**Линейные структуры данных.**

**Теоретический блок 1: Стек. (LIFO) [1]**

Кроме стандартных структур данных: список, множество, ассоциативный массив (словарь), часто используют и ряд других структур. Особенность каждой из них состоит в способе хранения данных и алгоритме доступа к элементам структуры. В каждой структуре есть ограничения — то, что структура "делает плохо". Например, добавление элемента в конец списка выполняется быстро, а удаление первого элемента выполняется за длину списка. Познакомимся с несколькими структурами, которые называют "линейными структурами данных", потому что здесь элементы хранятся в виде последовательности, один за другим. Например, список в языкe Python и вектор в языке C++ - линейные структуры данных, где есть операция доступа к i-му по счёту элементу, выполняемая за О(1), но операции удаления элемента из начала или вставка элемента в середину выполняются долго.

*Стек* (stack) — это линейная структура данных, в которой элементы добавляются и удаляются только с одного конца — вершины стека. Можно представить стек как стопку тарелок. Можно положить или снять тарелку только для вершины стопки, то есть за О(1), но если нужна нижняя тарелка, то только за О(n).

То есть если мы положим в стек последовательно числа 1, 2, 3, а потом удалим из стека последний элемент, то в стеке останутся 1 и 2. Если мы затем положим в стек числа 4 и 5, а затем последовательно удалим из стека все элементы, то они будут удаляться в порядке 5, 4, 2, 1.

Таким образом, стек поддерживает следующие операции (все выполняются за O(1):

|  |  |
| --- | --- |
| * Добавить элемент в конец стека (push) * Удалить последний элемент из стека (pop) | * Узнать значение последнего элемента стека (top) * Узнать размер стека |

Реализация стека в Python-е выполняется через список, реализующий операции добавления/удаления в последней позиции (вершине стека), за O(1). Добавление/чтение с удалением/просто чтение элемента есть методы append: stack.append(x) / x = stack.pop()/stack[-1]. Функция len(stack) вернёт число элементов.

**Стек с поддержкой минимального элемента**

Давайте реализуем стек, который будет содержать ещё одну операцию — "узнать значение наименьшего элемента во всём стеке". Это так называемый "стек с минимумом". Эта задача решается следующим образом: мы будем хранить два стека: в одном будут сами значения, а в другом стеке для каждого элемента будет храниться минимальное значение во всём стеке, когда мы добавили этот элемент.

Например, пусть мы положили в стек число 5. Оно же будет наименьшим элементом. Теперь положим в стек число 4. Наименьшим элементом во всём стеке теперь стало 4. Положим это число во второй стек. Теперь добавим в стек число 7. Наименьший элемент в стеке до этого — это 4, мы добавили большее число, поэтому значение минимума осталось равно 4, положим его во второй стек. Теперь добавим в стек число 2. Это число меньше минимума во всём стеке, поэтому во второй стек добавим число 2.

Если теперь из первого стека удалять элементы и одновременно удалять элементы из второго стека, то на вершине второго стека всегда будет наименьшее значение в первом стеке.

**Реализация функций**

stack = []; stack\_min = []

**def** push(elem):

stack.**append**(elem)

**if** stack\_min **and** stack\_min[-1] < elem: stack\_min.append(stack\_min[-1])

**else**: stack\_min.**append**(elem)

**def** top():

**return** stack[-1]

**def** get\_min():

**return** stack\_min[-1]

**def** pop():

stack.**pop**();

stack\_min.**pop**()

**Задача о правильной скобочной последовательности**

Пусть дана последовательность из трёх видов скобок: ()(), [][], {}{}. Проверим ее на правильность.

Правильной является последовательность, допустимая при записи арифметического выражения.

Пример правильной последовательности: ([]())[]([]())[]

Примеры неправильных последовательностей:

|  |  |  |  |
| --- | --- | --- | --- |
| (()(() | (}(} | )()( | ([)]([)] |

Формально правильная скобочная последовательность определяется таким образом:

* пустая строка является правильной скобочной последовательностью;
* правильная последовательность, взятая в скобки одного типа, — правильная последовательность;
* правильная последоватеьность с приписаной правильной последовательностью — тоже правильная.

Можно сформулировать простой принцип проверки скобочной последовательности на правильность – там всегда есть пара открывающаяся/закрывающаяся скобка одного вида, где внутри нет других скобок. Удалив их, мы опять найдём такую пару и т.д. Если в итоге удалим все скобки, то последовательность правильная, а если нет - неправильная. Но здесь сложность алгоритма O(n2). Действительно, рассмотрим правильную последовательность, где сначала идут n/2 открывающихся, потом n/2 закрывающихся скобок того же типа. Каждый раз удаляем 2 скобки из середины строки за O(n), поскольку нужно сдвигать половину строки. Это удаление будет выполняться n/2 раз, поэтому общая сложность будет O(n2).

Можно улучшить этот алгоритм, если использовать стек. Закрывающая скобка должна быть парной к последней открывающей. То есть если мы будем хранить последовательность из открывающих скобок, то закрывающая скобка удаляет последнюю открывающую, если она того же вида, в противном случае последовательность неправильная. Это означает, что последовательность открывающих скобок, которые не были ещё закрыты, представляет собой стек. Если мы встретили открывающую скобку, то она добавляется в конец стека. Если мы встретили закрывающую скобку, то должны выполняться следующие условия: стек не пуст и на вершине стека хранится скобка, парная данной закрывающей скобке. Если после обработки всех скобок стек оказался пуст, то последовательность правильная.

В каком случае последовательность будет неправильной? Если для очередной закрывающей скобки не нашлось парной открывающей, то есть если мы встретили закрывающую скобку, то либо стек пуст, либо на вершине его находится скобка другого вида. Или если мы для какой-то открывающей скобки не нашли закрывающую, это означает, что после рассмотрения всех скобок стек оказался не пуст.

Сложность алгоритма O(n), так как каждую скобку мы кладём в стек или удаляем из стека не более одного раза, а данные операции на стеке работают за O(1).

**Задача 1. Интерпретатор команд для стека**

stack=[]

**while** **True**:

x=**list**(**input**().**split**())

**if** **len**(x)==2:

**if** x[0]=='push': stack.**append**(**int**(x[1])); **print**('ok')

**else**: **print**('error')

**else**:

**if** x[0]=='exit': **print**('bye'); **break**

**elif** x[0]=='pop':

**if** **len**(stack)==0: **print**('error')

**else**: **print**(stack.**pop**())

**elif** x[0]=='back':

**if** **len**(stack)==0: **print**('error')

**else**: **print**(stack[-1])

**elif** x[0]=='size': **print**(**len**(stack))

**elif** x[0]=='clear': stack=[]; **print**('ok')

**else**: **print**('error')

**Задача 2. Правильная скобочная последовательность**

stack=[]; f=**True**; x=**input**()

**for** i **in** **range**(**len**(x)):

**if** x[i]=='[' **or** x[i]=='(' **or** x[i]=='{': stack.**append**(x[i])

**else**:

**if** **len**(stack)==0: f=**False**; **break**

**else**:

**if** (stack[-1]=='[' **and** x[i]==']')**or**(stack[-1]=='{' **and** x[i]=='}')**or**(stack[-1]=='(' **and** x[i]==')'): stack.**pop**()

**else**: f=**False**; **break**

**if** **len**(stack)==0 **and** f: **print**('yes')

**else**: **print**('no')

**Задача 3. Постфиксная запись**

В постфиксной записи (или обратной польской записи) операция записывается после двух операндов. Например, сумма двух чисел A и B записывается как A B +. Запись B C + D ∗ обозначает привычное нам (B+C)∗D, а запись A B C + D ∗ + означает A+(B+C)∗D. Такая запись не требует скобок и соглашений о приоритете операторов — всё выполняется подряд слева направо.

**Входные данные:** В одной строке дано постфиксное выражение, содержащее цифры и операции +, −, ∗. Цифры и операции разделены пробелами. В конце строки возможно произвольное число пробелов.

**Выходные данные:** Результат вычисления записанного выражения (по модулю не превосходит 2⋅109).

stack=[]; x=**list**(**input**().**split**())

**for** i **in** **range**(**len**(x)):

**if** x[i]==' ':**break** #пробел

**if** x[i]=='+': b=**int**(stack.**pop**()); a=**int**(stack.**pop**()); stack.**append**(a+b)

**elif** x[i]=='-': b=**int**(stack.**pop**()); a=**int**(stack.**pop**()); stack.**append**(a-b)

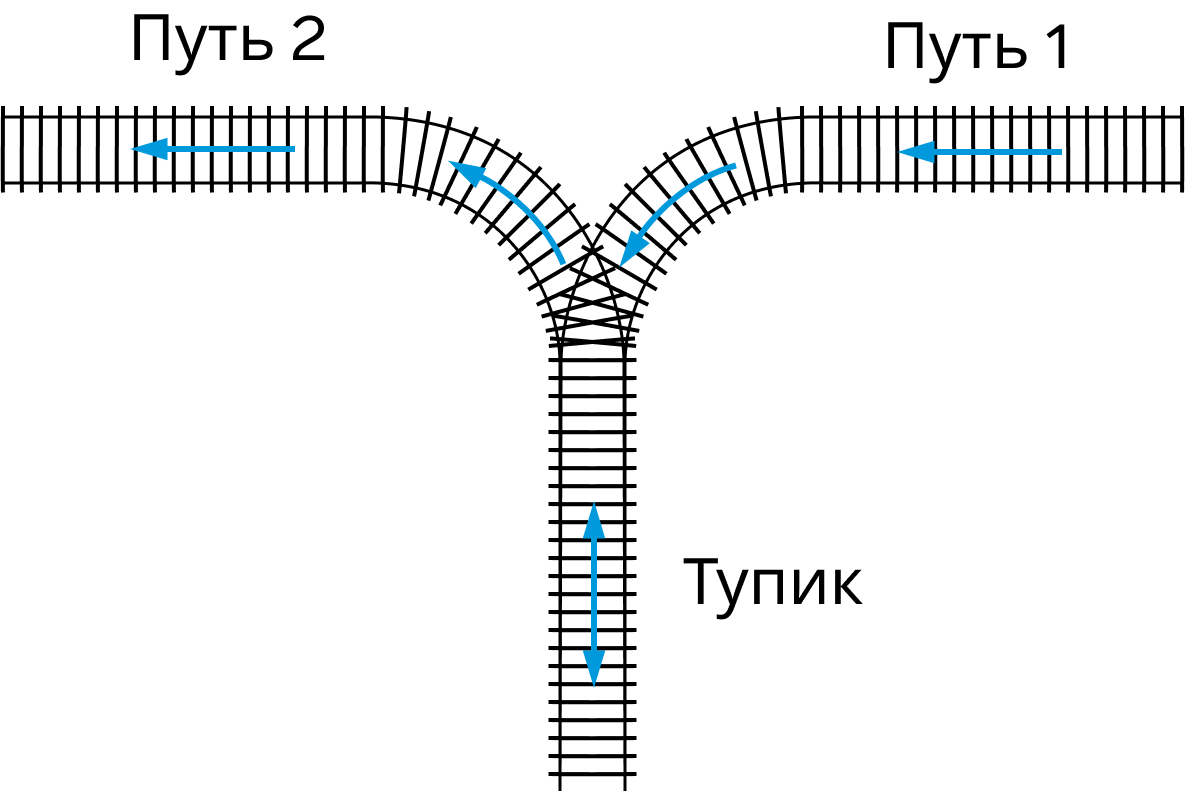
**elif** x[i]=='\*': b=**int**(stack.**pop**()); a=**int**(stack.**pop**()); stack.**append**(a\*b)

**else**: stack.**append**(x[i])

**print**(**int**(stack.**pop**()))

**Задача 4. Сортировка вагонов**

К тупику со стороны пути 1 (см. рисунок) подъехал поезд. Разрешается отцепить от поезда один или несколько первых вагонов и завезти их в тупик (при желании можно завезти в тупик и весь поезд). После этого часть из этих вагонов вывезти в сторону пути 2, а затем завезти в тупик ещё несколько вагонов и снова часть их вывезти в сторону пути 2, и так далее, но каждый вагон может лишь один раз заехать с пути 1 в тупик, а затем один раз выехать из тупика на путь 2. Заезжать в тупик с пути 2 или выезжать из тупика на путь 1 запрещается, и нельзя также с пути 1 попасть на путь 2, не заезжая в тупик.



Известно начальный порядок вагонов. Требуется с помощью указанных операций сделать так, чтобы вагоны следовали на пути 2 по порядку - от первого до последнего.

**Входные данные:** В первой строке входных данных указано количество вагонов в поезде (1⩽N⩽2000). Во второй строке через пробел идут номера вагонов в порядке от головы поезда, едущего по пути 1 в сторону тупика. Вагоны пронумерованы от 1 до N без повторов.

**Выходные данные:** Если можно упорядочить вагоны по пути 2, вывести YES, а если нет, то NO.

**Примечание:** простая, но полезная задача…

stack=[]; n=**int**(**input**()); x=**list**(**map**(**int**, **input**().**split**())); nxt=1; y=[]; f=**True**

**for** i **in** **range**(n):

**while** **len**(stack)>0 **and** nxt==stack[-1]:

y.**append**(stack.**pop**()); nxt+=1

**if** x[i]==nxt: y.**append**(x[i]); nxt+=1

**else**: stack.**append**(x[i])

**while** **len**(stack)>0: y.**append**(stack.**pop**())

**for** i **in** **range**(n):

**if** y[i]!=i+1: f=**False**; **break**

**if** f:**print**('YES')

**else**:**print**('NO')

**Теоретический блок 2: Очередь. (FIFO) [2]**

Другая структура данных, подобная стеку, — очередь (queue). Здесь новые элементы добавляются в конец очереди, а удаляются элементы из ее начала. Очередь поддерживает следующие операции:

|  |  |
| --- | --- |
| Добавить элемент в очередь (push);  Узнать значение первого элемента очереди (front); | Удалить элемент из очереди (pop);  Узнать размер очереди (size). |

Все операции должны выполняться за O(1). Для этого будем запоминать позицию головы очереди. Тогда при удалении элемента из очереди указатель ее головы увеличивается на единицу, а место удалённых элементов будет пропущенной областью памяти. Чтобы бороться с этой проблемой, есть два подхода. Один подход - логически замкнуть массив в кольцо, то есть за последним элементом будет идти первый. Здесь когда очередь достигнет конца массива, то она начнёт дописываться в его начало, если там есть достаточное количество ранее удаленных элементов, то есть размер массива должен позволять разместить столько элементов, сколько в максимуме могут храниться в очереди.

Другой подход — регулярное сжатие свободного места. Если в начале очереди накопилось слишком много пустого места (например, столько же, сколько элементов в самой очереди), то можно сразу удалить все эти ячейки, скопировав все элементы очереди в начало массива. То есть мы будем операцию реального удаления элементов выполнять редко, но удалять сразу много элементов, тогда средняя сложность в пересчёте на одну операцию удаления будет O(1).

Есть еще один интересный способ реализации очереди. Если последовательность элементов положить в стек, а потом излечь, то она развернётся в обратном порядке. Если это повторить ещё раз, то последовательность элементов опять развернётся и порядок будет исходным. Очередь же не меняет порядок элементов, поэтому можно реализовать очередь с использованием двух стеков.

Возьмём два стека: s1 и s2. Операцию добавления будем всегда делать в первый стек. А удалять элементы будем из второго стека. Но при этом второй стек изначально пустой, все добавленные элементы будут находиться в первом стеке. *Если необходимо удалить элемент, а второй стек пуст, то все элементы из первого стека перенесём во второй стек. При этом самый первый добавленный элемент, который был внизу первого стека, окажется на вершине второго стека и выйдет из очереди первым.* Если s2 не пуст, достаём элементы из него. Если s2 снова пуст, повторяем ту же операцию.

Кажется, что здесь изменится вычислительная сложность операций очереди, поскольку удаление может выполняться долго, так как все элементы перекладыются из одного стека в другой. Но такие операции будут происходить редко, и средняя сложность в пересчете на одну операцию невелика. Действительно, пусть мы n элементов добавили в очередь и n элементов удалили, как-то чередуя эти операции. Тогда каждый элемент сначала был добавлен в стек s1, после - перенесён в s2, затем - удалён из s2, то есть мы выполнили 2 операции добавления и 2 операции удаления, то есть 4n операций со стеком для реализации 2n операций с очередью, тогда на одну операцию в среднем требуется O(1) операций со стеком.

Эта идея может показаться сложной для реализации обычной очереди, но подходит для реалиазации другой структуры — очереди с минимумом или максимумом, то есть очереди, которая поддерживает ещё один запрос — узнать значение наименьшего или наибольшего элемента в очереди. Сложно реализовать операцию взятия минимума, когда из структуры данных удаляется один элемент. Пусть мы знали, что наименьшее значение всех элементов в структуре равно m и из структуры был удалён элемент, равный m. Какое стало теперь наименьшее значение во всей структуре? Оно может оказаться как m, так и больше m.

Ранее мы рассмотрели реализацию стека с минимумом, теперь для очереди с минимумом будем использовать два стека с минимумом, то есть минимумы в стеках будут поддерживаться автоматически. Значение минимума очереди равно минимум из минимальных значений в каждом из двух стеков.

**Задача 7. Игра в пьяницу (две очереди)**

В игре в пьяницу карточная колода (карточки с цифрами от 0 до 9) раздаётся поровну двум игрокам. Каждый вскрывает верхнюю карту, и тот, чья карта старше, и учитывая, что «0 старше 9», обе вскрытые карты кладет под низ своей колоды – сначала карту игрока 1, а затем – игрока 2. Оставшийся без карт — проигрывает. Напишите программу, моделирующую игру в пьяницу и определяющую выигравшего.

**Входные данные:** Программа получает на вход две строки по 5 чисел: первая - разделённых пробелами, номера карт первого игрока, вторая — аналогично для второго игрока. Карты перечислены сверху вниз, то есть каждая строка начинается с той карты, которая будет открыта первой.

**Выходные данные:** Программа должна вывести номер выигравшего - first или second, и, через пробел, количество слелпннвх ходов, а если за 106 ходов игра не закончилась, то вывести слово botva.

|  |  |  |
| --- | --- | --- |
| **Примеры:** | Ввод:  1 3 5 7 9  2 4 6 8 0 | Вывод  second 5 |

h1=0;h2=0;c=0

q1=**list**(**map**(**int**, **input**().**split**()))

q2=**list**(**map**(**int**, **input**().**split**()))

**while** **len**(q1)-h1>0 **and** (**len**(q2)-h2>0) **and** (c<1000001):

p1=q1[h1]; p2=q2[h2];h1+=1;h2+=1;c+=1

**if** ((p1==0) **and** (p2==9)):q1.**append**(p1);q1.**append**(p2)

**elif** ((p2==0) **and** (p1==9)):q2.**append**(p1);q2.**append**(p2)

**elif** p1>p2:q1.**append**(p1);q1.**append**(p2)

**else**:q2.**append**(p1);q2.**append**(p2)

**if** c>1000000:**print**('botva')

**elif** **len**(q1)-h1==0:**print**('second',c)

**else**:**print**('first',c)

**Задача 8. Очередь с поддержкой минимума**

**Входные данные**

Первая строка содержит число n — количество операций. В следующей строке - через пробел, числа ai (0≤ai≤10000). Если ai>0, то ai добавить в очередь, а если 0, то это команда удаления из очереди.

**Выходные данные**

На каждый запрос удаления элемента из очереди необходимо вывести значение минимального элемента очереди (учитывая удаляемый элемент). Если запрос удаления из пустой очереди, то вывести −1.

**Примеры.** Ввод: 9 5 4 3 6 0 0 0 0 0; Вывод: 3 3 3 6 -1

**Примечание:** следует аккуратно рассмотреть все случаи, быстродействие, в среднем, 4\*n, то есть O(1)

n=**int**(**input**())

s1=[];s2=[];m1=[];m2=[] #стеки для записи/чтения и текущих минимумов

p=**list**(**map**(**int**, **input**().**split**())) #чтение исходных данных

**for** i **in** **range**(n):

**if** p[i]>0: # если push - запрос

s1.**append**(p[i]) #добавляем к s1

**if** (**len**(m1)==0) **or** m1[-1]>p[i]: m1.**append**(p[i]) #корректируем текущий

**else**:m1.**append**(m1[-1]) # минимум (m1[-1]) для s1

**else**: # если pop - запрос

**if** **len**(s2)>0: #если pop-стек (s2) – не пуст

s2.**pop**()

**if** **len**(m1)>0: t=**min**(m1[-1],m2[-1]) #определяем текущий минимум в s2 + s1

**else**: t=m2[-1]

m2.**pop**(); **print**(t) #вывод текущего минимума, коррекция стека минимумов m2 (m1-не менятся)

**else**: #если pop-стек (s2) – пуст, то

**if** **len**(s1)==0:**print**(-1) #если туда пока нечего добавлять, то выводим -1

**else**: #а если есть, что добавить, то переписываем s1 в s2 в обратном порядке, сформировав m2

s2.**append**(s1.**pop**()); m2.**append**(s2[-1]); m1.**pop**()

**while** **len**(s1)>0:

s2.**append**(s1.**pop**()); m1.**pop**()

**if** s2[-1]<m2[-1]: m2.**append**(s2[-1])

**else**: m2.**append**(m2[-1])

s2.**pop**() #после переписывания выполняем нужные действия с pop - стеками

**if** **len**(m1)>0:t=**min**(m1[-1],m2[-1])

**else**: t=m2[-1]

m2.**pop**(); **print**(t)

**Теоретический блок 3: Дек (двусвязный список) [3]**

Если есть **стек**, где элементы кладут и извлекают с одного и того же конца, и **очередь**, где элементы кладут в один конец, а извлекают с другого, то логично существование структуры данных, где класть и извлекать можно с обоих концов: **дек** (*deque* или *double-ended queue*. Дек поддерживает операции:

|  |  |
| --- | --- |
| * Добавить в начало дека - O(1) * Добавить в конец дека - O(1) * Удалить из начала дека - O(1) * Удалить из конца дека - O(1) | * Узнать значение последнего элемента дека - O(1) * Узнать значение первого элемента дека - O(1) * Узнать размер дека - O(1)   **Эти же операции, но с серединой дека – O(n)** |

Операции получения значения элемента и удаления элемента могут быть совмещены. Дек можно реализовать так же, как и очередь, — на кольцевом массиве, но нужно учесть, что при добавлении элементов дек может ползти в любую сторону, а потому реализация сложнее, чем для очереди. Для реализации дека в Python есть класс **deque** модуля **collections**. Приведём пример работы с деком в Python:

**from** collections **import** deque

d = **deque**(); d.**append**(5); d.**appendleft**(4); d.**appendleft**(3); **print**(len(d), d.**pop**(), d.**popleft**());

Вывод: 3 5 3

Как видим, у дека в Python есть следующие операции:

|  |  |
| --- | --- |
| * **append**(x) - добавляет x в конец; * **appendleft**(x) - добавляет x в начало; * **clear**() - очищает дек. | * x = **pop**() - удаляет и возвращает последний элемент дека; * x = **popleft**() - удаляет и возвращает первый элемент дека; |

Чтобы обратиться к левому элементу дека, нужно, как в списке, использовать обращение по индексу d[0], а чтобы обратиться к самому последнему элементу — использовать отрицательный индекс d[−1]. Может показаться, что можно использовать и любой индекс, например, d[1], d[2] и так далее. Это верно, но дек в Python реализован так, что эти операции выполняются быстро только для элементов у краёв дека. Обращение к одному элементу выполняется за время, пропорциональное расстоянию до ближайшего конца дека, поэтому не следует использовать эту операцию для элементов, которые удалены от края дека.

Обратите внимание, что вызов инструкции pop() и popleft() на пустом деке даст ошибку.

Все эти операции выполняются за O(1), но у дека на Python есть операции, выполняемые за O(n): 1)count(x) — количество элементов, равных xx; 2)remove(value) удаляет первое вхождение value.

**Задача 9. Дек с защитой от ошибок (разрешается использование библиотечных модулей)**

Напишите программу, моделирующую работу дека, реализовав все указанные методы. Программа считывает цепочку команд и выполняет ту или иную операцию. После выполнения каждой команды программа должна вывести одну строчку. Команды для программы:

* **push\_front** — добавить (положить) в начало новый элемент. Программа выводит ok;
* **push\_back** — добавить (положить) в конец новый элемент. Программа выводит ok;
* **pop\_front** — извлечь первый элемент. Программа выводит его значение (или *error*);
* **pop\_back** — извлечь последний элемент. Программа выводит его значение (или *error*);
* **front** — значение первого элемента (без удаления). Программа выводит значение (или *error*);
* **back** — значение последнего элемента (без удаления). Программа выводит значение (или *error*);
* **size** — вывод количество элементов в деке;
* **clear** — очистить дек (удалить из него все элементы) и вывести ok;
* **exit** — программа выводит bye и завершает работу.

Количество элементов в деке всегда меньше 101. Перед исполнением операций pop\_front, pop\_back, front, back программа должна проверять дек на пустоту. Если при запросе на выполнение этих команд дек пуст, то программа должна вместо числового значения вывести строку error.

**Входные данные:** Набор команд для дека, по одной в строке. Среди них обязательно есть команда *exit*.

**Выходные данные:** Требуется вывести протокол работы дека, по одному сообщению в строке.

|  |  |  |
| --- | --- | --- |
| **Примеры:** | Ввод  size  push\_back 1  size  push\_back 2  size  push\_front 3  size  exit | Вывод  0  ok  1  ok  2  ok  3  bye |

**from** collections **import** deque

d=**deque**()

**while** **True**:

x=**list**(**input**().**split**())

if **len**(x)==2:

**if** x[0]=='push\_front': d.**appendleft**(**int**(x[1])); **print**('ok')

**elif** x[0]=='push\_back':d.**append**(**int**(x[1])); **print**('ok')

**else**:

**if** x[0]=='exit': **print**('bye'); **break**

**elif** x[0]=='pop\_front':

**if** **len**(d)==0: **print**('error')

**else**: **print**(d.**popleft**())

**elif** x[0]=='pop\_back':

**if** **len**(d)==0: **print**('error')

**else**: **print**(d.**pop**())

**elif** x[0]=='front':

**if** **len**(d)==0: **print**('error')

**else:** **print**(d[0])

**elif** x[0]=='back':

**if** **len**(d)==0: **print**('error')

**else**: **print**(d[-1])

**elif** x[0]=='size': **print**(len(d))

**elif** x[0]=='clear': d.**clear**(); **print**('ok')

### Задача 10. Гоблины и шаманы

Гоблины Мглистых гор любях ходить к шаманам, а потому образуются длинные, шумные очереди, которыя мешают шаманам, в связи с чем те установили правила касательно порядка в очереди.

Обычные гоблины должны вставать в конец очереди, а привилегированные, знающие особый пароль, встают ровно в ее середину, причем при нечетной длине очереди они встают сразу за центром.

Так как гоблины известны своим непочтительным отношением ко всяческим правилам и законам, шаманы попросили вас написать программу, которая бы отслеживала порядок гоблинов в очереди.

**Входные данные:** В первой строке входных данный записано число N (1≤N≤105) — количество запросов к программе. Следующие N строк содержат описание запросов в формате:

* "+ i" — гоблин с номером i (1≤i≤N) встает в конец очереди.
* "\* i" — привилегированный гоблин с номером i встает в середину очереди.
* "-" — первый гоблин из очереди уходит к шаманам. Гарантируется, что очередь не пуста.

**Выходные данные:** Для запроса типа "-" программа выводит номер гоблина, входящего к шаманам.

|  |  |  |
| --- | --- | --- |
| **Примеры** | Ввод  7  + 1  + 2  -  + 3  + 4  -  - | Вывод  1  2  3 |
| **Ограничения:** Время выполнения: 3 секунды  (задачу можно решать разными способами) |

**from** collections **import** deque

n=**int**(**input**()); d1=**deque**();d2=**deque**()

**for** i **in** **range**(n):

t=**list**(**input**().**split**())

**if** **len**(t)>1:

**if** t[0]=='+': d2.**append**(**int**(t[1]))

**else**: d1.**append**(**int**(t[1]))

**else**: **print**(d1.**popleft**())

**if** (**len**(d1)+**len**(d2))%2==0: # выравниваем деки так, чтобы \* писать в конец d1, а + в конец d2

**while** **len**(d1)<**len**(d2):d1.**append**(d2.**popleft**())

**while** **len**(d2)<**len**(d1):d2.**appendleft**(d1.**pop**())

**else**:

**while** **len**(d1)<**len**(d2):d1.**append**(d2.**popleft**())

**while** **len**(d2)<**len**(d1)-1:d2.**appendleft**(d1.**pop**())

Итак:

**Стек** - добавляет/удаляет/смотрит последний элемент за O(1), другие операции – недоступны.

**Очередь** - добавляет в конец, удалаяет первый, смотрит оба - за O(1), другие операции – недоступны.

**Дек** - добавляет/удаляет/смотрит крайние элементы за O(1), то же, но для остальных – за O(n).

Связный список позволяет добавляет/удаляет/везде за O(1), но поиск выполняет за O(n). Реализуем его!

**Теоретический блок 4: Односвязный список (One-Way Linked List).**

Односвязный список – структура, состоящая из элементов, каждый из которых, помимо основных данных, содержит ссылку – адрес элемента, следующего за ним. Последний элемент обычно содержит пустую ссылку – адресную константу, обычно существующую во всех языках программирования. Существуют также варианты кольцевых списков, где последний элемент ссылается на первый.

Адреса первого и последнего элемента, называемые обычно «Голова» и «Хвост» списка, хранятся в отдельных переменных, хотя в кольцевых списках достаточно одной адресной переменной. Графически односвязный список имеет вид (слева – обычный список, справа – кольцевой):

|  |  |
| --- | --- |
|  |  |

Каждый элемент списка – экземпляр (объект) класса, который опишем, как узел графа:

**class** Node:

**def** **\_\_init\_\_**(self, d):#конструктор \_\_init\_\_ - предопределенный изменяемый метод любого класса.

self.d=d; self.next=**None**#здесь показана структура конечного объекта

Ниже представлен пример реализации класса для односвязного списка, где описаны основные методы:

#добавить/удалить конец/начало/после известного указателя - O(1)

#поиск адреса по ключу, вывод списка - O(n)

**class** Node:#заголовок

**def** **\_\_init\_\_**(self, d):#поле d – данные, next – адрес следующего элемента списка

self.d=d; self.next=**None**

**def** nxt(self, v):#добавить после self, вернуть указатель на добавленный - O(1)

t=Node(v); t.next=self.next; self.next=t; **return** t

**def** prev(self, v):#добавить перед self, вернуть указатель на добавленный - O(1)

t=Node(v); t.next=self; **return** t

**def** searching(self, v):#ищем v в списке, возвращаем его адрес или адрес tail - O(n)

**while** self.next:

**if** self.d==v: **break**

self=self.next

**return** self

**def** del\_item(tail, item):#удаляем элемент после item, возвращаем tail - O(1)

**if** item.next==**None** **or** item==tail: **return** tail

item.next=item.next.next

**if** item.next==**None**: **return** item

**else**: **return** tail

**def** prn\_lst(head):#печать всего списка, начиная со значения, переданного в head - O(n)

**while** head: **print**(head.d, end=' '); head=head.next

**def** get\_item(head, i):#возвращаем i-й элемент, начиная с head, либо, если i>n, ссылку на tail – O(n)

**while** head!=**None**:

**if** i==0: **break**

i-=1; head=head.next

**return** head

**def** set\_item(head, i, v): #задаем i-му элементу значение v, вернув ссылку на него, либо на tail – O(n)

**while** head!=**None**:

**if** i==0: head.d=v; **break**

i-=1; head=head.next

**return** head

Обычно в головной ссылке содержится поле, хранящее количество элементов в списке.

Здесь представлена головная программа, демонстрирующая работу с односвязным списком:

tail=head=**Node**(0)#запоминаем голову и хвост(для первого элемента совпадают)

**for** i **in** **range**(1,11): tail=Node.**nxt**(tail,i) #добавляем к хвосту

Node.**prn\_lst**(head); **print**() #печатаем начиная с head (head не меняется)

Node.**prn\_lst**(head.next.next.next); **print**() #печатаем пропустив первые 3 элемента

**for** i **in** **range**(1,10): head=Node.prev(head,i)#добавляем перед head(head меняется)

Node.**prn\_lst**(head); **print**() #смотрим результат

Node.**nxt**(Node.**searching**(head,8),33)#вставляем 33 после найденного элемента (8)

Node.**prn\_lst**(head); **print**() #смотрим результат

p=head.next; tail=Node.**del\_item**(tail, p)#удаляем элемент после 8

Node.**prn\_lst**(head); **print**() #смотрим результат

tail=Node.**del\_item**(tail,tail)#удаляем элемент после хвоста (удаляется хвост)

Node.**prn\_lst**(head); **print**()#смотрим результат

Node.**set\_item**(head,5,12)#элементу с номером 5 присвоить значение 12

Node.**prn\_lst**(head); **print**() #смотрим результат

Node.**set\_item**(head,99,77)#несуществующему элементу присвоить 77 (ignore)

Node.**prn\_lst**(head); **print**() #смотрим результат

x=Node.**get\_item**(head,17)#x-найденный узел или None

**if** x: **print**(x.d)

**else**: **print**('такого элемента нет!')

#для удаления головы создают псевдо-голову перед головой или замыкают список

#(кольцевой список - отдельная тема)

#данная демонстрашка показывает общий принцип работы one way linked list,

#всё - из обязательной HL IB-программы, принцип построения и алгоритмы похожи

#можно показать, как создать список-массив (поиск,вставка,удаление - O(1)),

#а также двусвязный список (классический... так-то и здесь я уже всё показал)

Отличие от работы с обычными объектами в том, что здесь обращение к методам идет не как к методам объекта, а как к методам класса!

# **Приложения: Ссылки на видеолекции (всего около 0,5 часа)**

1. Стек. Кириенко Д.Б. Тренер по спорт.программированию,16 мин. <https://youtu.be/KfEeTWogJIg>

2. Очередь. Кириенко Д.Б., 10 мин. <https://youtu.be/ub2XRPsnur4>

3. Кириенко Д.Б., 5 мин. <https://youtu.be/7CLrRYPAo14>