Структура МНОЖСТВО

Для хранения конечного множества значений имеется удобная структура: массив. Заметим, что массив и множество, вообще говоря, не одно и то же. Во-первых, массив, в отличие от множества, может содержать одно и то же значение в нескольких экземплярах. Во-вторых, элементы в n-элементном множестве можно расположить n! способами, переставляя их.

Таким образом, массив содержит, кроме списка элементов, избыточную информацию об их упорядочении. С учётом того, что имеется n! различных способов упорядочить множество, состоящее из n элементов, количество дополнительной информации равно согласно формуле Хартли $log_2 n!$. Для множества из 10 элементов перерасход составит приблизительно 18,47 бит, а уже для 1000000-элементного — 18488864,88.

Кроме того, массивы устроены довольно сложно. На каждый элемент массива приходится много дополнительной информации.

Мы обсудим реализацию конечного множества неотрицательных чисел, не использующую массивов и лишённую перечисленных недостатков. Это битовая реализация числового множества. Присутствие числа i в множестве обозначается установкой i-го бита в массиве, отсутствие — очисткой соответствующего бита.

Вспомогательная тема: побитовые операции

Логические побитовые операции

Битовые операторы И (AND, &), ИЛИ (OR, |), НЕ (NOT, ~) используют те же таблицы истинности, что и их логические эквиваленты.

Побитовое И. Побитовое И используется для выключения битов. Любой бит, установленный в 0, вызывает установку соответствующего бита результата также в 0.

```
  11001010
& 11100010
  11000010
```

Побитовое ИЛИ. Побитовое ИЛИ используется для включения битов. Любой бит, установленный в 1, вызывает установку соответствующего бита результата также в 1.

```
  11001010
| 11100010
  11101010
```

Побитовое НЕ. Побитовое НЕ инвертирует состояние каждого бита исходной переменной.

```
~ 11001010
  00110101
```

Побитовые сдвиги

Операторы сдвига << и >> заставляют биты левого операнда сдвинуться влево или вправо на то количество позиций, которое указано во втором операнде.

Пример:

int a = 5; // в двоичной системе: 0000000000000101

int b = a << 3; // в двоичной системе: 0000000000101000, или 40 в десятичной

int c = b >> 3; // в двоичной системе: 0000000000000101, или снова 5, как было изначально

Следует иметь ввиду, что при сдвиге значения x на y бит (x<<y), самые левые y бит в исходном числе x теряются, т.к. они буквально выталкиваются за его пределы.

int a = 5; // в двоичной системе: 0000000000000101

int b = a << 14; // в двоичной системе: 0100000000000000 - первая 1 в 101 исчезла

Если вы уверены, что ни один из битов в сдвигаемом числе не пропадет, то для простоты можно считать, что оператор сдвига << умножает левый операнд на 2 в степени, показателем которой является правый операнд. Например, для получения степеней 2 могут быть использованы следующие выражения:

1 << 0 == 1

1 << 1 == 2

1 << 2 == 4

1 << 3 == 8

...

1 << 9 == 512

1 << 10 == 1024 ...

В переменных типа int старший бит является знаковым битом, определяющим является ли число положительным или отрицательным. Если переменная x имеет тип int, то при сдвиге x вправо знаковый бит копируется в младшие биты (по историческим причинам):

int x = -16; // в двоичной системе: 1111111111110000

int y = x >> 3; // в двоичной системе: 1111111111111110

int x = -16; // в двоичной системе: 1111111111110000

int y = (unsigned int)x >> 3; // в двоичной системе: 0001111111111110

Таким образом, если предотвращать эффект расширения знака, оператор сдвига вправо >> можно использовать для деления числа на степени 2. Например:

int x = 1000;

int y = x >> 3; // целочисленное деление 1000 на 8, в результате которого y = 125.

Наглядный пример и разбор функций

Лекции: http://www.itmm.unn.ru/files/2019/02/Metody-programmirovaniya-1-chast-2015.pdf

Пусть рассматривается множество неотриательных значений, не превышающих 45, например, A = {0, 1, 2,3 4, …, 19, 23, 32, 33, 44}.

В описании класса TBitField есть 3 поля: длина битового поля (максимальное количество бит) – BitLen, память для представления битового поля - массив unsigned int (TELEM) элементов – pMem, количество элементов в массиве pMem – MemLen.

Посмотрим на примере. В данном случае BitLen = 45.

Битовое поле, соответствующее данному множеству:

1 1 1 … 1 0 0 0 1 0 … 0 1 1 0 … 0 1

0 1 2 … 19 20 21 22 23 24 … 31 32 33 34 … 43 44

Но в памяти компьютера это битовое представление хранится иначе. Для начала определим, по какой формуле вычисляется длина массива pMem.

Замечание! В зависимости от архитектуры процессора unsigned int может занимать 2 байта (16 бит) или 4 байта (32 бита). В лекционных материалах излагается информация и приведены формулы для 16-битных величин. Так как у большинства всё же архитектура такова, что unsigned int 4-х байтовый, пересмотрим формулы, использующиеся в лабораторной.

Проще всего излагать матерал на примере.

Выделение памяти для представления битового поля

В лекционных материалах в первой же реализованной функции класса битовое поле имеет место магическая формула

```
TBitField :: TBitField(int len) : BitLen(len) {
  MemLen = ( len + 15 ) >> 4; // в эл-те pMem 16 бит (TELEM==int)
  pMem   = new TELEM[MemLen];
  if ( pMem != NULL )
    for ( int i=0; i<MemLen; i++ ) pMem[i] = 0;
}              /*-----------------------------------------*/
```

И указано, что в элементе pMem 16 бит. Однако у нас unsigned int 32 бита. Если используется другой тип данных, то можно получить и другое значение. Поэтому нужно понять, как эта формула связана с размером используемого типа данных и написать её в общем виде.

Смысл: нужно выбрать столько элементов, сколько хватит на хранение указанного числа бит. Очевидно, что для рассматриваемого примера это число 2: 32 бита в первом элементе, оставшиеся во втором. Рассмотрим точную формулу.

```
MemLen = (len + sizeof(TELEM) * 8 - 1) >> x;
```

Если бы размер TELEM был 2 байта, получили бы именно len + 15. Осталось понять, откуда берется сдвиг на 4 бита в той формуле. Сдвиг на 4 означает, что полученное значение сдвигается на 16 бит (2^4 = 16), соответственно для 32-битной реализации 2^x = 32 -> x = 5. Однако, лучше так не писать и представить это также в общем виде, то есть сдвиг будет

```
int x = log((int)sizeof(TELEM) * 8) / log(2);
```

Строго посчитаем по этой формуле требуемое число элементов pMem.

MemLen = (45 + 4 * 8 – 1) >> 5 = 51 = 10~~0110~~ >> 4 = 10 = 2

MemLen = 2 – для представления множества с максимальным числом бит 45 нам нужно 2 элемента по 4 байта (8 байт или 64 бит)

Индексы в pMem элементов – метод GetIndexMem

```
int TBitField :: GetMemIndex ( const int n ) const { // индекс Мем для бита n
  // преобразовать к int и разделить на 16
  return n >> 4; // в эл-те pMem 16 бит
}              /*-----------------------------------------*/
```

Аналогичным образом вместо приведённого в методичке сдвига на невнятные 4, приводим формулу к общему виду:

```
int x = log((int)sizeof(TELEM) * 8) / log(2);
return n >> x;
```

Посчитаем на примере.

Например, для 6-го бита индекс будет 110 >> 5 = ~~000110~~ >> 5 = 0

Например, для 20-го бита индекс будет 10100 >> 5 = ~~010100~~ >> 5 = 0

Например, для 31-го бита индекс будет 11111 >> 5 = ~~011111~~ >> 5 = 0

Например, для 32-го бита индекс будет 1~~00000~~ >> 5 = 1

Маски для битов

```
TELEM TBitField :: GetMemMask ( const int n ) const { // битовая маска для бита
n
  // преобразовать к int, найти остаток от деления на 16 и сдвинуть
  return 1 << (n & 15);
}              /*-----------------------------------------*/
```

```
1 << (n & (sizeof(TELEM) * 8 - 1));
```

Например, для 6-го бита индекс будет 1 << (6 & (4 * 8 - 1)) = 1 << 110 = 1 << 6 = 1000000

```
6 & (4 * 8 - 1) =  00110
                   11111
                   = 110
```

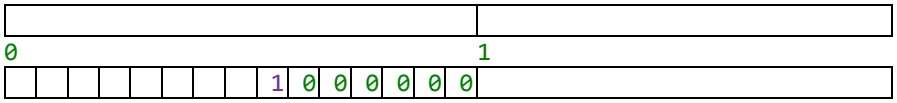
Метод установки бита SetMem

```
void TBitField :: SetBit ( const int n ) { // установить бит
  if ( (n > -1) && (n < BitLen) )
    pMem[GetMemIndex(n)] |= GetMemMask(n);
}              /*-----------------------------------------*/
```

Посмотрим это на примере

Рассмотрим операцию установки 6-го бита

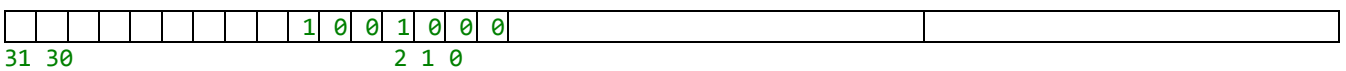
PMem



Pmem = 64 0

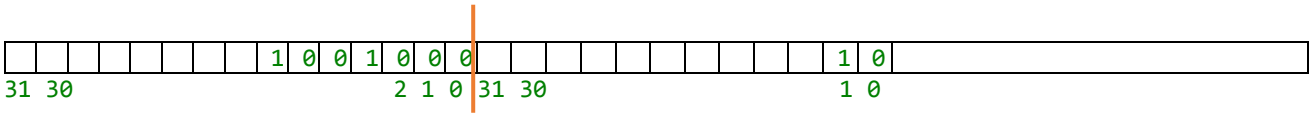
Рассмотрим операцию установки 3-го бита. Индекс будет 1 << (3 & (4 * 8 - 1)) = 1 << 11 = 1 << 3 = 1000 = 8

```
00011
11111
=  11
```

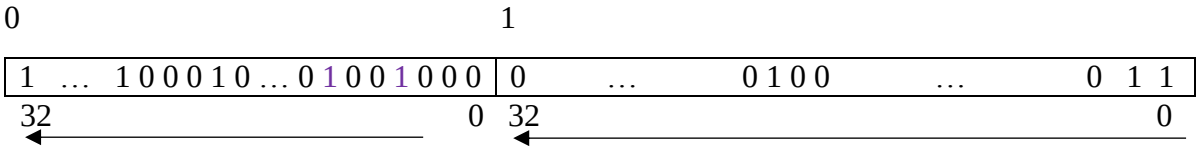


Например, для 33-го бита индекс будет 1 << (33 & (4 * 8 - 1)) = 1 << 1 = 10

```
100001
011111
=    1
```



pMem программиста



pMem человека

9437183	4099

Тестирование программы

Как разобраться в программе (пример показательного теста)

```
TEST(TBitField, my_test_for_understending_how_it_work)
{
  const int size = 20;
  TBitField bf(size);

  bf.SetBit(3);
  bf.SetBit(6);
  bf.SetBit(9);
  bf.SetBit(11);
  bf.SetBit(12);
  bf.SetBit(13);
  bf.SetBit(19);

  std::cout << "\nBitField Data:";
  std::cout << bf;

  std::cout << "\nBits:";
  for (int i = 1; i < 20; ++i)
    std::cout << " " << bf.GetBit(i);

  EXPECT_NE(0, bf.GetBit(6));
}

TEST(TBitField, my_test_for_understending_how_it_work_full_set)
{
  const int size = 45;
  TBitField bf(size);

  bf.SetBit(0);
  bf.SetBit(1);
  bf.SetBit(2);
  bf.SetBit(3);
  bf.SetBit(4);
  bf.SetBit(5);
  bf.SetBit(6);
  bf.SetBit(7);
  bf.SetBit(8);
  bf.SetBit(9);
  bf.SetBit(10);
  bf.SetBit(11);
  bf.SetBit(12);
  bf.SetBit(13);
  bf.SetBit(14);
  bf.SetBit(15);
  bf.SetBit(16);
  bf.SetBit(17);
```

```
  bf.SetBit(18);
  bf.SetBit(19);
  bf.SetBit(23);
  bf.SetBit(32);
  bf.SetBit(33);
  bf.SetBit(44);

  std::cout << "\nBitField Data:";
  std::cout << bf;

  std::cout << "\nBits:";
  for (int i = 0; i < 44; ++i) std::cout << " " << bf.GetBit(i);

  EXPECT_NE(0, bf.GetBit(6));
}
```