**Правительство Российской Федерации**

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

Кафедра «Компьютерная безопасность»

# ОТЧЕТ
# К ЛАБОРАТОРНОЙ РАБОТЕ №1

**по дисциплине**

**«Языки программирования»**

Работу выполнил
студент группы СКБ-201 _____ Г.П. Кашкин
подпись, дата

Работу проверил _____ С.А. Булгаков
подпись, дата

Москва 2021

# Содержание

# Постановка задачи

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

### Общая часть

Разработать набор классов, объекты которых реализуют типы данных, указанные ниже. Для классов разработать необходимые конструкторы, деструктор, конструктор копирования, а также методы, обеспечивающие изменение отдельных составных частей объекта. Используя перегрузку операторов (operator) разработать стандартную арифметику объектов, включающую арифметические действия над объектами и стандартными типами (целыми, вещественными, строками – в зависимости от вида объектов), присваивание, ввод и вывод в стандартные потоки (используя операторы «<<» и «>>»), приведение к/от базового типа данных. Организовать операции в виде конвейера значений, с результатом (новым объектом) и сохранением значений входных операндов.

### Задачи

а) Дата и время, представленные целочисленными переменными: год, месяц, день, час, минута, секунда. Базовый тип: `uint64_t` формат представления unix time. Реализовать возможность преобразования в/из формата представления filetime (целое 64-х разрядное значение, представляющее число интервалов по 100 наносекунд, прошедших с первого января 1601 года).

б) Целое произвольной длины (во внешней форме представления в виде строки символов-цифр). Базовый тип: `std::string`.

в) Год «от Адама»,имеющий внутреннее представление в виде целочисленных переменных: ин дикт, круг солнцу, круг луне. Диапазоны значений (циклические): индикт 1—15, круг солнцу 1—28, круг луне 1—19. Ежегодно каждая переменная увеличивается на 1. Итоговое значение вычисляется как произведение переменных(диапазона на некоторый множитель; переменные независимы), а хранимое значение является остатком от деления (на диапазон), при этом 0 соответствует максимум. Необходима возможность отображения/задания как в виде одного числа, так и виде трех. Реализовать возможность преобразования в/из формата представления «от рождества Христова» используя соответствие 1652 = 7160 «от Адама».

г) Разреженная матрица, представленная динамическим массивом структур, содержащих описания ненулевых коэффициентов: индексы местоположения коэффициента в матрице (целые) и значение коэффициента (вещественное).

# 1 Алгоритм решения задачи

## 1.1 Задание 1

Для решения задачи использовались функции стандартной библиотеки и базовые арифметические операции. Для синхронизации двух форм представления использовались функции `std::mktime` и `std::gmtime`, для преобразования в человекочитаемую строчку - `std::ctime`. Для конвертации в формат представления `Filetime` использовалась константа разницы эпох microsoft и unix.

## 1.2 Задание 3

Для решения задачи было необходимо разработать двустороннюю конверцию из тройки параметров в номер года. Преобразование в тройку параметров происходит через многократное деление с остатком на границы значений. В обратную сторону - происходит перебор всех возможных годов, которых всего $15 \cdot 28 \cdot 19 = 7890$, то есть алгоритм фактически имеет константную асимптотику. Для преобразования в года от Рождества Христова необходимо вычесть константу данную в условии (5508).

## 1.3 Задание 4

Для оптимального решения задачи используется декартово дерево. Классическая реализация последнего базируется на двух операциях - `merge` и `split`, которые объединяют и делят деревья соответственно. Алгоритмы рекурсивны и базируются на поддержании инвариантов дерева, то есть для каждой ноды верно следующее: ключ у всех элементов левого поддерева строго меньше ключа ноды, а у всех элементов правового - больше (помимо этого поддерживается аналогичный инвариант по приоритетам, но я не буду их рассматривать, так как они случайны и необходимы только для лучшей средней полноты дерева). Проверка инварианта работает через отключаемый флагом макрос и делается повсюду.
Таким образом, вставка и удаление элементов так же имплементированы через `split` и `merge`. Поиск по ключу реализован как спуск от корня дерева, поэтому для него необходимы только индексы потомков. Напротив для итераторов требуется нахождение следующей и предыдущей по ключу вершины, что задает необходимость в хранении предков ноды. Хотя увеличение итератора формально и имеет асимптотику $O(\log n)$, но усредненно работает за $O(1)$, поэтому может классифицироваться как bidirectional. Для запланированной разработки `random_access_iterator` уже заложен подсчет размера поддерева для каждой ноды, при помощи него получится за $O(\log n)$ перемещаться на произвольное число нод.

# 2 Выполнение задания

## 2.1 Задание 1

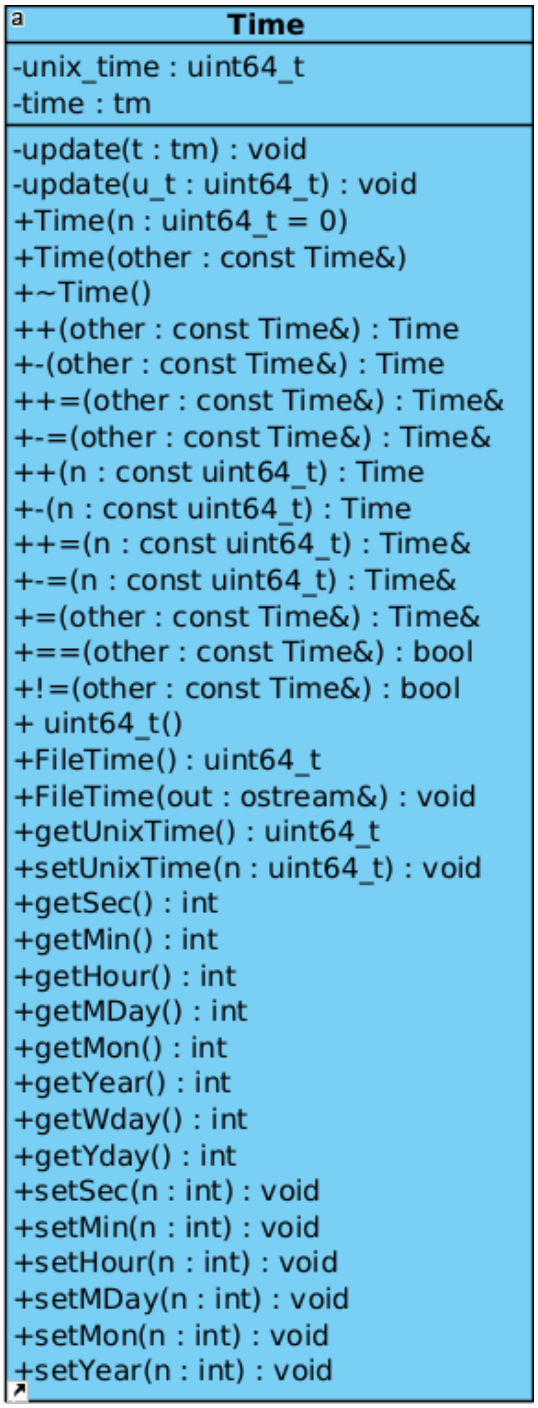


**Рис. 1. UML 2.0 diagram for Time class**

Для решения данной задачи разработан класс `Time` хранящий в себе одновременно структуру `std::tm` и переменную `uint64_t unix_time`, содержащие совокупную информацию о дате и времени в целочисленном формате и количество секунд с начала unix эры соответственно. При модификации любого из полей запускается функция `update`, которая обновляет синхронизирует второе поле. Синхронизация происходит посредством функций `std::mktime` и `std::gmtime` конвертирующими один формат значений в другой. Перегрузка `operator<<` основана на функции `std::ctime` возвращающей красивую строку с информацией о дате и времени по указателю на `unix_time`. Помимо этого в классе реализована базовая арифметика, explicit конверция в `uint64_t`, функции доступа (с префиксами get и set), конвертация в формат `Filetime`. Так как все поля тривиальны - используется деструктор по умолчанию, для конструкторов используются списки инициализации. Реализованы copy и move семантики.

## 2.2 Задание 3

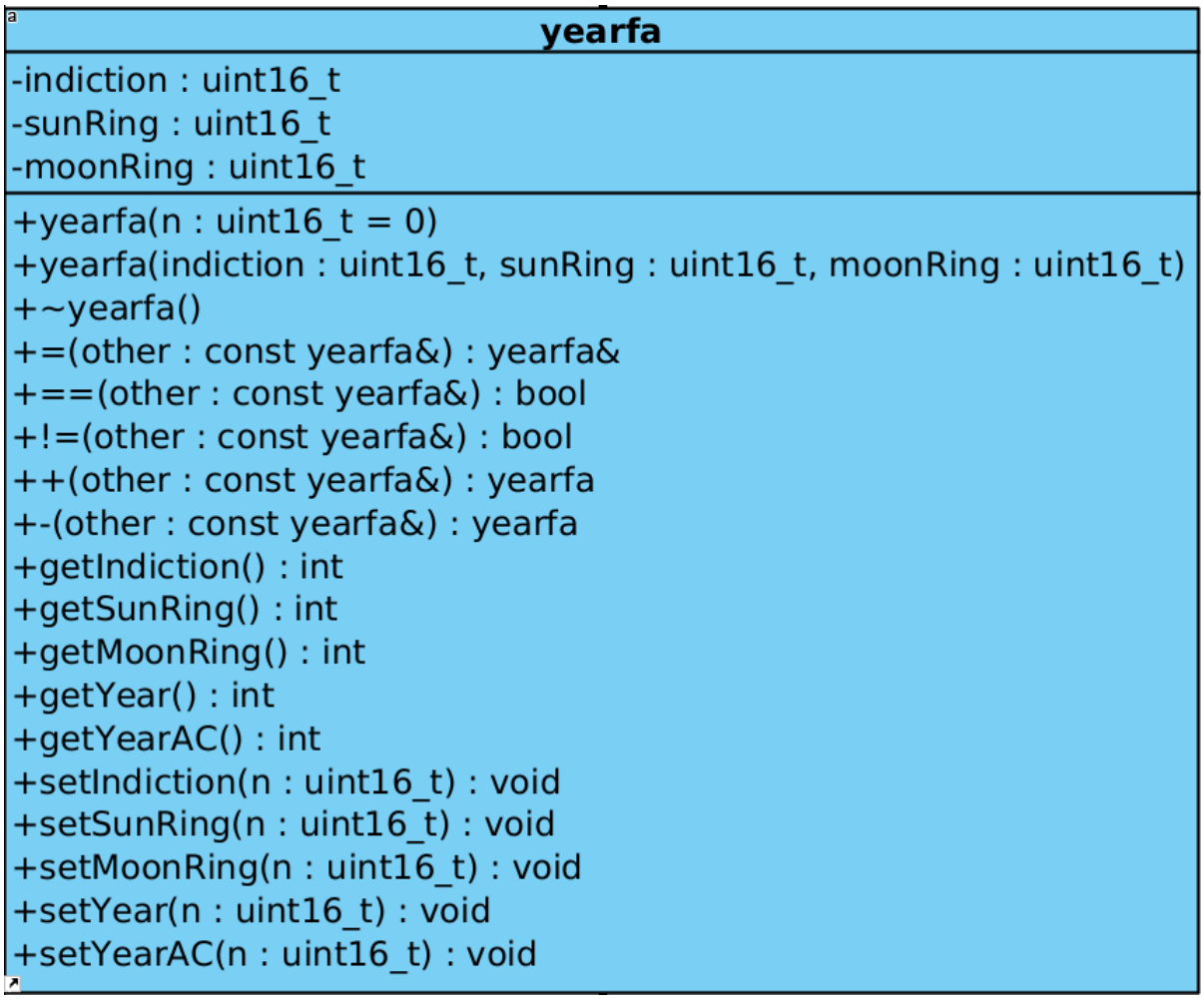


**Рис. 2. UML 2.0 diagram for yearfa class**

Для решения данной задачи разработан класс `yearFA`, описывающий календарь «от Адама». Концепция самого календаря базовая, поэтому все функции, кроме рассчитывающих год по параметрам прописаны в `inline` формате внутри `yearfa.hpp`. Последние реализованы посредством перебора, данный календарь поддерживает только 7890 лет, следовательно асимптотика алгоритма - $O(1)$. Так как все поля тривиальны - используется деструктор по умолчанию, для конструкторов используются списки инициализации, копирование так же тривиально и реализуется компилятором.

## 2.3 Задание 4

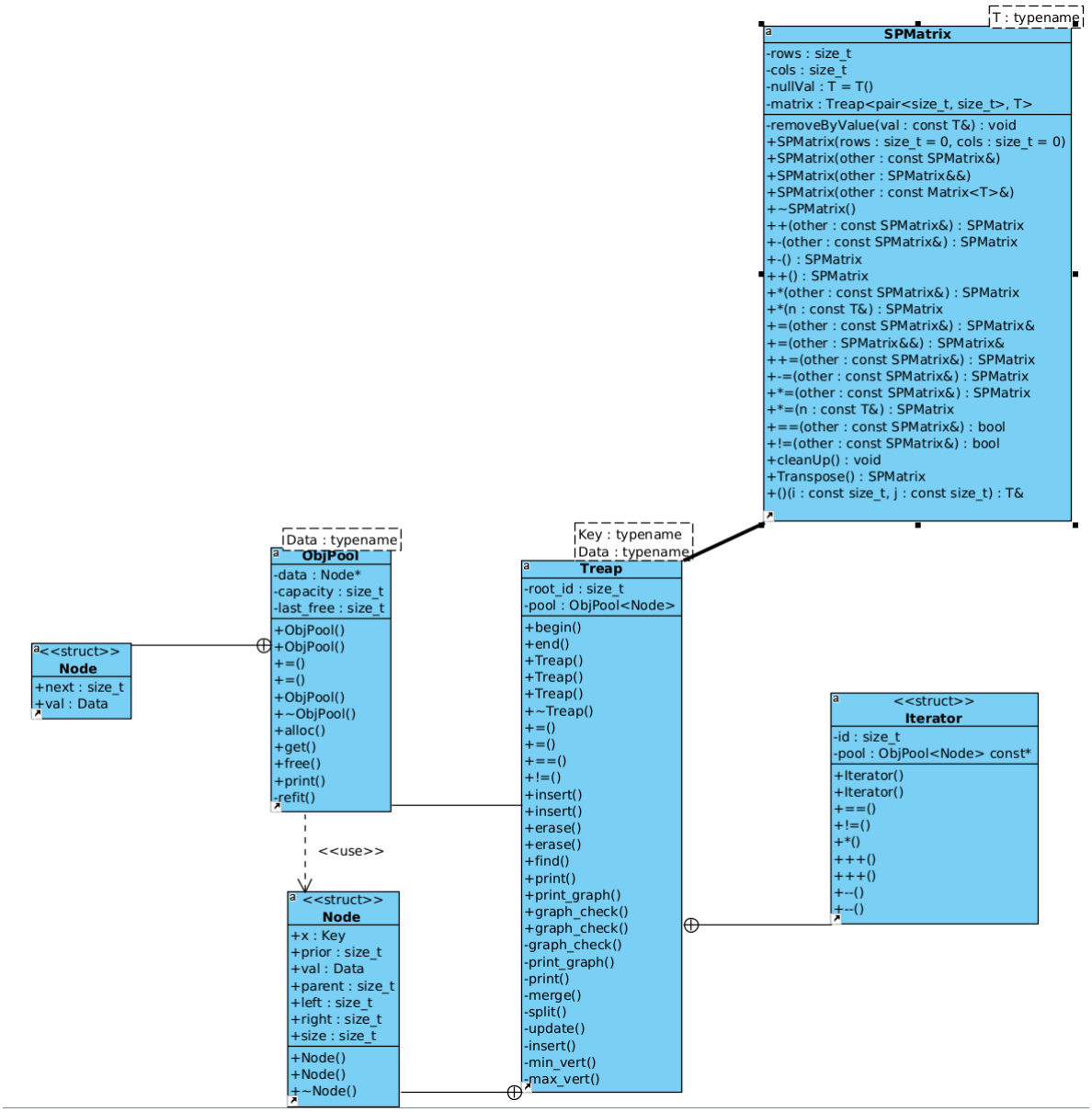


**Рис. 3. UML 2.0 diagram for spmatrix class**

Для решения данной задачи разработан класс `spmatrix`, представляющий разреженную матрицу (sparse matrix), он базируется на классе `treap` - декартовом дереве, позволяющем усредненно за $O(\log(n))$ вставку, поиск и удаление элемента, построение дерева происходит за $O(n)$. Для `treap` определена внутренняя структура ноды, хранящая необходимые сервисные значения, то есть индекс предков и детей, случайный приоритет, размер поддерева данной вершины, ключ и полезную нагрузку. Для оптимизации выделения и освобождения памяти используется класс `ObjPool` (object pool), представляющий из себя связанный список доступных для выделения нод `treap`. При аллоцировании занимается крайняя в списке нода, если в `ObjPool` заканчиваются свободные ноды, он переаллоцируется целиком на память в два раза больше. Так как при переаллоцировании ломаются все указатели на выделенные ноды, приходится использовать индексы. На декартовом дереве реализована поддержка `bidirectional_access_iterator`, функции вставки, поиска и удаления элементов по ключу. Помимо этого есть несколько сервисных функций облегчающих отладку и небольшой макрос того же свойства. Хранение размеров поддеревьев в нодах `treap` реализовано для расширения итераторов в `random_access`, но последнее

пока не готово. Разыменование итератора возвращает пару из ключа и его полезной нагрузки. `treap` как и `spmatrix` базируется на механике шаблонов, таким образом декартово дерево можно использовать в других проектах, а матрицы поддерживают произвольные типы данных полезной нагрузки. Доступ к объектам матрицы реализован через перегрузки `operator()`, для чтения может выдаваться сервисный нулевой объект, для модификации ищется или аллоцируется нода по ключу. Кроме базовых арифметических операций реализовано также умножение, работающее за $O(n \cdot m)$, где $n$ - число ненулевых элементов первой матрицы, а $m$ - число столбцов второй. Для всех классов реализована copy и move семантика.

# 3 Получение исполняемых модулей

Для всего проекта использовалась система сборки `cmake`. В конфигурации системы сборки прописаны три режима компиляции: basic, sanitizer и debug с разными уровнями придирчивости (использование посредством флага `-DCMAKE_BUILD_TYPE=*`), в каждом из них прописано использование требуемого стандарта `c++14` и компилятора `clang`. Помимо этого `cmake` автоматически скачивает с github и подключает библиотеку для unit тестов `GoogleTest`, которая используется для проверки корректности программ. Для каждого задания создан отдельный конфиг, который рекурсивно подключается в корневом `CMakeLists.txt`.

# 4 Тестирование

Тестирование производится при помощи библиотеки `GoogleTest`, везде, где позволяет логика используется многократный проход теста с генератором случайных значений.

## 4.1 Задание 1

### 4.1.1 Операторы

Операторы тестируются на повторяемость результатов операций, корректность копирования и стрессоустойчивость всех методов.

### 4.1.2 Функции доступа

Функции доступа многократно тестируются на стрессоустойчивость на случайных значениях, в том числе выходящих за рамки ожидаемых.

## 4.2 Задание 3

### 4.2.1 Операторы

Немногочисленные операторы тестируются на на повторяемость результатов операций, корректность копирования и стрессоустойчивость.

### 4.2.2 Фунции доступа

Функции доступа проверяются на стабильность и соответствию инварианта, указанного в условии.

## 4.3 Задание 4

### 4.3.1 Treap

Класс декартового дерева как и ObjPool тестировался вручную при помощи красивых функций вывода состояния и проверки встроенной инвариантов на крайних значениях. Всего три различных теста для Treap и один для ObjPool.

### 4.3.2 Простое умножение

Группа тестов проверяющих корректность операции умножения. Состоит из базового теста на корректность умножения на единичную матрицу, предподсчитаного умножения матриц небольших размеров.

### 4.3.3 Умножение случайных

Группа тестов проверяющих корректность операции умножения на случайных матрицах. Заполнение матриц происходит при помощи класса Matrix разработанного ранее и стабильного. С ним же и сравниваются результаты операций.

### 4.3.4 Сложение случайных

Как и в прошлой группе проверяется корректность сложения случайных матриц произвольного размера, результат сравнивается с заведомо корректным результатом Matrix.

### 4.3.5 Транспонирование

Проверка корректности операции транспонирования разреженной матрицы.

### 4.3.6 Копирование и перемещение

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

# Приложение А

## А.1 Файл time.hpp

```
#ifndef TIME_HPP
#define TIME_HPP

#include <ctime>
#include <iostream>

#define EPOCH_DIFF 11644482617LL


class Time{
    private:
        uint64_t unix_time;
        std::tm time;

        /*
        struct std::tm {

        int sec;   // seconds of minutes from 0 to 61
        int min;   // minutes of hour from 0 to 59
        int hour;  // hours of day from 0 to 24
        int mday;  // day of month from 1 to 31
        int mon;   // month of year from 0 to 11
        int year;  // year since 1900
        int wday;  // days since sunday         [ Ignored by updater ]
        int yday;  // days since January 1st    [ Ignored by updater ]
        int isdst; // hours of daylight savings time

        }
        */


        void update(std::tm t);
        void update(uint64_t u_t);

    public:
        explicit Time(uint64_t n = 0) { update(n); time.tm_isdst = false; }
        Time(const Time &other) : unix_time(other.unix_time), time(other.time)
    {time.tm_isdst = false; }
        ~Time() {}

        Time operator+(const Time &other) const;
        Time operator-(const Time &other) const;
        Time& operator+=(const Time &other);
        Time& operator-=(const Time &other);

        Time operator+(const uint64_t n) const;
        Time operator-(const uint64_t n) const;
        Time& operator+=(const uint64_t n);
        Time& operator-=(const uint64_t n);
```

```
49
50          Time& operator=(const Time &other);
51
52          bool operator==(const Time &other) const { return unix_time == other.
        unix_time; }
53          bool operator!=(const Time &other) const { return unix_time != other.
        unix_time; }
54
55          explicit operator uint64_t() const { return unix_time; }
56
57          uint64_t FileTime() const;
58
59          void FileTime(std::ostream &out) const;
60
61          uint64_t getUnixTime() const { return unix_time; }
62          void setUnixTime(uint64_t n) { unix_time = n; }
63
64          int getSec()  const { return time.tm_sec; }
65          int getMin()  const { return time.tm_min; }
66          int getHour() const { return time.tm_hour; }
67          int getMDay() const { return time.tm_mday; }
68          int getMon()  const { return time.tm_mon; }
69          int getYear() const { return time.tm_year + 1900; }
70          int getWday() const { return time.tm_wday; }
71          int getYday() const { return time.tm_yday; }
72
73          void setSec (int n) { time.tm_sec = n; update(time); }
74          void setMin (int n) { time.tm_min = n; update(time); }
75          void setHour(int n) { time.tm_hour = n; update(time); }
76          void setMDay(int n) { time.tm_mday = n; update(time); }
77          void setMon (int n) { time.tm_mon = n; update(time); }
78          void setYear(int n) { time.tm_year = n - 1900; update(time); }
79
80      friend Time operator+(const uint64_t n, const Time &other);
81      friend Time operator-(const uint64_t n, const Time &other);
82
83      friend std::ostream& operator<<(std::ostream &out, const Time &T);
84
85      friend std::istream& operator>>(std::istream &in, Time &T);
86 };
87
88
89 #endif
```

## A.2 Файл time.cpp

```
1 #include "time.hpp"
2
3 void Time::update(std::tm t)
4 {
5     time = t;
6     unix_time = std::mktime(&time);
7 }
8
```

```
9  void Time::update(uint64_t u_t)
10 {
11     unix_time = u_t;
12     time_t t = static_cast<time_t>(u_t);
13     time = *std::gmtime(&t);
14 }
15
16
17 Time Time::operator+(const Time &other) const
18 {
19     Time result(other);
20     result.unix_time += this->unix_time;
21     result.update(result.unix_time);
22     return result;
23 }
24
25 Time Time::operator-(const Time &other) const
26 {
27     Time result(other);
28     result.unix_time -= this->unix_time;
29     result.update(result.unix_time);
30     return result;
31 }
32
33 Time& Time::operator+=(const Time &other)
34 {
35     this->unix_time += other.unix_time;
36     update(unix_time);
37     return (*this);
38 }
39
40 Time& Time::operator-=(const Time &other)
41 {
42     this->unix_time -= other.unix_time;
43     update(unix_time);
44     return (*this);
45 }
46
47 Time Time::operator+(const uint64_t n) const {
48     Time result(unix_time + n);
49     result.update(result.unix_time);
50     return result;
51 }
52
53 Time Time::operator-(const uint64_t n) const {
54     Time result(unix_time - n);
55     result.update(result.unix_time);
56     return result;
57 }
58
59 Time& Time::operator+=(const uint64_t n) {
60     unix_time += n;
61     update(unix_time);
```

```
62      return (*this);
63  }
64
65  Time& Time::operator-=(const uint64_t n) {
66      unix_time -= n;
67      update(unix_time);
68      return (*this);
69  }
70
71  Time& Time::operator=(const Time &other) {
72      unix_time = other.unix_time;
73      time = other.time;
74      return (*this);
75  }
76
77  size_t Time::FileTime() const {
78      return unix_time - EPOCH_DIFF;
79  }
80
81  void Time::FileTime(std::ostream &out) const {
82      out << Time(unix_time - EPOCH_DIFF);
83  }
84
85  Time operator+(const uint64_t n, const Time &other){
86      Time result(other.unix_time + n);
87      result.update(result.unix_time);
88      return result;
89  }
90
91  Time operator-(const uint64_t n, const Time &other){
92      Time result(other.unix_time - n);
93      result.update(result.unix_time);
94      return result;
95  }
96
97
98  std::ostream& operator<<(std::ostream &out, const Time &T){
99      std::time_t t(T.unix_time);
100     out << std::ctime(&t);
101     return out;
102 }
103
104 std::istream& operator>>(std::istream &in, Time &T){
105     in >> T.unix_time;
106     return in;
107 }
```

## A.3 Файл test-time.cpp

```
1  #include "time.hpp"
2
3  #include <random>
4  #include <string>
5  #include <gtest/gtest.h>
```

```
6
7
8  std::mt19937 rnd(179);
9
10 TEST(Basic, operators) {
11     for (int i = 0; i < 1000; ++i){
12         Time t1(rnd()), t2(rnd());
13         t1 += t2;
14         t1 += rnd();
15         t2 += rnd();
16         t2 -= t1;
17         Time t3(t1), t4(t2), t5, t6;
18         t5 = t1;
19         t6 = t2;
20
21         EXPECT_EQ( t1, t3 );
22         EXPECT_EQ( t5, t3 );
23         EXPECT_EQ( t2, t4 );
24         EXPECT_EQ( t6, t2 );
25         int64_t n = rnd();
26         EXPECT_EQ( t1 + n, t3 + n );
27         n = rnd();
28         EXPECT_EQ( t4 - n, t6 - n );
29         EXPECT_EQ( t1 - t2 - n, t3 - t6 - n );
30         t4.FileTime();
31         std::stringstream out;
32         t4.FileTime(out);
33         int64_t k = t1.getUnixTime();
34         t2.setUnixTime(k);
35         EXPECT_EQ( t1, t2 );
36         out >> t6;
37         EXPECT_NE( t1, t6 );
38     }
39 }
40
41 TEST(Basic, GetSetTime) {
42     for (int i = 0; i < 1000; ++i){
43         Time t1(rnd()), t2(rnd());
44         t1.setSec(rnd() % 61);
45         t2.setMin(rnd() % 61);
46         t1.setHour(rnd() % 25);
47         t2.setMDay(rnd() % 32);
48         t1.setMon(rnd() % 13);
49         t2.setYear(rnd());
50   Time t3(t1);
51         t1 += t2;
52         t2 += t3;
53         EXPECT_EQ(t1, t2);
54     }
55
56 }
```

# Приложение Б

## Б.1 Файл yearfa.hpp

```
#ifndef YEARFA_HPP
#define YEARFA_HPP

#include <iostream>
#include <cassert>

static constexpr uint16_t armageddon = 15 * 28 * 19;

class yearfa{
    private:
        uint16_t indiction;      // from 1 to 15
        uint16_t sunRing;        // from 1 to 28
        uint16_t moonRing;       // from 1 to 19

    public:
        explicit yearfa(uint16_t n = 0);
        yearfa(uint16_t indiction, uint16_t sunRing, uint16_t moonRing) : indiction(indiction), sunRing(sunRing), moonRing(moonRing) {}
        //yearfa(const yearfa &other) indiction(other.in)

        ~yearfa() = default;

        //yearfa& operator=( const yearfa &other ) { indiction = other.indiction; sunRing = other.sunRing; moonRing = other.moonRing; return (*this); }
        bool operator==(const yearfa &other) const { return indiction == other.indiction && sunRing == other.sunRing && moonRing == other.moonRing; }
        bool operator!=(const yearfa &other) const { return indiction != other.indiction || sunRing != other.sunRing || moonRing != other.moonRing; }

        yearfa operator+( const yearfa &other) const { yearfa result(getYear() + other.getYear()); return result; }
        yearfa operator-( const yearfa &other) const { yearfa result(getYear() - other.getYear()); return result; }

        int getIndiction() const { return indiction; }
        int getSunRing()   const { return sunRing; }
        int getMoonRing()  const { return moonRing; }
        int getYear()   const;
        int getYearAC() const { return getYear() - 5508; }

        void setIndiction(uint16_t n) { indiction = n; }
        void setSunRing(  uint16_t n) { sunRing = n; }
        void setMoonRing( uint16_t n) { moonRing = n; }
        void setYear(     uint16_t n);
        void setYearAC(   uint16_t n) { setYear(n + 5508); }     // set by year according to Christ

```

```
42          friend std::ostream& operator<<(std::ostream &out, const yearfa &y) {
    out << y.getYear(); return out; }
43          friend std::istream& operator>>(std::istream &in, yearfa &y) { y.
    setYear(in.get()); return in; }
44
45 };
46
47
48 #endif
```

## Б.2 Файл yearfa.cpp

```
1 #include "yearfa.hpp"
2
3 yearfa::yearfa(uint16_t year){
4     indiction = year % 15;
5     if (!indiction)
6         indiction = 15;
7     sunRing = year % 28;
8     if (!sunRing)
9         sunRing = 28;
10    moonRing = year % 19;
11    if (!moonRing)
12        moonRing = 19;
13 }
14
15 void yearfa::setYear(uint16_t year){
16    indiction = year % 15;
17    if (!indiction)
18        indiction = 15;
19    sunRing = year % 28;
20    if (!sunRing)
21        sunRing = 28;
22    moonRing = year % 19;
23    if (!moonRing)
24        moonRing = 19;
25 }
26
27 int yearfa::getYear() const{
28    for (uint16_t year = 1; year < 10000; ++year){
29        if ((year % 15 == indiction || (year % 15 == 0 && indiction == 15)) &&
    \
30            (year % 28 == sunRing   || (year % 28 == 0 && sunRing == 28)) &&
      \
31            (year % 19 == moonRing  || (year % 19 == 0 && moonRing == 19)))
32                return year;
33    }
34    assert(true);
35    return 0;
36 }
```

## Б.3 Файл test-yearfa.cpp

```
1 #include "yearfa.hpp"
```

```
2
3 #include <random>
4 #include <string>
5 #include <gtest/gtest.h>
6
7
8 std::mt19937 rnd(179);
9
10 TEST(Basic, operators) {
11     for (int i = 0; i < 1000; ++i){
12         yearfa Y1(rnd() % 3000 + 1), Y2(rnd() % 3000 + 1), Y3;
13         Y3 = Y1 + Y2;
14         yearfa Y4(Y1 + Y2);
15         EXPECT_EQ(Y3, Y4);
16         std::stringstream out;
17         out << Y1 << Y2 << Y3 << Y4;
18     }
19 }
20
21 TEST(Basic, getSetTime) {
22     for (int i = 0; i < 1000; ++i){
23         yearfa Y1(rnd() % 7000 + 1), Y2(rnd() % 7000 + 1), Y3;
24         Y3 = Y2;
25         yearfa Y4(Y1);
26         EXPECT_EQ(Y3, Y2);
27         EXPECT_EQ(Y1, Y4);
28         Y1.getIndiction();
29         Y1.getSunRing();
30         Y1.getMoonRing();
31         Y1.getYear();
32         Y1.getYearAC();
33
34         Y1.setYear(7160);
35         EXPECT_EQ(Y1.getYearAC(), 1652);
36         EXPECT_EQ(Y1.getYear(), 7160);
37         Y2.setYearAC(1652);
38         EXPECT_EQ(Y2.getYear(), 7160);
39         uint16_t q = rnd() % 2000 + 10;
40         Y3.setYearAC(q);
41         EXPECT_EQ(Y3.getYearAC(), q);
42     }
43 }
```

# Приложение B

## B.1 Файл spmatrix.hpp

```
#ifndef SPMATRIX_HPP
#define SPMATRIX_HPP

#include <iostream>
#include <vector>
#include <cassert>

#include "treap.hpp"

#include "Matrix.hpp" //DEBUG

template<typename T>
class SPMatrix{                                                    //TODO fix cleanUp;
    private:

        size_t rows;
        size_t cols;
        T nullVal = T();

        Treap<std::pair<size_t, size_t>, T> matrix;

        void removeByValue(const T &val);


    public:
        SPMatrix( size_t rows = 0, size_t cols = 0 ) : rows(rows), cols(cols) {};
        SPMatrix( const SPMatrix &other ) : rows(other.rows), cols(other.cols), matrix(other.matrix) { this->cleanUp(); }
        SPMatrix( SPMatrix &&other ) : rows  (std::exchange( other.rows, 0 )), \
                                       cols  (std::exchange( other.cols, 0 )) \
                                       { matrix = std::move(other.matrix); this->cleanUp(); }

        SPMatrix( const Matrix<T> &other ) : rows(other.rows), cols(other.cols) { //DEBUG
            for (size_t r = 0; r < rows; ++r)
                for(size_t c = 0; c < cols; ++c)
                    (*this)(r, c) = other(r, c);

            this->cleanUp();
        }

        ~SPMatrix() {}

        SPMatrix operator+( const SPMatrix &other ) const; //TODO optimise it
        SPMatrix operator-( const SPMatrix &other ) const;
```

```
44          SPMatrix operator-() const { return -1 * (*this); }
45          SPMatrix operator+() const { return (*this); }
46
47          SPMatrix operator*( const SPMatrix &other ) const;
48          SPMatrix operator*( const T &n ) const;
49
50          SPMatrix& operator=( const SPMatrix & other );
51          SPMatrix& operator=( SPMatrix&& other );
52
53          SPMatrix operator+=( const SPMatrix &other );
54          SPMatrix operator-=( const SPMatrix &other );
55          SPMatrix operator*=( const SPMatrix &other );
56          SPMatrix operator*=( const T &n );
57
58          bool operator==( const SPMatrix &other ) const;
59          bool operator!=( const SPMatrix &other ) const { return !((*this) ==
        other); }
60
61          void cleanUp() {return; removeByValue(nullVal); }  //TODO fix "Treap
        error in merge at 526" (I assume problem is in erase);
62
63          SPMatrix Transpose() const;
64
65          const T& operator() ( const size_t i, const size_t j ) const;
66          T& operator() ( const size_t i, const size_t j );
67
68
69      friend std::ostream& operator<<( std::ostream &out, const SPMatrix &M){
70          for(size_t i=0; i < M.rows; ++i){
71              for(size_t j=0; j < M.cols; ++j)
72                  out << M(i, j) << ' ';
73              out << '\n';
74          }
75          out << "\n";
76          return out;
77      }
78
79      friend SPMatrix operator*( const T &n, const SPMatrix &M){
80          SPMatrix result(M);
81          result *= n;
82          return result;
83      }
84
85
86
87  };
88
89  template<typename T>
90  SPMatrix<T> SPMatrix<T>::Transpose() const {
91      SPMatrix result(cols, rows);
92      for (auto elem : matrix) {
93          size_t x = elem.first.first, y = elem.first.second;
94          result(y, x) = elem.second;
```

```
95      }
96      return result;
97  }
98
99  template<typename T>
100 void SPMatrix<T>::removeByValue(const T &val) {
101     std::vector<std::pair<size_t, size_t>> toBeRemoved;
102     for (auto elem : matrix)
103         if (elem.second == val)
104             toBeRemoved.push_back(elem.first);
105
106     for (auto elem : toBeRemoved)
107         matrix.erase(elem);
108 }
109
110 template<typename T>
111 SPMatrix<T>& SPMatrix<T>::operator=(const SPMatrix<T> &other) {
112     rows = other.rows;
113     cols = other.cols;
114     matrix = other.matrix;
115     this->cleanUp();
116     return (*this);
117 }
118
119 template<typename T>
120 SPMatrix<T>& SPMatrix<T>::operator=(SPMatrix<T> &&other) {
121     rows = std::exchange(other.rows, 0);
122     cols = std::exchange(other.cols, 0);
123     matrix = std::move(other.matrix);
124     this->cleanUp();
125     return (*this);
126 }
127
128 template<typename T>
129 const T& SPMatrix<T>::operator() (const size_t i, const size_t j) const{
130     assert(i >= 0 && i < rows && j >= 0 && j < cols);
131     T* result = matrix.find(std::make_pair(i, j));
132     if (result)
133         return *result;
134     return nullVal;
135 }
136
137 template<typename T>
138 T& SPMatrix<T>::operator() (const size_t i, const size_t j) {
139     assert(i >= 0 && i < rows && j >= 0 && j < cols);
140     T* node = matrix.find(std::make_pair(i, j));
141     if (node)
142         return *node;
143     node = matrix.insert({i, j});
144
145     return *node;
146 }
147
```

```
148 template<typename T>
149 SPMatrix<T> SPMatrix<T>::operator+(const SPMatrix<T> &other) const {
150     assert(rows == other.rows && cols == other.cols);
151     SPMatrix result(other);
152     for (auto elem : matrix)
153         result(elem.first.first, elem.first.second) += elem.second;
154 
155     return result;
156 }
157 
158 template<typename T>
159 SPMatrix<T> SPMatrix<T>::operator-(const SPMatrix<T> &other) const {
160     return (*this) + -other;
161 }
162 
163 template<typename T>
164 SPMatrix<T> SPMatrix<T>::operator*(const SPMatrix<T> &other) const {
165     assert(cols == other.rows);
166 
167     SPMatrix<T> result(rows, other.cols);
168     for (const auto pair : matrix) {
169         T elem = pair.second;
170         size_t x = pair.first.first, y = pair.first.second;
171         for (size_t r = 0; r < other.cols; ++r){
172             result(x, r) += elem * other(y, r);
173         }
174     }
175     return result;
176 }
177 
178 template<typename T>
179 SPMatrix<T> SPMatrix<T>::operator*(const T &n) const {
180     SPMatrix result(*this);
181     for (auto elem : result.matrix)
182         elem.second *= n;
183     return result;
184 }
185 
186 template<typename T>
187 SPMatrix<T> SPMatrix<T>::operator+=(const SPMatrix<T> &other) {
188     assert(rows == other.rows && cols == other.cols);
189     for (auto elem : other.matrix)
190         (*this)(elem.first.first, elem.first.second) += elem.second;
191     this->cleanUp();
192     return (*this);
193 }
194 
195 
196 template<typename T>
197 SPMatrix<T> SPMatrix<T>::operator-=(const SPMatrix<T> &other) {
198     (*this) += -other;
199     return (*this);
200 }
```

```
201
202 template<typename T>
203 SPMatrix<T> SPMatrix<T>::operator*=(const T &n) {
204     for (auto elem : matrix)
205         elem.second *= n;
206     this->cleanUp();
207     return *this;
208 }
209
210 template<typename T>
211 SPMatrix<T> SPMatrix<T>::operator*=(const SPMatrix<T> &other) {
212     (*this) = (*this) * other;
213     this->cleanUp();
214     return (*this);
215 }
216
217 template<typename T>
218 bool SPMatrix<T>::operator==(const SPMatrix &other) const {
219     if (rows != other.rows || cols != other.cols)
220         return false;
221     T *val;
222     for (auto elem : matrix)
223         if (elem.second != nullVal && ((other(elem.first.first, elem.first.
    second) != elem.second)))
224             return false;
225
226     for (auto elem : other.matrix)
227         if (elem.second != nullVal && (((*this)(elem.first.first, elem.first.
    second) != elem.second)))
228             return false;
229     return true;
230 }
231
232 #endif
```

## B.2 Файл treap.hpp

```
1 #ifndef TREAP_HPP
2 #define TREAP_HPP
3
4 #include <cstdint>
5 #include <cstddef>
6 #include <random>
7 #include <vector>
8 #include <iostream>
9 #include <cassert>
10 #include <set>
11
12 std::mt19937 rnd(179);
13
14 template<class T, class U = T>
15 T exchange(T& obj, U&& new_value)
16 {
17     T old_value = std::move(obj);
```

```
18      obj = std::forward<U>(new_value);
19      return old_value;
20  }
21
22  //==================================
23  // Object pool
24
25  template<typename Data>
26  class ObjPool{
27  public:
28
29      ObjPool(const ObjPool &other)
30      {
31          capacity = other.capacity;
32          last_free = other.last_free;
33          data = new Node[capacity];
34          std::copy(other.data, other.data + other.capacity, data);
35      }
36
37      ObjPool(ObjPool &&other)
38      {
39          capacity = exchange(other.capacity, 0);
40          last_free = exchange(other.last_free, -1);
41          data = exchange(other.data, nullptr);
42      }
43
44      ObjPool& operator=(const ObjPool &other)
45      {
46          capacity = other.capacity;
47          last_free = other.last_free;
48          if (data)
49              delete[] data;
50          data = new Node[capacity];
51          std::copy(other.data, other.data + other.capacity, data);
52          return (*this);
53      }
54
55      ObjPool& operator=(ObjPool &&other)
56      {
57          capacity = exchange(other.capacity, 0);
58          last_free = exchange(other.last_free, -1);
59          if (data)
60              delete[] data;
61          data = exchange(other.data, nullptr);
62          return (*this);
63      }
64
65
66      ObjPool(size_t capacity=1) : capacity(capacity)
67      {
68          data = new Node [capacity];
69          for (size_t i=0; i<capacity - 1; ++i)
70              data[i].next = i + 1;
```

```
71          data[capacity - 1].next = -1;
72
73          last_free = 0;
74      }
75      ~ObjPool()
76      {
77          delete [] data;
78      }
79
80      size_t alloc()
81      {
82          refit();
83          size_t result = last_free;
84          last_free = data[last_free].next;
85          return result;
86      }
87
88      Data *get(size_t id) const
89      {
90          // std::cerr << (id == -1) << ' ' << id << '\n'; //DEBUG
91          assert(id != -1);
92          assert(id < capacity);
93          return &data[id].val;
94      }
95
96      void free(size_t id)
97      {
98          data[id].next = last_free;
99          last_free = id;
100     }
101
102     void print(std::ostream& out)
103     {
104         for (size_t id = last_free; id != -1; id = data[id].next)
105         {
106             out << "(" << id << ") -> ";
107         }
108         out << '\n';
109     }
110
111
112 private:
113     struct Node{
114         size_t next;
115         Data val;
116         // ~Node() {};
117     };
118
119     Node *data;
120     size_t capacity;
121     size_t last_free;
122
123     void refit()
```

```
124     {
125         if (last_free != -1)
126             return;
127         Node *nbuf = new Node[capacity * 2];
128         std::copy(data, data + capacity, nbuf);
129         assert(data);
130         delete [] data;
131         data = nbuf;
132         capacity *= 2;
133         for (size_t i = capacity / 2; i < capacity - 1; ++i)
134             data[i].next = i + 1;
135         data[capacity - 1].next = -1;
136         last_free = capacity / 2;
137     }
138 };
139
140 //===================================
141 // Treap
142
143 #ifndef NDEBUG
144 #define TREAP_CHECK(v) { \
145     if (!graph_check(v)) \
146     { \
147         std::cerr << "Treap error in " << __func__ << " at " << __LINE__ << "\
    n"; \
148         std::cerr.flush(); \
149         std::exit(0); \
150     } }
151 #else
152 #define TREAP_CHECK(v) {}
153 #endif
154
155 template<typename Key, typename Data>
156 class Treap
157 {
158 private:
159      struct Node
160     {
161         Key x;
162         size_t prior;
163         Data val = Data();
164         size_t parent;
165         size_t left, right;
166         size_t size;
167
168         Node()                : prior(rnd()), parent(-1), left(-1), right(-1),
     size(1) {}
169         Node(Key x, Data val) : x(x), prior(rnd()), val(val), parent(-1), left
    (-1), right(-1), size(1) {}
170
171         ~Node() {};
172     };
173
```

```
174     size_t root_id;
175     ObjPool<Node> pool;
176 
177 public:
178     struct Iterator //TODO add random_access_iterator support
179     {
180         using iterator_category = std::bidirectional_iterator_tag;
181         using difference_type   = std::ptrdiff_t;
182         using value_type        = Node;
183 
184         Iterator( size_t id=-1, const ObjPool<Node>* pool=nullptr ) : id(id), pool(pool) {}
185         Iterator( const Iterator &other )                           : id(other.id), pool(other.pool) {};
186 
187         bool operator==( const Iterator other ) { return id == other.id;}
188         bool operator!=( const Iterator other ) { return id != other.id;}
189 
190         std::pair<Key, Data&> operator*() { Node* v = pool->get(id); \
191                                             return {v->x, v->val}; }
192 
193         const std::pair<const Key, const Data&> operator*() const { Node* v = pool->get(id); \
194                                             return std::make_pair(v->x, v->val); }
195 
196 
197         Iterator operator++()
198         {
199             assert(id != -1);
200             Node* v = pool->get(id);
201             if (v->right != -1)
202             {
203                 id = v->right;
204                 while ((v = pool->get(id))->left != -1)
205                     id = v->left;
206                 return (*this);
207             }
208             while ((v = pool->get(id))->parent != -1)
209             {
210                 if (pool->get(v->parent)->left == id)
211                 {
212                     id = v->parent;
213                     return (*this);
214                 }
215                 id = v->parent;
216             }
217             id = -1;
218             return (*this);
219         }
220 
221         Iterator operator++(int)
222         {
```

```
223             assert(id != -1);
224             Node* v = pool->get(id);
225             Iterator result(*this);
226             if (v->right != -1)
227             {
228                 id = v->right;
229                 while ((v = pool->get(id))->left != -1)
230                     id = v->left;
231                 return result;
232             }
233             while ((v = pool->get(id))->parent != -1)
234             {
235                 if (pool->get(v->parent)->left == id)
236                 {
237                     id = v->parent;
238                     return result;
239                 }
240                 id = v->parent;
241             }
242             id = -1;
243             return result;
244         }
245 
246         Iterator operator--()
247         {
248             assert(id != -1);
249             Node* v = pool->get(id);
250             if (v->left != -1)
251             {
252                 id = v->left;
253                 while ((v = pool->get(id))->right != -1)
254                     id = v->right;
255                 return (*this);
256             }
257             while ((v = pool->get(id))->parent != -1)
258             {
259                 if (pool->get(v->parent)->right == id)
260                 {
261                     id = v->parent;
262                     return (*this);
263                 }
264                 id = v->parent;
265             }
266             id = -1;
267             return (*this);
268         }
269 
270         Iterator operator--(int)
271         {
272             assert(id != -1);
273             Node* v = pool->get(id);
274             Iterator result(*this);
275             if (v->left != -1)
```

```
276             {
277                 id = v->left;
278                 while ((v = pool->get(id))->right != -1)
279                     id = v->right;
280                 return result;
281             }
282             while ((v = pool->get(id))->parent != -1)
283             {
284                 if (pool->get(v->parent)->right == id)
285                 {
286                     id = v->parent;
287                     return result;
288                 }
289                 id = v->parent;
290             }
291             id = -1;
292             return result;
293         }
294
295
296
297         private:
298             size_t id;
299             const ObjPool<Node>* pool;
300     };
301
302     Iterator begin() const { return Iterator(min_vert(root_id), &pool); }
303     Iterator end()   const { return Iterator(); }
304
305
306     //=====================================
307     // TREAP internal functions
308
309     Treap() : root_id(-1) {}
310     Treap(const Treap &other) : root_id(other.root_id), pool(other.pool) {}
311     Treap(Treap &&other);
312     ~Treap() {}
313
314
315     Treap& operator=( const Treap &other );
316     Treap& operator=( Treap &&other );
317
318     bool  operator==( const Treap &other ) const;
319     bool  operator!=( const Treap &other ) const { return !((*this) == other);
        }
320
321     void   insert( Key x, Data val );
322     Data*  insert( Key x );
323
324     void   erase ( Key x ) { if (root_id != -1)  root_id = erase(root_id, x);
      }
325     size_t erase ( size_t id, Key x );
326
```

```
327     Data* find( Key x ) const;
328 
329     void print      ( std::ostream &out ) { print(out, root_id); out << '\n';
    }
330     void print_graph( std::ostream &out )
331     {
332         static size_t dumpn = 0;
333         out << "digraph tree" << dumpn++ <<  "{\n"
334                "    node [shape=record];\n";
335         if (root_id != -1)
336             print_graph(out, root_id);
337         out << "};\n";
338     }
339 
340     bool graph_check()
341     {
342         if (root_id != -1 && pool.get(root_id)->parent != -1)
343             return false;
344         return graph_check(root_id);
345     }
346 
347     bool graph_check(size_t id)
348     {
349         std::set<size_t> S;
350         return graph_check(id, S);
351     }
352 
353 private:
354 
355     bool graph_check( size_t id, std::set<size_t> &S );
356     void print_graph( std::ostream &out, size_t id );
357     void print( std::ostream &out, size_t id );
358 
359     size_t                    merge( size_t tl_id, size_t tr_id );
360     std::pair<size_t, size_t> split( size_t t_id, Key k );
361 
362     void update( size_t id );
363     void insert( Node &node);                          //TODO write it to
    emplement faster 0 nodes removal
364 
365     size_t min_vert( size_t v_id ) const;
366     size_t max_vert( size_t v_id ) const;
367 };
368 
369 
370 template<typename Key, typename Data>
371 Treap<Key, Data>::Treap(Treap &&other)
372 {
373     root_id = exchange(other.root_id, -1);
374     pool = std::move(other.pool);
375 }
376 
377 
```

```
378 template<typename Key, typename Data>
379 Treap<Key, Data>& Treap<Key, Data>::operator=(const Treap<Key, Data> &other)
380 {
381     root_id = other.root_id;
382     pool = other.pool;
383     return (*this);
384 }
385
386 template<typename Key, typename Data>
387 Treap<Key, Data>& Treap<Key, Data>::operator=(Treap<Key, Data> &&other)
388 {
389     root_id = exchange(other.root_id, -1);
390     pool = std::move(other.pool);
391     return (*this);
392 }
393
394
395 template<typename Key, typename Data>
396 bool Treap<Key, Data>::operator==(const Treap<Key, Data> &other) const {
397     if (root_id != other.root_id)
398         return false;
399
400     auto iter_this = this->begin(), iter_other = other.begin();
401     while (iter_this != this->end() && iter_other != other.end()){
402         if (*iter_this != *iter_other)
403             return false;
404         ++iter_this;
405         ++iter_other;
406     }
407     if (iter_this != this->end() || iter_other != other.end())
408         return false;
409     return true;
410 }
411
412 template<typename Key, typename Data>
413 void Treap<Key, Data>::insert(Key x, Data val)
414 {
415     Data* q = find(x);
416     if (q)
417     {
418         *q = val;
419         return;
420     }
421
422     auto [tl_id, tr_id] = split(root_id, x);
423     size_t tm_id = pool.alloc();
424     assert(tm_id != -1);
425     Node *v = pool.get(tm_id);
426     v->val = val;
427     v->x = x;
428     root_id = merge(merge(tl_id, tm_id), tr_id);
429     TREAP_CHECK(root_id);
430 }
```

```
431
432 template<typename Key, typename Data>
433 Data* Treap<Key, Data>::insert(Key x)
434 {
435     Data* q = find(x);
436     if (q)
437         return q;
438
439     auto [tl_id, tr_id] = split(root_id, x);
440     size_t tm_id = pool.alloc();
441     assert(tm_id != -1);
442     Node *v = pool.get(tm_id);
443     v->x = x;
444     root_id = merge(merge(tl_id, tm_id), tr_id);
445     TREAP_CHECK(root_id);
446     return &(v->val);
447 }
448
449
450 template<typename Key, typename Data>
451 size_t Treap<Key, Data>::erase(size_t id, Key x) //TODO find error
452 {
453     if (id == -1)
454         return -1;
455     Node *v = pool.get(id);
456     if (v->x == x)
457     {
458         size_t tl_id = v->left;
459         size_t tr_id = v->right;
460
461         v->parent = -1;
462         if (tr_id != -1)
463             pool.get(tr_id)->parent = -1;
464         if (tl_id != -1)
465             pool.get(tl_id)->parent = -1;
466
467         pool.free(id);
468         return merge(tl_id, tr_id);
469     }
470
471     if (v->x < x)
472         v->right = erase(v->right, x);
473     else
474         v->left = erase(v->left, x);
475
476     update(id);
477     TREAP_CHECK(root_id);
478     return id;
479 }
480
481 template<typename Key, typename Data>
482 Data* Treap<Key, Data>::find(Key x) const
483 {
```

```
484     size_t cur_id = root_id;
485     Node *v;
486     while (cur_id != -1 && ((v = pool.get(cur_id))->x != x))
487     {
488         if (v->x > x)
489             cur_id = v->left;
490         else
491             cur_id = v->right;
492     }
493     if (cur_id != -1)
494         return &v->val;
495     return nullptr;
496 }
497
498 template<typename Key, typename Data>
499 bool Treap<Key, Data>::graph_check(size_t id, std::set<size_t> &S)
500 {
501     if (id == -1)
502         return true;
503     if (S.find(id) != S.end())
504         return false;
505     S.insert(id);
506     Node *v = pool.get(id);
507     if (v->right != -1)
508         if (!graph_check(v->right, S) || pool.get(v->right)->parent != id)
509             return false;
510     if (v->left != -1)
511         if (!graph_check(v->left, S) || pool.get(v->left)->parent != id)
512             return false;
513     return true;
514 }
515
516 template<typename Key, typename Data>
517 void Treap<Key, Data>::print_graph(std::ostream &out, size_t id)
518 {
519     assert(id != -1);
520     Node *v = pool.get(id);
521     out << "struct" << id << " [label=\"" << id << " | { key = " << v->x << "
        | data = " << v->val
522     << " }\"];\n";
523     if (v->left != -1)
524     {
525         out << "struct" << id << " -> " << "struct" << v->left << ";\n";
526         print_graph(out, v->left);
527     }
528     if (v->right != -1)
529     {
530         out << "struct" << id << " -> " << "struct" << v->right << ";\n";
531         print_graph(out, v->right);
532     }
533 }
534
535
```

```
536 template<typename Key, typename Data>
537 size_t Treap<Key, Data>::merge(size_t tl_id, size_t tr_id)
538 {
539     TREAP_CHECK(tl_id);
540     TREAP_CHECK(tr_id);
541     if (tl_id == -1)
542         return tr_id;
543     if (tr_id == -1)
544         return tl_id;
545     Node *tl = pool.get(tl_id);
546     Node *tr = pool.get(tr_id);
547     if (tl->prior < tr->prior)
548     {
549         tl->right = merge(tl->right, tr_id);
550         update(tl_id);
551         TREAP_CHECK(tl_id);
552         return tl_id;
553     }
554     else
555     {
556         tr->left = merge(tl_id, tr->left);
557         update(tr_id);
558         TREAP_CHECK(tr_id);
559         return tr_id;
560     }
561 }
562
563 template<typename Key, typename Data>
564 std::pair<size_t, size_t> Treap<Key, Data>::split(size_t t_id, Key k)
565 {
566     if (t_id == -1)
567         return {-1, -1};
568     Node *t = pool.get(t_id);
569
570     if (t->x <= k)
571     {
572         auto [tl_id, tr_id] = split(t->right, k);
573         t->right = tl_id;
574         update(t_id);
575         if (tr_id != -1)
576             pool.get(tr_id)->parent = -1;
577         t->parent = -1;
578         TREAP_CHECK(tr_id);
579         TREAP_CHECK(t_id);
580         return {t_id, tr_id};
581     }
582     else
583     {
584         auto [tl_id, tr_id] = split(t->left, k);
585         t->left = tr_id;
586         update(t_id);
587         if (tl_id != -1)
588             pool.get(tl_id)->parent = -1;
```

```
589         t->parent = -1;
590         TREAP_CHECK(tl_id);
591         TREAP_CHECK(t_id);
592         return {tl_id, t_id};
593     }
594 }
595
596 template<typename Key, typename Data>
597 void Treap<Key, Data>::update(size_t id)
598 {
599     assert(id != -1);
600
601     Node* v = pool.get(id);
602     v->size = 1;
603     if (v->left != -1)
604     {
605         Node* tl = pool.get(v->left);
606         v->size += tl->size;
607         tl->parent = id;
608     }
609     if (v->right != -1)
610     {
611         Node* tr = pool.get(v->right);
612         v->size += tr->size;
613         tr->parent = id;
614     }
615 }
616
617 template<typename Key, typename Data>
618 size_t Treap<Key, Data>::min_vert(size_t v_id) const
619 {
620     if (v_id == -1)
621         return -1;
622     Node *v;
623     while ((v = pool.get(v_id))->left != -1)
624         v_id = v->left;
625     return v_id;
626 }
627
628 template<typename Key, typename Data>
629 void Treap<Key, Data>::print(std::ostream &out, size_t id)
630 {
631     TREAP_CHECK(id);
632     if (id == -1) return;
633     Node *v = pool.get(id);
634     print(out, v->left);
635
636     out << '(' << v->x << ", " << v->val << ", " << v->size << ") ";
637
638     print(out, v->right);
639 }
640
641 template<typename Key, typename Data>
```

```
642 size_t Treap<Key, Data>::max_vert(size_t v_id) const
643 {
644     if (v_id == -1)
645         return -1;
646     Node *v;
647     while ((v = pool.get(v_id))->right != -1)
648         v_id = v->right;
649     return v_id;
650 }
651
652 #endif
```

## B.3 Файл test-spmatrix.cpp

```
1  #include "spmatrix.hpp"
2  #include "Matrix.hpp"
3
4  #include <gtest/gtest.h>
5
6  constexpr size_t n = 2;
7  constexpr size_t k = 17;
8
9  TEST(Multiplication, toEandConst) {
10     Matrix<int> M1(n, n), M2(n, n);
11     M1(0, 0) = 1;
12     M1(0, 1) = 2;
13     M1(1, 0) = 3;
14     M1(1, 1) = 4;
15     M2(0, 0) = 1;
16     M2(0, 1) = 0;
17     M2(1, 0) = 0;
18     M2(1, 1) = 1;
19     SPMatrix<int> S1(M1), S2(M2);
20
21     EXPECT_EQ( SPMatrix<int>(M1 * M2), S1 * S2 );
22     EXPECT_EQ( SPMatrix<int>(M2 * M1), S2 * S1 );
23     EXPECT_EQ( S1 * S1, S1 * S1);
24     EXPECT_EQ( S2 * S2, S2 * S2);
25
26     EXPECT_EQ( S1 * S2 * n, n * S1 * S2 );
27     EXPECT_EQ( S2 * n * S1, S1 * n * S2 );
28 }
29
30 TEST(Multiplication, diffSizeCheckByValue) {
31     Matrix<int> M3(3, 2), M4(2, 3);
32     M3(0, 0) = 1;
33     M3(0, 1) = 2;
34     M3(1, 0) = 3;
35     M3(1, 1) = 4;
36     M3(2, 0) = 5;
37     M3(2, 1) = 6;
38
39     M4(0, 0) = 7;
40     M4(0, 1) = 8;
```

```
41      M4(0, 2) = 9;
42      M4(1, 0) = 10;
43      M4(1, 1) = 11;
44      M4(1, 2) = 12;
45
46      SPMatrix<int> S3(M3), S4(M4);
47
48      EXPECT_EQ( SPMatrix<int>(M3 * M4), S3 * S4 );
49      EXPECT_EQ( S3 * n * S4, n * S3 * S4);
50  }
51
52  TEST(Multiplication, RandomSquares) {
53      for(int i = 0; i < 10; ++i){
54          int a = rnd() % 20 + 4;
55          Matrix<int> M1(a, a), M2(a, a);
56          M1.FillMatrixRandom();
57          M2.FillMatrixRandom();
58
59          SPMatrix<int> S1(M1), S2(M2);
60          EXPECT_EQ( SPMatrix<int>(M1 * M2), S1 * S2 );
61          EXPECT_EQ( SPMatrix<int>(M2 * M1), S2 * S1 );
62          EXPECT_EQ( SPMatrix<int>(M1 * M1), S1 * S1 );
63          EXPECT_EQ( SPMatrix<int>(M2 * M2), S2 * S2 );
64          EXPECT_NE( SPMatrix<int>(M1 * M2 * M1), S1 * S2 );
65      }
66  }
67
68  TEST(Multiplication, RandomDiffSize) {
69      for(int i = 0; i < 10; ++i){
70          int a = rnd() % 20 + 3, b = rnd() % 20 + 3, c = rnd() % 20 + 3;
71          Matrix<int> M1(a, b), M2(b, c);
72          M1.FillMatrixRandom();
73          M2.FillMatrixRandom();
74
75          SPMatrix<int> S1(M1), S2(M2);
76          EXPECT_EQ( SPMatrix<int>(M1 * M2), S1 * S2 );
77      }
78  }
79
80
81  TEST(Summing, RandomDiffSize) {
82      for(int i = 0; i < 20; ++i){
83          int a = rnd() % 50 + 3, b = rnd() % 50 + 3;
84          Matrix<int> M1(a, b), M2(a, b);
85          M1.FillMatrixRandom();
86          M2.FillMatrixRandom();
87
88          SPMatrix<int> S1(M1), S2(M2);
89          EXPECT_EQ( SPMatrix<int>(M1 + M2), S1 + S2 );
90          EXPECT_EQ( SPMatrix<int>(M2 + M1), S2 + S1 );
91          EXPECT_EQ( SPMatrix<int>(M1 + M1), S1 + S1 );
92          EXPECT_EQ( SPMatrix<int>(M2 + M2), S2 + S2 );
93          EXPECT_NE( SPMatrix<int>(M1 + M1 + M1), S1 + S2 );
```

```
94      }
95  }
96
97  TEST(Summing, RandomDiffSizeSubraction) {
98      for(int i = 0; i < 40; ++i){
99          Matrix<int> M1(4, 12), M2(4, 12);
100         M1.FillMatrixRandom();
101         M2.FillMatrixRandom();
102
103         SPMatrix<int> S1(M1), S2(M2);
104         EXPECT_EQ( SPMatrix<int>(M1 - M2), S1 - S2 );
105         EXPECT_EQ( SPMatrix<int>(M2 - M1), S2 - S1 );
106         EXPECT_EQ( SPMatrix<int>(M1 - M1), S1 - S1 );
107         EXPECT_EQ( SPMatrix<int>(M2 - M2), S2 - S2 );
108         EXPECT_NE( SPMatrix<int>(M1 - M1 + M1), S1 - S2 );
109     }
110 }
111
112 TEST(Summing, RandomMultiplByConst) {
113     size_t cnst = 19;
114     for(int i = 0; i < 40; ++i){
115         Matrix<int> M1(4, 12), M2(4, 12);
116         M1.FillMatrixRandom();
117         M2.FillMatrixRandom();
118
119         SPMatrix<int> S1(M1), S2(M2);
120         EXPECT_EQ( SPMatrix<int>(M1 + M2 * cnst), S1 + S2 * cnst );
121         EXPECT_EQ( SPMatrix<int>(M2 + M1 * cnst), S2 + S1 * cnst );
122         EXPECT_EQ( SPMatrix<int>(M1 + M1), S1 + S1 );
123         EXPECT_EQ( SPMatrix<int>(-M1), -S1);
124         EXPECT_EQ( SPMatrix<int>(M2), +S2);
125
126
127         EXPECT_EQ( SPMatrix<int>(M2 + 3 * M2 * 2 * cnst), 3 * S2 * 2 * cnst +
        S2 );
128         EXPECT_NE( SPMatrix<int>(M1 + M1 + M1), S1 * 13 + S2 );
129     }
130 }
131
132 TEST(Other, Transposing) {
133     for(int i = 0; i < 50; ++i){
134         Matrix<int> M1(4, 12), M2(4, 12);
135         M1.FillMatrixRandom();
136         M2.FillMatrixRandom();
137
138         SPMatrix<int> S1(M1), S2(M2);
139
140         EXPECT_EQ( SPMatrix<int>(M1.Transposition()), S1.Transpose() );
141         EXPECT_EQ( SPMatrix<int>(M2.Transposition()), S2.Transpose() );
142         EXPECT_NE( SPMatrix<int>(M1.Transposition()), S2.Transpose() );
143     }
144 }
145
```

```
146 TEST(Other, Copy) {
147     for (int i = 0; i < 3; ++i){
148         Matrix<int> M1(4, 18);
149         M1.FillMatrixRandom();
150         Matrix<int> M2(M1);
151         M2.FillMatrixRandom();
152         SPMatrix<int> S1(M1), S2(M2);
153         SPMatrix<int> S3 = S1 + S2;
154         Matrix<int> M3 = M1 + M2;
155         SPMatrix<int> S4(S1 + S3);
156         Matrix<int> M4(M3 + M1);
157 
158         EXPECT_EQ(SPMatrix<int>(M3), S3);
159         EXPECT_EQ(SPMatrix<int>(M4), S4);
160         SPMatrix<int> S5;
161         S5 = S1 * 17 + S3;
162         Matrix<int> M5;
163         M5 = M1 * 17 + M3;
164         EXPECT_EQ(SPMatrix<int>(M5), S5);
165         S5 += S4;
166         M5 = M5 + M4;
167         EXPECT_EQ(SPMatrix<int>(M5), S5);
168         S5 -= S3;
169         M5 = M5 - M3;
170         EXPECT_EQ(SPMatrix<int>(M5), S5);
171 
172         Matrix<int> MM1(12, 23), MM2(23, 19);
173         MM1.FillMatrixRandom();
174         MM2.FillMatrixRandom();
175         SPMatrix<int> SS1(MM1), SS2(MM2);
176         SS1 *= SS2;
177         MM1 = MM1 * MM2;
178         SS1 *= 127 * i;
179         MM1 = MM1 * 127 * i;
180         EXPECT_EQ(SPMatrix<int>(MM1), SS1);
181         EXPECT_NE(SPMatrix<int>(MM1), S1);
182         EXPECT_NE(SPMatrix<int>(M4), S2);
183     }
184 
185 }
186 
187 TEST(Other, Move) {
188     for (int i = 0; i < 1; ++i){
189         Matrix<int> M1(18, 39), M2, M3, M4;
190         M1.FillMatrixRandom();
191         M2 = M1;
192         M2.FillMatrixRandom();
193         SPMatrix<int> S1(M1), S2(M2), S3, S4;
194         S3 = std::move(S1);
195         S4 = std::move(S2);
196         M3 = std::move(M1);
197         M4 = std::move(M2);
198 
```

```
199         EXPECT_EQ(SPMatrix<int>(M3), S3);
200         EXPECT_EQ(SPMatrix<int>(M4), S4);
201         SPMatrix<int> S5(std::move(S3));
202         Matrix<int> M5(std::move(M3));
203         EXPECT_EQ(SPMatrix<int>(M5), S5);
204     }
205 }
```

## B.4 Файл Matrix.hpp

```
1  #ifndef MATRIX_h
2  #define MATRIX_h
3
4  #include <iostream>
5  #include <cassert>
6  #include <random>
7  #include <utility>
8
9  const long int INF = 1e2;
10
11 template<typename T> //DEBUG
12 class SPMatrix;
13
14 template<typename T>
15 class Matrix
16 {
17     friend SPMatrix<T>;
18     private:
19
20         size_t rows;
21         size_t cols;
22         T determinant;
23         bool determinantIsNaN;
24
25         T *matrix;
26
27
28     public:
29         // Matrix() = default;
30         Matrix( size_t rows = 0, size_t cols = 0);
31         Matrix( const Matrix & other);
32         Matrix( Matrix&& other);
33
34         ~Matrix();
35
36         void WriteMatrix() const;
37         void ReadMatrix();
38
39         void FillMagickSE(); // DEBUG
40         void FillMatrix();   // DEBUG
41         void FillMatrixOp(); // DEBUG
42
43         void FillMatrixRandom(T (*CustomRandom)());    // DEBUG
44         void FillMatrixRandom();                       // DEBUG
```

```
45
46          T CalcDeterminant();
47          Matrix GaussianMethod() const;
48          void swapRows(size_t i, size_t j);
49          T MinorsMethod() const;
50          Matrix Minor(size_t i, size_t j) const;
51
52          Matrix Transposition() const;
53
54          Matrix operator+( const Matrix & other ) const;
55          Matrix operator-( const Matrix & other ) const;
56          Matrix operator-() const;
57          Matrix operator*( const Matrix & other ) const;
58          Matrix operator*( const T &n) const;
59          Matrix& operator=( const Matrix & other );
60          Matrix& operator=( Matrix&& other);
61
62          T& operator() (const size_t i, const size_t j) const;
63          T& operator() (const size_t i, const size_t j);
64
65      friend std::ostream& operator<<(std::ostream &out, const Matrix &a)
66      {
67          for(size_t i=0; i < a.rows; ++i){
68              for(size_t j=0; j < a.cols; ++j)
69                  out << a(i, j) << ' ';
70              out << '\n';
71          }
72          out << "\n";
73          return out;
74      }
75
76      friend Matrix operator*( const T &n, const Matrix &M)
77      {
78          Matrix<T> result = Matrix<T>(M.rows, M.cols);
79          for(size_t i=0; i < M.rows * M.cols; ++i)
80             result.matrix[i] = M.matrix[i] * n;
81          return result;
82      }
83
84 };
85
86
87 template<typename T>
88 T& Matrix<T>::operator() (const size_t i, const size_t j) const
89 {
90      return matrix[i * cols + j];
91 }
92
93 template<typename T>
94 T& Matrix<T>::operator() (const size_t i, const size_t j)
95 {
96      return matrix[i * cols + j];
97 }
```

```
98
99
100 template<typename T>
101 Matrix<T> Matrix<T>::operator+(const Matrix<T> & other) const
102 {
103     assert(rows == other.rows && cols == other.cols);
104
105     Matrix result = Matrix(other);
106     for(size_t i=0; i < rows * cols; ++i)
107         result.matrix[i] += matrix[i];
108
109     return result;
110 }
111
112 template<typename T>
113 Matrix<T> Matrix<T>::operator-(const Matrix<T> & other) const
114 {
115     assert(rows == other.rows && cols == other.cols);
116
117     return *this + -1 * other;
118 }
119
120 template<typename T>
121 Matrix<T> Matrix<T>::operator-() const
122 {
123     return -1 * (*this);
124 }
125
126 template<typename T>
127 Matrix<T> Matrix<T>::operator*(const Matrix<T> & other) const
128 {
129     assert(cols == other.rows);
130
131
132     Matrix result(rows, other.cols);
133     for(size_t i=0; i < rows; ++i)
134         for(size_t j=0; j < other.cols; ++j){
135             T sum = 0;
136             for(size_t r=0; r < cols; ++r)
137                 sum += (*this)(i, r) * other(r, j);
138             result(i, j) = sum;
139         }
140     return result;
141 }
142
143 template<typename T>
144 Matrix<T> Matrix<T>::operator*(const T &n) const
145 {
146     Matrix result = Matrix(rows, cols);
147     for(size_t i=0; i < rows * cols; ++i)
148         result.matrix[i] = matrix[i] * n;
149     return result;
150 }
```

```
151
152
153 template<typename T>
154 Matrix<T>& Matrix<T>::operator=(Matrix&& other)
155 {
156     if(this == &other)
157         return *this;
158     delete[] matrix;
159     matrix = std::exchange(other.matrix, nullptr);
160     rows = std::exchange(other.rows, 0);
161     cols = std::exchange(other.cols, 0);
162
163     determinantIsNaN = other.determinantIsNaN;
164     determinant = other.determinant;
165
166     return *this;
167 }
168
169 template<typename T>
170 Matrix<T>& Matrix<T>::operator=(const Matrix<T> &other)
171 {
172     if (this == &other)
173         return *this;
174
175     rows = other.rows;
176     cols = other.cols;
177
178     determinant = other.determinant;
179     determinantIsNaN = other.determinantIsNaN;
180
181     T *temp = matrix;
182
183     try{
184         matrix = new T[rows * cols];
185         for(size_t i=0; i < rows*cols; ++i)
186             matrix[i] = other.matrix[i];
187         delete[] temp;
188     }
189     catch(...)
190     {
191         matrix = temp;
192
193         assert(false);
194     }
195
196     return *this;
197 }
198
199
200 template<typename T>
201 Matrix<T>::~Matrix()
202 {
203     delete[] matrix;
```

```
204
205     matrix = nullptr;
206 }
207
208 template<typename T>
209 Matrix<T>::Matrix(const Matrix<T> &other)
210 {
211     rows = other.rows;
212     cols = other.cols;
213     determinant = other.determinant;
214     determinantIsNaN = other.determinantIsNaN;
215
216     try{
217         matrix = new T[rows * cols];
218
219         for(size_t i = 0; i < rows * cols; ++i)
220                 matrix[i] = other.matrix[i];
221     }
222     catch(...)
223     {
224         assert(false);
225     }
226 }
227
228 template<typename T>
229 Matrix<T>::Matrix( Matrix&& other)
230 {
231     matrix = std::exchange(other.matrix, nullptr);
232     rows = std::exchange(other.rows, 0);
233     cols = std::exchange(other.cols, 0);
234     determinantIsNaN = std::exchange(other.determinantIsNaN, true);
235     determinant = std::exchange(other.determinant, 0);
236 }
237
238 template<typename T>
239 Matrix<T>::Matrix(size_t r, size_t c) : rows(r), cols(c), determinantIsNaN(
    true), determinant(0)
240 {
241     //rows = r;
242     //cols = c;
243     //determinantIsNaN = true;
244
245     try{
246         matrix = new T[rows * cols];
247         for(size_t i=0; i < rows * cols; ++i)
248             matrix[i] = 0;
249     }
250     catch(...)
251     {
252         assert(false);
253     }
254 }
255
```

```
256
257 template<typename T>
258 Matrix<T> Matrix<T>::Transposition() const
259 {
260     Matrix result = Matrix(cols, rows);
261     for(size_t i=0; i < rows; ++i)
262         for(size_t j=0; j < cols; ++j)
263             result(j, i) = (*this)(i, j);
264     return result;
265 }
266
267
268 template<typename T>
269 T Matrix<T>::CalcDeterminant()
270 {
271     if (!determinantIsNaN)
272         return determinant;
273
274     if (rows != cols)
275         return MinorsMethod();
276
277
278     determinant = 1;
279     determinantIsNaN = false;
280     Matrix Triangular = GaussianMethod();
281     if (Triangular.matrix == nullptr){
282         determinant = MinorsMethod();
283         return determinant;
284     }
285     for(size_t i=0; i < rows; ++i)
286         determinant *= Triangular(i, i);
287
288     return determinant;
289 }
290
291 template<typename T>
292 Matrix<T> Matrix<T>::GaussianMethod() const
293 {
294     Matrix result = Matrix(*this);
295     int determinant_ratio = 1;
296     for(size_t k=0; k < rows; ++k)
297     {
298         if (result(k, k) == T(0)){
299             size_t i = k + 1;
300             while (i < cols && result(k, i) == T(0))
301                 ++i;
302             if (i != cols){
303                 result.swapRows(k, i);
304                 determinant_ratio *= -1;
305             }
306             else
307                 return Matrix<T>();
308         }
```

```
309
310         for (size_t i=k+1; i < rows; ++i)
311         {
312             T ratio = result(i, k) / result(k, k);
313             for (size_t j = 0; j < cols; ++j){
314                 result(i, j) -= result(k, j) * ratio;
315             }
316         }
317     }
318     return result * determinant_ratio;
319 }
320
321 template<typename T>
322 void Matrix<T>::swapRows(size_t row_1, size_t row_2)
323 {
324     for(size_t i=0; i<cols; ++i){
325         T temp = (*this)(row_1, i);
326         (*this)(row_1, i) = (*this)(row_2, i);
327         (*this)(row_2, i) = temp;
328     }
329 }
330
331 template<typename T>
332 T Matrix<T>::MinorsMethod() const
333 {
334     T result = 0;
335     if (rows==2 && cols==2)
336         return (*this)(0,0) * (*this)(1,1) - (*this)(0, 1) * (*this)(1, 0);
337
338     for(size_t i=0; i < rows; ++i){
339         result += (*this)(0, i) * T(i%2==0?1:-1) * Minor(0, i).MinorsMethod();
340     }
341     return result;
342 }
343
344 template<typename T>
345 Matrix<T> Matrix<T>::Minor(size_t i, size_t j) const
346 {
347     Matrix result(rows-1, cols-1);
348     bool flag_row = false; // flag for skipping r == i
349
350     for(size_t r=0; r<rows-1; ++r){
351         bool flag_col = false; // flag for skipping c = j every r
352
353         for(size_t c=0; c<cols-1; ++c){
354             if (r == i) flag_row = true;
355             if (c == j) flag_col = true;
356
357             result(r, c) = (*this)(r + flag_row, c + flag_col);
358         }
359     }
360     return result;
361 }
```

```
362
363
364 template<typename T>
365 void Matrix<T>::FillMagickSE() // Funcs to fill matrix in a weird way for
        testing purposes
366 {
367     size_t cnt = 0;
368     for (size_t i = 0; i < rows; ++i)
369         for (size_t j = 0; j < cols; ++j){
370             if (cols <= j + i){
371                 ++cnt;
372                 (*this)(i, j) = cnt;
373             }
374     }
375 }
376
377 template<typename T>
378 void Matrix<T>::FillMatrix()
379 {
380     size_t cnt = 0;
381     for(size_t i = 0; i < rows; ++i)
382         for(size_t j = 0; j < cols; ++j)
383         {
384             (*this)(i, j) = cnt;
385             ++cnt;
386         }
387 }
388
389 template<typename T>
390 void Matrix<T>::FillMatrixOp()
391 {
392     size_t cnt = rows * cols;
393     for(size_t i = 0; i < rows; ++i)
394         for(size_t j = 0; j < cols; ++j)
395         {
396             (*this)(i, j) = cnt;
397             --cnt;
398         }
399 }
400
401
402 template<typename T>
403 T Random()
404 {
405     static std::random_device r;
406     static std::default_random_engine e1(r());
407     std::uniform_int_distribution<T> uniform_dist(-INF, INF);
408     return uniform_dist(e1);
409 }
410
411 template<>
412 double Random()
413 {
```

```
414     static std::random_device r;
415     static std::default_random_engine e1(r());
416     static std::uniform_real_distribution<double> uniform_dist(-INF, INF);
417     return uniform_dist(e1);
418 }
419
420 template<typename T>
421 void Matrix<T>::FillMatrixRandom(T (*CustomRandom)())
422 {
423     for(size_t i = 0; i < rows; ++i)
424         for(size_t j = 0; j < cols; ++j){
425             (*this)(i, j) = (*CustomRandom)();
426         }
427 }
428
429 template<typename T>
430 void Matrix<T>::FillMatrixRandom()
431 {
432    for(size_t i = 0; i < rows; ++i)
433         for(size_t j = 0; j < cols; ++j){
434             (*this)(i, j) = Random<T>();
435         }
436 }
437
438
439 template<typename T>
440 void Matrix<T>::WriteMatrix() const
441 {
442     std::cout << rows << ' ' << cols << std::endl;
443
444     for(size_t i = 0; i < rows; ++i){
445         for(size_t j = 0; j < cols; ++j)
446             std::cout << (*this)(i, j) << ' ';
447         std::cout << std::endl;
448     }
449 }
450
451
452 template<typename T>
453 void Matrix<T>::ReadMatrix()
454 {
455     size_t r, c;
456     std::cin >> r >> c;
457     if (r != rows || c != cols)
458     {
459         T temp = matrix;
460         try{
461             matrix = new T[rows * cols];
462             rows = r;
463             cols = c;
464         }
465         catch(...)
466         {
```

```
467                 matrix = temp;
468                 assert(false);
469             }
470         }
471 
472         for(size_t i=0; i<rows; ++i)
473             for(size_t j=0; j<cols; ++j)
474                 std::cin >> (*this)(i, j);
475 }
476 
477 
478 #endif
```