МИНИСТЕРСТВО НАУКИ И ВЫСШЕГООБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

ЛАБОРАТОРНАЯ РАБОТА

на тему:

**«БИТОВЫЕ ПОЛЯ И МНОЖЕСТВА»**

**Выполнил(а):** студент(ка) группы

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Шпынов Н.А. /

Подпись

**Проверил:** к.т.н., доцент каф. ВВиСП

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Кустикова В.Д. /

Подпись

Нижний Новгород  
2023

# Оглавление

**[Введение 1](#_Toc25862)**

**[2 Постановка задачи 3](#_Toc5658)**

**[3 Руководство пользователя 4](#_Toc8539)**

**[4 Руководство программиста 9](#_Toc16586)**

**[Заключение 14](#_Toc25428)**

**[Литература 15](#_Toc800)**

[Приложение А. Реализация класса TBitField 16](#_Toc31696)

[Приложение Б. Реализация класса TSet 16](#_Toc3076)

# Введе**н**ие

## Актуальность

Битовые поля - это структуры данных, которые позволяют компактно хранить и работать с набором булевских значений. Каждое поле представляет собой группу битов в памяти компьютера, каждому из которых присваивается определённая позиция.

Использование битовых полей даёт большую гибкость, обеспечивает оптимизацию памяти и удобство работы с различными флагами и настройками в программе. Также они позволяют управлять информацией и хранить её в компактном формате, что является ценным ресурсом во многих областях разработки программного обеспечения.

## Значимость

Битовые поля являются важным и значимым инструментом в программировании

1. Оптимизация использования памяти

 Использование битовых полей позволяет сократить расход памяти. С их помощью можно упаковывать несколько различных флагов или настроек в одно целочисленное поле, вместо того, чтобы использовать отдельные переменные или массивы.

1. Удобство чтения и записи значений

Битовые поля позволяют мы легко читать и записывать значения в отдельные биты поля. Например, если объекту присвоить своё битовое поле, мы можем использовать каждый бит для отображения различных его свойств или состояний. Это делает код более читаемым и удобным для работы.

1. Кросс-платформенность

Битовые поля позволяют достичь определённой степени аппаратной независимости. Программа, использующая битовые поля, может работать на разных архитектурах и платформах без изменений в логике.

## Применимость

Использование битовых полей даёт множество преимуществ в реализации многих структур и программ. Примеры использования битовых полей:

1. Флаги состояний

битовые поля можно использовать для представления и управления флагами состояния объектов или системы. Каждому состоянию "включено/выключено" или "открыто/закрыто" ставится в соответствие свой бит, значение которого и будет указывать на текущее положение.

1. Настройки конфигурации

Битовые поля позволяют упаковать различные настройки или флаги конфигурации в одно поле, что делает управление, использование и применение настроек какого то объекта более удобными.

1. Сериализация данных

 Битовые поля могут быть использованы для сериализации или десериализации данных (*перевод структуры данных в битовую последовательность и наоборот*), когда требуется компактно передать или сохранить данные.

1. *Реализация множеств*

Битовые поля являются одним из простейших способов реализации алгебры логики в программе. Каждому элементу в универсе ставится в соответствие свой бит, обозначающий его наличие или отсутствие в данном множестве

# Постановка задачи

Цель – реализовать представление множеств на языке C++ при помощи битовых полей

Задачи:

1. Разработка представления битовых полей

Для существования системы множеств на основе битовых полей, следует создать реализацию их самих.

1. Определить способ хранения битовых полей
2. Создать структуру данных с битовыми полями
3. Реализовать необходимые для работы поля и методы
4. Разработка структуры данных для представления множества

Определить, каким образом будут храниться элементы множества с использованием битовых полей. Потребуется создать специальную структуру данных, в которой каждый бит будет соответствовать наличию или отсутствию определённого элемента.

1. Определение операций над множествами

Разработать алгоритмы и функции для выполнения операций над множествами, таких как объединение, пересечение, разность, проверка на принадлежность элемента, добавление/удаление элементов из множества и другие.

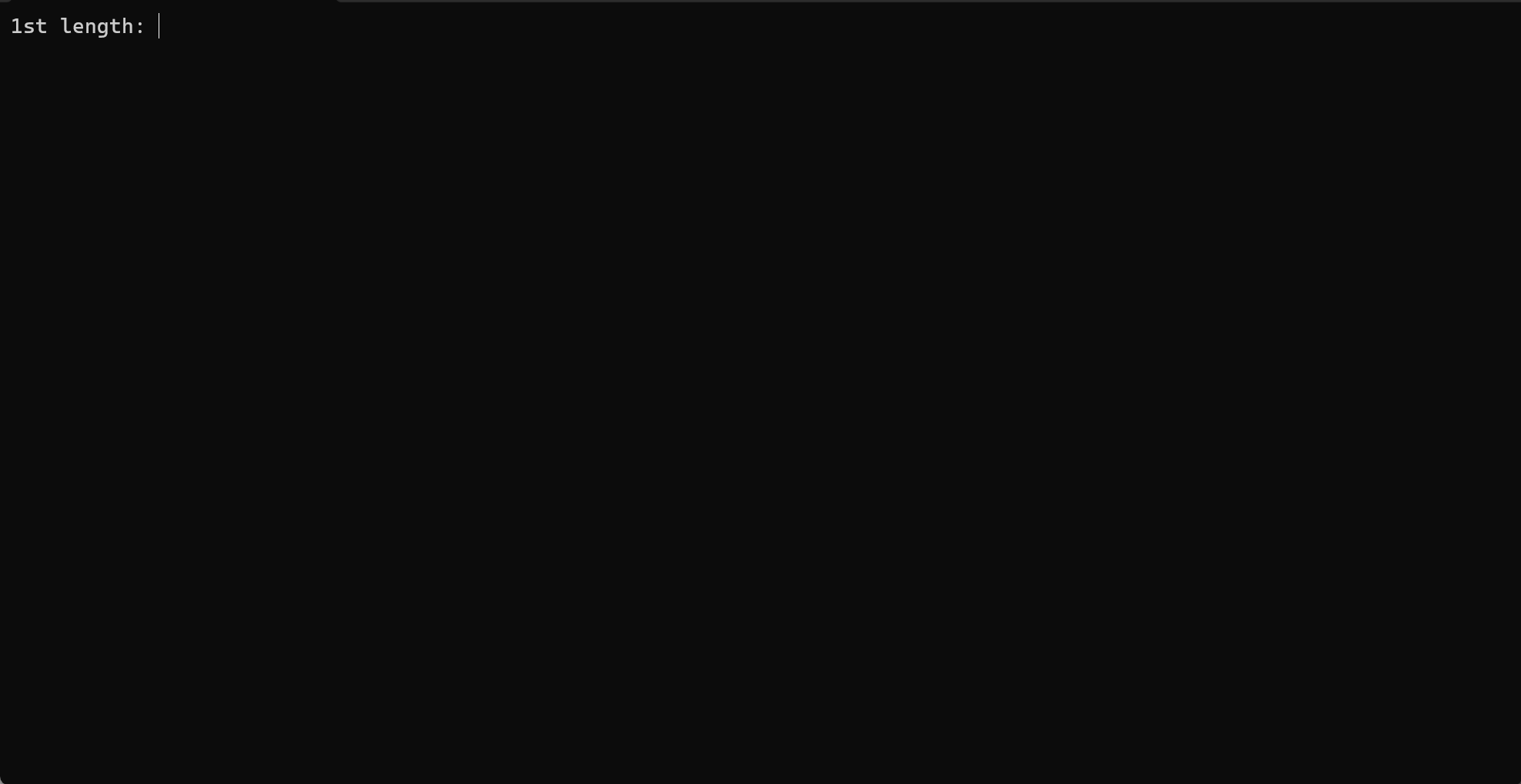
1. Тестирование и отладка.

Как и при любой разработке программного обеспечения, в конце необходимо провести тестирование и отладку кода. Следует обнаружить и исправить возможные ошибки и проблемы, а также убедиться в правильности функционирования.

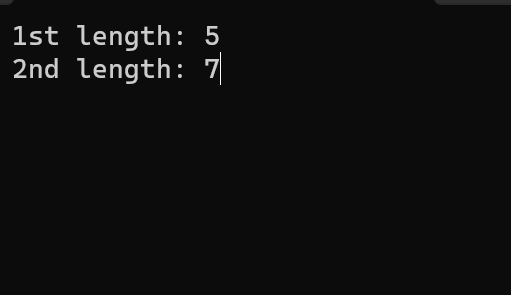
# Руководство пользователя

## Приложение для демонстрации работы битовых полей

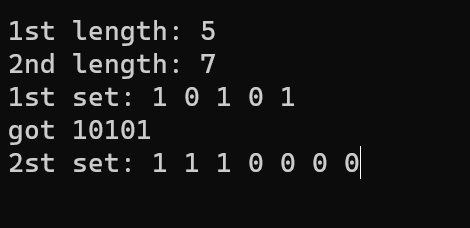
1. Запустить sample\_tbitfield.exe. В результате появится следующее окно (Рис. 1):



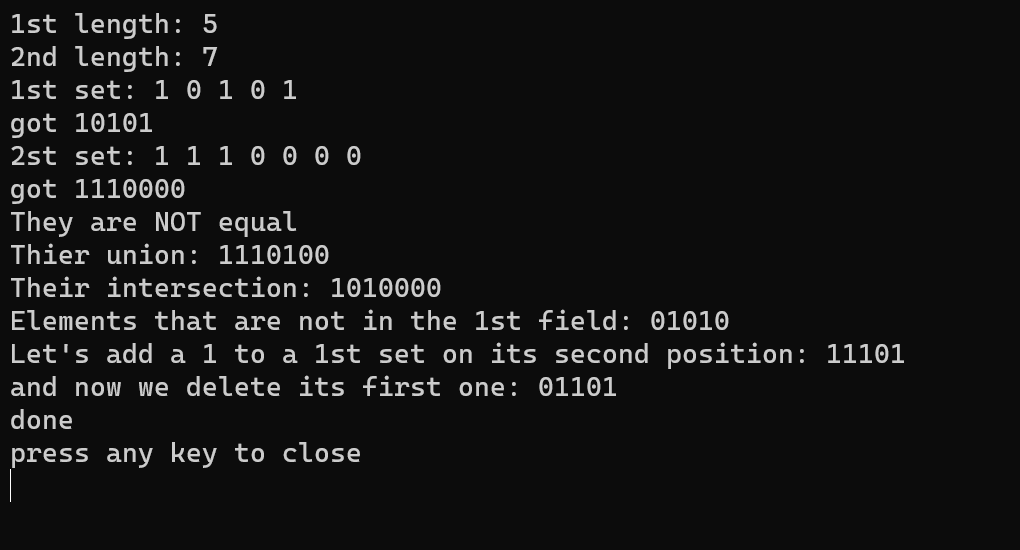
1. Основное окно приложения
2. Затем программа попросит ввести длины (количество элементов) двух полей (Рис. 2):



1. ввод длин
2. Далее потребуется ввести поля соответственно следующим образом. Сначала вводятся все элементы первого поля по очереди через пробел, 0 или 1. Затем точно также и у второго поля (Рис. 3):



1. результат ввода полей
2. Программа покажет примеры результатов операций над полями. После чего попросит ввести любое значение для выхода (Рис. 4).



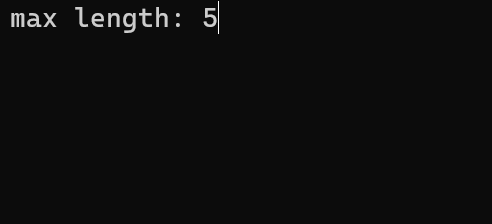
1. итог

## Приложение для демонстрации работы множеств

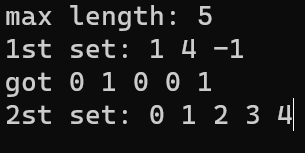
1. Запустить sample\_tset.exe. В результате появится следующее окно (Рис. 1):



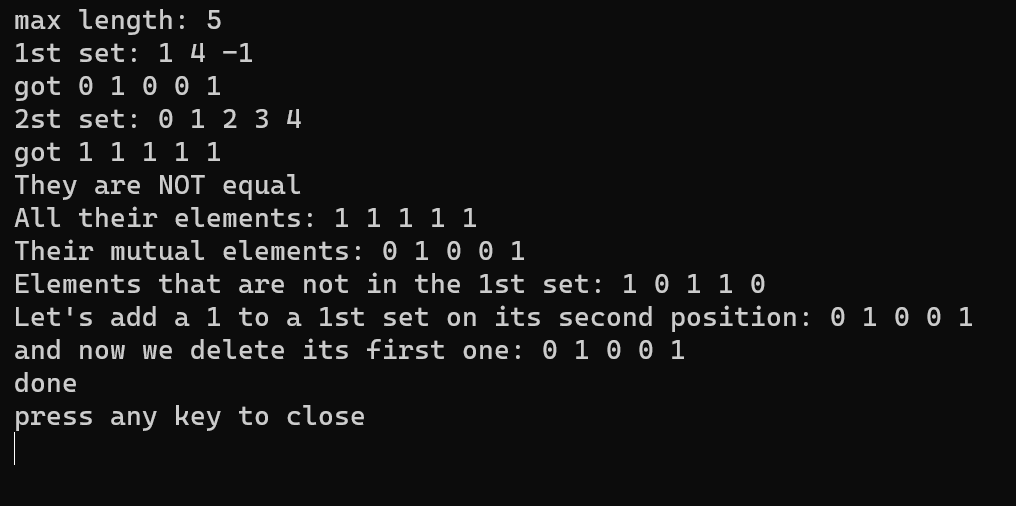
1. Основное окно приложения
2. Программа запросит длину универса (максимальное количество элементов) (Рис. 2):



1. Ввод длины
2. Далее потребуется ввести множества соответственно следующим образом. Сначала вводятся все элементы первого множества через пробел, вводятся элементы в любом порядке по их номеру до того, пока множество не станет максимальным или не будет введено орицательное число, обозначающее, что ввод множества закончен. Затем точно также и у второго множества (Рис. 3):



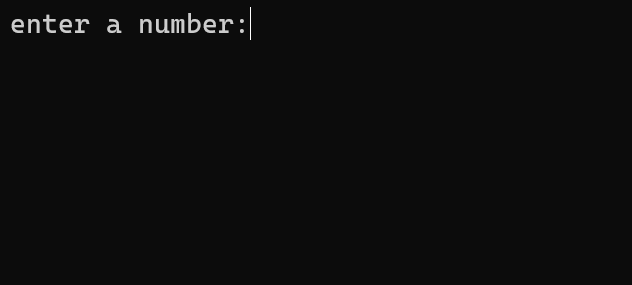
1. результат ввода множеств
2. Программа покажет примеры результатов операций над множествами. После чего попросит ввести любое значение для выхода (Рис. 4).



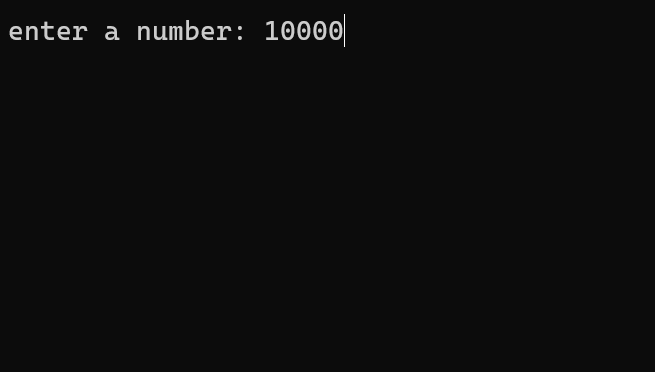
1. Итог

## Приложение «решето Эратосфена»

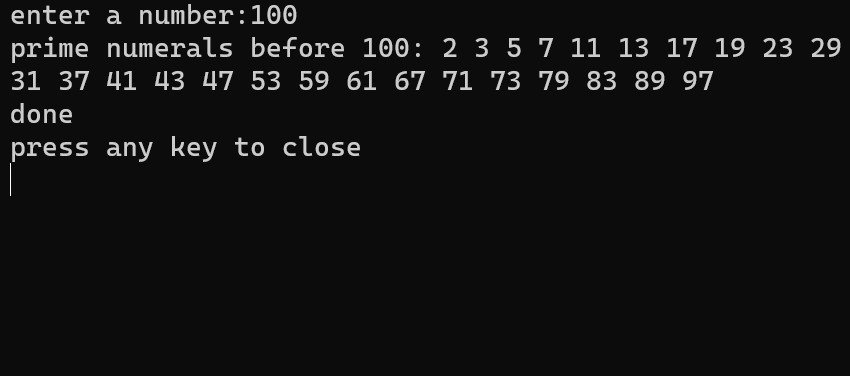
1. Запустить sample\_primenumbers.exe. В результате появится следующее окно (Рис. 1):



1. Основное окно приложения
2. Программа запросит число, в рамках которого будет искать все простые числа, нужно его ввести (Рис. 2):



1. Ввод числа
2. Программа выдаст все простые числа, меньше введённого (Рис. 3):



1. Результат

# Руководство программиста

## Использованнные алгоритмы

### Битовые поля

Битовое поле - это класс, полями которого являются его длина в битах, его длина во вспомогательных единицах, массив этих единиц.

Операции над полями реализованы в качестве методов класса полей

### Множества

Множество - это класс, полями которого является длина его универса и битовое поле, хранящее информацию о включении элементов в это множество массив этих единиц.

Операции над полями реализованы в виде методов класса множество

### Алгоритм «решето Эратосфена»

1. Получить число n, до которого требуется искать простые числа
2. Создать вспомогательное битовое поле длины n+1, заполненное единицами
3. Пройти по полю до максимального числа, квадрат которого меньше n
4. Для каждого из полученных чисел пройди по массиву от удвоенного номера этого числа до конца с шагом в это число и удалить все полученные элементы
5. Вывести результат

## Описание классов

### Класс TbitField

Объявление класса:

class TBitField

{

private:

int BitLen;

TELEM \*pMem;

int MemLen;

int GetMemIndex(const int n) const;

TELEM GetMemMask (const int n) const;

public:

TBitField(int len);

TBitField(const TBitField &bf);

~TBitField();

int GetLength(void) const;

void SetBit(const int n);

void ClrBit(const int n);

int GetBit(const int n) const;

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

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

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

TBitField operator|(const TBitField &bf);

TBitField operator&(const TBitField &bf);

TBitField operator~(void);

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

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

};

Поля:

BitLen – длина битового поля.

pMem – массив вспомогательных единиц

MemLen – количество вспомогательных единц

Конструкторы:

TBitField(int len) - конструктор с параметром

TBitField(const TBitField &bf) - конструктор копирования

Деструктор:

~TBitField();

Методы:

int GetMemIndex(const int n) const;

Назначение: получение индекса элемента, где хранится бит.

Входные данные:

n – номер бита.

Выходные данные:

индекс элемента, где хранится бит с номером n.

TELEM GetMemMask (const int n) const;

Назначение: получение маски элемента, где хранится бит.

Входные данные:

n – номер бита.

Выходные данные:

Число, являющееся маской бита с номером n

int GetLength(void) const;

Назначение: получение длины поля вне метода.

Входные данные:

Нет

Выходные данные:

Длина поля

void SetBit(const int n);

Назначение: установка бита с индексом n единицей

Входные данные:

n - номер

Выходные данные:

нет

void ClrBit(const int n);

Назначение: установка бита с индексом n нулем

Входные данные:

n - номер

Выходные данные:

нет

int GetBit(const int n) const;

Назначение: получение значения бита под индексом n

Входные данные:

n - номер

Выходные данные:

Значение бита под индексом n

Операторы:

int operator==(const TBitField &bf) const - сравнение

int operator!=(const TBitField &bf) const - сравнение на неравенство

сonst TBitField& operator=(const TBitField &bf) - присваивание

TBitField operator|(const TBitField &bf) - операция "или"

TBitField operator&(const TBitField &bf) - операция "и"

TBitField operator~(void); - отрицание

friend istream &operator>>(istream &istr, TBitField &bf) - ввод

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

### Класс TSet

class TSet

{

private:

int MaxPower;

TBitField BitField;

public:

TSet(int mp);

TSet(const TSet &s);

TSet(const TBitField &bf);

operator TBitField();

int GetMaxPower(void) const;

void InsElem(const int Elem);

void DelElem(const int Elem);

int IsMember(const int Elem) const;

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

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

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

TSet operator+ (const int Elem);

TSet operator- (const int Elem);

TSet operator+ (const TSet &s);

TSet operator\* (const TSet &s);

TSet operator~ (void);

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

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

};

Поля:

int MaxPower;  - мощность универса для этого множества

TBitField BitField; - поле, характеризующее множество

Конструкторы:

TSet(int mp);

TSet(const TSet &s);

TSet(const TBitField &bf);

Методы:

**int GetMaxPower() const;**

Назначение: получение максимальной мощности множества, то есть размера универса

Входные данные:

нет

Выходные данные:

максимальная мощность множества.

void InsElem(const int Elem);

Назначение: включение элемента в множество.

Входные данные:

Elem – номер включаемого элемента.

Выходные данные:

нет

void DelElem(const int Elem);

Назначение: удаление элемента из множества.

Входные данные:

Elem – номер удаляемого элемента.

Выходные данные:

нет

int IsMember(const int Elem) const;

Назначение: проверка элемента на наличие в множестве.

Входные данные:

Elem – номер проверяемого элемента.

Выходные данные:

1 – элемент присутствует в множестве

0 – элемента в множестве нет

Операторы:

int operator== (const TSet &s) const - сравнение

int operator!= (const TSet &s) const - сранение (неравенство)

const TSet& operator=(const TSet &s) - присваивание

TSet operator+ (const int Elem) - включение элемента в множество

TSet operator- (const int Elem) - исключение элемента из множества

TSet operator+ (const TSet &s) - сумма множеств

TSet operator\* (const TSet &s) - пересечение множеств

TSet operator~ (void) - отрицание множжества

friend istream &operator>>(istream &istr, TSet &bf) - вывод множества

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

# Заключение

# Реализация множеств при помощи битовых полей оказалась эффективным подходом, позволяющим компактно хранить и оперировать наборами элементов. Этот метод позволяет сократить затраты на память, обеспечивает быстрые операции с множествами и может быть применён в решении различных задач, связанных с манипуляцией данными и их анализом.

# Литература

1. <https://ru.wikipedia.org/>
2. <https://prog-cpp.ru/>
3. <https://en.cppreference.com/w/>
4. <https://github.com/>

# Приложения

## Приложение А. Реализация класса TBitField

TBitField::TBitField(int len)

{

    if (len < 0)

        throw "negative length caught";

    BitLen = len;

    MemLen = BitLen / bitsInElem + 1;

    pMem = new TELEM[MemLen];

    memset(pMem, 0, Amount);

}

TBitField::TBitField(const TBitField &bf)

{

    BitLen = bf.BitLen;

    MemLen = bf.MemLen;

    pMem = new TELEM[MemLen];

    memcpy(pMem, bf.pMem, Amount);

}

TBitField::~TBitField()

{

    delete[] pMem;

}

int TBitField::GetMemIndex(const int n) const

{

    if (n < BitLen)

        return n >> shiftSize;

    else

        throw "no such bit";

}

TELEM TBitField::GetMemMask(const int n) const

{

   return 1 << (n & (bitsInElem - 1));

}

int TBitField::GetLength(void) const

{

    return BitLen;

}

void TBitField::SetBit(const int n)

{

     if ((n >= BitLen) || (n < 0))

        throw "out of range";

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

}

void TBitField::ClrBit(const int n)

{

    if ((n >= BitLen) || (n < 0))

        throw "out of range";

    pMem[GetMemIndex(n)] = pMem[GetMemIndex(n)] & (~GetMemMask(n));

}

int TBitField::GetBit(const int n) const

{

    if (n >= BitLen || n < 0)

        throw "out of range";

    if ((pMem[GetMemIndex(n)] & GetMemMask(n)) == 0)

        return 0 ;

    else

        return 1;

}

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

{

    if (\*this == bf)

        return \*this;

    if (BitLen != bf.BitLen)

    {

        delete[] pMem;

        BitLen = bf.BitLen;

        MemLen = bf.MemLen;

        pMem = new TELEM[MemLen];

    }

    memcpy(pMem, bf.pMem, Amount);

    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;

}

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

{

  return (!(\*this == bf));

}

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

{

    int maxlen = max(BitLen, bf.BitLen);

    TBitField Ftmp(maxlen);

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

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

    return Ftmp;

}

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

{

    int maxlen = max(BitLen, bf.BitLen);

    TBitField Ftmp(maxlen);

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

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

    return Ftmp;

}

TBitField TBitField::operator~(void)

{

    TBitField Ftmp(BitLen);

    for (int i = 0; i < BitLen; i++)

        if (GetBit(i))

            Ftmp.ClrBit(i);

        else Ftmp.SetBit(i);

    return Ftmp;

}

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

{

    int bt;

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

            istr >> bt;

            if (!bt) {

                bf.ClrBit(i);

            }

            else if (bt == 1) {

                bf.SetBit(i);

            }

            else{

                cout << "not 0 or 1, setting to 1";

                bf.SetBit(i);

                throw "not 0 or 1";

            }

        }

        return istr;

}

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

{

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

        if (bf.GetBit(i)) ostr << "1";

        else ostr << "0";

    }

    return ostr;

}

## Приложение Б. Реализация класса TSet

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

TSet::TSet(const TSet &s) : BitField(s.BitField), MaxPower(s.GetMaxPower()) {}

TSet::TSet(const TBitField &bf) : BitField(bf), MaxPower(bf.GetLength()){}

int TSet::GetMaxPower(void) const

{

    return MaxPower;

}

int TSet::IsMember(const int Elem) const

{

    return BitField.GetBit(Elem);

}

void TSet::InsElem(const int Elem)

{

    BitField.SetBit(Elem);

}

void TSet::DelElem(const int Elem)

{

    BitField.ClrBit(Elem);

}

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

{

    if (\*this == s)

        return \*this;

    BitField = s.BitField;

    MaxPower = s.MaxPower;

    return \*this;

}

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

{

    if (MaxPower != s.MaxPower)

        return 0;

    return (BitField == s.BitField);

}

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

    return !(\*this == s);

}

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

{

    TSet TStmp(max(MaxPower, s.GetMaxPower()));

    TStmp.BitField = BitField | s.BitField;

    return TStmp;

}

TSet TSet::operator+(const int Elem)

{

    TSet TStmp(\*this);

    TStmp.InsElem(Elem);

    return TStmp;

}

TSet TSet::operator-(const int Elem)

{

    TSet TStmp(\*this);

    TStmp.DelElem(Elem);

    return TStmp;

}

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

{

    TSet TStmp(max(MaxPower, s.GetMaxPower()));

    TStmp.BitField = BitField & s.BitField;

    return TStmp;

}

TSet TSet::operator~(void)

{

    TSet TStmp(MaxPower);

    TStmp.BitField = ~BitField;

    return TStmp;

}

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

{

   int elem;

   int n;

   cout << "set length: ";

   istr >> n;

   for (int i = 0; i < n; i++) {

       istr >> elem;

       s.InsElem(elem);

   }

    return istr;

}

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

{

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

        if (s.IsMember(i))

            ostr << i << " ";

    return ostr;

}