Федеральное агентство по образованию Российской Федерации

Государственное образовательное учреждение

Высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий математики механики

БитОвые поля

Отчет по лабораторной работе

Выполнил:

студент ИИТММ гр. 381706-2

Антипин А.С.\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил:

ассистент каф. МОСТ, ИИТММ

Лебедев И.Г.\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Нижний Новгород

2018 г.

Содержание:

[Введение 3](#_Toc533188730)

[Постановка задач и целей 4](#_Toc533188731)

[Постановка задачи: 4](#_Toc533188732)

[Цели работы: 4](#_Toc533188733)

[Руководство пользователя 5](#_Toc533188734)

[Руководство программиста 7](#_Toc533188735)

[Описание структуры программы 7](#_Toc533188736)

[Описание структур данных 8](#_Toc533188737)

[Описание алгоритмов 8](#_Toc533188738)

[Эксперименты 12](#_Toc533188739)

[Время выполнения 12](#_Toc533188740)

[Заключение 14](#_Toc533188741)

[Литература 15](#_Toc533188742)

[Приложение 16](#_Toc533188743)

# Введение

При разработке современного программного обеспечения для десктопных устройств, разработчик, как правило, не старается экономить память, которую будет занимать программа при выполнении. Однако, в некоторых случаях это просто необходимо, ввиду ее недостатка на устройстве, под которое ведется разработка, или в связи с нехваткой вычислительной мощностью этого устройства. В отдельных случаях, необходимую информацию можно представить в виде последовательности бит. Это эффективно, когда основной задачей стоит определение присутствия элементов в множестве. Все возможные элементы нумеруются, затем каждому присваивается 1 если этот он присутствует в множестве и 0 если он не присутствует.

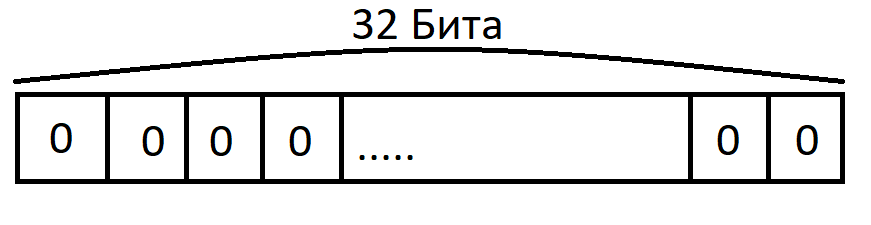
Разберем пример: пусть у нас есть переменная типа integer, она кодируется 4 байтами, в каждом байте 8 бит, соответственно всего у нас есть 32 бита. Если присвоить этой переменной значение 0, то и все биты «занулятся» (рис. 1), если же присвоить этой переменной значение 1, то 31 старших бит останутся равными 0, а последний 32 бит примет значение равное 1 (рис. 2). Соответственно при 2 у нас будет 30 старших бит с 0 значением, 31-ый с 1-ым и 32-ой с 0-ым (рис. 3).

рис. 2

рис. 1

рис. 3

Таким образом тратится гораздо меньше памяти на хранение информации. В C++ уже реализованы операции работы с битами, но только для конкретных типов данных. А что если элементов в множестве больше чем количество бит, которое требуется для представления этого элемента? Тогда нужно написать собственный класс битовых полей, который не будет зависеть от разрядности типа данных и в котором будут перегружены операции над битами.

# Постановка задач и целей

## Постановка задачи:

Создать класс «Битовое поле», в котором будут перегружены основные битовые операции, от него унаследовать класс «Набор», в котором будут реализованы методы работы с множествами. Для проверки, реализовать «Решето Эратосфена» с помощью методов класса «Битовое поле» и «Набор».

## Цели работы:

* Познакомиться с битовыми операциями;
* Познакомиться с системой контроля версий – GIT;
* Познакомится с фреймворком gtest и пройти все тесты, написанные для классов «Битовое поле» и «Набор».

# Руководство пользователя

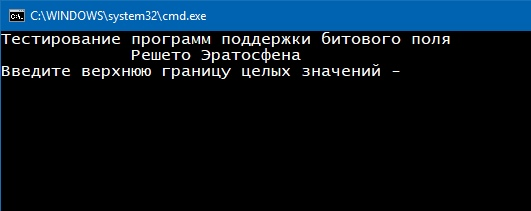
При запуске программы, в консоль будут написаны следующие слова (рис. 4):

рис. 4 (слова, которые выводятся при запуске программы)

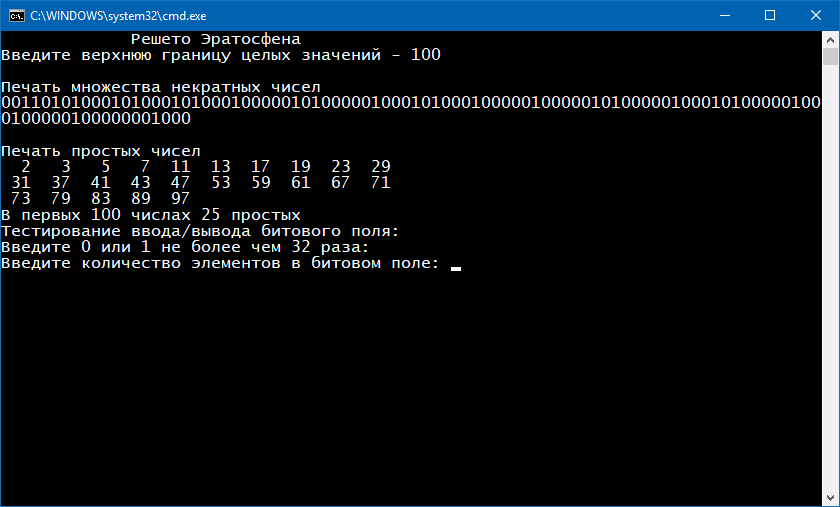
Здесь пользователю потребуется ввести любое натуральное число, после нажать кнопку Enter, после чего программа найдет все простые числа в интервале от 2 до заданного числа (рис 5)

рис. 5 (вывод простых чисел и просьба ввести какую-либо последовательность бит)

Далее программа тестирует потоковый ввод/вывод и просит ввести последовательность 0 и 1 для записи ее в битовое поле и вывода этой последовательности (рис 6.). От пользователя требуется ввести количество элементов в битовом поле и само поле.

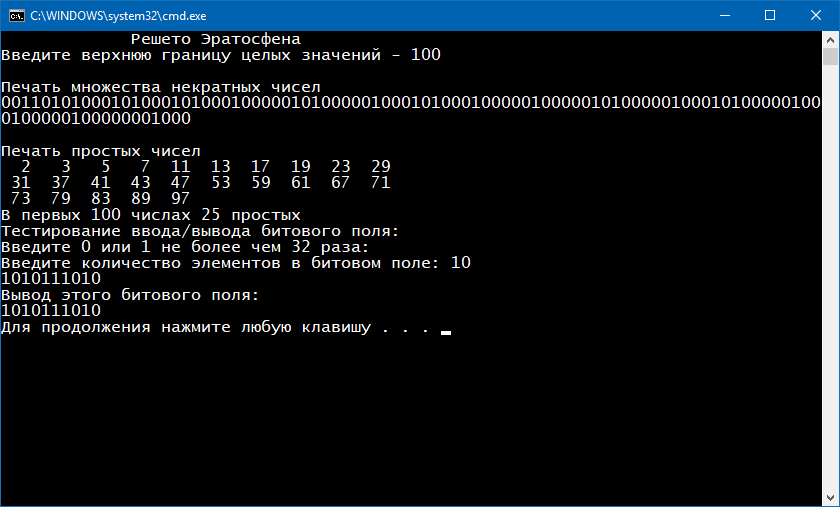


рис. 6 (окончательный вывод программы)

# Руководство программиста

## Описание структуры программы

Программа состоит из 4 основных модулей:

* Класс «TBitField»;
* Класс «TSet»;
* Проект, использующий фреймворк Google Test, для проверки правильности работы классов «TBitField» и «TSet»;
* Проект, реализующий решето Эратосфена.

Древо классов

**Класс gtest**

**test\_main.cpp**

**test\_tbitfield.cpp**

**test\_tset.cpp**

**Проект sample\_prime\_numbers**

**sample\_prime\_numbers.cpp**

**tset.h**

**tset.cpp**

**Класс Tset**

**tbitfield.h**

**tbitfield.cpp**

**Класс Tbitfield**

**Класс Tbitfield:**

Класс tbitfield содержит реализацию класса «Битовое поле», описанного во введении работы. В файле tbitfield.h написано объявление этого класса, а в tbitfield.cpp написана его реализация.

**Класс Tset:**

Класс tset содержит реализацию класса «Набор», который упрощает работу с множествами. В нем реализованы такие методы, как «включить элемент в множество», «проверить наличие элемента в множестве», «пересечение», «объединение», «дополнение» множеств и др. В файле tset.h написано объявление этого класса, а в tset.cpp написана его реализация.

**Класс gtest:**

Класс gtest реализует тестирование классов tbitfield и tset, по средствам фреймворка Google Test. Тесты пишутся для каждого метода классов, каждого ветвления этих методов и для всех возможных исключений этих методов. В файле «test\_tbitfield.cpp» реализованы тесты для класса tbitfield, а в файле «test\_tsetcpp» соответственно для класса tset.

**Проект sample\_prime\_numbers:**

В данном проекте реализован примет использования битовых полей и множеств, который в конечном итоге и будет доступен пользователю. Здесь используется предкомпиляция, поэтому можно увидеть, как будет вести себя программа при использовании битовых полей, и при использовании класса «Набор» (идентичное поведение).

## Описание структур данных

**Класс TBitField:**

Поля:

typedef unsigned int TELEM

int BitLen; - длина битового поля — максимальное количество битов

TELEM \*pMem; - память для представления битового поля

int MemLen; - количество элементов Мем для представления битового поля

**Класс TSet:**

Поля:

int MaxPower; - максимальная мощность множества

TBitField BitField; - битовое поле для хранения характеристического вектора

## Описание алгоритмов

TBitField::TBitField(int len) — конструктор с параметром, который принимает целое число, равное числу битов, необходимых пользователю для использования. BitLen = len, MemLen = len / (8 \* sizeof(TELEM)) + 1 — вычисляется количество элементов заданного типа, необходимые для представления нужного числа битов, выделяется память необходимого размера и зануляются все элементы: pMem = new TELEM[MemLen]; for (int i = 0; i < MemLen; i++) pMem[i] = 0.

TBitField::TBitField(const TBitField &bf) — конструктор копирования, который принимает константную ссылку на объект своего типа. Копирует поля другого элемента, где необходимо использует цикл.

TbitField::~TBitField() - деструктор, который необходим для очистки памяти от массива элементов TELEM.if (pMem != NULL) delete []pMem.

int TBitField::GetMemIndex(const int n) const — метод, который возвращает индекс элемента, в котором находится нужный бит. Принимает целочисленный параметр n. return n / (8 \* sizeof(TELEM)) — поиск этого индекса.

TELEM TBitField::GetMemMask(const int n) const — битовая маска для бита n. Элемент, битовое представление которого соответствует всем нулям и единице в n-ой позиции. Вычисление позиции этого бита в типе данных int - int bit = n % (8 \* sizeof(TELEM)), побитовый сдвиг единицы - return 1 << bit.

int TBitField::GetLength(void) const — возврат количества битов.

void TBitField::SetBit(const int n) — сделать активным бит в указанном положении, т.е. бит в этой позиции становится равным 1. Метод принимает целочисленный аргумент — номер бита. int i = GetMemIndex(n) — вычисление в каком элементе из массива нужно поставить бит, int bit = GetMemMask(n) — вычисление самого бита, pMem[i] = pMem[i] | bit — логическое ИЛИ, которое позволяет поставить бит.

void TBitField::ClrBit(const int n) — очистка нужного бита, т.е. присвоение ему значение 0. Аналогично предыдущему методу принимает номер бита и вычисляется номер элемента в массиве, номер бита в этом элементе, меняется только способ присвоения 0 - pMem[i] = pMem[i] & ~bit.

int TBitField::GetBit(const int n) const — получить значение бита по его номеру. n – номер бита. Схожая реализация с предыдущими, меняется только последняя строка, в которой возвращается значение этого бита с помощью логического И - return pMem[i] & bit.

TBitField& TBitField::operator=(const TBitField &bf) — перегрузка оператора присваивание. Принимает константную ссылку на объект типа TbitField и возвращает ссылку на объект того же типа. if (pMem != bf.pMem) — проверка на присвоение объекта самому себе. Далее идет копия всех полей класса TbitField.

int TBitField::operator==(const TBitField &bf) const – перегрузка оператора равно. Принимает константную ссылку на объект типа TbitField и возвращает 0 если объекты не равны и 1 если они равны. for (int i = 0; i < MemLen; i++), if (pMem[i] != bf.pMem[i]) — проверка каждого элемента из массива на равенство, если хотя бы один из них не равен соответствующему элементу другого массива, то и битовые поля не равны.

int TBitField::operator!=(const TBitField &bf) const — перегрузка оператора не равно. Реализация этого метода схожа с реализацией перегрузке оператора равно.

TBitField TBitField::operator|(const TBitField &bf) — перегрузка битовой операции ИЛИ. Принимает константную ссылку на объект своего типа, и возвращает объект этого же типа. if (bf.BitLen < BitLen){ less = bf.BitLen; more = BitLen; lessM = Memlen; moreM = MemLen;} - нахождение длины наибольшего массива элементов из 2 битовых полей. for (int i = 0; i < lessM; i++) tmp.pMem[i] = pMem[i] | bf.pMem[i]; - копирование в результирующий массив результата выполнения логического ИЛИ с меньшим количеством элементов из 2 массивов. Далее в результирующим массив копируются все оставшиеся данные из наибольшего массива, которые не были преобразованы с помощью операции ИЛИ.

TBitField TBitField::operator&(const TBitField &bf) – перегрузка битовой операции И. Реализация этого метода схожа с реализацией перегрузки оператора ИЛИ, с той лишь разницей, что если длины битовых полей различны, то в результирующее поле записывается результат выполнения операции только пока есть элементы в наименьшем поле, а остальные элементы в результирующем поле зануляются.

TBitField TBitField::operator~(void) – перегрузка оператора ОТРИЦАНИЕ для битов. Возвращает объект типа TBitField. for (int i = 0; i < MemLen; i++) rez.pMem[i] = ~pMem[i]; - каждый элемент в массиве «отрицается», rez.pMem[MemLen - 1] &= g >> (MemLen \* sizeof(TELEM) \* 8 - BitLen); - зануляются все элементы, которые не входят в данное битовое поле.

TSet::TSet(int mp) : BitField(mp) – конструктор с параметром, который наследуется от конструктора инициализатора битового поля. Принимает количество элементов в множестве.

TSet::TSet(const TSet &s) : BitField(s.BitField) – конструктор копирования множества, который также наследуется от конструктора в TBitField. Принимает константную ссылку на объект этого же класса.

TSet::TSet(const TBitField &bf) : BitField(bf) – конструктор преобразования типа из TBitField в TSet.

TSet::operator TBitField() – оператор преобразования TSet к TBitField.

int TSet::GetMaxPower(void) const – возвращает максимальное количество элементов в множестве.

int TSet::IsMember(const int Elem) const – проверка на присутствие элемента с данным индексом в множестве.

void TSet::InsElem(const int Elem) – включить элемент в множество.

void TSet::DelElem(const int Elem) – исключить элемент из множества.

TSet& TSet::operator=(const TSet &s) – перегрузка оператора присваивания для множества.

int TSet::operator==(const TSet &s) const – перегрузка оператора сравнения.

int TSet::operator!=(const TSet &s) const – перегрузка оператора не равно.

TSet TSet::operator+(const TSet &s) – перегрузка оператора объединение для множеств. Принимает и возвращает объект типа TSet. if (MaxPower >= s.MaxPower) – поиск множества наибольшей длины. TSet tmp(MaxPower); tmp.BitField = BitField | s.BitField; - создание результирующего множества и присвоение ему результата логического ИЛИ двух битовых полей.

TSet TSet::operator+(const int Elem) – объединение с элементом.

TSet TSet::operator-(const int Elem) – разность с элементом.

TSet TSet::operator\*(const TSet &s) – перегрузка оператора пересечение для множества. Принимает и возвращает объект типа TSet. Реализация схожа с реализацией оператора объединения, с разницей лишь в использовании логического И в битовом поле.

TSet TSet::operator~(void) – дополнение для множества.

# Эксперименты

## Время выполнения

Рассмотрим время выполнения некоторых методов:

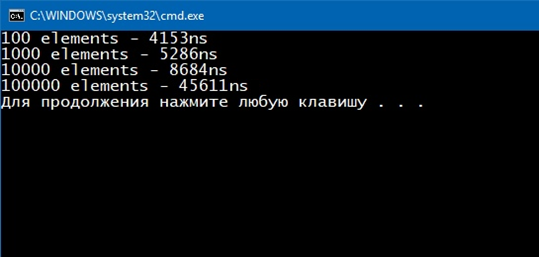
Логическое И – теоретическая сложность O(n). На практике получаются следующие числа (рис 11):

рис. 11 (время выполнения операции «логическое ИЛИ»)

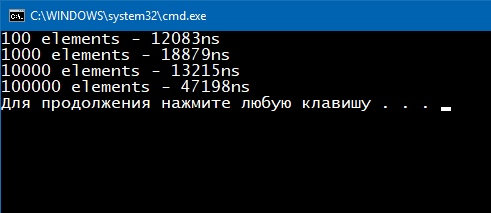
Логическое отрицание - теоретическая сложность O(n). На практике получаются следующие числа: (рис 12)

рис. 12 (время выполнения операции «логическое отрицание»)

Составим диаграмму с временем работы методов, от разного количества элементов:

время выполнения операции логических операций

С учетом погрешности измерения, можно сказать, что оба алгоритма работают линейно.

Тесты проводились на системе:

Процессор Intel Core i5 7200U;

Оперативная память 12 GB.

# Заключение

В заключении можно сказать, что все поставленные цели и задачи были выполнены, а именно: созданы классы «TBitField» и «TSet», написаны к ним тесты, и они успешно пройдены. С помощью этих классов была реализована небольшая, но полезная программа – «Решето Эратосфена», которая помогает находить простые числа. По ходу выполнения работы получены навыки работы с GIT.

# Литература

* Учебные материалы к учебному курсу «Методы программирования» - Гергель В.П.

# Приложение

**tbitfield.h:**

// ННГУ, ВМК, Курс "Методы программирования-2", С++, ООП

//

// tbitfield.h - Copyright (c) Гергель В.П. 07.05.2001

// Переработано для Microsoft Visual Studio 2008 Сысоевым А.В. (19.04.2015)

//

// Битовое поле

#ifndef \_\_BITFIELD\_H\_\_

#define \_\_BITFIELD\_H\_\_

#include <iostream>

using namespace std;

typedef unsigned int TELEM;

class TBitField

{

private:

int tBitLen; // длина битового поля - макс. к-во битов

TELEM \*pMem; // память для представления битового поля

int tMemLen; // к-во эл-тов Мем для представления бит.поля

// методы реализации

int GetMemIndex(const int n) const; // индекс в pМем для бита n (#О2)

TELEM GetMemMask (const int n) const; // битовая маска для бита n (#О3)

public:

TBitField(int len); // (#О1)

TBitField(const TBitField &bf); // (#П1)

~TBitField(); // (#С)

// доступ к битам

int GetLength(void) const; // получить длину (к-во битов) (#О)

void SetBit(const int n); // установить бит (#О4)

void ClrBit(const int n); // очистить бит (#П2)

int GetBit(const int n) const; // получить значение бита (#Л1)

// битовые операции

int operator==(const TBitField &bf) const; // сравнение (#О5)

int operator!=(const TBitField &bf) const; // сравнение

TBitField& operator=(const TBitField &bf); // присваивание (#П3)

TBitField operator|(const TBitField &bf); // операция "или" (#О6)

TBitField operator&(const TBitField &bf); // операция "и" (#Л2)

TBitField operator~(void); // отрицание (#С)

friend istream &operator>>(istream &istr, TBitField &bf); // (#О7)

friend ostream &operator<<(ostream &ostr, const TBitField &bf); // (#П4)

};

// Структура хранения битового поля

// бит.поле - набор битов с номерами от 0 до tBitLen

// массив pМем рассматривается как последовательность tMemLen элементов

// биты в эл-тах pМем нумеруются справа налево (от младших к старшим)

// О8 Л2 П4 С2

#endif

**tbitfield.cpp:**

// ННГУ, ВМК, Курс "Методы программирования-2", С++, ООП

//

// tbitfield.cpp - Copyright (c) Гергель В.П. 07.05.2001

// Переработано для Microsoft Visual Studio 2008 Сысоевым А.В. (19.04.2015)

//

// Битовое поле

#include "tbitfield.h"

TBitField::TBitField(int len)

{

if (len >= 0)

{

tBitLen = len;

tMemLen = len / (8 \* sizeof(TELEM)) + 1;

pMem = new TELEM[tMemLen];

for (int i = 0; i < tMemLen; i++)

pMem[i] = 0;

}

else

throw(1);

}

TBitField::TBitField(const TBitField &bf) // конструктор копирования

{

tMemLen = bf.tMemLen;

tBitLen = bf.tBitLen;

pMem = new TELEM[tMemLen];

for (int i = 0; i < tMemLen; i++)

pMem[i] = bf.pMem[i];

}

TBitField::~TBitField()

{

if (pMem != NULL)

delete[]pMem;

}

int TBitField::GetMemIndex(const int n) const // индекс Мем для бита n

{

if (n < 0 || n > tBitLen)

throw(1);

return n / (8 \* sizeof(TELEM));

}

TELEM TBitField::GetMemMask(const int n) const // битовая маска для бита n

{

if (n < 0 || n > tBitLen)

throw(1);

int bit = n % (8 \* sizeof(TELEM));

return 1 << bit;

}

// доступ к битам битового поля

int TBitField::GetLength(void) const // получить длину (к-во битов)

{

return tBitLen;

}

void TBitField::SetBit(const int n) // установить бит

{

if (n < 0 || n > tBitLen)

throw(1);

int i = GetMemIndex(n);

int bit = GetMemMask(n);

pMem[i] = pMem[i] | bit;

}

void TBitField::ClrBit(const int n) // очистить бит

{

if (n < 0 || n > tBitLen)

throw(1);

int i = GetMemIndex(n);

int bit = GetMemMask(n);

pMem[i] = pMem[i] & ~bit;

}

int TBitField::GetBit(const int n) const // получить значение бита

{

if (n < 0 || n > tBitLen)

throw(1);

int i = GetMemIndex(n);

int bit = GetMemMask(n);

return (pMem[i] & bit) != 0;

}

// битовые операции

TBitField& TBitField::operator=(const TBitField &bf) // присваивание

{

if (pMem != bf.pMem)

{

tBitLen = bf.tBitLen;

tMemLen = bf.tMemLen;

if (pMem)

delete[]pMem;

pMem = new TELEM[tMemLen];

for (int i = 0; i < tMemLen; i++)

pMem[i] = bf.pMem[i];

}

return \*this;

}

int TBitField::operator==(const TBitField &bf) const // сравнение

{

bool rez = true;

if (tBitLen != bf.tBitLen || tMemLen != bf.tMemLen)

rez = false;

else

{

for (int i = 0; i < tMemLen; i++)

if (pMem[i] != bf.pMem[i])

rez = false;

}

return rez;

}

int TBitField::operator!=(const TBitField &bf) const // сравнение

{

bool rez = true;

if (tBitLen != bf.tBitLen || tMemLen != bf.tMemLen)

return rez;

else

{

for (int i = 0; i < (tBitLen / (sizeof(int) \* 8)); i++)

if (pMem[i] != bf.pMem[i])

return rez;

}

}

TBitField TBitField::operator|(const TBitField &bf) // операция "или"

{

int less = tBitLen;

int more = bf.tBitLen;

int lessM = tMemLen;

int moreM = bf.tMemLen;

if (bf.tBitLen < tBitLen)

{

lessM = bf.tMemLen;

moreM = tMemLen;

less = bf.tBitLen;

more = tBitLen;

}

TBitField tmp(more);

for (int i = 0; i < lessM; i++)

tmp.pMem[i] = pMem[i] | bf.pMem[i];

if (tBitLen > bf.tBitLen)

for (int i = lessM; i < tMemLen; i++)

tmp.pMem[i] = pMem[i];

else

for (int i = lessM; i < bf.tMemLen; i++)

tmp.pMem[i] = bf.pMem[i];

return tmp;

}

TBitField TBitField::operator&(const TBitField &bf) // операция "и"

{

int less = tBitLen;

int more = bf.tBitLen;

int lessM = tMemLen;

int moreM = bf.tMemLen;

if (bf.tBitLen < tBitLen)

{

lessM = bf.tMemLen;

moreM = tMemLen;

less = bf.tBitLen;

more = tBitLen;

}

TBitField tmp(more);

for (int i = 0; i < lessM; i++)

tmp.pMem[i] = pMem[i] & bf.pMem[i];

if (tBitLen > bf.tBitLen)

for (int i = lessM; i < tMemLen; i++)

tmp.pMem[i] = 0;

else

for (int i = lessM; i < bf.tMemLen; i++)

tmp.pMem[i] = 0;

return tmp;

}

TBitField TBitField::operator~(void) // отрицание

{

TBitField rez(tBitLen);

TELEM g = ~(TELEM(0));

for (int i = 0; i < tMemLen; i++)

rez.pMem[i] = ~pMem[i];

rez.pMem[tMemLen - 1] &= g >> (tMemLen \* sizeof(TELEM) \* 8 - tBitLen);

return rez;

}

// ввод/вывод

istream &operator>>(istream &istr, TBitField &bf) // ввод

{

char ch;

for(int i = 0; i < bf.tBitLen; i++)

{

istr >> ch;

if (ch == '0')

bf.ClrBit(i);

else if (ch == '1')

bf.SetBit(i);

else if (ch == ' ')

break;

else if (ch == '\n')

break;

else break;

}

return istr;

}

ostream &operator<<(ostream &ostr, const TBitField &bf) // вывод

{

for (int i = 0; i < bf.tBitLen; i++)

ostr << bf.GetBit(i);

return ostr;

}

**tset.h:**

// ННГУ, ВМК, Курс "Методы программирования-2", С++, ООП

//

// tset.h - Copyright (c) Гергель В.П. 07.05.2001

// Переработано для Microsoft Visual Studio 2008 Сысоевым А.В. (19.04.2015)

//

// Множество

#ifndef \_\_SET\_H\_\_

#define \_\_SET\_H\_\_

#include "tbitfield.h"

class TSet

{

private:

int tMaxPower; // максимальная мощность множества

TBitField tBitField; // битовое поле для хранения характеристического вектора

public:

TSet(int mp);

TSet(const TSet &s); // конструктор копирования

TSet(const TBitField &bf); // конструктор преобразования типа

operator TBitField(); // преобразование типа к битовому полю

// доступ к битам

int GetMaxPower(void) const; // максимальная мощность множества

void InsElem(const int elem); // включить элемент в множество

void DelElem(const int elem); // удалить элемент из множества

int IsMember(const int elem) const; // проверить наличие элемента в множестве

// теоретико-множественные операции

int operator== (const TSet &s) const; // сравнение

int operator!= (const TSet &s) const; // сравнение

TSet& operator=(const TSet &s); // присваивание

TSet operator+ (const int elem); // объединение с элементом

// элемент должен быть из того же универса

TSet operator- (const int elem); // разность с элементом

// элемент должен быть из того же универса

TSet operator+ (const TSet &s); // объединение

TSet operator\* (const TSet &s); // пересечение

TSet operator~ (void); // дополнение

friend istream &operator>>(istream &istr, TSet &bf);

friend ostream &operator<<(ostream &ostr, const TSet &bf);

};

#endif

**tset.cpp:**

// ННГУ, ВМК, Курс "Методы программирования-2", С++, ООП

//

// tset.cpp - Copyright (c) Гергель В.П. 04.10.2001

// Переработано для Microsoft Visual Studio 2008 Сысоевым А.В. (19.04.2015)

//

// Множество - реализация через битовые поля

#include "tset.h"

TSet::TSet(int mp) : tBitField(mp)

{

if (mp < 0)

throw(1);

tMaxPower = mp;

}

// конструктор копирования

TSet::TSet(const TSet &s) : tBitField(s.tBitField)

{

tMaxPower = s.tMaxPower;

}

// конструктор преобразования типа

TSet::TSet(const TBitField &bf) : tBitField(bf)

{

tMaxPower = bf.GetLength();

}

TSet::operator TBitField()

{

return tBitField;

}

int TSet::GetMaxPower(void) const // получить макс. к-во эл-тов

{

return tMaxPower;

}

int TSet::IsMember(const int Elem) const // элемент множества?

{

return tBitField.GetBit(Elem);

}

void TSet::InsElem(const int Elem) // включение элемента множества

{

tBitField.SetBit(Elem);

}

void TSet::DelElem(const int Elem) // исключение элемента множества

{

tBitField.ClrBit(Elem);

}

// теоретико-множественные операции

TSet& TSet::operator=(const TSet &s) // присваивание

{

tMaxPower = s.tMaxPower;

tBitField = s.tBitField;

return \*this;

}

int TSet::operator==(const TSet &s) const // сравнение

{

if (tMaxPower == s.tMaxPower && tBitField == s.tBitField)

return 1;

else

return 0;

}

int TSet::operator!=(const TSet &s) const // сравнение

{

if (tMaxPower == s.tMaxPower && tBitField == s.tBitField)

return 0;

else

return 1;

}

TSet TSet::operator+(const TSet &s) // объединение

{

if (tMaxPower >= s.tMaxPower)

{

TSet tmp(tMaxPower);

tmp.tBitField = tBitField | s.tBitField;

return tmp;

}

if (tMaxPower < s.tMaxPower)

{

TSet tmp(s.tMaxPower);

tmp.tBitField = tBitField | s.tBitField;

return tmp;

}

}

TSet TSet::operator+(const int Elem) // объединение с элементом

{

tBitField.SetBit(Elem);

return \*this;

}

TSet TSet::operator-(const int Elem) // разность с элементом

{

tBitField.ClrBit(Elem);

return \*this;

}

TSet TSet::operator\*(const TSet &s) // пересечение

{

if (tMaxPower >= s.tMaxPower)

{

TSet tmp(tMaxPower);

tmp.tBitField = tBitField & s.tBitField;

return tmp;

}

if (tMaxPower < s.tMaxPower)

{

TSet tmp(s.tMaxPower);

tmp.tBitField = tBitField & s.tBitField;

return tmp;

}

}

TSet TSet::operator~(void) // дополнение

{

tBitField = ~tBitField;

return \*this;

}

// перегрузка ввода/вывода

istream &operator>>(istream &istr, TSet &s) // ввод

{

int temp;

char ch;

do

{

istr >> ch;

} while (ch != '{');

do

{

istr >> temp;

s.InsElem(temp);

do

{

istr >> ch;

} while ((ch != ',') && (ch != '}'));

} while (ch != '}');

return istr;

}

ostream& operator<<(ostream &ostr, const TSet &s) // вывод

{

int i, n;

char ch = ' ';

ostr << "{";

n = s.GetMaxPower();

for (i = 0; i < n; i++)

if (s.IsMember(i))

{

ostr << ch << ' ' << i;

ch = 'i';

}

ostr << " }";

return ostr;

}

**sample\_prime\_numbers.cpp:**

// ННГУ, ВМК, Курс "Методы программирования-2", С++, ООП

//

// sample\_prime\_numbers.cpp - Copyright (c) Гергель В.П. 20.08.2000

// Переработано для Microsoft Visual Studio 2008 Сысоевым А.В. (19.04.2015)

//

// Тестирование битового поля и множества

#include <iomanip>

#include <iostream>

//#define USE\_SET // Использовать класс TSet,

// закоментировать, чтобы использовать битовое поле

#ifndef USE\_SET // Использовать класс TBitField

#include "tbitfield.h"

int main()

{

int n, m, k, count;

setlocale(LC\_ALL, "Russian");

cout << "Тестирование программ поддержки битового поля" << endl;

cout << " Решето Эратосфена" << endl;

cout << "Введите верхнюю границу целых значений - ";

cin >> n;

TBitField s(n + 1);

// заполнение множества

for (m = 2; m <= n; m++)

s.SetBit(m);

// проверка до sqrt(n) и удаление кратных

for (m = 2; m \* m <= n; m++)

// если m в s, удаление кратных

if (s.GetBit(m))

for (k = 2 \* m; k <= n; k += m)

if (s.GetBit(k))

s.ClrBit(k);

// оставшиеся в s элементы - простые числа

cout << endl << "Печать множества некратных чисел" << endl << s << endl;

cout << endl << "Печать простых чисел" << endl;

count = 0;

k = 1;

for (m = 2; m <= n; m++)

if (s.GetBit(m))

{

count++;

cout << setw(3) << m << " ";

if (k++ % 10 == 0)

cout << endl;

}

cout << endl;

int counts = 0;

cout << "В первых " << n << " числах " << count << " простых" << endl;

cout << "Тестирование ввода/вывода битового поля:\nВведите 0 или 1 не более чем 32 раза:\n";

cout << "Введите количество элементов в битовом поле: ";

cin >> counts;

TBitField field(counts);

cin >> field;

cout << "Вывод этого битового поля:\n";

cout << field << endl;

}

#else

#include "tset.h"

int main()

{

int n, m, k, count;

setlocale(LC\_ALL, "Russian");

cout << "Тестирование программ поддержки множества" << endl;

cout << " Решето Эратосфена" << endl;

cout << "Введите верхнюю границу целых значений - ";

cin >> n;

TSet s(n + 1);

// заполнение множества

for (m = 2; m <= n; m++)

s.InsElem(m);

// проверка до sqrt(n) и удаление кратных

for (m = 2; m \* m <= n; m++)

// если м в s, удаление кратных

if (s.IsMember(m))

for (k = 2 \* m; k <= n; k += m)

if (s.IsMember(k))

s.DelElem(k);

// оставшиеся в s элементы - простые числа

cout << endl << "Печать множества некратных чисел" << endl << s << endl;

cout << endl << "Печать простых чисел" << endl;

count = 0;

k = 1;

for (m = 2; m <= n; m++)

if (s.IsMember(m))

{

count++;

cout << setw(3) << m << " ";

if (k++ % 10 == 0)

cout << endl;

}

cout << endl;

cout << "В первых " << n << " числах " << count << " простых" << endl;

}

#endif

**test\_tbitfield.cpp:**

#include "tbitfield.h"

#include <gtest.h>

TEST(TBitField, can\_create\_bitfield\_with\_positive\_length)

{

ASSERT\_NO\_THROW(TBitField bf(3));

}

TEST(TBitField, can\_get\_length)

{

TBitField bf(3);

EXPECT\_EQ(3, bf.GetLength());

}

TEST(TBitField, new\_bitfield\_is\_set\_to\_zero)

{

TBitField bf(100);

int sum = 0;

for (int i = 0; i < bf.GetLength(); i++)

{

sum += bf.GetBit(i);

}

EXPECT\_EQ(0, sum);

}

TEST(TBitField, can\_set\_bit)

{

TBitField bf(10);

EXPECT\_EQ(0, bf.GetBit(3));

bf.SetBit(3);

EXPECT\_NE(0, bf.GetBit(3));

}

TEST(TBitField, can\_clear\_bit)

{

TBitField bf(10);

int bitIdx = 3;

bf.SetBit(bitIdx);

EXPECT\_NE(0, bf.GetBit(bitIdx));

bf.ClrBit(bitIdx);

EXPECT\_EQ(0, bf.GetBit(bitIdx));

}

TEST(TBitField, throws\_when\_create\_bitfield\_with\_negative\_length)

{

ASSERT\_ANY\_THROW(TBitField bf(-3));

}

TEST(TBitField, throws\_when\_set\_bit\_with\_negative\_index)

{

TBitField bf(10);

ASSERT\_ANY\_THROW(bf.SetBit(-3));

}

TEST(TBitField, throws\_when\_set\_bit\_with\_too\_large\_index)

{

TBitField bf(10);

ASSERT\_ANY\_THROW(bf.SetBit(11));

}

TEST(TBitField, throws\_when\_get\_bit\_with\_negative\_index)

{

TBitField bf(10);

ASSERT\_ANY\_THROW(bf.GetBit(-3));

}

TEST(TBitField, throws\_when\_get\_bit\_with\_too\_large\_index)

{

TBitField bf(10);

ASSERT\_ANY\_THROW(bf.GetBit(11));

}

TEST(TBitField, throws\_when\_clear\_bit\_with\_negative\_index)

{

TBitField bf(10);

ASSERT\_ANY\_THROW(bf.ClrBit(-3));

}

TEST(TBitField, throws\_when\_clear\_bit\_with\_too\_large\_index)

{

TBitField bf(10);

ASSERT\_ANY\_THROW(bf.ClrBit(11));

}

TEST(TBitField, can\_assign\_bitfields\_of\_equal\_size)

{

const int size = 2;

TBitField bf1(size), bf2(size);

for (int i = 0; i < size; i++)

{

bf1.SetBit(i);

}

bf2 = bf1;

EXPECT\_NE(0, bf2.GetBit(0));

EXPECT\_NE(0, bf2.GetBit(1));

}

TEST(TBitField, assign\_operator\_changes\_bitfield\_size)

{

const int size1 = 2, size2 = 5;

TBitField bf1(size1), bf2(size2);

for (int i = 0; i < size1; i++)

{

bf1.SetBit(i);

}

bf2 = bf1;

EXPECT\_EQ(2, bf2.GetLength());

}

TEST(TBitField, can\_assign\_bitfields\_of\_non\_equal\_size)

{

const int size1 = 2, size2 = 5;

TBitField bf1(size1), bf2(size2);

for (int i = 0; i < size1; i++)

{

bf1.SetBit(i);

}

bf2 = bf1;

EXPECT\_NE(0, bf2.GetBit(0));

EXPECT\_NE(0, bf2.GetBit(1));

}

TEST(TBitField, compare\_equal\_bitfields\_of\_equal\_size)

{

const int size = 2;

TBitField bf1(size), bf2(size);

for (int i = 0; i < size; i++)

{

bf1.SetBit(i);

}

bf2 = bf1;

EXPECT\_EQ(bf1, bf2);

}

TEST(TBitField, or\_operator\_applied\_to\_bitfields\_of\_equal\_size)

{

const int size = 4;

TBitField bf1(size), bf2(size), expBf(size);

// bf1 = 0011

bf1.SetBit(2);

bf1.SetBit(3);

// bf2 = 0101

bf2.SetBit(1);

bf2.SetBit(3);

// expBf = 0111

expBf.SetBit(1);

expBf.SetBit(2);

expBf.SetBit(3);

EXPECT\_EQ(expBf, bf1 | bf2);

}

TEST(TBitField, or\_operator\_applied\_to\_bitfields\_of\_non\_equal\_size)

{

const int size1 = 4, size2 = 5;

TBitField bf1(size1), bf2(size2), expBf(size2);

// bf1 = 0011

bf1.SetBit(2);

bf1.SetBit(3);

// bf2 = 01010

bf2.SetBit(1);

bf2.SetBit(3);

// expBf = 01110

expBf.SetBit(1);

expBf.SetBit(2);

expBf.SetBit(3);

EXPECT\_EQ(expBf, bf1 | bf2);

}

TEST(TBitField, or\_operator\_applied\_to\_bitfields\_of\_non\_equal\_size\_1)

{

const int size1 = 4, size2 = 33;

TBitField bf1(size1), bf2(size2), expBf(size2);

// bf1 = 0011

bf1.SetBit(2);

bf1.SetBit(3);

// bf2 = 01010

bf2.SetBit(1);

bf2.SetBit(3);

bf2.SetBit(32);

// expBf = 01110

expBf.SetBit(1);

expBf.SetBit(2);

expBf.SetBit(3);

expBf.SetBit(32);

EXPECT\_EQ(expBf, bf1 | bf2);

}

TEST(TBitField, and\_operator\_applied\_to\_bitfields\_of\_equal\_size)

{

const int size = 4;

TBitField bf1(size), bf2(size), expBf(size);

// bf1 = 0011

bf1.SetBit(2);

bf1.SetBit(3);

// bf2 = 0101

bf2.SetBit(1);

bf2.SetBit(3);

// expBf = 0001

expBf.SetBit(3);

EXPECT\_EQ(expBf, bf1 & bf2);

}

TEST(TBitField, and\_operator\_applied\_to\_bitfields\_of\_non\_equal\_size)

{

const int size1 = 4, size2 = 5;

TBitField bf1(size1), bf2(size2), expBf(size2);

// bf1 = 0011

bf1.SetBit(2);

bf1.SetBit(3);

// bf2 = 01010

bf2.SetBit(1);

bf2.SetBit(3);

// expBf = 00010

expBf.SetBit(3);

EXPECT\_EQ(expBf, bf1 & bf2);

}

TEST(TBitField, and\_operator\_applied\_to\_bitfields\_of\_non\_equal\_size\_1)

{

const int size1 = 4, size2 = 33;

TBitField bf1(size1), bf2(size2), expBf(size2);

// bf1 = 0011

bf1.SetBit(2);

bf1.SetBit(3);

// bf2 = 01010

bf2.SetBit(1);

bf2.SetBit(3);

bf2.SetBit(32);

// expBf = 00010

expBf.SetBit(3);

EXPECT\_EQ(expBf, bf1 & bf2);

}

TEST(TBitField, can\_invert\_bitfield)

{

const int size = 2;

TBitField bf(size), negBf(size), expNegBf(size);

// bf = 01

bf.SetBit(1);

negBf = ~bf;

// expNegBf = 10

expNegBf.SetBit(0);

EXPECT\_EQ(expNegBf, negBf);

}

TEST(TBitField, can\_invert\_large\_bitfield)

{

const int size = 38;

TBitField bf(size), negBf(size), expNegBf(size);

bf.SetBit(35);

negBf = ~bf;

for (int i = 0; i < size; i++)

expNegBf.SetBit(i);

expNegBf.ClrBit(35);

EXPECT\_EQ(expNegBf, negBf);

}

TEST(TBitField, invert\_plus\_and\_operator\_on\_different\_size\_bitfield)

{

const int firstSze = 4, secondSize = 8;

TBitField firstBf(firstSze), negFirstBf(firstSze), secondBf(secondSize), testBf(secondSize);

// firstBf = 0001

firstBf.SetBit(0);

negFirstBf = ~firstBf;

// negFirstBf = 1110

// secondBf = 00011000

secondBf.SetBit(3);

secondBf.SetBit(4);

// testBf = 00001000

testBf.SetBit(3);

EXPECT\_EQ(secondBf & negFirstBf, testBf);

}

TEST(TBitField, can\_invert\_many\_random\_bits\_bitfield)

{

const int size = 38;

TBitField bf(size), negBf(size), expNegBf(size);

std::vector<int> bits;

bits.push\_back(0);

bits.push\_back(1);

bits.push\_back(14);

bits.push\_back(16);

bits.push\_back(33);

bits.push\_back(37);

for (unsigned int i = 0; i < bits.size(); i++)

bf.SetBit(bits[i]);

negBf = ~bf;

for (int i = 0; i < size; i++)

expNegBf.SetBit(i);

for (unsigned int i = 0; i < bits.size(); i++)

expNegBf.ClrBit(bits[i]);

EXPECT\_EQ(expNegBf, negBf);

}

TEST(TBitField, bitfields\_with\_different\_bits\_are\_not\_equal)

{

const int size = 4;

TBitField bf1(size), bf2(size);

bf1.SetBit(1);

bf1.SetBit(3);

bf2.SetBit(1);

bf2.SetBit(2);

EXPECT\_NE(bf1, bf2);

}

**test\_tset.cpp:**

#include "tset.h"

#include <gtest.h>

TEST(TSet, can\_get\_max\_power\_set)

{

const int size = 5;

TSet set(size);

EXPECT\_EQ(size, set.GetMaxPower());

}

TEST(TSet, can\_insert\_non\_existing\_element)

{

const int size = 5, k = 3;

TSet set(size);

set.InsElem(k);

EXPECT\_NE(set.IsMember(k), 0);

}

TEST(TSet, can\_insert\_existing\_element)

{

const int size = 5;

const int k = 3;

TSet set(size);

set.InsElem(k);

set.InsElem(k);

EXPECT\_NE(set.IsMember(k), 0);

}

TEST(TSet, can\_delete\_non\_existing\_element)

{

const int size = 5, k = 3;

TSet set(size);

set.DelElem(k);

EXPECT\_EQ(set.IsMember(k), 0);

}

TEST(TSet, can\_delete\_existing\_element)

{

const int size = 5, k = 3;

TSet set(size);

set.InsElem(k);

EXPECT\_GT(set.IsMember(k), 0);

set.DelElem(k);

EXPECT\_EQ(set.IsMember(k), 0);

}

TEST(TSet, compare\_two\_sets\_of\_non\_equal\_sizes)

{

const int size1 = 4, size2 = 6;

TSet set1(size1), set2(size2);

EXPECT\_EQ(1, set1 != set2);

}

TEST(TSet, compare\_two\_equal\_sets)

{

const int size = 4;

TSet set1(size), set2(size);

// set1 = set2 = {1, 3}

set1.InsElem(1);

set1.InsElem(3);

set2.InsElem(1);

set2.InsElem(3);

EXPECT\_EQ(set1, set2);

}

TEST(TSet, compare\_two\_non\_equal\_sets)

{

const int size = 4;

TSet set1(size), set2(size);

// set1 = {1, 3}

set1.InsElem(1);

set1.InsElem(3);

// set2 = {1, 2}

set2.InsElem(1);

set2.InsElem(2);

EXPECT\_EQ(1, set1 != set2);

}

TEST(TSet, can\_assign\_set\_of\_equal\_size)

{

const int size = 4;

TSet set1(size), set2(size);

// set1 = {1, 3}

set1.InsElem(1);

set1.InsElem(3);

set2 = set1;

EXPECT\_EQ(set1, set2);

}

TEST(TSet, can\_assign\_set\_of\_greater\_size)

{

const int size1 = 4, size2 = 6;

TSet set1(size1), set2(size2);

// set1 = {1, 3}

set1.InsElem(1);

set1.InsElem(3);

set2 = set1;

EXPECT\_EQ(set1, set2);

}

TEST(TSet, can\_assign\_set\_of\_less\_size)

{

const int size1 = 6, size2 = 4;

TSet set1(size1), set2(size2);

// set1 = {1, 3, 5}

set1.InsElem(1);

set1.InsElem(3);

set1.InsElem(5);

set2 = set1;

EXPECT\_EQ(set1, set2);

}

TEST(TSet, can\_insert\_non\_existing\_element\_using\_plus\_operator)

{

const int size = 4;

const int k = 3;

TSet set(size), updatedSet(size);

set.InsElem(0);

set.InsElem(2);

updatedSet = set + k;

EXPECT\_NE(0, updatedSet.IsMember(k));

}

TEST(TSet, throws\_when\_insert\_non\_existing\_element\_out\_of\_range\_using\_plus\_operator)

{

const int size = 4;

const int k = 6;

TSet set(size), updatedSet(size);

set.InsElem(0);

set.InsElem(2);

ASSERT\_ANY\_THROW(updatedSet = set + k);

}

TEST(TSet, can\_insert\_existing\_element\_using\_plus\_operator)

{

const int size = 4;

const int k = 3;

TSet set(size), updatedSet(size);

set.InsElem(0);

set.InsElem(k);

updatedSet = set + k;

EXPECT\_NE(0, set.IsMember(k));

}

TEST(TSet, check\_size\_of\_the\_combination\_of\_two\_sets\_of\_equal\_size)

{

const int size = 5;

TSet set1(size), set2(size), set3(size);

// set1 = {1, 2, 4}

set1.InsElem(1);

set1.InsElem(2);

set1.InsElem(4);

// set2 = {0, 1, 2}

set2.InsElem(0);

set2.InsElem(1);

set2.InsElem(2);

set3 = set1 + set2;

EXPECT\_EQ(size, set3.GetMaxPower());

}

TEST(TSet, can\_combine\_two\_sets\_of\_equal\_size)

{

const int size = 5;

TSet set1(size), set2(size), set3(size), expSet(size);

// set1 = {1, 2, 4}

set1.InsElem(1);

set1.InsElem(2);

set1.InsElem(4);

// set2 = {0, 1, 2}

set2.InsElem(0);

set2.InsElem(1);

set2.InsElem(2);

set3 = set1 + set2;

// expSet = {0, 1, 2, 4}

expSet.InsElem(0);

expSet.InsElem(1);

expSet.InsElem(2);

expSet.InsElem(4);

EXPECT\_EQ(expSet, set3);

}

TEST(TSet, check\_size\_changes\_of\_the\_combination\_of\_two\_sets\_of\_non\_equal\_size)

{

const int size1 = 5, size2 = 7;

TSet set1(size1), set2(size2), set3(size1);

// set1 = {1, 2, 4}

set1.InsElem(1);

set1.InsElem(2);

set1.InsElem(4);

// set2 = {0, 1, 2}

set2.InsElem(0);

set2.InsElem(1);

set2.InsElem(2);

set3 = set1 + set2;

EXPECT\_EQ(size2, set3.GetMaxPower());

}

TEST(TSet, can\_combine\_two\_sets\_of\_non\_equal\_size)

{

const int size1 = 5, size2 = 7;

TSet set1(size1), set2(size2), set3(size1), expSet(size2);

// set1 = {1, 2, 4}

set1.InsElem(1);

set1.InsElem(2);

set1.InsElem(4);

// set2 = {0, 1, 2, 6}

set2.InsElem(0);

set2.InsElem(1);

set2.InsElem(2);

set2.InsElem(6);

set3 = set1 + set2;

// expSet = {0, 1, 2, 4, 6}

expSet.InsElem(0);

expSet.InsElem(1);

expSet.InsElem(2);

expSet.InsElem(4);

expSet.InsElem(6);

EXPECT\_EQ(expSet, set3);

}

TEST(TSet, can\_intersect\_two\_sets\_of\_equal\_size)

{

const int size = 5;

TSet set1(size), set2(size), set3(size), expSet(size);

// set1 = {1, 2, 4}

set1.InsElem(1);

set1.InsElem(2);

set1.InsElem(4);

// set2 = {0, 1, 2}

set2.InsElem(0);

set2.InsElem(1);

set2.InsElem(2);

set3 = set1 \* set2;

// expSet = {1, 2}

expSet.InsElem(1);

expSet.InsElem(2);

EXPECT\_EQ(expSet, set3);

}

TEST(TSet, can\_intersect\_two\_sets\_of\_non\_equal\_size)

{

const int size1 = 5, size2 = 7;

TSet set1(size1), set2(size2), set3(size1), expSet(size2);

// set1 = {1, 2, 4}

set1.InsElem(1);

set1.InsElem(2);

set1.InsElem(4);

// set2 = {0, 1, 2, 4, 6}

set2.InsElem(0);

set2.InsElem(1);

set2.InsElem(2);

set2.InsElem(4);

set2.InsElem(6);

set3 = set1 \* set2;

// expSet = {1, 2, 4}

expSet.InsElem(1);

expSet.InsElem(2);

expSet.InsElem(4);

EXPECT\_EQ(expSet, set3);

}

TEST(TSet, check\_negation\_operator)

{

const int size = 4;

TSet set(size), set1(size), expSet(size);

// set1 = {1, 3}

set.InsElem(1);

set.InsElem(3);

set1 = ~set;

// expSet = {0, 2}

expSet.InsElem(0);

expSet.InsElem(2);

EXPECT\_EQ(expSet, set1);

}