ThreeMultiSet in C++

Проблем:

Искаме да поддържаме структура от данни, която представя мултимножество $S \subseteq \{0, 0, 0, 1, \ldots, U-1, U-1, U-1\}$, в което всеки елемент може да се съдържа най-много 3 пъти и е по-малък от някое дадено U. Освен това искаме да можем да добавяме и премахваме елементи, да казваме колко пъти се среща даден елемент и да намираме сечение на две мултимножества, като желаем структурата да бъде възможно най-оптимална откъм памет, а операциите, които се извършват върху нея, да бъдат “бързи”.

Идея:

Лесно може да забележим, че двоичното представяне на числата 0, 1, 2, 3 заема точно два бита – $00_{(2)}$, $01_{(2)}$, $10_{(2)}$, $11_{(2)}$, което значи, че може да направим Bitset, в който всяко число се представя с не един, а два бита и така ще може в един int (bucket), който има 32 бита, да пазим информацията за 32 / 2 = 16 числа. За улеснение на обясненията тук ще ползваме bucket, който има 16 бита и 8 различни числа.

Ето например как ще изглежда един bucket за числата от 0 до 7:

0		1		2		3		4		5		6		7	
0	1	0	0	1	1	1	0	0	0	1	1	1	0	0	1

Множеството, представено тук, е $S = \{0, 2, 2, 2, 3, 3, 5, 5, 5, 6, 6, 7\}$.

Решение:

Ясно е, че ще използваме динамичен масив, като нека да е от тип Bucket, който е alias на uint16_t (с alias лесно ще може да променяме големината на bucket-а).

```
using Bucket = uint16_t;
```

Броят различни числа в един bucket се определя по следния начин:

```
const short NUMS_IN_BUCKET = sizeof(Bucket) * BITS_IN_BYTE / BITS_PER_NUM;
```

където BITS_IN_BYTE = 8 и BITS_PER_NUM = 2.

Данните, които ще пазим в класа, са:

```
Bucket *buckets; // масивът
size_t size;     // размерът на масива
size_t u;        // константата U
```

Първото нещо, което трябва съобразим, е колко ще е голям масивът – дължината му се определя от U – 1, което е най-голямото число в множеството, и размерът ще е индексът на bucket-а, в който се намира U – 1, плюс 1 (защото броим от 0). Понеже често ще ни трябва да разбираме едно число в кой bucket е, може да си изнесем формулата във функцията:

```
static size_t bucket(size_t num) {
    return num / NUMS_IN_BUCKET;
}
```

Тогава конструкторът на класа ще изглежда така:

```
ThreeMultiSet(size_t u) : u(u), size(bucket(u - 1) + 1) {
    buckets = new Bucket[size];
}
```

Разбира се, трябва да си разпишем и голямата четворка, понеже използваме динамична памет, но това оставяме за упражнение на читателя.

Така, да продължим с функциите, като нека започнем с функцията, която връща колко пъти се съдържа дадено число в множеството. Нека да се върнем на примера горе и да забележим, че ако искаме да проверим колко пъти се съдържа числото 3, най-лесно е да направим побитово & на bucket-а с маска, в която има 11 точно на позицията, на която се намират двата бита за 3, и да направим побитово >>, за да преместим битовете вдясно.

0		1		2		3		4		5		6		7	
0	1	0	0	1	1	1	0	0	0	1	1	1	0	0	1

&

0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0

=

0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

>>

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Получава се числото 2 – точно колкото пъти се среща числото 3 в мултимножестово. За да направим тези побитови операции правилно, ни трябва, освен да знам bucket-а, да можем да намерим маската за позицията и отместването на битовете, така че да получим числото, което ни трябва.

Нека това да става с функциите position(size_t num) и shift(size_t num), които ще разпишем след малко. С тях функцията count(size_t num) ще изглежда по следния начин:

```
int count(size_t num) const {
    if (num >= u) {
        return 0;
    }

    return (buckets[bucket(num)] & position(num)) >> shift(num);
}
```

Най-лесно да намерим маската, която връща функцията position(size_t num), е като просто преместим битовете 11 от дясно на правилната позиция с отместването, което ни връща функцията shift(size_t num). Тук MAX_COUNT_OF_NUMBER = 3 = $11_{(2)}$.

```
static size_t position(size_t num) {
    return MAX_COUNT_OF_NUMBER << shift(num);
}
```

В примера с числото 3 отместването на маската е точно 8 бита от дясно наляво. В един bucket има 8 числа, които се намират на позиции от 0 до 7 при броене от ляво надясно, което значи, че позицията на едно число е точно остатъкът, който то дава при деление с 8. Ние обаче искаме да правим побитово shift-ване от дясно наляво, затова ще трябва да сменим посоката на броене, като просто от 7 извадим получения остатък. Остава и да умножим по броя битове, които заема дадено число в един bucket, и получаваме, че shift(3) = 2 * (7 - 3 % 8) = 8.

По този начин получаваме формулата за отместване:

```
static size_t shift(size_t num) {
    return BITS_PER_NUM * (NUMS_IN_BUCKET - 1 - num % NUMS_IN_BUCKET);
}
```

Сега остава да видим как добавяме число в множество (няма да разглеждаме премахване, защото е аналогично). Това, което ни трябва да знаем, е колко пъти се среща числото, но вече имаме функцията count(size_t num), която прави точно това. Ясно е, че ако тя върне 3, то не трябва да правим а нищо, но ако върне стойност k < 3, то трябва на позицията, на която се намират битовете на числото, да запишем битовете на (k + 1), което става лесно на две стъпки:

1. Сваляме битовете на тази позиция, което става с побитово & с маска, която има 00 само на позицията им, а тя се получава като обърнем всички битове на маската, която връща функцията position(size_t num).

2. На тази позиция записваме битовете на (k + 1) с побитово |, като преди това ги отместим с shift(size_t num).

Например да видим как ще добавим 6 в множеството, което имаме.

count(6) = 2 => трябва да запишем битовете 2 + 1 = 3 = $11_{(2)}$, което става така:

1)

0		1		2		3		4		5		6		7	
0	1	0	0	1	1	1	0	0	0	1	1	1	0	0	1

&

(

~

0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0

=

1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1

)

=

0	1	0	0	1	1	1	0	0	0	1	1	0	0	0	1

2)

0	1	0	0	1	1	1	0	0	0	1	1	0	0	0	1

|

(

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

<< shift(6) = 2 * (7 - 6 % 8) = 2

0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0

)

=

0		1		2		3		4		5		6		7	
0	1	0	0	1	1	1	0	0	0	1	1	1	1	0	1

Тези две стъпки ще ги изнесем във функцията replace(size_t num, size_t count), като така функцията за добавяне на число ще изглежда по следния начин:

```
void insert(size_t num) {
    int countNum = count(num);
    if (num >= u || countNum == MAX_COUNT_OF_NUMBER) {
        return;
    }

    ++countNum;
    replace(num, countNum);
}
```

```
void replace(size_t num, size_t count) {
    size_t bucketIdx = bucket(num);

    buckets[bucketIdx] &= ~position(num);
    buckets[bucketIdx] |= (count << shift(num));
}
```

Нека да видим последно как можем да намираме сечение на две мултимножества. Логично е, че ако в едното множество имаме три пъти 2, а в другото – само един път, то в сечението трябва да имаме веднъж 2, т.е. трябва да вземем по-малкото от двете числа. Така функцията за сечение е:

```
ThreeMultiSet intersect(const ThreeMultiSet &lhs, const ThreeMultiSet &rhs) {
    size_t minU = std::min(lhs.u, rhs.u);
    ThreeMultiSet intersection(minU);

    for (size_t i = 0; i < minU; ++i) {
        intersection.replace(i, std::min(lhs.count(i), rhs.count(i)));
    }

    return intersection;
}
```