Министерство науки и высшего образования 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 множества»**

**Выполнил**:

студент группы 381908-2

Серебренников М.М.

**Проверил**:

доцент кафедры программной инженерии к.т.н., доцент с.н.с,

Карпенко С.Н.

Нижний Новгород

2020

Содержание

[Введение 3](#_Toc23876641)

[1. Постановка задачи 4](#_Toc23876642)

[2. Руководство пользователя 5](#_Toc23876643)

[3. Руководство программиста 6](#_Toc23876644)

[3.1. Описание структуры программы 6](#_Toc23876645)

[3.2. Описание алгоритмов 7](#_Toc23876646)

[Заключение 11](#_Toc23876647)

[Литература 12](#_Toc23876648)

[Приложение 13](#_Toc23876649)

# Введение

Множество является одним из ключевых понятий математики; это математический объект, являющийся набором каких-либо объектов, которые называются элементами этого множества и обладают общим для всех их характеристическим свойством. Это понятие фундаментально не только математики, но и всего окружающего мира. Для упрощения работы с множествами и их объектами была создана эта программа.

# Постановка задачи

Разработать структуру данных **Множество** и использовать ее для эффективного хранения множеств и выполнения основных операций над множествами.

Выполнение работы предполагает решение следующих задач:

1. Разработка интерфейса класса BitField.
2. Реализация методов класса BitField.
3. Разработка интерфейса класса Set.
4. Реализация методов класса Set.

# Руководство пользователя

Вначале пользователю предлагается ввести границу значений (т.е. до какого числа будет реализован поиск простых чисел).

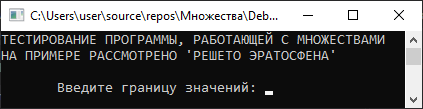


Рис. 1. Начало работы

При неправильном вводе программа просит заново ввести значение.

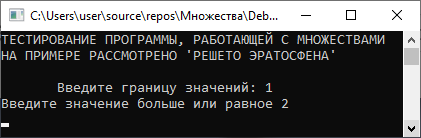


Рис. 2. Пример неправильного ввода

После ввода корректного арифметического выражения программа выводит на экран список простых чисел и их количество в заданном диапазоне.

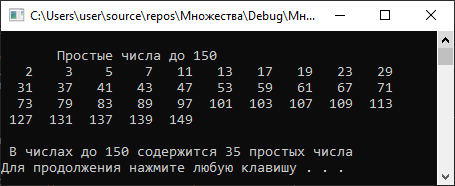


Рис. 3. Ввод простых чисел

Затем предлагается выйти из программы или ввести новое значение.

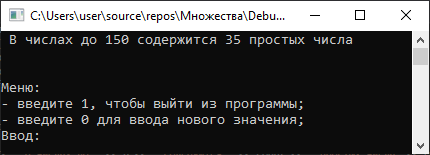


Рис. 4. Выход из программы

## Руководство программиста

## Описание структуры программы

Программа написана на языке программирования C++. Состоит из файлов BitField.h, BitField.cpp, Set.h, Set.cpp, main.cpp.

В программе реализованы следующие классы:

1. **Класс «BitField»**

Описание: реализует структуру хранения битовых полей.

|  |
| --- |
| **BitField** |
| - **BitLen**:int  - \***pMem**:UI (=unsigned int)  - **MenLen**:int |
| “**constructor**”  BitField(int len)  BitField(const BitField &bf)  “**destructor**”  ~BitField()  “**interface**”  GetMemIndex (const int n) const: int  GetMemMask (const int n) const:UI(=unsigned int)  GetLength (void) const: int  SetBit(const int n): void  ClrBit(const int n): void  GetBit(const int n) const: int  operator== (const BitField &bf): int  &operator=(const BitField &bf): BitField  &operator|(const BitField &bf): BitField  &operator&(const BitField &bf): BitField  &operator~(void): BitField |

В следующей таблице указано применение данных и методов класса.

|  |  |  |
| --- | --- | --- |
| **№** | **Данные** | **Хранит** |
| 1 | BitLen | длина битового поля – макс. кол-во битов |
| 2 | \*pMem | память для представления битового поля (указатель на начало) |
| 3 | MemLen | кол-во элементов битового поля |
|  | | |
| **№** | **Методы** | **Применение** |
| 1 | GetLength | возвращает длину строки |
| 2 | SetBit | устанавливает бит из позиции n |
| 3 | ClrBit | удаляет бит из позиции n |
| 4 | GetBit | получает значение бита из позиции n |
| 5 | operator== | оператор сравнения |
| 6 | &operator= | оператор присваивания |
| 7 | &operator| | оператор пересечение множеств (или) |
| 8 | &operator& | оператор объединения множеств (и) |
| 9 | &operator~ | оператор логического отрицания (не) |

1. **Класс «Set»**

Описание: реализует обработку множеств.

|  |
| --- |
| **Postfix** |
| - **MaxPower**:int  - **BF**:BitField |
| “**constructor**”  Set(const Set&s)  Set(int mp)  Set(const BitField &bf)  “**interface**”  GetMaxPower(void) const: int  InsElem(const int n): void  DelElem(const int n): void  IsMember(const int n) const: int    operator== (const Set &s): int  &operator=(const Set &s): Set  operator+ (const int n): Set  operator- (const int n): Set  operator\* (const Set &s): Set  operator~ (void): Set  operator+ (const Set &s): Set |

В следующей таблице указано применение данных и методов класса.

|  |  |  |
| --- | --- | --- |
| **№** | **Данные** | **Хранит** |
| 1 | MaxPower | хранит максимальную мощность множества |
| 2 | BF | служит для работы методов, где необходимо обратиться к методам класса BitField |
|  | | |
| **№** | **Методы** | **Применение** |
| 1 | GetMaxPower | возвращает максимальное количество элементов |
| 2 | InsElem | добавляет элемент в множество |
| 3 | DelElem | удаляет элемент из множества |
| 4 | IsMember | проверяет элемент на принадлежность множеству |
| 5 | operator== | оператор сравнения |
| 6 | &operator= | оператор присваивания |
| 7 | operator+ | объединение множеств |
| 8 | operator\* | пересечение множеств |
| 9 | operator~ | дополнение множеств |

## Описание алгоритмов

В классе «BitField» реализованы методы для работы с битовыми полями.

1) Метод «GetLength»

Данный метод возвращает значение длинны битового поля.

Ниже приведен псевдокод алгоритма:

**Функция GetLength (void) const**

{

**вернуть** BitLen;

}

2) Метод «SetBit»

Данный метод позволяет установить бит (т.е. 1) в позицию n.

Ниже приведен псевдокод алгоритма:

**Функция SetBit (const int n)**

{

**если**  ((n > -1) **и** (n < BitLen)) **тогда**

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

}

3) Метод «ClrBit »

Данный метод позволяет очистить бит в позиции n.

Ниже приведен псевдокод алгоритма:

**Функция ClrBit (const int n)**

{

**если**  ((n > -1) **и** (n < BitLen)) **тогда**

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

}

4) Метод «GetBit»

Данный метод позволяет получить значение бита в позиции n.

Ниже приведен псевдокод алгоритма:

**Функция GetBit (const int n) const**

{

**если** ((n > -1) **и** (n < BitLen)) **тогда**

**вернуть** pMem[GetMemIndex(n)] & GetMemMask(n);

**вернуть** 0;

}

5) Метод «operator=»

Данный метод перегружает операцию присваивания (знак «=»).

Ниже приведен псевдокод алгоритма:

**Функция operator=(const BitField &bf)**

{

BitLen = bf.BitLen;

**если** (MemLen **неравно** bf.MemLen) **тогда**  {

MemLen = bf.MemLen;

**если** (pMem **неравно** NULL) **тогда**

**удалить** pMem;

pMem = **создать память** UI[MemLen];

}

**если** (pMem **неравно** NULL) **тогда**

**цикл от** int i = 0; **до** i < MemLen; **с шагом** i++)

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

**вернуть неявный указатель на объект**;

}

6) Метод «operator==»

Данный метод перегружает операцию сравнения и возвращает 1 – истина или 0 - ложь (знак «==»).

Ниже приведен псевдокод алгоритма:

**Функция operator==(const BitField &bf)**

{

int res = 1;

**если** (BitLen неравно bf.BitLen) **тогда**

res = 0;

**иначе**

**цикл от** int i=0; **до** i<MemLen; **с шагом** i++

**если**  (pMem[i] **неравно** bf.pMem[i]) **тогда**  {

res = 0;

**прекрати**;

}

**вернуть** res;

}

7) Метод «operator|»

Данный метод перегружает операцию «ИЛИ» (знак «|»).

Ниже приведен псевдокод алгоритма:

**Функция operator|(const BitField &bf)**

{

int i, len = BitLen;

**если** (bf.BitLen > len) **тогда**

len = bf.BitLen;

BitField temp(len);

**цикл от** i = 0; **до** i < MemLen; **с шагом** i++)

temp.pMem[i] = pMem[i];

**цикл от** i = 0; **до** i < bf.MemLen; **с шагом** i++)

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

**вернуть** temp;

}

8) Метод «operator&»

Данный метод перегружает операцию «И» (знак «&»).

Ниже приведен псевдокод алгоритма:

**Функция operator& (const BitField &bf)**

{

int i, len = BitLen;

**если** (bf.BitLen > len) **тогда**

len = bf.BitLen;

BitField temp(len);

**цикл от** int i = 0; **до** i < MemLen; **с шагом** i++)

temp.pMem[i] = pMem[i];

**цикл от** i = 0; **до** i = bf.MemLen; **с шагом** i++)

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

**вернуть** temp;

}

9) Метод «operator~»

Данный метод перегружает операцию «отрицание» (знак «~»).

Ниже приведен псевдокод алгоритма:

**Функция operator~ (void)**

{

int i, len = BitLen;

BitField temp(len);

**цикл от** int i = 0; **до** i < MemLen; **с шагом** i++)

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

**вернуть** temp;

}

В классе «Set» реализованы методы для работы с элементами множества и множествами.

1) Метод «GetMaxPower»

Данный метод возвращает максимальную мощность множества.

Ниже приведен псевдокод алгоритма:

**Функция GetMaxPower(void) const**

{

**вернуть** MaxPower;

}

2) Метод «IsMember»

Данный метод проверяет наличие элемента в множестве, возвращает работу метода GetBit .

Ниже приведен псевдокод алгоритма:

**Функция IsMember(const int Elem) const**

{

**верунть** BF.GetBit(Elem);

}

3) Метод «InsElem»

Данный метод позволяет включить элемент в множество.

Ниже приведен псевдокод алгоритма:

**Функция InsElem(const int Elem)**

{

BF.SetBit(Elem);

}

4) Метод «DelElem»

Данный метод позволяет удалить элемент из множества.

Ниже приведен псевдокод алгоритма:

**Функция DelElem(const int Elem)**

{

BF.ClrBit(Elem);

}

5) Метод «operator=»

Данный метод перегружает операцию присваивания (знак «=»).

Ниже приведен псевдокод алгоритма:

**Функция operator=** **(const Set &s)**

{

BF = s.BF;

MaxPower = s.GetMaxPower();

**вернуть** **неявный указатель на объект**;

}

6) Метод «operator==»

Данный метод перегружает операцию сравнения и возвращает 1 – истина или 0 - ложь (знак «==»).

Ниже приведен псевдокод алгоритма:

**Функция operator==(const Set &s)**

{

**вернуть** BF == s.BF;

}

7) Метод «operator+»

Данный метод перегружает операцию «объединение» (знак «+»).

Ниже приведен псевдокод алгоритма:

**Функция operator+(const Set &s)**

{

Set temp(BF | s.BF);

**вернуть** temp;

}

8) Метод «operator\*»

Данный метод перегружает операцию «пересечение» (знак «\*»).

Ниже приведен псевдокод алгоритма:

**Функция operator\* (const Set &s)**

{

Set temp(BF & s.BF);

**вернуть** temp;

}

9) Метод «operator~»

Данный метод перегружает операцию «дополнение» (знак «~»).

Ниже приведен псевдокод алгоритма:

**Функция operator~ (void)**

{

Set temp(~BF);

**вернуть** temp;

}

# Заключение

При выполнении лабораторной работы реализованы два класса **BitField** и **Set**, позволяющие работать с множествами.

# Литература

1. Лафоре Р. Объектно-ориентированное программирование в C++. Классика Computer Science. 4-е изд. – СПБ.: Питер, 2018. – 928 с.
2. Страуструп Бьерн Язык программирования C++. Специальное издание. Пер. с англ. – М.: Издательство Бином, 2017 г. – 1136 с.
3. Шилдт Г. С++ Базовый курс. 3-е изд. – М.: Издательство Вильямс, 2018. 624 с.

# Приложение

Set.h

#pragma once

#ifndef \_\_SET\_H

#define \_\_SET\_H

#include "BitField.h"

using namespace std;

class Set

{

private:

int MaxPower;

BitField BF;

public:

Set(const Set&s);

Set(int mp);

Set(const BitField &bf);

int GetMaxPower(void) const;

void InsElem(const int n);

void DelElem(const int n);

int IsMember(const int n) const;

int operator== (const Set &s);

Set & operator=(const Set &s);

Set operator+ (const Set &s);

Set operator\* (const Set &s);

Set operator~ (void);

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

friend ostream& operator<<(ostream& ostr, const Set &);

};

#endif

BitField.h

#pragma once

#ifndef \_\_BITFIELD\_H

#define \_\_BITFIELD\_H

using namespace std;

#include <iostream>

#include <stdlib.h>

typedef unsigned int UI; // unsigned int == UI

class BitField

{

private:

int BitLen; // dlinna bitovogo polya

UI \*pMem; // videleni dinamicheskoi pamyati

int MemLen; // kol-vo elementov bitovogo polya

int GetMemIndex (const int n) const;

UI GetMemMask (const int n) const;

public:

BitField(int len);

BitField(const BitField &bf);

~BitField();

int GetLength (void) const;

void SetBit(const int n);

void ClrBit(const int n);

int GetBit(const int n) const;

int operator== (const BitField &bf);

BitField & operator=(const BitField &bf);

BitField & operator|(const BitField &bf);

BitField & operator&(const BitField &bf);

BitField & operator~(void);

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

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

};

#endif

Set.cpp

#include "Set.h"

#include "BitField.h"

#include "Set.h"

using namespace std;

Set::Set(int mp) :

MaxPower(mp), BF(mp) {

}

Set::Set(const Set&s) :

MaxPower(s.MaxPower), BF(s.BF) {

}

Set :: Set(const BitField &bf) :

MaxPower(bf.GetLength()), BF(bf) {

}

int Set::GetMaxPower(void) const {

return MaxPower;

}

int Set:: IsMember(const int Elem) const {

return BF.GetBit(Elem);

}

void Set::InsElem(const int Elem) {

BF.SetBit(Elem);

}

void Set::DelElem(const int Elem) {

BF.ClrBit(Elem);

}

Set & Set ::operator= (const Set &s) {

BF = s.BF;

MaxPower = s.GetMaxPower();

return \*this;

}

int Set :: operator== (const Set &s) {

return BF == s.BF;

}

Set Set :: operator+ (const Set &s) {

Set temp(BF | s.BF);

return temp;

}

Set Set :: operator\* (const Set &s) {

Set temp(BF & s.BF);

return temp;

}

Set Set :: operator~ ( void ) {

Set temp(~BF);

return temp;

}

istream& operator>> (istream &istr, Set &s) {

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 Set &s) {

int i, n; char ch = ' ';

ostr << "{";

n = s.GetMaxPower();

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

if (s.IsMember(i)) {

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

ch = ',';

}

}

ostr << " }";

return ostr;

}

BitField.cpp

#include "BitField.h"

#include <stdio.h>

#include <iostream>

#include <math.h>

using namespace std;

BitField::BitField(int len) : BitLen(len) {

double tmp;

tmp = len / 32.0;

MemLen = ceil(tmp);

pMem = new UI[MemLen];

if (pMem != NULL)

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

pMem[i] = 0;

}

BitField::BitField(const BitField &bf) {

BitLen = bf.BitLen;

MemLen = bf.MemLen;

pMem = new UI[MemLen];

if (pMem != NULL)

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

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

}

BitField :: ~BitField() { // деструктор

delete pMem;

pMem = NULL;

}

int BitField::GetMemIndex(const int n) const {

return n >> 4;

}

UI BitField::GetMemMask(const int n) const {

return 1 << (n & 15);

}

int BitField::GetLength(void) const { // возвращает длину поля

return BitLen;

}

void BitField::SetBit(const int n) {

if ((n > -1) && (n < BitLen))

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

}

void BitField::ClrBit(const int n) {

if ((n > -1) && (n < BitLen))

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

}

int BitField::GetBit(const int n) const {

if ((n > -1) && (n < BitLen))

return pMem[GetMemIndex(n)] & GetMemMask(n);

return 0;

}

// битовые операции

BitField & BitField :: operator=(const BitField &bf) {

BitLen = bf.BitLen;

if (MemLen != bf.MemLen) {

MemLen = bf.MemLen;

if (pMem != NULL) delete pMem;

pMem = new UI[MemLen];

}

if (pMem != NULL)

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

return \*this;

}

int BitField :: operator==(const BitField &bf) {

int res = 1;

if (BitLen != bf.BitLen)

res = 0;

else

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

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

res = 0;

break;

}

return res;

}

BitField & BitField :: operator| (const BitField &bf) {

int i, len = BitLen;

if (bf.BitLen > len)

len = bf.BitLen;

BitField temp(len);

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

temp.pMem[i] = pMem[i];

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

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

return temp;

}

BitField & BitField :: operator& (const BitField &bf) {

int i, len = BitLen;

if (bf.BitLen > len)

len = bf.BitLen;

BitField temp(len);

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

temp.pMem[i] = pMem[i];

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

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

return temp;

}

BitField & BitField :: operator~ (void) {

int i, len = BitLen;

BitField temp(len);

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

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

return temp;

}

istream& operator>> (istream &istr, BitField &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 BitField &bf) {

int len = bf.GetLength();

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

if (bf.GetBit(i))

ostr << '1';

else

ostr << '0';

return ostr;

}

main.cpp

#pragma hdrstop

#include <iomanip>

#include <conio.h>

#include "BitField.h"

#include "Set.h"

#include <stdio.h>

#include <stdlib.h>

using namespace std;

int main() {

BitField x(90), x1(40), x2(0);

x2 = x&x1;

int work = 1;

while (work) {

setlocale(LC\_ALL, "Russian");

int n, m, k, count;

cout << "ТЕСТИРОВАНИЕ ПРОГРАММЫ, РАБОТАЮЩЕЙ С МНОЖЕСТВАМИ" << endl;

cout << "НА ПРИМЕРЕ РАССМОТРЕНО " << "'" << "РЕШЕТО ЭРАТОСФЕНА" << "'" << endl << endl;

cout << " Введите границу значений: ";

cin >> n;

if (n < 2) {

do {

cout << "Введите значение больше или равное 2" << endl;

cin >> n;

} while (n < 2);

}

Set s(n + 1);

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

s.InsElem(m);

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

if (s.IsMember(m))

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

if (s.IsMember(k)) s.DelElem(k);

cout << endl;

cout << " Простые числа до " << n << endl;

count = 0;

k = 1;

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

if (s.IsMember(m)) {

count++;

cout << setw(4) << m << " ";

if (k++ % 10 == 0)

cout << endl;

}

cout << endl << endl;

cout << " В числах до " << n << " содержится " << count << " простых числа" << endl;

cout << "\n\nМеню:\n- введите 1, чтобы выйти из программы;";

cout << "\n- введите 0 для ввода нового значения;\nВвод: ";

int stop = 1;

cin >> stop ;

cout << endl;

if (stop)

{

work = 0;

}

}

system("pause");

return 0;

}