Задача 1. Распечатать список

Источник:	базовая
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

Дан односвязный список, который хранится в массиве. Массив состоит из узлов, в каждом узле лежит какое-то вещественное значение и индекс следующего элемента списка в том же массиве. Нужно распечатать вещественные значения всех узлов в порядке их следования в связном списке.

Формат входных данных

В первой строке содержится два целых числа: N — количество узлов в массиве и F — индекс первого элемента связного списка ($0 \leqslant F < N \leqslant 100\,000$).

В каждой k-ой из оставшихся N строк записано содержимое k-ого элемента массива. Сначала записывается значение (вещественное число с трёмя или меньше знаками после точки, по модулю не превышает 10^4), а затем индекс следующего элемента.

Все индексы элементов нумеруются с нуля. В последнем узле связного списка индекс следующего элемента равен -1.

Формат выходных данных

Нужно вывести N строк, по одному вещественному числу в каждой строке — значения всех узлов в порядке их следования в связном списке. Вещественные числа нужно выводить хотя бы с тремя знаками после точки, например используя формат `"%0.3lf"`.

Пример

input.txt	output.txt
5 3 4.283 2 2.718 4 5.0 -1 1.111 1 3.141 0	1.111 2.718 3.141 4.283 5.000

Пояснение к примеру

В примере порядок элементов получается: 3 (1.111) -> 1 (2.718) -> 4 (3.141) -> 0 (4.283) -> 2 (5.0).

Задача 2. Односвязный список

Источник:	базовая
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

Требуется реализовать односвязный список на массиве. В каждом узле списка хранится строковое значение и индекс следующего узла. Дано начальное состояние списка, а также последовательность операций двух видов: добавление и удаление узла. Нужно выполнить все операции, и после этого вывести значения всех узлов списка в порядке их следования в цепи.

У каждого узла списка есть индекс в едином массиве, в котором всё хранится. В начальном состоянии списка есть N узлов, имеющих индексы от 0 до $N-1$ в порядке их задания. При добавлении нового узла он дописывается в конец массива. То есть первый добавленный узел имеет индекс N, второй добавленный — $N+1$, и так далее.

Нужно выполнять операции двух видов:

0. *Добавление узла.* При этом указывается индекс узла, после которого нужно вставить новый узел, и строковое значение для нового узла. Если индекс равен -1, то узел надо вставить в начало списка (перед самым первым элементом). После выполнения операции нужно вывести индекс нового узла (они назначаются по порядку, см. выше).
1. *Удаление узла.* При этом указывается индекс узла, и удалить нужно тот узел, который идёт сразу после него. Гарантируется, что узел с указанным индексом не последний. Если указан индекс -1, значит нужно удалить самый первый элемент списка (гарантируется, что он есть). После выполнения операции нужно вывести строковое значение удалённого узла.

В задаче используется “мультитест”: в одном входном файле записано много отдельных тестов.

Формат входных данных

В первой строке файла записано одно целое число T — количество тестов в файле. Далее в файле идут тесты (T штук) подряд, один за другим.

Первая строка теста начинается с трёх целых чисел: N — изначальное количество узлов в связном списке, F — индекс первого элемента списка и Q — количество операций, которые нужно выполнить ($0 \leqslant F < N \leqslant 10^5$, $0 \leqslant Q \leqslant 10^5$).

Затем идёт N строк, в которых описываются узлы связного списка в порядке увеличения индекса. Описание каждого узла состоит из его строкового значения и индекса следующего узла в списке (или -1, если следующего нет).

Наконец, в тесте идут Q строк, которые описывают операции над списком. В каждой строке сначала записан тип операции: 0 — добавление, 1 — удаление. Затем указан индекс узла, после которого нужно вставить/удалить узел. Если описывается операция вставки узла, то в конце также задано строковое значение нового узла.

У каждого узла строковое значение имеет длину от 1 до 7 символов включительно, и состоит из произвольных печатаемых символов ASCII (такие символы имеют код от 33 до 126 включительно).

Сумма N по всем тестах не превышает 10^5, аналогично для суммы всех Q.

Формат выходных данных

Для каждого теста нужно вывести следующее:

1. результаты выполнения операций (Q строк)
2. строка "===" (три знака равенства)
3. строковые значения всех узлов списка после выполнения операций (в порядке их следования в списке)
4. снова строка "==="

Пример

input.txt	output.txt
3 5 3 0 4.283 2 2.718 4 5.0 -1 1.111 1 3.141 0 1 0 5 zero -1 0 -1 one 0 -1 two 0 1 three 0 0 four 1 2 4 2 2 45 1 %drill# 3 &qw6: 0 *a-+r -1 1 1 0 2 \num\	=== 1.111 2.718 3.141 4.283 5.0 === 1 2 3 4 one === two three zero four === *a-+r 4 === &qw6: \num\ 45 %drill# ===

Пояснение к примеру

В примере три теста.

Первый тест полностью совпадает с примером к задаче “Распечатать список”: дано 5 узлов и 0 операций. Т.к. операций нет, в ответе записана сразу строка "===". Потом записан ответ как в задаче “Распечатать список” и ещё одна строка "===".

Во втором тесте изначально есть только один узел. Заметим, что в каждом узле значение равно индексу, записанному по-английски: `zero`, `one`, `two`, `three`, `four`.

Первые четыре операции задают вставку элементов: сначала вставляется два узла в начало, получается список `two`, `one`, `zero`. Затем вставляется узел после узла `one` и ещё один узел после узла `zero`, получается список `two`, `one`, `three`, `zero`, `four`. Последняя операция удаления: удаляется узел, который стоит **после** узла `two`, то есть узел `one`.

Задача 3. Хранение строк

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

Нужно реализовать систему хранения строк в динамической памяти.

Система должна обрабатывать три вида запросов/операций:

0. *Создать строку*: указывается длина создаваемой строки и собственно содержимое строки. Нужно выделить блок памяти при помощи `malloc` и записать в него содержимое.
1. *Удалить строку*: указывается идентификатор ранее созданной строки. Нужно освободить память, выделенную ранее для этой строки, при помощи `free`.
2. *Вывести строку*: указывается идентификатор ранее созданной строки. Нужно распечатать содержимое этой строки в выходной файл.
3. *Сколько символов в строке*: указывается идентификатор ранее созданной строки и один символ. Нужно вывести в выходной файл одно целое число: сколько раз этот символ встречается в указанной строке.

Идентификатором строки является номер запроса на её создание в общей нумерации запросов. При обращении к строке по идентификатору гарантируется, что эта строка ещё **не** была удалена.

Замечание: Не все созданные строки удаляются в запросах. В целях аккуратности удалите все остающиеся строки при помощи `free` в конце программы.

Формат входных данных

В первой строке записано целое число N — количество количество запросов ($1 \leqslant N \leqslant 10^5$). В остальных N строках описаны запросы, по одному запросу в строке.

Запрос начинается с целого числа t, обозначающего тип запроса. Если $t = 0$, то это запрос создания, и тогда далее указана длина строки l и сама строка ($1 \leqslant l \leqslant 10^5$). Если $t = 1$, то это запрос удаления, а если $t = 2$ — то это запрос на вывод. В обоих случаях далее указано целое число k — идентификатор строки ($0 \leqslant k < N$). Если $t = 3$, то это запрос о количестве символов, и тогда далее указан идентификатор строки k и ещё один символ через пробел.

Все строки состоят из произвольных печатаемых символов ASCII (коды от 33 до 126 включительно). То же верно и про символы в запросах на количество.

Сумма всех длин l по всем запросам создания не превышает $5 \cdot 10^5$. Суммарный размер всех строк, которые вам нужно вывести, не превышает $5 \cdot 10^5$. Сумма длин строк по всем запросам о количестве символов не превышает 10^8.

Пример

input.txt	output.txt
12 0 3 aba 2 0 3 0 a 3 0 b 1 0 0 6 malloc 0 5 %d%s% 2 1 2 2 1 1 3 2 % 2 2	aba 2 1 malloc %d%s% 3 %d%s%

Пояснение к примеру

Сначала создаётся строка “aba” с идентификатором 0. Далее она распечатывается, и выводится количество букв 'a' и 'b' в ней. Наконец, нулевая строка удаляется.

Затем создаются ещё две строки: строка “malloc” и строка “%d%s%” с идентификаторами 1 и 2 соответственно. Потом они распечатываются и строка “malloc” удаляется. В конце выводится количество процентов в строке “%d%s%” и она распечатывается ещё раз.

Задача 4. Разделить на слова

Источник:	основная*
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

Требуется реализовать функции для выделения слов из заданной строки. Слова отделены друг от друга символами-разделителями, которые передаются в строку как параметр.

Сигнатура функций должна быть такой:

```
typedef struct Tokens_s {
    int num;          //количество слов в строке
    char **arr;       //массив слов (каждый элемент -- строка, т.е. char*)
} Tokens;
//tokens: структура, в которую нужно положить результаты
//str: строка, в которой нужно искать слова
//delims: все символы-разделители в виде строки
void tokensSplit(Tokens *tokens, const char *str, const char *delims);
//удаляет все ресурсы внутри tokens
void tokensFree(Tokens *tokens);
```

Память для массива слов `tokens->arr` следует выделять динамически ровно на столько элементов, сколько слов в строке. Поскольку заранее это количество неизвестно, то нужно запускать алгоритм два раза: первый раз, чтобы узнать сколько слов, и второй раз, чтобы записать слова в массив.

Ваша реализация вышеописанных функцию должна работать согласно следующим договорённостям:

1. Вызывающий гарантирует, что параметр `tokens` **не** нулевой (для обеих функций) и указывает на структуру `Tokens`.
2. Если при вызове `tokensSplit` указатель `tokens->arr` нулевой, то внутри функции нужно только посчитать количестов слов и записать его в `tokens->num`.
3. Если при вызове `tokensSplit` указатель `tokens->arr` **не** нулевой, то он должен указывать на массив, в который точно войдут все слова. В этом случае реализация функции должна записать в `tokens->num` количество слов, а в `tokens->arr[i]` записать i-ое слово, самостоятельно выделив под него память с помощью `malloc`.
4. Функция `tokensFree` должна удалять массив слов и сами строки-слова с помощью `free`. При этом программа должна работать корректно, даже если эту функцию случайно вызовут два или три раза подряд.

Таким образом, вызывающий может завести структуру `tokens`, потом определить количество слов первым вызовом `tokensSplit`, затем выделить память на массив `tokens->arr`, и, наконец, найти все слова вторым вызовом `tokensSplit`.

С помощью этих функций нужно решить тестовую задачу. Дана одна строка длиной до 10^6, состоящая из букв латинского алфавита и знаков препинания четырёх типов: точка, запятая, точка с запятой, двоеточие. Нужно найти слова в этой строке, состоящие из букв, и вывести их в файл в таком же формате, как показано в примере.

Пример

input.txt	output.txt
..ko,.Privet:kreved,.,;ko:;,.	4 ko Privet kreved ko

Комментарий

Следует выводить слова ровно в том порядке, в котором они встречаются в строке.

Задача 5. Растущий массив

Источник:	основная
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

В данной задаче нужно реализовать массив переменного размера, в который можно дописывать элементы, не зная заранее его окончательный размер. Используя эту структуру данных, нужно решить приведённую ниже задачу.

В первой строке записано целое число N — количество записей ($1 \leqslant N \leqslant 2 \cdot 10^5$). В остальных N строках содержатся записи, по одной в строке.

Для каждой записи указаны ключ и значение через пробел. Ключ — это целое число в диапазоне от 0 до 10^6 включительно, а значение — это строка от одного до семи символов включительно, состоящая только из маленьких букв латинского алфавита.

Требуется вывести ровно те же самые N записей, но в другом порядке. Записи должны быть упорядочены по возрастанию ключа. Если у нескольких записей ключ равный, то нужно упорядочить их в том порядке, в котором они встречаются по входном файле.

Важно: Решать задачу **нужно** следующим образом (другие решения засчитываться **не** будут). Нужно завести 10^6 **массивов** переменного размера, и в каждый k-ый массив складывать все записи с ключом, равным k. После раскидывания записей по массивам достаточно будет пробежаться по массивам в порядке увеличения k и распечатать их.

Пример

input.txt	output.txt
7 3 qwerty 3 string 6 good 1 a 3 ab 2 hello 5 world	1 a 2 hello 3 qwerty 3 string 3 ab 5 world 6 good

Пояснение к примеру

В примере 7 записей с ключами 1, 2, 3, 5 и 6 — именно в таком порядке записи и выведены в выходном файле. Обратите внимание, что есть три записи с ключом 3: `qwerty`, `string`, `ab`. Они выведены ровно в том порядке, в котором они идут во входном файле.

Задача 6. Построение списка добавлением в голову

Источник:	основная
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

По заданной последовательности целых чисел построить односвязный динамический список. Каждое новое число добавлять в начало списка. Затем пройти по построенному списку и посчитать количество отрицательных чисел, входящих в список, кратных 7. После этого память освободить.

Формат входных данных

Входной файл содержит заданную последовательность целых чисел. Числа в файле записаны через пробел. Их величина по модулю не превосходит 1000. Количество чисел может изменяться от 1 до 1000.

Формат выходных данных

В выходной файл нужно вывести одно целое число — количество отрицательных чисел, кратных 7.

Пример

input.txt	output.txt
10 -1 14 8 -21 -35 35 16	2

Задача 7. Построение списка добавлением в хвост

Источник:	основная
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

По заданной последовательности целых чисел построить односвязный динамический список. Каждое новое число добавлять в конец списка. Затем пройти по построенному списку и посчитать среднее арифметическое чисел, входящих в список. После этого память освободить.

Формат входных данных

Входной файл содержит заданную последовательность целых чисел. Числа в файле записаны через пробел. Их величина по модулю не превосходит 1000. Количество чисел может изменяться от 1 до 1000.

Формат выходных данных

В выходной файл нужно вывести одно целое число — среднее арифметическое элементов списка.

Пример

input.txt	output.txt
1 5 4 6 3	3

Задача 8. Построение упорядоченного списка

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

По заданной последовательности целых чисел построить односвязный динамический список. Каждое новое число добавлять в список так, чтобы он оставался упорядоченным по возрастанию. Если такое число в списке уже есть, то его не добавлять. Затем пройти по построенному списку от начала до конца и распечатать его элементы. После этого память освободить.

Формат входных данных

Входной файл содержит заданную последовательность целых чисел. Числа в файле записаны через пробел. Их величина по модулю не превосходит 1000. Количество чисел может изменяться от 1 до 1000.

Формат выходных данных

В выходной файл нужно вывести упорядоченную последовательность заданных чисел без повторений. Числа выводить через пробел в одну строку.

Пример

input.txt	output.txt
10 -1 14 8 -21 -3 35 16 -3 10	-21 -3 -1 8 10 14 16 35

Задача 9. Сумма чётных 2

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

Требуется реализовать функцию `foreach`, которая пробегает по всем элементам заданного **массива** целых чисел и вызывает указанный callback для каждого из этих элементов. Аналогичную функцию `foreach` нужно реализовать и для **связного списка**. Передаваемый callback должен поддерживать контекст.

Пример сигнатуры функции `foreach` для массива приведён ниже. Для связного списка реализуйте функцию `foreach` с аналогичной сигнатурой, чтобы можно было передавать в неё те же самые callback-функции.

```
//тип указателя на функцию, которую можно передавать как callback
typedef void (*callback)(void *ctx, int *value);
//здесь ctx -- это контекст, который передаётся в func первым аргументом
//последние два параметра задают массив, по элементам которого нужно пройтись
void arrayForeach(void *ctx, callback func, int *arr, int n);
```

Эти две функции требуется применить для решения следующей тестовой задачи: *дана последовательность целых чисел, требуется посчитать сумму всех её чётных элементов.* Запишите данную последовательность как в массив, так и в связный список, и вызовите в каждом случае `foreach`. Функция-callback, которую вы передаёте в `foreach`, не должна обращаться к глобальным переменным: все необходимые для её работы данные передавайте через контекст.

Формат входных данных

В первой строке содержится целое число N — количество элементов последовательности ($1 \leqslant N \leqslant 100$). Во второй строке записано N целых чисел через пробел — сама последовательность. Все элементы последовательности по абсолютной величине не превышают 100.

Формат выходных данных

Выведите два целых числа: сумму всех чётных элементов последовательности. Первое число должно соответствовать сумме, вычисленной с помощью `foreach` по массиву, а второе — сумме, вычисленной с помощью `foreach` по связному списку.

Пример

input.txt	output.txt
8 2 3 7 6 8 3 1 2	18 18

Задача 10. 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).