Задача 6. Хеш-таблица

Источник:	основная I
Имя входного файла:	---
Имя выходного файла:	---
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

В этой задаче нужно реализовать "map" (отображение) с помощью хеш-таблицы.

По сути, отображение — это функция, отображающая «ключи» в «значения», и которую можно изменять. В map хранится набор пар «ключ-значение», у всех пар всегда разные ключи. Можно выполнять операции: установить заданное значение для заданного ключа, узнать значение для заданного ключа.

Например, рассмотрим отображение целочисленных ключей в строковые значения:

$$5 \to five \quad 3 \to three \quad 13 \to bignumber \quad 17 \to bignumber \quad 7 \to seven$$

Если установить для ключа 6 значение six, а для ключа 3 значение $triple$, то отображение примет вид:

$$5 \to five \quad 3 \to triple \quad 13 \to bignumber \quad 17 \to bignumber \quad 7 \to seven \quad 6 \to six$$

Чтобы быстро находить элемент в отображении по заданному ключу, вам нужно реализовать хеш-таблицу. В хеш-таблице хранятся лишь указатели на ключ и на значение, которые задаёт пользователь. Чтобы хеш-таблицу можно было применять для любых типов ключей и значений, эти указатели нетипизированные (`void*`, точнее `const void*`). Сами ключи и значения хранит пользователь: он обеспечивает их хранение. Гарантируется, что каждые занесённые в хеш-таблицу ключ и значение остаются валидными как минимум пока их не удалят из хеш-таблицы.

Очевидно, ключи разного типа сравниваются между собой по-разному. Поэтому пользователь хеш-таблицы задаёт указатель на функцию `EqualFunc`, с помощью которых хеш-таблица может проверить для любых двух ключей, совпадают ли они. Кроме того, для хеш-таблицы необходима хеш-функция, которая также задаётся пользователем (`HashFunc`). Гарантируется, что эти функции корректны: `EqualFunc` является эквивалентностью (есть транзитивность, симметричность и рефлексивность), `HashFunc` работает детерминированно и всегда выдаёт одинаковый хеш для равных ключей. Функции `EqualFunc` и `HashFunc` можно вызывать для всех ключей, которые сейчас лежат в таблице, а также для ключа, который передан явно в выполняемую функцию. Естественно, вызывать их для нулевого указателя или для указателя на уже удалённый из таблицы ключ нельзя.

При создании хеш-таблицы пользователь задаёт количество ячеек в ней. Пользователь гарантирует, что в хеш-таблице всегда будет хотя бы одна свободная ячейка. Более того, если вы используйте открытую адресацию (линейное пробирование и т.п.), то степень заполненности таблицы будет ограничена разумной константой. Хеш-функция возвращает 32-битное число, которое достаточно хорошо распределено на всему диапазону от 0 до $2^{32} - 1$. Благодаря этим условиям ваша хеш-таблица должна работать достаточно быстро. Естественно, вы можете реализовать разрешение коллизий через цепочки: даже в этом случае рекомендуется заводить предписанное количество ячеек в таблице.

Вы должны оформить вашу хеш-таблицу как динамическую библиотеку. В хедере `hashmap.h` должна быть объявлена структура `HashMap`, а также следующие типы и функции:

```
//pointer to key or value (untyped)
typedef const void *cpvoid;
//returns 1 if and only if two keys pointed by [a] and [b] are equal
//returns 0 otherwise
typedef int (*EqualFunc)(cpvoid a, cpvoid b);
//returns 32-bit hash of a key pointed by [key]
typedef uint32_t (*HashFunc)(cpvoid key);

//creates and returns new hash table with:
// [ef] -- function which compares keys for equality
// [hf] -- function which produces a hash for a key
// [size] -- prescribed size/capacity of the hash table (number of cells)
HashMap HM_Init(EqualFunc ef, HashFunc hf, int size);
//frees memory of hash map [self]
//note: called exactly once for every hash map created by HM_Init
void HM_Destroy(HashMap *self);

//returns value corresponding to the specified key [key] in hash map [self]
//if [key] is not present in the map, then returns NULL
cpvoid HM_Get(const HashMap *self, cpvoid key);
//sets value [value] for the key [key] in hash map [self]
//if [self] already has some value for [key], it is overwritten
void HM_Set(HashMap *self, cpvoid key, cpvoid value);
```

Кроме хедера `hashmap.h` вы можете отправить любое количество файлов исходного кода или хедеров.

Все файлы исходного кода будут собираться в `hashmap.so` с указанием макроса `HASHMAP_EXPORTS`. Этот макрос можно использовать для экспортирования функций в общепринятом порядке. Далее тестирующий код жюри будет собираться с созданной библиотекой `hashmap.so`.

Пример кода, который должен собираться вместе с библиотекой и работать:

```
#include "hashmap.h"
#include <assert.h>
#include <string.h>

#define INTOF(ptr) (*(int*)(ptr))
int IntEqualFunc(cpvoid a, cpvoid b) {
    return (INTOF(a) - INTOF(b)) % 1000 == 0;
}
uint32_t IntHashFunc(cpvoid key) {
    int t = INTOF(key) % 1000;
    if (t < 0)
        t += 1000;
    return t * 0xDEADBEEF;
}

int main() {
    int data[] = {13, 174, 1013, -987, 0, 1};

    HashMap h = HM_Init(IntEqualFunc, IntHashFunc, 5);
    HM_Set(&h, &data[0], "hello");
    HM_Set(&h, &data[1], "world");
    assert(strcmp(HM_Get(&h, &data[2]), "hello") == 0);
    assert(strcmp(HM_Get(&h, &data[3]), "hello") == 0);
    HM_Set(&h, &data[4], "zero");
    assert(strcmp(HM_Get(&h, &data[4]), "zero") == 0);
    assert(strcmp(HM_Get(&h, &data[1]), "world") == 0);
    assert(HM_Get(&h, &data[5]) == 0);
    HM_Set(&h, &data[5], "one");
    //note: one empty cell left => cannot add more!
    HM_Destroy(&h);

    h = HM_Init(IntEqualFunc, IntHashFunc, 100000); //create larger table
    assert(HM_Get(&h, &data[5]) == 0);
    HM_Set(&h, &data[1], "newtable");
    HM_Set(&h, &data[1], "newtableX");
    assert(strcmp(HM_Get(&h, &data[1]), "newtableX") == 0);
}
```

Заметим, что здесь ключом является остаток от деления на 1000. В частности, целые числа 13, 1013, -987 считаются равными, т.к. они сравнимы по модулю 1000. Равенство подтверждается задаваемой функцией `EqualFunc`, и функция `HashFunc` выдаёт одинаковый хеш для всех этих ключей. При реализации хеш-таблицы вам не нужно думать об этом: надо лишь вызывать заданные функции для сравнения ключей.