Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное

учреждение высшего образования

"Национальный исследовательский Нижегородский государственный

университет им. Н. И. Лобачевского"

Институт информационных технологий, математики и механики

Отчет по лабораторной работе

СТРУКТУРА ХРАНЕНИЯ МНОЖЕСТВ

Выполнил: студент группы 381808-2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Шульман Е. А.

Подпись

Проверил:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Панов А. А.

Подпись

Нижний Новгород

2019

Содержание

[**Введение** 2](#_Toc22317839)

[**Постановка задачи** 3](#_Toc22317840)

[**Руководство пользователя** 4](#_Toc22317841)

[**Руководство программиста** 7](#_Toc22317842)

[Описание структуры программы 7](#_Toc22317843)

[Описание структуры данных 8](#_Toc22317844)

[Описание алгоритмов 9](#_Toc22317845)

[**Заключение** 12](#_Toc22317846)

[**Литература** 13](#_Toc22317847)

[**Приложения** 14](#_Toc22317848)

[Приложение 1 14](#_Toc22317849)

[Приложение 2 15](#_Toc22317850)

[Приложение 3 15](#_Toc22317851)

[Фрагменты исходного кода программы 16](#_Toc22317852)

**Введение**

В математике множества — это коллекции объектов, у которых есть что-то общее. Например, мы можем объявить множество четных положительных целых чисел:

[2, 4, 6, 8, 10, ...]

Или множество нечетных положительных целых:

[1, 3, 5, 7, 9, ...]

В этих двух множествах нет общих элементов. Давайте теперь посмотрим на множество делителей числа 100:

[1, 2, 4, 5, 10, 20, 25, 50, 100]

Теперь мы можем узнать, какие делители числа 100 — нечетные, просто глядя на множество нечетных чисел и на множество делителей и выбирая те из чисел, которые присутствуют в обоих. Мы также можем ответить на вопрос: «Какие нечетные числа не являются делителями ста?» или «Какие положительные целые, четные или нечетные, не являются делителями ста?».

На первый взгляд это кажется не очень полезным, но это искусственный пример. Предположим, что у нас есть множество всех работников предприятия и множество работников, прошедших ежемесячную проверку. Тогда мы с легкостью сможем ответить на вопрос: «Кто из работников не прошел проверку?».

Мы также можем добавить различные множества и построить более сложный запрос, например: «Кто из штатных работников отдела продаж, имеющих корпоративную кредитную карточку, не прошел обязательный курс повышения квалификации?».

Программная реализация множества может выполняться различными способами (в соответствии с требованиями конкретной задачи или с общих позиций) и обычно тесно связана с использованием битовых операций в выбранном языке программирования.

Данная работа посвящена изучению одного из возможных подходов к хранению и обработке множеств.

**Постановка задачи**

Реализовать структуру хранения данных, в виде класса TSet. Структура множество должна эффективно хранить элементы множества и поддерживать следующие операции:

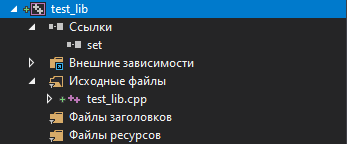
* Включение элемента в множество
* Исключение элемента из множества
* Проверка наличия элемента в множестве
* Сложение множеств
* Пересечение множеств
* Разность множеств
* Копирование множества
* Вычисление мощности множества

А также элементы множества должны быть проиндексированы (каждому элементу соответствует уникальный индекс), множество индексов элементов составляет непрерывный диапазон целых значений и размер структуры из *n* элементов должен занимать меньше чем n\*sizeof(int) байт.

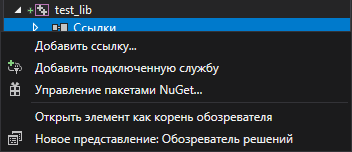
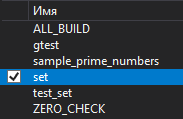
**Руководство пользователя**

По результатам работы были получена статическая библиотека set.lib для того, чтобы ее использовать необходимо:

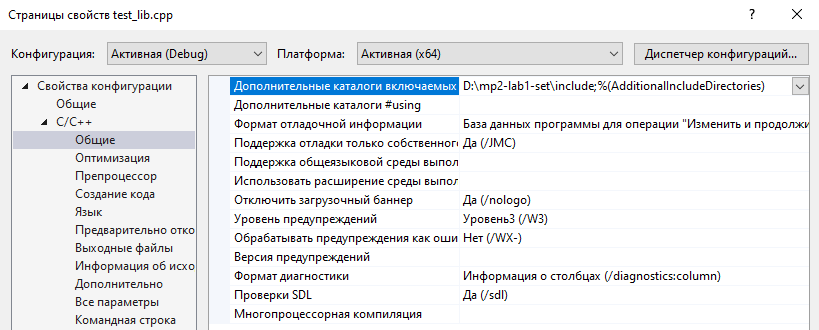
1. Создать новый проект.



1. Указываем путь до статической библиотеки и её имя. Все это сделается автоматически если добавить библиотеку в ссылку.

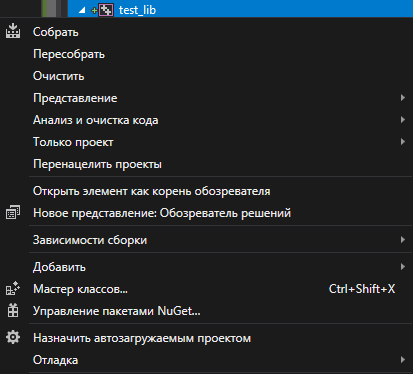
1. В настройках проекта указываем, где искать заголовочные файлы.



1. Подключаем заголовочный файл.

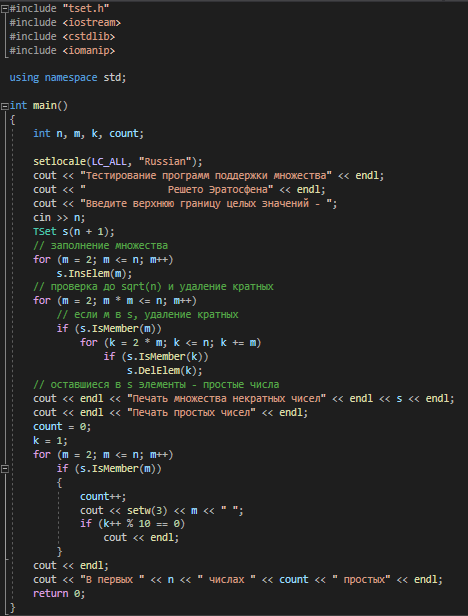
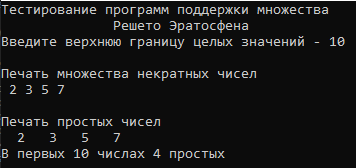


1. Назначаем наш проект автозагружаемым.



1. Теперь нашему проекту доступна библиотека set.lib.

Проверим это:

**Руководство программиста**

Описание структуры программы

Программа состоит из 4 проектов:

* gtest – библиотека google тестов;
* sample\_prime\_numbers – решето Эратосфена алгоритм поиска простых чисел, используется для тестирования множества;
* set – класс tbitfield и tset;

BitField.h, BitField.cpp – модуль с классом, реализующим операции над Битовыми полями.

Set.h, Set.cpp – модуль с классом, реализующим обработку Множеств.

* test\_set – тесты для классов tbitfield и tset;

Описание структуры данных

Класс tset реализован путем агрегации класса tbitfield.

Поля класса:

* int MaxPower - максимальная мощность множества
* TBitField BitField - битовое поле для хранения характеристического вектора

Если в tbitfield n-ый бит = 1, то элемент n – принадлежит множеству.

Класс tbitfield представлен массивом из unsigned int длины BitLen, каждый элемент битового поля соответствует определенному биту одного из элементов массива:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |

pMem[0]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

pMem[1]

Т. е. каждый бит представляет один элемент множества, и элемент находится в этом множестве, если соответствующий бит равен 1. В данном случае целые значения 1, 2, 3, 4, 22, 28 и 31 принадлежат битовому полю (остальные элементы не принадлежат).

Описание алгоритмов

Класс TBitField

Наиболее сложные методы класса

* TBitField(int len);

Конструктор создает битовое поле длины len. Если [len < 0] вызывается исключение.

Память выделяется по следующей формуле:

MemLen = len / (sizeof(int)\*8) + 1;

После выделения памяти всем элементам массива pMem присваивается 0.

* TBitField(const TBitField& bf);

Конструктор копирования создает новое битовое поле, копирую данные из существующего bf.

* int GetMemIndex(const int n) const;

Данный метод возвращает индекс элемента, содержащего n-ый бит.

* TELEM GetMemMask(const int n) const;

Данный метод создает число в битах которого единица есть только в n-ом бите.

* void SetBit(const int n);

Данный метод устанавливает n-ому биту значение 1, если n – выходит за пределы [0, BitLen) вызывается исключение.

* void ClrBit(const int n);

Данный метод устанавливает n-ому биту значение 0, если n – выходит за пределы [0, BitLen) вызывается исключение.

* void GetBit(const int n);

Данный метод возвращает значение n-ого бита, если n – выходит за пределы [0, BitLen) вызывается исключение.

TBitField& operator=(const TBitField& bf);

Перегруженный оператор = копирует данные битового поля bf в \*this. Если [this = &bf], то смысла выполнять присваивание нет - сразу возвращаем \*this. Если длины битовых полей различны – пересоздаем массив this.

* int operator==(const TBitField& bf) const;

Перегруженный оператор == сравнивает левый и правый операнды и возвращает «1», если они равны, иначе возвращает «0». Если длины битовых полей различны – сразу возвращаем «0».

* Int operator!=(const TBitField& bf) const;

Перегруженный оператор != сравнивает левый и правый операнды и возвращает «0», если они равны, иначе возвращает «1». Если длины битовых полей различны – сразу возвращаем «1».

* TBitField operator|(const TBitField& bf);

Перегруженный оператор | возвращает результат выполнения логической операции «или» между элементами pMem и bf.pMem.

* TBitField operator&(const TBitField& bf);

Перегруженный оператор & возвращает результат выполнения логической операции «и» между элементами pMem и bf.pMem.

* TBitField operator~(void);

Перегруженный оператор ~ возвращает результат выполнения логической операции «не» относительно элементов массива pMem.

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

Перегруженный оператор >> позволяет вводить битовое поле.

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

Перегруженный оператор << позволяет выводить через пробел все элементы битового поля.

Класс TSet

Наиболее сложные методы класса

* TSet& operator=(const TSet &s);

Присваивает данные множества “ s ” множеству this.

* int operator==(const TBitField &bf) const;

Сравнивает 2 множества на основе сравнения битовых полей.

* TSet operator+ (const int Elem);

Добавляет в множество this элемент Elem и возвращает новое множество.

* TSet operator- (const int Elem);

Удаляет из множества this элемент Elem и возвращает новое множество.

* TSet operator+ (const TSet &s);

Объединяет множество “ s ” c множеству this.

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

Находит общие элементы в множестве “ s ” и множестве this, возвращает новое множество только с найденными элементами.

* TSet operator~ (void);

Возвращает дополнение к множеству на основе перегрузки операции ~ в TBitField.

* friend istream &operator>>(istream &istr, TBitField &bf);

Перегруженный оператор >> позволяет вводить множество.

* friend istream &operator<<(istream &istr, TBitField &bf);

Перегруженный оператор << позволяет выводить все элементы множества.

**Заключение**

В данной лабораторной работе был разработан класс на основе которого была создана библиотека, позволяющая хранить целочисленные данные в диапазоне от 0 до MAX\_INT\*sizeof(int)\*8 – 1.

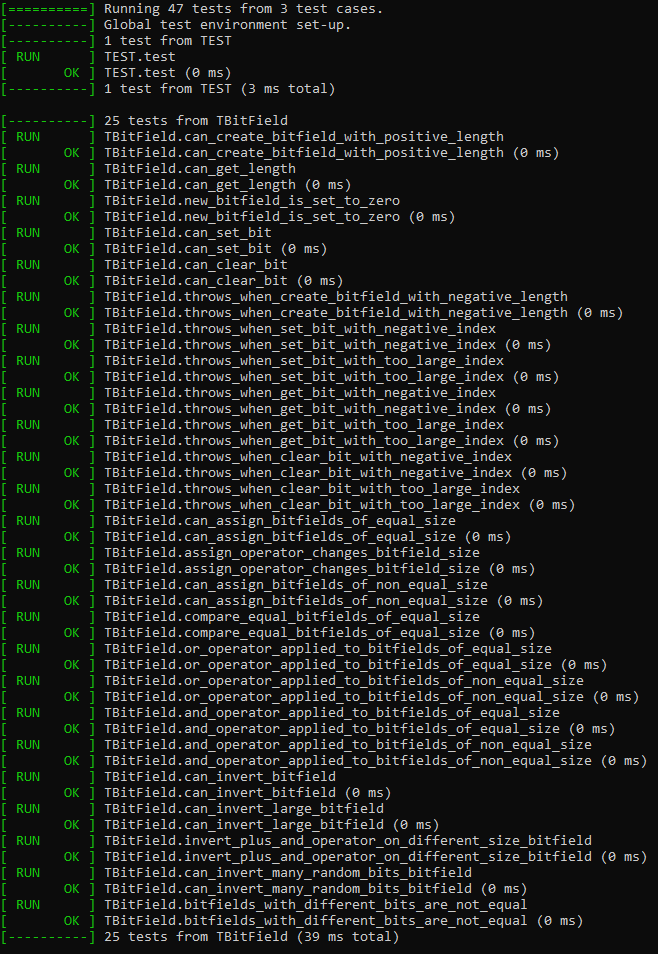
Битовые поля обеспечивают удобный доступ к отдельным битам данных. Они позволяют формировать объекты с длиной, не кратной байту.

Обращение к битовым полям требует дополнительных команд процессора для маскирования и сдвига, и потому медленнее обращения к словам/байтам. Поэтому битовые поля применяются для максимально полной упаковки информации в местах, где не важна скорость доступа к информации.

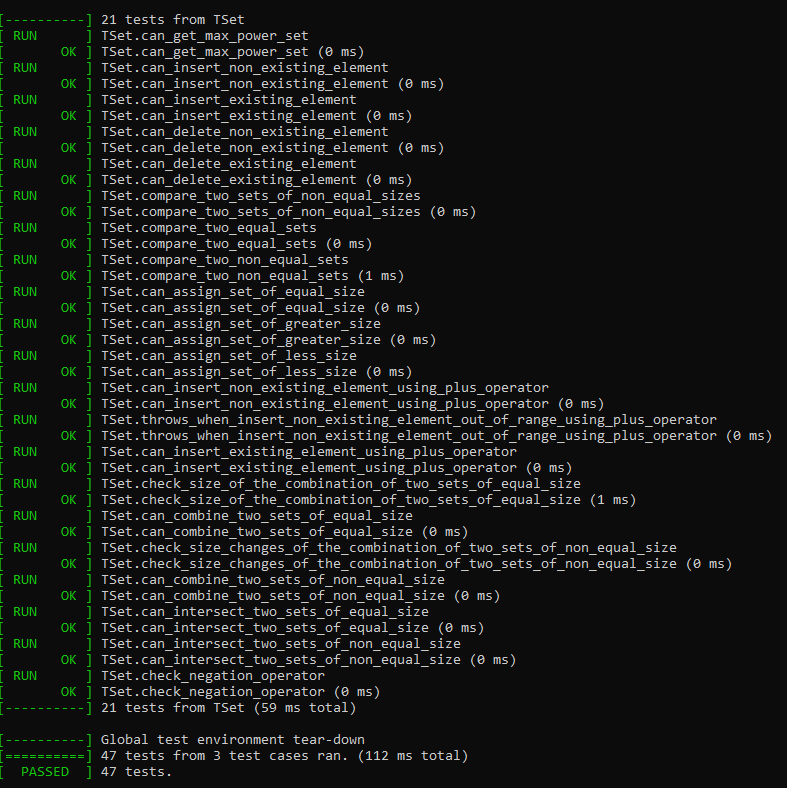
**Литература**

**Приложения**

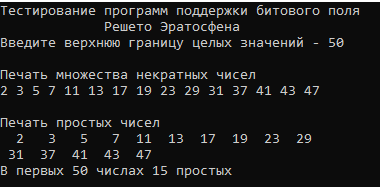
Приложение 1



Приложение 2



Приложение 3



## Фрагменты исходного кода программы

TBitField::TBitField(int len)

{

if (len < 0)

throw "Error";

BitLen = len;

MemLen = BitLen / (sizeof(int)\*8) + 1;

pMem = new TELEM[MemLen];

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

pMem[i] = 0;

}

TBitField::TBitField(const TBitField& bf)

{

BitLen = bf.BitLen;

MemLen = bf.MemLen;

pMem = new TELEM[MemLen];

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

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

}

TBitField::~TBitField()

{

if (MemLen)

delete[] pMem;

BitLen = 0;

MemLen = 0;

}

void TBitField::SetBit(const int n)

{

if ((n < 0) || (n >= BitLen))

throw "Error";

pMem[GetMemIndex(n)] |= GetMemMask(n);

}

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

{

if (this != &bf)

{

BitLen = bf.BitLen;

if (MemLen != bf.MemLen)

{

MemLen = bf.MemLen;

if (pMem != NULL)

delete[]pMem;

pMem = new TELEM[MemLen];

}

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

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

}

return \*this;

}

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

{

if (BitLen != bf.BitLen)

return 0;

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

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

return 0;

return 1;

}

TBitField TBitField::operator|(const TBitField& bf)

{

int MaxBitLen = BitLen > bf.BitLen ? BitLen : bf.BitLen;

TBitField tmp(MaxBitLen);

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

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

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

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

return tmp;

}

TBitField TBitField::operator&(const TBitField& bf)

{

int MaxBitLen = BitLen > bf.BitLen ? BitLen : bf.BitLen;

TBitField tmp(MaxBitLen);

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

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

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

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

return tmp;

}

TBitField TBitField::operator~(void)

{

TBitField tmp(BitLen);

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

if (!GetBit(i)) tmp.SetBit(i);

return tmp;

}

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

{

int i = 0;

char ch;

do {

istr >> ch;

} while (ch != ' ');

while (1) {

istr >> ch;

if (ch == '0')

bf.ClrBit(i++);

else if (ch == '1')

bf.SetBit(i++); else break;

}

return istr;

}

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

{

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

if (bf.GetBit(i)) ostr << i << " ";

return ostr;

}

TSet::TSet(int mp) : BitField(mp), MaxPower(mp)

{

}

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

{

MaxPower = s.MaxPower;

}

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

{

return BitField == s.BitField;

}

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

{

return BitField != s.BitField;

}

TSet TSet::operator+(const int Elem)

{

TBitField tmp(MaxPower);

tmp = BitField;

tmp.SetBit(Elem);

TSet temp(tmp);

return temp;

}

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

{

TSet tmp(BitField & s.BitField);

return tmp;

}

TSet TSet::operator~(void)

{

TSet tmp(~BitField);

return tmp;

}

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

{

int i = 0;

istr >> i;

while ((i >= 0) && (i<s.MaxPower))

{

s.InsElem(i);

istr >> i;

}

return istr;

}

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

{

for (int i = 0; i<s.GetMaxPower(); i++)

if (s.IsMember(i)) ostr << i;

return ostr;

}