Задача 8. Аллокатор

Источник:	повышенной сложности*
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	**специальное**

Функции `malloc` и `free` позволяют динамически выделять блоки памяти и возвращать их назад в кучу. К сожалению, иногда эти функции работают медленнее, чем того хотелось бы. В таких случаях программисты порой реализуют собственные алгоритмы выделения памяти взамен `malloc`/`free`, которые применимы в одной конкретной задаче, зато работают при этом намного быстрее.

В этой задаче нужно реализовать специальный алгоритм для выделения блоков памяти **одинакового** (и маленького) размера. Алгоритм работает очень просто. Изначально выделяется большой кусок памяти (по сути массив) размером ровно в N блоков. Когда пользователь запрашивает у аллокатора новый блок памяти, аллокатор выбирает любой незанятый блок из этого массива и возвращает его адрес пользователю. Если все блоки заняты, аллокатор должен вернуть нулевой указатель, так как заведуемая им память закончилась. Когда пользователь освобождает блок памяти, аллокатор помечает его как свободный, чтобы в будущем можно было его переиспользовать.

Кроме собственно массива блоков, аллокатор также должен хранить множество незанятых блоков, чтобы знать, какие блоки сейчас можно выдавать пользователю, а какие нет. Для этого используется система под названием free list. Все незанятые блоки объединяются в односвязный список, причём узлами этого списка становятся сами незанятые блоки памяти. То есть в каждом незанятом блоке аллокатор хранит указатель на следующий такой незанятый блок.

Обратите внимание, что узлы односвязного списка **физически расположены внутри того самого массива**, блоки которого выдаются пользователю, а не где-то ещё снаружи! Так получается аллокатор без накладных расходов: помимо собственно куска памяти из N блоков не нужно никакой дополнительной памяти, кроме $O(1)$ памяти где-то в головной структуре аллокатора.

В тестовой задаче нужно реализовать аллокатор для выделения блоков размером в 8 байт и записи туда вещественных значений типа `double`. Нужно реализовать следующие функции:

```
//головная структура аллокатора
typedef struct MyDoubleHeap_s {
    ???     //можно хранить здесь всякие данные
} MyDoubleHeap;
//создать новый аллокатор с массивом на slotsCount блоков
MyDoubleHeap initAllocator(int slotsCount);
//запросить блок памяти под число типа double
double *allocDouble(MyDoubleHeap *heap);
//освободить блок памяти, на который смотрит заданный указатель
void freeDouble(MyDoubleHeap *heap, double *ptr);
```

Далее нужно обработать набор операций/запросов.

Формат входных данных

В первой строке задано два целых числа: N — на сколько блоков нужно изначально создать массив (`slotsCount`) и Q — сколько операций нужно после этого выполнить ($2 \leqslant N, Q \leqslant 3 \cdot 10^5$). В остальных Q строках описаны операции.

Каждая операция начинается с целого числа t — типа операции. Если $t = 0$, то это операция выделения блока памяти. Тогда далее записано вещественное число, которое нужно сохранить в этом блоке памяти. При выполнении этой операции нужно вывести в выходной файл адрес, который вернула функция `allocDouble`. Этот адрес должен должен делиться на 8, чтобы double был корректно выровнен по своему размеру.

Если $t = 1$, то это операция освобождения блока памяти, и далее записано целое число k — номер операции, в которой был выделен тот блок памяти, который сейчас нужно удалить. Если $t = 2$, то нужно просто распечатать содержимое того блока памяти, который был выделен на k-ой операции, как вещественное число.

Все запросы нумеруются по порядку номерами от 0 до $Q - 1$. Для вывода в файл адреса/указателя используйте формат `"%p"`. Все вещественные числа заданы с не более чем 5 знаками после десятичной точки, и не превышают 10^4 по модулю. Вещественные числа следует выводить в аналогичном виде (например, используя формат `"%0.5lf"`).

Гарантируется, что никакой выделенный блок памяти не будет удалён дважды, и что у вас не попросят распечатать содержимое уже освобождённого блока. Гарантируется, что если запрос на выделение памяти возвращает нулевой указатель (когда все N блоков заняты), то на эту операцию не ссылаются никакие другие запросы, т.е. этот невыделенный блок не попытаются освободить или распечатать.

Формат выходных данных

Выведите результаты выполнения операций (для операций типа $t = 0$ и $t = 2$).

Пример

input.txt	output.txt
5 12 0 0.1 0 1.1 0 2.1 0 3.1 0 4.1 0 5.1 2 3 2 1 1 3 0 123.0 2 9 0 -1	001A0480 001A0488 001A0490 001A0498 001A04A0 00000000 3.100000000000000 1.100000000000000 001A0498 123.000000000000000 00000000

Пояснение к примеру

Изначально вызываем `initAllocator` с `slotsCount` = 5. Внутри он создаёт массив на 5 элементов, и объединяет их все в односвязный список (т.к. изначально все блоки незаняты).

Далее делается шесть запросов на выделение памяти. Первые пять срабатывают успешно, и в выходном файле распечатаны адреса блоков (они идут подряд с шагом в 8 байт). Для шестого запроса свободных блоков нет (ведь $N = 5$), поэтому блок не выделяется и выводится нулевой указатель.

Далее распечатываются вещественные числа по адресам `001A0498` и `001A0488`. Потом блок по адресу `001A0498` освобождается, и сразу же выделяется обратно для числа 123.0. Наконец, распечатывается содержимое для только что выделенного блока (т.е. 123.0) и выполняется ещё один неуспешный запрос на выделение памяти.

Комментарий

Для решения тестовой задачи рекомендуется завести глобальный массив `double *idToHeap[301000]`, чтобы отслеживать, какой указатель `double*` соответствует каждому номеру операции k (см. формат входных данных).