tree3[15]->val, nullptr);
75
76                  tree3.erase(2);
77
78                  EXPECT_EQ(tree3.length(), 15);
79                  EXPECT_EQ(tree3[1]->val, tmp[0]);
80                  EXPECT_EQ(tree3[2]->val, 0);
81                  EXPECT_EQ(tree3[3]->val, tmp[7]);
82                  //EXPECT_EQ(tree3[4]->val, &tmp4);
83                  //EXPECT_EQ(tree3[5]->val, &tmp5);
84                  EXPECT_EQ(tree3[6]->val, tmp[2]);
85                  //EXPECT_EQ(tree3[7]->val, nullptr);
86                  //EXPECT_EQ(tree3[8]->val, nullptr);
87                  //EXPECT_EQ(tree3[9]->val, nullptr);
88                  //EXPECT_EQ(tree3[10]->val, nullptr);
89                  //EXPECT_EQ(tree3[11]->val, nullptr);
90                  EXPECT_EQ(tree3[12]->val, tmp[5]);
91                  EXPECT_EQ(tree3[13]->val, tmp[6]);
92                  //EXPECT_EQ(tree3[14]->val, nullptr);
93                  //EXPECT_EQ(tree3[15]->val, nullptr);
94          }
95  }
```

```
96
97  TEST(tree, Iterator) {
98          for (std::size_t j{ 0 }; j < 1000; ++j)
99          {
100                 Tree<int> tree1(3), tree2;
101                 std::vector<int>  tmp(8);
102
103                 std::random_device rd;
104                 std::mt19937 mersenne(rd());
105
106                 for (std::size_t i{ 0 }; i < 8; ++i)
107                 {
108                         tmp[i] = mersenne();
109                 }
110
111                 tree1[1]->val = tmp[0];
112                 tree1[2]->val = tmp[1];
113                 tree1[3]->val = tmp[2];
114                 tree1[4]->val = tmp[3];
115                 tree1[5]->val = tmp[4];
116                 tree1[6]->val = tmp[5];
117                 tree1[7]->val = tmp[6];
118
119                 std::swap(tree1, tree2);
120
121                 std::size_t i{ 0 };
122
123                 for (auto elem : tree2)
124                 {
125                         EXPECT_EQ(elem, tmp[i]);
126                         ++i;
127                 }
128         }
129 }
130
131 int main(int argc, char** argv)
132 {
133         ::testing::InitGoogleTest(&argc, argv);
134         return RUN_ALL_TESTS();
135 }
```