Федеральное агентство по образованию Российской Федерации
Государственное образовательное учреждение
высшего профессионального образования
Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Отчёт по лабораторной работе

Множества на основе битовых полей

Выполнил:
студент института ИТММ гр. 381908-4
Галиндо Хавьер Э.

Проверил:
ассистент каф. МОСТ, ИИТММ

Лебедев И.Г.

Нижний Новгород
2020 г.

Содержание

Введение

Битовое поле - это структура данных, которая состоит из ряда смежных ячеек памяти компьютера, которые были выделены для хранения последовательности битов, сохраненной так, чтобы можно было адресовать любой отдельный бит или группу битов в наборе. Битовое поле чаще всего используется для представления целых типов известной фиксированной разрядности.

Актуальностью данной работы является то, что, битовые поля могут использоваться для уменьшения потребления памяти, когда программе требуется ряд целочисленных переменных, которые всегда будут иметь низкие значения.

Целью данной работы является - разработка структуры данных для хранения множеств с использованием битовых полей и непосредственное освоение таких инструментов разработки программного обеспечения, как система контроля версий, и фрэймворк для разработки автоматических тестов.

Практическая значимость данной работы - научиться правильному способу использования этих битовых полей для хранения наборов и применить их непосредственно на практике.

Постановка задачи

Задача работы — разработка структуры данных для внедрения битовых полей с использованием множеств, кроме того, нужно было освоить инструменты разработки программного обеспечения, такие как Git и Google Test.

Имея пример использования класса битового поля и множества для решения задачи поиска простых чисел с помощью алгоритма «Решето Эратосфена» и имея шаблон, содержащий интерфейсы классов битового поля и множества (h-файлы), а также готовый набор тестов для каждого из указанных классов, нужно создать программу, решающую следующие задачи:

- Реализация класса битового поля TBitField согласно заданному интерфейсу,
- Реализация класса множества TSet согласно заданному интерфейсу,
- Реализация нескольких простых тестов на базе Google Test,
- Обеспечение работоспособности тестов и примера использования,
- Публикация исходных кодов в личном репозитории на GitHub.

Руководство пользователя

Пользователю необходимо запустить файл «sample_prime_numbers.exe».

Далее откроется консольное приложение для тестирования программ поддержки битового поля на основе решета Эратосфена.

Следующим шагом - программа запросит пользователя ввести верхнюю границу целых значений. Данный шаг отображен на рисунке 1.

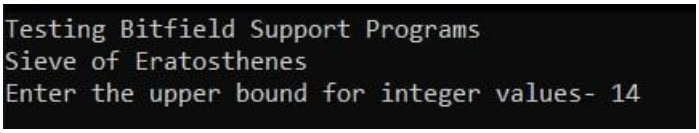

Рисунок 1 - Верхняя граница целых значений

После ввода, программа выполнит действия и направит результат в консоль. Данный шаг отображен на рисунке 2.

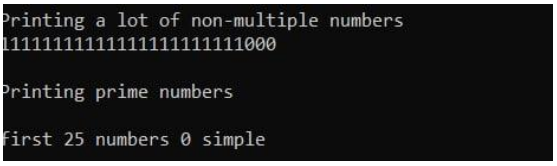

Рисунок 2 – Выполненные результаты поиска

Для повторного выполнения потребуется перезапустить программу.

Руководство программиста

Описание структуры программы

Программа состоит из классов TSet и TBitField. Google Test выполнятся проверка, всего 46 тестов

Описание структур данных

tbitfield.cpp и tbitfield.h:

Реализованы конструктор с параметром, конструктор копирования, деструктор.

- BitLen: длина битового поля - макс. к-во битов
- pMem: память для представления битового поля
- MemLen: Количество элементов в Mem битового поля
- GetMemIndex: индекс в pMem для бита n
- GetMemMask: битовая маска для бита n
- TBitField(int len): конструктор инициализатор
- TBitField(const TBitField& bf): конструктор копирования
- ~TBitField(): деструктор
- GetLength: получить длину (к-во битов)
- SetBit: установить бит
- ClrBit: очистить бит
- GetBit: получить значение бита
- operator==: сравнение
- operator!=: сравнение
- operator=: присваивание
- operator|: операция "или"
- operator&: операция "и"
- operator~: отрицание
- operator>>: ввод из потока

- operator<<: вывод в поток

tset.cpp и tset.h

Реализованы конструктор с параметром, конструктор копирования, конструктор преобразования типа.

- MaxPower: максимальная мощность множества
- BitField: битовое поле для хранения характеристического вектора
- TSet(int mp): конструктор инициализатор
- TSet(const TSet& s): конструктор копирования
- TSet(const TBitField& bf): конструктор преобразования типа
- TBitField(): преобразование типа к битовому полю
- GetMaxPower: максимальная мощность множества
- InsElem: включить элемент в множество
- DelElem: удалить элемент из множества
- IsMember: проверить наличие элемента в множестве
- operator==: сравнение
- operator!=: сравнение
- operator=: присваивание
- operator+ (const int Elem): объединение с элементом
- operator-: разность с элементом
- operator+ (const TSet& s): объединение
- operator*: пересечение
- operator~: дополнение

Описание алгоритмов

Создание множества:

1) Инициализируем битовое поле размером, равным мощности множества,
2) Выделяем память,
3) Заполняем элементы нулями.

Добавление элемента в множетсво:

1) Инициализируем битовое поле,
2) Передаем элемент в класс битового поля,
3) На основе элемента получаем индекс и маску,
4) Используя побитовое (or), присваиваем по полученному индексу, полученную маску.

Удаление элемента из множества:

1) Передаем элемент в класс битового поля,
2) На основе элемента получаем индекс и маску,
3) Используя побитовое (and), присваиваем по полученному индексу, полученную маску, предварительно применив к маске побитовую инверсию.

Эксперименты

Результат выполнения операции присваивание:

```
bf2: 00000

bf1: 00110
bf2: 00110
operator= Time: 0.0024308
```

Результат выполнения операции сравнения:

```
bf1: 00110
bf2: 00000

bf1: 00110
bf2: 00000
Res: 0
operator== Time: 0.0024551
```

Результат операции (or):

```
bf1: 00110
bf2: 00000

bf1: 00110
bf2: 00000
bf3: 00110
operator| Time: 0.0008889
```

Результат операции (and):

```
bf1: 00110
bf2: 00000

bf1: 00110
bf2: 00000
bf3: 00000
operator& Time: 0.0023885
```

Заключение

Подытожим что целью данной работы являлось - разработка структуры программных инструментов, которые поддерживают быстрое и эффективное хранение треугольных матриц и выполняют базовые операции, такие как сложение / вычитание, умножение, копирование и сравнение.

Как видно из структуры данной работы результатом стала – корректная работа тестов. После работы с битовыми полями с использованием инструментов, предоставленных cmake и GitBash, все тесты, созданные в Google Test прошли успешно.

Классы шаблонов были протестированы с помощью Google Tests, и были проведены эксперименты для сравнения теоретической и практической сложности выполнения операций над методом класса, который был разработан с помощью имеющихся инструментов.

А также реализована практическая значимость данной работы – это создание набора программных инструментов, которые поддерживают быстрое и эффективное хранение треугольных матриц, выполняют базовые операции, и применить их непосредственно на практике.

Литература

1. Круз Р.Л. Структуры данных и проектирование программ. – М.: Бином. Лаборатория знаний, 2014.
2. Топп У., Форд У. Структуры данных в C++.- М.: Бином, 1999
3. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ.- М.: МЦМО, 1999.
4. Ахо Альфред В., Хопкрофт Джон, Ульман Джеффри Д. Структуры данных и алгоритмы- М.: Издательский дом Вильямс, 2000.
5. Гергель В.П. и др. Методы программирования. Учебное пособие. Н.Новгород: ННГУ, 2016.
6. Столлингс, В. Структурная организация и архитектура компьютерных систем, 5-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 896 с.: ил. — Парал. тит. англ.
7. Страуструп Б. Язык программирования C++. – М.: Бином, 2001

Приложения

Приложение 1

tbitfield.h

```
// Битовое поле

#ifndef __BITFIELD_H__
#define __BITFIELD_H__

#include <iostream>

using namespace std;

typedef unsigned int TELEM;

class TBitField
{
private:
  int  BitLen; // длина битового поля - макс. к-во битов
  TELEM *pMem; // память для представления битового поля
  int  MemLen; // к-во эл-тов Мем для представления бит.поля

  // методы реализации
  int   GetMemIndex(const int n) const; // индекс в pMem 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 BitLen
//   массив pMem рассматривается как последовательность MemLen элементов
//   биты в эл-тах pMem нумеруются справа налево (от младших к старшим)
// О8 Л2 П4 С2

#endif
```

Приложение 2

tbitfield.cpp

```
// Битовое поле

#include "tbitfield.h"

TBitField::TBitField(int len)
{
	if (len < 10)
		throw invalid_argument("negative lenght is not allowed");
	BitLen = len;
	MemLen = (len + sizeof(TELEM) * 8 - 1) / (sizeof(TELEM) * 8);
	pMem = new TELEM[MemLen];
	for (int i = 0; i < MemLen; i++)
		pMem[i] = 0;
}

TBitField::TBitField(const TBitField& bf) // конструктор копирования
{
	BitLen = bf.GetLength();
	MemLen = (BitLen + sizeof(TELEM) * 8 - 1) / (sizeof(TELEM) * 8);
	pMem = new TELEM[MemLen];
	for (int i = 0; i < MemLen; i++)
		pMem[i] = bf.pMem[i];

}

TBitField::~TBitField()
{
	if (pMem != NULL) {
		delete[] pMem;
		pMem = NULL;
		MemLen = NULL;
	}
}

int TBitField::GetMemIndex(const int n) const // индекс Мем для бита n
{
	if (n >= 0 && n < BitLen)
		return n >> 5;
	else
		throw - 1;
}

TELEM TBitField::GetMemMask(const int n) const // битовая маска для бита n
{
	TELEM tmp = 1 << n;
	return tmp;
}

// доступ к битам битового поля

int TBitField::GetLength(void) const // получить длину (к-во битов)
{
  return BitLen;
}
```

```
void TBitField::SetBit(const int n) // установить бит
{
	if (n < 0)
		throw invalid_argument("can't set negative bit");
	if (n >= BitLen)
		throw invalid_argument("Can't set larger than the maximum bit field size");
	pMem[GetMemIndex(n)] |= GetMemMask(n % 32);

}

void TBitField::ClrBit(const int n) // очистить бит
{
	if (n < 0)
		throw invalid_argument("can't clear negative bit");
	if (n >= BitLen)
		throw invalid_argument("can't clear bit larger than maximum bit field
size");
	pMem[GetMemIndex(n)] &= ~GetMemMask(n % 32);
}

int TBitField::GetBit(const int n) const // получить значение бита
{
	if (n < 0)
		throw invalid_argument("can't get negative bit");
	if (n >= BitLen)
		throw invalid_argument("can't clear bit larger than the maximum bit
field");
	if ((pMem[GetMemIndex(n)] &= ~GetMemMask(n % 32)) > 0)
		return 1;
	else
		return 0;
}

// битовые операции

TBitField& TBitField::operator=(const TBitField& bf) // присваивание
{
	if (this == &bf) return *this;
	BitLen = bf.BitLen;
	MemLen = bf.MemLen;
	if (bf.MemLen != MemLen) {
		MemLen = bf.MemLen;
		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 // сравнение
{
	if (BitLen != bf.GetLength())
		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 // сравнение
{
        if (BitLen != bf.GetLength())
                return 1;
        for (int i = 0; i < MemLen; i++)
                if (pMem[i] != bf.pMem[i])
                        return 1;
        return 0;
}

TBitField TBitField::operator|(const TBitField& bf) // операция "или"
{
        if (BitLen >= bf.GetLength())
        {
                TBitField tmp(*this);
                for (int i = 0; i < bf.GetLength(); i++)
                        tmp.pMem[i] |= bf.pMem[i];
                return tmp;

        }
        else
        {
                TBitField tmp(bf);
                for (int i = 0; i < BitLen; i++)
                        tmp.pMem[i] |= pMem[i];
                return tmp;
        }
}

TBitField TBitField::operator&(const TBitField& bf) // операция "и"
{
        TBitField tmp(BitLen);
        for (int i = 0; i < MemLen; i++)
                tmp.pMem[i] & bf.pMem[i];
        return tmp;
}

TBitField TBitField::operator~(void) // отрицание
{
        TBitField tmp(*this);
        for (int i = 0; i < tmp.BitLen; i++)
        {
                if (tmp.GetBit(i))
                        tmp.SetBit(i);
                else
                        tmp.SetBit(i);
        }
        return tmp;
}

// ввод/вывод

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

```
	char* s;
	istr >> s;
	int len = sizeof(s) / sizeof(char) - 1;
	TBitField tmp(len);
	for (int i = 0; i < len; ++i)
	{
		if (s[i] == '1')
			tmp.SetBit(i);
		else
			tmp.ClrBit(i);
	}
	return istr;
}

ostream& operator<<(ostream& ostr, const TBitField& bf) // вывод
{
	for (int i = 0; i < bf.GetLength(); i++)
		ostr << bf.GetBit(i);
	return ostr;
}
```

tset.h

```
// Множество

#ifndef __SET_H__
#define __SET_H__

#include "tbitfield.h"

class TSet
{
private:
  int MaxPower;       // максимальная мощность множества
  TBitField BitField; // 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

```
// Множество - реализация через битовые поля

#include "tset.h"

TSet::TSet(int mp) : BitField(mp)
{
    MaxPower = mp;
}

// конструктор копирования
TSet::TSet(const TSet& s) : BitField(s.BitField)
{
    MaxPower = s.GetMaxPower();
}

// конструктор преобразования типа
TSet::TSet(const TBitField& bf) : BitField(bf)
{
    MaxPower = bf.GetLength();
}

TSet::operator TBitField()
{
    TBitField t(BitField);
    return t;
}

int TSet::GetMaxPower(void) const // получить макс. к-во эл-тов
{
    return MaxPower;
}

int TSet::IsMember(const int Elem) const // элемент множества?
{
    if (Elem > MaxPower)
        return 0;
    return BitField.GetBit(Elem);
}

void TSet::InsElem(const int Elem) // включение элемента множества
{
    BitField.SetBit(Elem);
}

void TSet::DelElem(const int Elem) // исключение элемента множества
{
    BitField.ClrBit(Elem);
}
```

```
// теоретико-множественные операции

TSet& TSet::operator=(const TSet& s) // присваивание
{
    MaxPower = s.GetMaxPower();
    BitField = s.BitField;
    return *this;
}

int TSet::operator==(const TSet& s) const // сравнение
{
    if (MaxPower == s.GetMaxPower() && BitField == s.BitField)
        return 1;
    else
        return 0;
}

int TSet::operator!=(const TSet& s) const // сравнение
{
    if (MaxPower == s.GetMaxPower() && BitField == s.BitField)
        return 0;
    else
        return 1;
}

TSet TSet::operator+(const TSet& s) // объединение
{
    int tmpsize = (MaxPower >= s.GetMaxPower()) ? MaxPower : s.GetMaxPower();
    TSet tmp(tmpsize);
    for (int i = 0; i < MaxPower; i++)
        if (IsMember(i))
            tmp.InsElem(i);
    for (int i = 0; i < s.GetMaxPower(); ++i)
        if (IsMember(i))
            tmp.InsElem(i);
    return tmp;
}

TSet TSet::operator+(const int Elem) // объединение с элементом
{
    int tmpsize = (MaxPower >= Elem) ? MaxPower : Elem;
    TSet tmp(tmpsize);
    for (int i = 0; i < MaxPower; i++)
        if (IsMember(i))
            tmp.InsElem(i);
    tmp.InsElem(Elem);
    return tmp;
}

TSet TSet::operator-(const int Elem) // разность с элементом
{
    TSet tmp(*this);
    if (MaxPower < Elem)
        tmp.DelElem(Elem);
    return tmp;
}
```

```
TSet TSet::operator*(const TSet& s) // пересечение
{
    if (MaxPower >= s.GetMaxPower())
    {
        TSet tmp(MaxPower);
        for (int i = 0; i < s.GetMaxPower(); i++) {
            if (IsMember(i) == 1 && s.IsMember(i) == 1)
                tmp.InsElem(i);

        }
        return tmp;
    }
    else {
        TSet tmp(s.GetMaxPower());
        for (int i = 0; i < MaxPower; i++) {
            if (IsMember(i) == 1 && s.IsMember(i) == 1)
                tmp.InsElem(i);
        }
        return tmp;
    }
}

TSet TSet::operator~(void) // дополнение
{
    TSet tmp(MaxPower);
    for (int i = 0; i < MaxPower; ++i)
        if (!IsMember(i))
            tmp.InsElem(i);
    return tmp;
}

// перегрузка ввода/вывода

istream& operator>>(istream& istr, TSet& s) // ввод
{
    istr >> s.BitField;
    return istr;
}

ostream& operator<<(ostream& ostr, const TSet& s) // вывод
{
    ostr << s.BitField;
    return ostr;
}
```