Правительство Российской Федерации

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

Кафедра «Компьютерная безопасность»

ОТЧЕТ
К ЛАБОРАТОРНОЙ РАБОТЕ №2

по дисциплине

«Языки программирования»

Работу выполнил
студент группы СКБ-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

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.

1 Алгоритм решения задачи

1.1 Задача 1

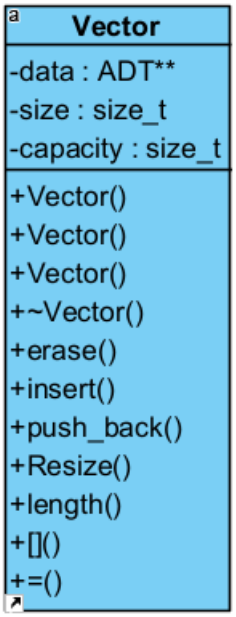

Рис. 1. UML-диаграмма класса Vector.

Для решения данной задачи был разработан класс Vector, UML диаграмма которого приведена на рис. 1, содержащий закрытые поля size, capacity типа std::size_t и data типа ADT**, 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 erase, принимающая индекс элемента в массиве и удаляющая его;
- функция insert, принимающая индекс элемента в массиве и указатель на объект типа ADT, которая вставляет в это место переданный указатель;
- функция push_back, принимающая указатель на объект типа ADT, которая добавляет в конец массива переданный указатель;
- 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

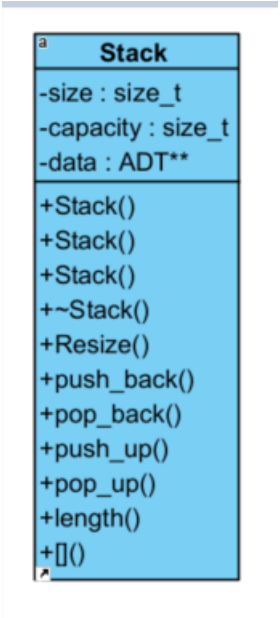

Рис. 2. UML-диаграмма класса Stack.

Для решения данной задачи был разработан класс Stack, UML диаграмма которого приведена на рис. 2, содержащий закрытые поля size, capacity типа std::size_t и data типа ADT**, 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 pop_back, удаляющая последний элемент из массива;

– функция push_back, принимающая указатель на объект типа ADT, которая добавляет в конец массива переданный указатель;

– функция pop_up, удаляющая первый элемент из массива;

– функция push_up, принимающая указатель на объект типа ADT, которая добавляет в начало массива переданный указатель;

– 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

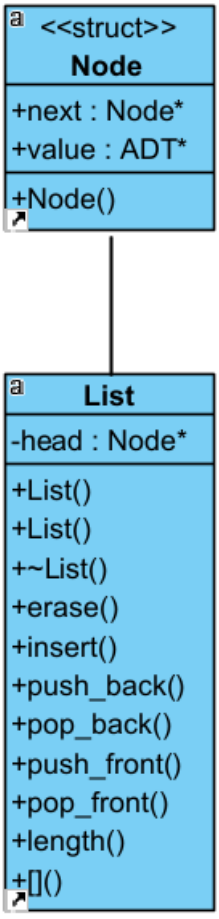

Рис. 3. UML-диаграмма класса List.

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 указатель на ADT.

Также класс содержит:

– конструктор по умолчанию;

– конструктор копирования;

– декструктор;

Помимо этого в классе имеются:

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

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

– функция push_back, принимающая указатель на объект типа ADT, которая добавляет в конец массива переданный указатель;

– функция pop_back, удаляющая последний элемент из массива;

– функция pop_front, удаляющая первый элемент из массива;

– функция push_front, принимающая указатель на объект типа ADT, которая добавляет в начало массива переданный указатель;

– 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

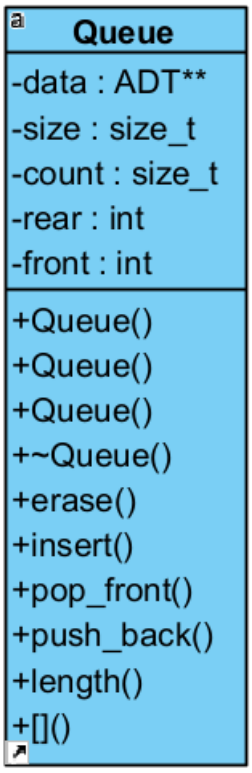

Рис. 4. UML-диаграмма класса Queue.

Для решения данной задачи был разработан класс Queue, UML диаграмма которого приведена на рис. 4, содержащий закрытые поля data тип ADT**, 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 erase, принимающая индекс элемента в очереди и удаляющая его;
- функция insert, принимающая индекс элемента в очереди и указатель на объект типа ADT, которая вставляет в это место переданный указатель;
- функция push_back, принимающая указатель на объект типа ADT, которая добавляет в конец очереди переданный указатель;
- 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 класс 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 erase, принимающая индекс элемента в дереве и удаляющая его и его потмков;
- функция insert, принимающая индекс элемента в очереди и указатель на объект типа ADT, которая вставляет в это место переданный указатель;
- 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;

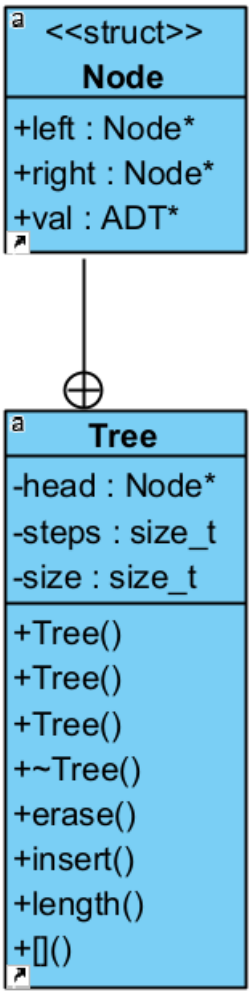

Рис. 5. UML-диаграмма класса Tree.

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 указатель-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 указатель-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 nullptr 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 nullptr.

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 nullptr.

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);

Файлы из которых собирается исполняемый модуль:

– main0 составляется из файлов Test.cpp, Vector.h, Vector.cpp;

– main1 составляется из файлов Test1.cpp, Stack.h, Stack.cpp;

– main2 составляется из файлов Test2.cpp, List.h, List.cpp;

– main3 составляется из файлов Test3.cpp, Queue.h, Queue.cpp;

– main4 составляется из файлов Test4.cpp, Tree.h, Tree.cpp;

Листинг-1. Файл CMakeLists.txt

```
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 -l

project(Laba2)

add_executable(main0 Test.cpp Vector.h Vector.cpp
DateAndTime.h DateAndTime.cpp
BigNumber.h BigNumber.cpp
YearFromAdam.cpp YearFromAdam.h
Matrix.h Matrix.cpp)

add_executable(main1 Test1.cpp Stack.h Stack.cpp
DateAndTime.h DateAndTime.cpp
BigNumber.h BigNumber.cpp
YearFromAdam.cpp YearFromAdam.h
Matrix.h Matrix.cpp)

add_executable(main2 Test2.cpp List.h List.cpp
DateAndTime.h DateAndTime.cpp
BigNumber.h BigNumber.cpp
YearFromAdam.cpp YearFromAdam.h
Matrix.h Matrix.cpp)

add_executable(main3 Test3.cpp Queue.h Queue.cpp
DateAndTime.h DateAndTime.cpp
BigNumber.h BigNumber.cpp
YearFromAdam.cpp YearFromAdam.h
```

```
33 Matrix.h Matrix.cpp)
34
35 add_executable(main4 Test4.cpp Tree.h Tree.cpp
36 DateAndTime.h DateAndTime.cpp
37 BigNumber.h BigNumber.cpp
38 YearFromAdam.cpp YearFromAdam.h
39 Matrix.h Matrix.cpp)
```

4 Тестирование

4.1 Тест №1

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 четырех, 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. Затем размер обоих векторов изменяется с помощью функции Resize на значение 2. 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.

4.1.3 Проверка работоспособности остальных функций

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 четырех, третьего вектора и четырех.

4.2 Тест №2

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 остальных функций

Создается один объект класса 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. В конце с помощью макроса EXPECT_EQ из GoogleTest проверяется состав стеков (с помощью операции индексации) и их размерность.

4.3 Тест №3

Создается объект A класса List, 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.

4.4 Тест №4

Создается объект A класса Queue, 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.

Приложение А

А.1 Файл Vector.h

```
#ifndef Vector_H
#define Vector_H

#include "ADT.h"

class Vector
{
private:
        ADT** data;
        std::size_t size, capacity;

public:
        Vector();
        Vector(std::size_t s);
        Vector(const Vector& other);
   ~Vector();

   void erase(std::size_t pos);
   void insert(std::size_t pos, ADT* val);
   void push_back(ADT* value);
   void Resize(std::size_t NewSize); // with losing data
   std::size_t length() const;

   ADT*& operator[](std::size_t index);
   ADT*& operator[](std::size_t index) const;
   Vector& operator=(const Vector& other);
};

#endif //Vector_H
```

А.2 Файл Vector.cpp

```
#include <iostream>
#include "Vector.h"

Vector::Vector()
    :data{ nullptr }, size{ 0 }, capacity{ 0 }
{}

Vector::Vector(std::size_t s)
    : size{ s }, capacity{ 2 * s }
{
    data = new ADT * [2 * s];
}

Vector::Vector(const Vector& other)
    : size{ other.size }, capacity{ other.capacity }
```

```
16  {
17      data = new ADT * [capacity];
18
19      for (std::size_t i{ 0 }; i < size; ++i)
20          data[i] = other.data[i];
21  }
22
23  Vector::~Vector()
24  {
25      if (data != nullptr)
26      {
27          delete[] data;
28          data = nullptr;
29      }
30  }
31
32  void Vector::erase(std::size_t pos)
33  {
34      if (pos >= size)
35          throw std::runtime_error
36              ("Index is outside the bounds of the vector");
37
38      for (std::size_t i{ pos }; i < size-1; ++i)
39      {
40          (*this)[i] = (*this)[i + 1];
41      }
42
43      --size;
44  }
45
46  void Vector::insert(std::size_t pos, ADT* val)
47  {
48      if(pos >= size)
49          throw std::runtime_error
50              ("Index is outside the bounds of the vector");
51
52      if (size + 1 > capacity)
53      {
54          Vector temp(*this);
55          this->Resize(size+1);
56          for (std::size_t i{ 0 }; i < size-1; ++i)
57          {
58              (*this)[i] = temp[i];
59          }
60      }
61      else
62      {
63          ++size;
64      }
65
```

```
66
67      for (std::size_t i{ size-1 }; i > pos; --i)
68      {
69          ADT* temp = (*this)[i];
70          (*this)[i] = (*this)[i - 1];
71          (*this)[i - 1] = temp;
72      }
73
74      (*this)[pos] = val;
75
76  }
77
78  void Vector::push_back(ADT* value)
79  {
80      if (size + 1 > capacity)
81      {
82          Vector res(*this);
83
84          this->Resize(size + 1);
85
86          for (std::size_t i{ 0 }; i < size; ++i)
87              (*this)[i] = res[i];
88      }
89      else
90      {
91          ++size;
92      }
93
94      (*this)[size - 1] = value;
95  }
96
97  void Vector::Resize(std::size_t NewSize)
98  {
99                  data = new ADT * [2 * NewSize];
100
101                 capacity = 2 * NewSize;
102                 size = NewSize;
103 }
104
105 std::size_t Vector::length() const
106 {
107     return size;
108 }
109
110 ADT*& Vector::operator[](std::size_t index)
111 {
112     if (index >= size)
113         throw std::runtime_error
114             ("Index is outside the bounds of the vector");
115
```

```
116     return data[index];
117 }
118
119 ADT*& Vector::operator[](std::size_t index) const
120 {
121     if (index >= size)
122         throw std::runtime_error
123             ("Index is outside the bounds of the vector");
124
125     return data[index];
126 }
127
128 Vector& Vector::operator=(const Vector& other)
129 {
130     if (this != &other)
131     {
132         if (data != nullptr)
133         {
134             delete[] data;
135             data = nullptr;
136         }
137
138         size = other.size;
139         capacity = other.capacity;
140         data = new ADT * [capacity];
141
142         for (std::size_t i{ 0 }; i < size; ++i)
143             data[i] = other.data[i];
144     }
145
146     return *this;
147 }
```

A.3 Файл Test.cpp

```
1  #include "gtest/gtest.h"
2  #include "gmock/gmock.h"
3  #include "Vector.h"
4
5  TEST(vector, ConstructorsAndLength) {
6          Vector A, B(4), C(B);
7          EXPECT_EQ(A.length(), 0);
8          EXPECT_EQ(B.length(), 4);
9          EXPECT_EQ(C.length(), 4);
10 }
11
12 TEST(vector, ResizeAndPushBack) {
13         Vector A, B(4);
14         A.Resize(2); B.Resize(2);
15
16         ADT* tmp = new Matrix(3, 3);
```

```
17
18          B.push_back(tmp);
19          A.push_back(new BigNumber("1234567"));
20
21          EXPECT_EQ(A.length(), 3);
22          EXPECT_EQ(B.length(), 3);
23  }
24
25  TEST(vector, EraseInsertOperator) {
26          Vector A(2), B(4);
27
28          Matrix tmp1(3, 3);
29          BigNumber tmp2("1234567");
30          YearFromAdam tmp3;
31          DateAndTime tmp4;
32
33          B.push_back(&tmp1);
34          A.push_back(&tmp2);
35          A.insert(0, &tmp3);
36          A.erase(1);
37          B[0] = &tmp4;
38
39          YearFromAdam K(2, 3, 4);
40          B[2] = &K;
41
42          Matrix tmp5(2, 2);
43          BigNumber tmp6("12345");
44
45          A.insert(2, &tmp5);
46          A.insert(1, &tmp6);
47          A.erase(2);
48
49          EXPECT_EQ(A[0], &tmp3);
50          EXPECT_EQ(A[1], &tmp6);
51          EXPECT_EQ(A[2], &tmp5);
52          EXPECT_EQ(A[3], &tmp2);
53          EXPECT_EQ(B[4], &tmp1);
54          EXPECT_EQ(A.length(), 4);
55          EXPECT_EQ(B.length(), 5);
56          EXPECT_EQ(B[0],&tmp4);
57          EXPECT_EQ(B[2],&K);
58  }
59
60  int main(int argc, char** argv)
61  {
62          ::testing::InitGoogleTest(&argc, argv);
63          return RUN_ALL_TESTS();
64  }
```

Приложение Б

Б.1 Файл Stack.h

```
/*#include "DateAndTime.h"
#include "BigNumber.h"
#include "YearFromAdam.h"
#include "Matrix.h"*/

#ifndef STACK_H
#define STACK_H

#include "ADT.h"

class Stack
{
private:
        std::size_t size, capacity;
        ADT** data;

public:
        Stack();
        Stack(const Stack& other);
        Stack(std::size_t s);
   ~Stack();

    void Resize(std::size_t NewSize); //with losing data
        void push_back(ADT* val);
        void pop_back();
        void push_up(ADT* val);
        void pop_up();
        std::size_t length() const;

        //ADT*& operator[](std::size_t index);
        //CAN WE CHANGE ELEMENTS IN STACK?

        ADT*& operator[](std::size_t index) const;
};

#endif //STACK_H
```

Б.2 Файл Stack.cpp

```
#include "Stack.h"

Stack::Stack()
        :data{ nullptr }, size{ 0 }, capacity{ 0 }
{}

Stack::Stack(const Stack& other)
        : size{ other.size }, capacity{ other.capacity }
```

```
{
        data = new ADT * [capacity];

        for (std::size_t i{ 0 }; i < size; ++i)
                data[i] = other.data[i];
}

Stack::Stack(std::size_t s)
        : size{ s }, capacity{ 2 * s }
{
        data = new ADT * [2 * s];
}

Stack::~Stack()
{
        if (data != nullptr)
        {
                delete[] data;
                data = nullptr;
        }
}

void Stack::Resize(std::size_t NewSize)
{
        data = new ADT * [2 * NewSize];

        capacity = 2 * NewSize;
        size = NewSize;
}

void Stack::push_back(ADT* val)
{
        if (size + 1 > capacity)
        {
                Stack res(*this);

                this->Resize(size + 1);

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

        (*this)[size - 1] = val;
}

void Stack::pop_back()
```

```
59  {
60          if (size == 0) throw std::runtime_error("Stack is empty");
61
62          --size;
63  }
64
65  void Stack::push_up(ADT* val)
66  {
67          if (size + 1 > capacity)
68          {
69                  Stack temp(*this);
70                  this->Resize(size + 1);
71                  for (std::size_t i{ 1 }; i < size; ++i)
72                  {
73                          (*this)[i] = temp[i-1];
74                  }
75          }
76          else
77          {
78                  ++size;
79
80                  for (std::size_t i{ size - 1 }; i > 0; --i)
81                  {
82                          ADT* temp = (*this)[i];
83                          (*this)[i] = (*this)[i - 1];
84                          (*this)[i - 1] = temp;
85                  }
86          }
87
88          (*this)[0] = val;
89  }
90
91  void Stack::pop_up()
92  {
93          if (size == 0) throw std::runtime_error("Stack is empty");
94
95          for (std::size_t i{ 0 }; i < size - 1; ++i)
96                  (*this)[i] = (*this)[i + 1];
97
98          --size;
99  }
100
101 std::size_t Stack::length() const
102 {
103         return size;
104 }
105
106 /*ADT*& Stack::operator[](std::size_t index)
107 {
108         if (index >= size)
```

```
109                 throw std::runtime_error
110                         ("Index is outside the bounds of the vector");
111
112         return data[index];
113 }*/
114
115 ADT*& Stack::operator[](std::size_t index) const
116 {
117         if (index >= size)
118                 throw std::runtime_error
119                         ("Index is outside the bounds of the vector");
120
121         return data[index];
122 }
```

Б.3 Файл Test1.cpp

```
1  #include "gtest/gtest.h"
2  #include "Stack.h"
3
4  TEST(stack, ConstructorsLengthResize) {
5          Stack A, B(4), C(B);
6          A.Resize(10);
7          B.Resize(5);
8          C.Resize(0);
9
10         EXPECT_EQ(A.length(), 10);
11         EXPECT_EQ(B.length(), 5);
12         EXPECT_EQ(C.length(), 0);
13 }
14
15 TEST(stack, Other) {
16         Stack A;
17
18         DateAndTime tmp1;
19         BigNumber tmp2("123456789");
20         YearFromAdam tmp3(1, 2, 3);
21
22         A.push_back(&tmp1); // {tmp1}
23         A.push_back(&tmp2); // {tmp1, tmp2}
24         A.push_up(&tmp3);   // {tmp3, tmp1, tmp2}
25         A.push_up(&tmp2);   // {tmp2, tmp3, tmp1, tmp2}
26
27         Stack B(A);
28
29         B.pop_back();
30         B.pop_up();
31         B.pop_back();
32
33         EXPECT_EQ(A[0], &tmp2);
34         EXPECT_EQ(A[1], &tmp3);
```

```
35          EXPECT_EQ(A[2], &tmp1);
36          EXPECT_EQ(A[3], &tmp2);
37          EXPECT_EQ(A.length(), 4);
38          EXPECT_EQ(B.length(), 1);
39          EXPECT_EQ(B[0], &tmp3);
40  }
41
42  int main(int argc, char** argv)
43  {
44          ::testing::InitGoogleTest(&argc, argv);
45          return RUN_ALL_TESTS();
46  }
```

Приложение В

В.1 Файл List.h

```
#ifndef LIST_H
#define LIST_H

#include "ADT.h"
class List
{
private:
        struct Node
        {
                Node* next = nullptr;
                ADT* value;

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

        //std::size_t size = 0;
        Node* head;

public:
        List();
        List(const List& other) = default;
    ~List() = default;

    void erase(std::size_t index);
    void insert(std::size_t index, ADT* val);
    void push_back(ADT* val);
    void pop_back();
    void push_front(ADT* val);
    void pop_front();

    std::size_t length() const;

    Node*& operator[](std::size_t index);
    Node*& operator[](std::size_t index) const;

};

#endif //LIST_H
```

В.2 Файл List.cpp

```
#include <iostream>
#include "List.h"

List::List()
```

```
5           :head{ nullptr }
6   {}
7
8   void List::erase(std::size_t index)
9   {
10          if (index >= length())
11                  throw std::runtime_error
12                          ("index was outside the bounds of the list");
13          if (length() == 1)
14                  throw std::runtime_error
15                          ("Attempt to delete the head of the list");
16
17          if (index == length() - 1) pop_back();
18          else if (index == 0) pop_front();
19          else (*this)[index - 1]->next = (*this)[index]->next;
20  }
21
22  void List::insert(std::size_t index, ADT* val)
23  {
24          if (index >= length())
25                  throw std::runtime_error
26                          ("index was outside the bounds of the list");
27
28          if (index == 0) push_front(val);
29          else {
30                  Node* tmp = new Node(val);
31
32                  tmp->next = (*this)[index];
33                  (*this)[index-1]->next = tmp;
34          }
35  }
36
37  void List::push_back(ADT* val)
38  {
39          Node* tmp = new Node(val);
40          if (head == nullptr)
41                  head = tmp;
42          else {
43                  Node* current = head;
44
45                  while (current->next != nullptr)
46                          current = current->next;
47
48                  current->next = tmp;
49          }
50  }
51
52  void List::pop_back()
53  {
54          if (length() == 0)
```

```
55                  throw std::runtime_error
56                          ("List is empty");
57
58          if (length() == 1)
59                  throw std::runtime_error
60                          ("Attempt to delete the head of the list");
61
62          (*this)[length() - 2]->next = nullptr;
63  }
64
65  void List::push_front(ADT* val)
66  {
67          Node* tmp = new Node(val);
68          tmp->next = head;
69          head = tmp;
70  }
71
72  void List::pop_front()
73  {
74          if (length() == 0)
75                  throw std::runtime_error
76                          ("List is empty");
77
78          if (length() == 1)
79                  throw std::runtime_error
80                          ("Attempt to delete the head of the list");
81
82          head = (*this)[1];
83  }
84
85  std::size_t List::length() const
86  {
87          if (head == nullptr) return 0;
88
89          std::size_t size{ 1 };
90          Node* current = head;
91
92          while (current->next != nullptr)
93          {
94                  current = current->next;
95                  ++size;
96          }
97
98          return size;
99  }
100
101 List::Node*& List::operator[](std::size_t index)
102 {
103         if (index>=length())
104                 throw std::runtime_error
```

```
105                         ("index was outside the bounds of the list");
106
107         Node* current = head;
108
109         for (std::size_t pos{ 0 }; pos < index; ++pos)
110         {
111                 current = current->next;
112         }
113
114         return current;
115 }
116
117 List::Node*& List::operator[](std::size_t index) const
118 {
119         if (index >= length())
120                 throw std::runtime_error
121                         ("index was outside the bounds of the list");
122
123         Node* current = head;
124
125         for (std::size_t pos{ 0 }; pos < index; ++pos)
126         {
127                 current = current->next;
128         }
129
130         return current;
131 }
```

B.3 Файл Test2.cpp

```
1  #include "gtest/gtest.h"
2  #include "List.h"
3
4  TEST(list, All) {
5
6          List A;
7
8          BigNumber tmp1;
9          DateAndTime tmp2;
10         YearFromAdam tmp3;
11         Matrix tmp4;
12         BigNumber tmp5;
13         DateAndTime tmp6;
14
15         A.push_back(&tmp1); // {tmp1}
16
17         EXPECT_EQ(A[0]->value, &tmp1);
18
19         A.push_front(&tmp2); // {tmp2, tmp1}
20
21         EXPECT_EQ(A[0]->value, &tmp2);
```

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

```
72
73          A.erase(1); // {tmp2, tmp4, tmp1, tmp1, tmp6}
74
75          EXPECT_EQ(A[0]->value, &tmp2);
76          EXPECT_EQ(A[1]->value, &tmp4);
77          EXPECT_EQ(A[2]->value, &tmp1);
78          EXPECT_EQ(A[3]->value, &tmp1);
79          EXPECT_EQ(A[4]->value, &tmp6);
80
81          A.erase(1); // {tmp2, tmp1, tmp1, tmp6
82
83          EXPECT_EQ(A[0]->value, &tmp2);
84          EXPECT_EQ(A[1]->value, &tmp1);
85          EXPECT_EQ(A[2]->value, &tmp1);
86          EXPECT_EQ(A[3]->value, &tmp6);
87
88          List B(A);
89
90          EXPECT_EQ(B[0]->value, &tmp2);
91          EXPECT_EQ(B[1]->value, &tmp1);
92          EXPECT_EQ(B[2]->value, &tmp1);
93          EXPECT_EQ(B[3]->value, &tmp6);
94
95          A.pop_front(); // {tmp1, tmp1, tmp6}
96
97          EXPECT_EQ(A[0]->value, &tmp1);
98          EXPECT_EQ(A[1]->value, &tmp1);
99          EXPECT_EQ(A[2]->value, &tmp6);
100
101         A.pop_back(); // {tmp1, tmp1}
102
103         EXPECT_EQ(A[0]->value, &tmp1);
104         EXPECT_EQ(A[1]->value, &tmp1);
105
106         A.erase(0); // {tmp1}
107
108         EXPECT_EQ(A[0]->value, &tmp1);
109 }
110
111 int main(int argc, char** argv)
112 {
113         ::testing::InitGoogleTest(&argc, argv);
114         return RUN_ALL_TESTS();
115 }
```

Приложение Г

Г.1 Файл Queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

#include "ADT.h"

class Queue
{
private:
        ADT** data;
        std::size_t size, count;
        int rear, front;


public:
        Queue();
        Queue(const Queue& other);
        Queue(std::size_t size);
   ~Queue();

    void erase(std::size_t index);
    void insert(std::size_t index, ADT* val);
        void pop_front();
        void push_back(ADT* val);

        std::size_t length() const;

        ADT*& operator[](std::size_t index);
        ADT*& operator[](std::size_t index) const;
};

#endif //QUEUE_H
```

Г.2 Файл Queue.cpp

```
#include "Queue.h"
#include <algorithm>

Queue::Queue()
    :data{ nullptr }, size{ 1 }, count{ 0 }, rear{ 0 }, front{ -1 }
{}

Queue::Queue(const Queue& other)
    : size{ other.size }, count{ other.count }, rear{ other.rear },
      front{ other.front }
{
    data = new ADT * [size];

```

```
14      std::copy(other.data, other.data + size, data);
15  }
16
17  Queue::Queue(std::size_t size)
18      : size{ size }, count{ 0 }, rear{ 0 }, front{ -1 }
19  {
20      data = new ADT * [size];
21
22      std::fill(data, data + size, nullptr);
23  }
24
25  Queue::~Queue()
26  {
27      if (data != nullptr)
28      {
29          delete[] data;
30          data = nullptr;
31      }
32  }
33
34  void Queue::erase(std::size_t index)
35  {
36      if (index >= size)
37          throw std::runtime_error
38              ("Index is outside the bounds of the queue");
39
40      if (index == front) pop_front();
41      else if (index != rear)
42      {
43          int ind = index;
44          std::size_t end = (ind > rear) ? (rear + size) : rear;
45
46          for (std::size_t i{ index + 1 }; i < end; ++i)
47          {
48              ADT* temp = (*this)[i % size];
49              (*this)[i % size] = (*this)[(i - 1) % size];
50              (*this)[(i - 1) % size] = temp;
51          }
52          int a = rear - 1;
53          a = (a >= 0) ? a : (a + size);
54          std::swap((*this)[rear], (*this)[a]);
55
56          (*this)[rear] = 0;
57
58          --count;
59      }
60  }
61
62  void Queue::insert(std::size_t index, ADT* val)
63  {
```

```
64      if (index >= size)
65          throw std::runtime_error
66              ("Index is outside the bounds of the queue");
67
68      if (index == rear+1) push_back(val);
69
70      if ((*this)[index] == nullptr)
71          (*this)[index] = val;
72      else {
73          if (index != front)
74          {
75              std::size_t end = (index - rear > 0) ? rear + size : rear;
76              for (std::size_t i{ index + 1 }; i < end; ++i)
77              {
78                  ADT* temp = (*this)[i % size];
79                  (*this)[i % size] = (*this)[(i - 1) % size];
80                  (*this)[(i - 1) % size] = temp;
81              }
82              std::swap((*this)[index], (*this)[rear]);
83              (*this)[index] = val;
84          }
85      }
86  }
87
88  void Queue::pop_front()
89  {
90      if (count == 0)
91          throw std::runtime_error
92              ("Queue is empty");
93
94      (*this)[front] = nullptr;
95
96      for (std::size_t i = front; i < size + front; ++i)
97      {
98          ADT* temp = (*this)[i % size];
99          (*this)[i % size] = (*this)[(i - 1) % size];
100         (*this)[(i - 1) % size] = temp;
101     }
102
103     rear = (rear - 1) % size;
104
105     rear = (rear >= 0) ? rear : rear + size;
106
107     int a = (rear) % size;
108     int b = (front - 1) % size;
109     b = (b >= 0) ? b : b + size;
110
111     std::swap((*this)[a], (*this)[b]);
112
113     --count;
```

```
114 }
115
116 void Queue::push_back(ADT* val)
117 {
118     if ((rear + 1) % size == front)
119         front = (front + 1) % size;
120
121     if (count == 0)
122         ++front;
123
124     if (count < size)
125         ++count;
126
127     if ((*this)[rear] != nullptr)
128         rear = (rear + 1) % size;
129
130     (*this)[rear] = val;
131 }
132
133 std::size_t Queue::length() const
134 {
135         return size;
136 }
137
138 ADT*& Queue::operator[](std::size_t index)
139 {
140     if (index >= size)
141         throw std::runtime_error
142             ("Index is outside the bounds of the queue");
143
144     return data[index];
145 }
146
147 ADT*& Queue::operator[](std::size_t index) const
148 {
149     if (index >= size)
150         throw std::runtime_error
151             ("Index is outside the bounds of the queue");
152
153     return data[index];
154 }
```

Г.3 Файл Test3.cpp

```
1 #include "gtest/gtest.h"
2 #include "Queue.h"
3
4 TEST(queue, All) {
5         Queue A(4);
6
7         DateAndTime  tmp1, tmp5, tmp9;
```

```
8           BigNumber     tmp2, tmp6;
9           YearFromAdam tmp3, tmp7;
10          Matrix              tmp4, tmp8;
11
12          A.push_back(&tmp1); // {tmp1, - , - , - }
13
14          EXPECT_EQ(A[0], &tmp1);
15
16          A.insert(1, &tmp2); // {tmp1, tmp2 , - , - }
17
18          EXPECT_EQ(A[0], &tmp1);
19          EXPECT_EQ(A[1], &tmp2);
20
21          A.push_back(&tmp3); A.push_back(&tmp4); // {tmp1, tmp2, tmp3, tmp4}
22
23          EXPECT_EQ(A[0], &tmp1);
24          EXPECT_EQ(A[1], &tmp2);
25          EXPECT_EQ(A[2], &tmp3);
26          EXPECT_EQ(A[3], &tmp4);
27
28          A.push_back(&tmp5);       A.push_back(&tmp6);
29          // {tmp5, tmp6 tmp3, tmp4}
30
31          EXPECT_EQ(A[0], &tmp5);
32          EXPECT_EQ(A[1], &tmp6);
33          EXPECT_EQ(A[2], &tmp3);
34          EXPECT_EQ(A[3], &tmp4);
35
36          A.pop_front(); // {tmp6 , - , tmp4, tmp5}
37
38          EXPECT_EQ(A[0], &tmp6);
39          EXPECT_EQ(A[1], nullptr);
40          EXPECT_EQ(A[2], &tmp4);
41          EXPECT_EQ(A[3], &tmp5);
42
43          A.insert(1, &tmp7); // {tmp6 , tmp7 , tmp4, tmp5}
44
45          EXPECT_EQ(A[0], &tmp6);
46          EXPECT_EQ(A[1], &tmp7);
47          EXPECT_EQ(A[2], &tmp4);
48          EXPECT_EQ(A[3], &tmp5);
49
50          A.erase(0); // {tmp7, - , tmp4, tmp5}
51
52          EXPECT_EQ(A[0], &tmp7);
53          EXPECT_EQ(A[1], nullptr);
54          EXPECT_EQ(A[2], &tmp4);
55          EXPECT_EQ(A[3], &tmp5);
56
57          A.push_back(&tmp8);       A.push_back(&tmp9);
```

```
58          // {tmp7, tmp8, tmp9, tmp5}
59
60          EXPECT_EQ(A[0], &tmp7);
61          EXPECT_EQ(A[1], &tmp8);
62          EXPECT_EQ(A[2], &tmp9);
63          EXPECT_EQ(A[3], &tmp5);
64  }
65
66  int main(int argc, char** argv)
67  {
68          ::testing::InitGoogleTest(&argc, argv);
69          return RUN_ALL_TESTS();
70  }
```

Приложение Д

Д.1 Файл Tree.h

```
#ifndef TREE_H
#define TREE_H

#include "ADT.h"

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

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

public:
        Tree();
        Tree(const Tree& other);
        Tree(std::size_t steps);
    ~Tree() = default;

    //void Resize(std::size_t NewSteps);

    void erase(std::size_t index);
    //Deleting all children starting from index

    void insert(std::size_t index, ADT* val);

    std::size_t length() const;

    Node*& operator[](std::size_t index);
    Node*& operator[](std::size_t index) const;


};

#endif
```

Д.2 Файл Tree.cpp

```
#include <iostream>
#include "Tree.h"
#include <cmath>
```

```cpp
#include <vector>

static
std::vector<bool> Dec2Bin(std::size_t param)
{
    std::vector<bool> bits;

    while (param)
    {
        bits.push_back(param % 2);
        param >>= 1;
    }

    std::reverse(bits.begin(), bits.end());

    return bits;
}

Tree::Tree()
    :head{ nullptr }
{}

Tree::Tree(const Tree& other)
    : size{ other.size }, steps{ other.steps }
{
    head = new Node;
    head = other.head;
}
Tree::Tree(std::size_t steps)
    : steps{ steps }
{
    head = new Node;
    size = std::pow(2, steps) - 1;

    for (std::size_t i{ 2 }; i <= size / 2; ++i)
    {
        (*this)[i] = new Node;
        (*this)[i]->val = nullptr;
        /*if (i % 2)
            (*this)[i / 2]->right = (*this)[i];
        else
            (*this)[i / 2]->left = (*this)[i];*/
    }
}

void Tree::erase(std::size_t index)
{
    if (index > size || index == 0)
        throw std::runtime_error
            ("Index is outside the bounds of the vector");
```

```
54
55      (*this)[index]->val = nullptr;
56      (*this)[index]->left = nullptr;
57      (*this)[index]->right = nullptr;
58  }
59
60  void Tree::insert(std::size_t index, ADT* val)
61  {
62      if (index > size || index == 0)
63          throw std::runtime_error
64              ("Index is outside the bounds of the vector");
65
66      double step = std::log2(index);
67      std::size_t delta = steps - static_cast<int>(step) - 1;
68
69      steps += delta;
70
71      size = std::pow(2, steps) - 1;
72
73      Node* cur = new Node;
74
75      cur->left = (*this)[index];
76      cur->right = nullptr;
77      cur->val = val;
78
79      (index % 2) ? ((*this)[index / 2]->right = cur)
80          : ((*this)[index / 2]->left = cur);
81  }
82
83  std::size_t Tree::length() const
84  {
85      return size;
86  }
87
88  Tree::Node*& Tree::operator[](std::size_t index)
89  {
90      /*if (head == nullptr)
91          throw std::runtime_error
92              ("Tree is empty");*/
93
94      if (index > size || index == 0)
95          throw std::runtime_error
96              ("Index is outside the bounds of the vector");
97
98      if (index == 1) return head;
99
100     std::vector<bool> bits = Dec2Bin(index);
101     bits.erase(bits.begin());
102
103     Node* current = head;
```

```
104
105     for (bool elem : bits)
106         if (elem)
107             current = current->right;
108         else
109             current = current->left;
110
111     if (current == nullptr)
112         current = new Node;
113
114     (index % 2) ? ((*this)[index / 2]->right = current)
115         : ((*this)[index / 2]->left = current);
116
117     return current;
118 }
119
120 Tree::Node*& Tree::operator[](std::size_t index) const
121 {
122     if (head == nullptr)
123         throw std::runtime_error
124             ("Tree is empty");
125
126     if (index >= size || index == 0)
127         throw std::runtime_error
128             ("Index is outside the bounds of the vector");
129
130     std::vector<bool> bits = Dec2Bin(index);
131     bits.erase(bits.begin());
132
133     Node* current = head;
134
135     for (bool elem : bits)
136         if (elem)
137             current = current->right;
138         else
139             current = current->left;
140
141     if (current == nullptr)
142         throw std::runtime_error
143             ("Element doesn't exist");
144
145     return current;
146 }
```

Д.3 Файл Test4.cpp

```
1 #include "gtest/gtest.h"
2 #include "Tree.h"
3
4 TEST(tree, All) {
5         DateAndTime  tmp1, tmp5;
```

```
6           BigNumber          tmp2, tmp6;
7           YearFromAdam tmp3, tmp7;
8           Matrix             tmp4, tmp8;
9
10          Tree tree1(3), tree2(4);
11
12          EXPECT_EQ(tree1.length(), 7);
13          EXPECT_EQ(tree2.length(), 15);
14
15          tree1[1]->val = &tmp1;
16          tree1[2]->val = &tmp2;
17          tree1[3]->val = &tmp3;
18          tree1[4]->val = &tmp4;
19          tree1[5]->val = &tmp5;
20          tree1[6]->val = &tmp6;
21          tree1[7]->val = &tmp7;
22
23          EXPECT_EQ(tree1[1]->val, &tmp1);
24          EXPECT_EQ(tree1[2]->val, &tmp2);
25          EXPECT_EQ(tree1[3]->val, &tmp3);
26          EXPECT_EQ(tree1[4]->val, &tmp4);
27          EXPECT_EQ(tree1[5]->val, &tmp5);
28          EXPECT_EQ(tree1[6]->val, &tmp6);
29          EXPECT_EQ(tree1[7]->val, &tmp7);
30
31          tree1.insert(3, &tmp8);
32
33          EXPECT_EQ(tree1.length(), 15);
34          EXPECT_EQ(tree1[1]->val, &tmp1);
35          EXPECT_EQ(tree1[2]->val, &tmp2);
36          EXPECT_EQ(tree1[3]->val, &tmp8);
37          EXPECT_EQ(tree1[4]->val, &tmp4);
38          EXPECT_EQ(tree1[5]->val, &tmp5);
39          EXPECT_EQ(tree1[6]->val, &tmp3);
40          //EXPECT_EQ(tree1[7]->val, nullptr);
41          //EXPECT_EQ(tree1[8]->val, nullptr);
42          //EXPECT_EQ(tree1[9]->val, nullptr);
43          //EXPECT_EQ(tree1[10]->val, nullptr);
44          //EXPECT_EQ(tree1[11]->val, nullptr);
45          EXPECT_EQ(tree1[12]->val, &tmp6);
46          EXPECT_EQ(tree1[13]->val, &tmp7);
47          //EXPECT_EQ(tree1[14]->val, nullptr);
48          //EXPECT_EQ(tree1[15]->val, nullptr);
49
50          Tree tree3(tree1);
51
52          EXPECT_EQ(tree3.length(), 15);
53          EXPECT_EQ(tree3[1]->val, &tmp1);
54          EXPECT_EQ(tree3[2]->val, &tmp2);
55          EXPECT_EQ(tree3[3]->val, &tmp8);
```

```
56          EXPECT_EQ(tree3[4]->val, &tmp4);
57          EXPECT_EQ(tree3[5]->val, &tmp5);
58          EXPECT_EQ(tree3[6]->val, &tmp3);
59          //EXPECT_EQ(tree3[7]->val, nullptr);
60          //EXPECT_EQ(tree3[8]->val, nullptr);
61          //EXPECT_EQ(tree3[9]->val, nullptr);
62          //EXPECT_EQ(tree3[10]->val, nullptr);
63          //EXPECT_EQ(tree3[11]->val, nullptr);
64          EXPECT_EQ(tree3[12]->val, &tmp6);
65          EXPECT_EQ(tree3[13]->val, &tmp7);
66          //EXPECT_EQ(tree3[14]->val, nullptr);
67          //EXPECT_EQ(tree3[15]->val, nullptr);
68
69          tree3.erase(2);
70
71          EXPECT_EQ(tree3.length(), 15);
72          EXPECT_EQ(tree3[1]->val, &tmp1);
73          EXPECT_EQ(tree3[2]->val, nullptr);
74          EXPECT_EQ(tree3[3]->val, &tmp8);
75          //EXPECT_EQ(tree3[4]->val, &tmp4);
76          //EXPECT_EQ(tree3[5]->val, &tmp5);
77          EXPECT_EQ(tree3[6]->val, &tmp3);
78          //EXPECT_EQ(tree3[7]->val, nullptr);
79          //EXPECT_EQ(tree3[8]->val, nullptr);
80          //EXPECT_EQ(tree3[9]->val, nullptr);
81          //EXPECT_EQ(tree3[10]->val, nullptr);
82          //EXPECT_EQ(tree3[11]->val, nullptr);
83          EXPECT_EQ(tree3[12]->val, &tmp6);
84          EXPECT_EQ(tree3[13]->val, &tmp7);
85          //EXPECT_EQ(tree3[14]->val, nullptr);
86          //EXPECT_EQ(tree3[15]->val, nullptr);
87  }
88
89  int main(int argc, char** argv)
90  {
91          ::testing::InitGoogleTest(&argc, argv);
92          return RUN_ALL_TESTS();
93  }
```