Занятие №11. Стек и очередь

Важно не только хранить данные в памяти, но и иметь возможность выполнять операции над ними по-возможности быстро.

«Структура данных» - программная единица (в языке Java удобнее всего для этого использовать класс), **объединяющая данные с возможностью выполнения определенных над ними определенных операций** (обычно с указанием временной эффективности).

До сих пор мы имели дело только с массивами. Основная и единственная встроенная операция в этой структуре данных – индексация. Обычно она реализована так, что можно получить доступ к любой ячейке за O(1). Поиск минимума в массиве требует порядка O(N) операций. Добавление элемента в массив вообще говоря невозможно. Массив создается на определенное количество ячеек, которое после создания нельзя изменить. Но, если условно считать, что в конце массива есть «пустое место», то вставка элемента в худшем случае (в начало) требует $O(N)$ операций. Мы умеем сортировать массив за $O(N^2)$. Теоретически доказано, что операциями обмена ячеек быстрее, чем за $O(NlogN)$ произвольный массив отсортировать нельзя. Оказывается, что можно организовать (то есть хранить определенным образом, использовать специальные алгоритмы для обработки) данные таким образом, чтобы эти и другие операции выполнялись быстрее.

В этом модуле мы поговорим о двух базовых структурах стеке и очереди. Сначала о том, что они из себя представляют. Потом о том, как они могут быть устроены внутри, мы будем их строить на базе массивов. И затем поговорим об их применениях.

В листингах мы будем использовать везде тип `T`. Для конкретных применений надо заменить `T` на конкретный тип. Например, `int` или `String`.

Стек (Stack)

Представим себе стопку. Допустим стопку тетрадей. Обычно учитель работает с ней так: смотрит на верхнюю тетрадь, проверяет ее, если добавляет элемент – тетрадь - в стопку, то наверх, а не в середину. Таким образом, первая сданная тетрадь (самого шустрого ученика) будет проверена последней, а последняя сданная – первой.

LIFO – Last Input First Output. «Первым вошел, последним вышел».

В стеке доступен только один элемент данных: тот, который был в него вставлен последним. Удалив этот элемент, пользователь получает доступ к предпоследнему

элементу и т. д. Такой механизм доступа, как мы увидим, удобен во многих ситуациях, связанных с программированием.

Операции работы с стеком:

Функция	Действие	Временная эффективность
void clear()	Очистка стека	$O(1)$ обратите внимание на код!
boolean empty()	Проверка на пустоту. Важная операция, чтобы не брать из пустого стека.	$O(1)$
boolean full()	Проверка на заполненность. Теоретически стек бесконечен. Но на практике это не так.	$O(1)$
void push(T v)	Добавить в стек новое значение	$O(1)$
T peek()	Получить верхнее значение без удаления	$O(1)$
T pop(T v)	Взять верхний элемент. Возвращается значение верхнего элемента, при этом он удалется из стека	$O(1)$

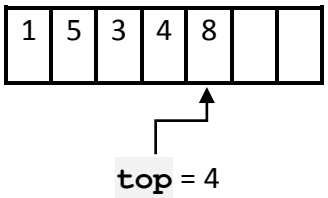
Мы видим, что Стек работает быстро.

Все операции выполняются за константу. Реализовать его также не сложно.

Будем хранить элементы стека в массиве и в отдельной ячейке индекс верхнего (последнего) элемента. Название Stack зарезервировано в стандартной библиотеке классов Java, поэтому мы должны немного изменить имя.

Данные (поля) класса MyStack

Массив `stackArray` (`maxSize` = 7)




```
class MyStack
 {
     private int maxSize; // Размер массива
     private T[] stackArray;
     private int top; // Вершина стека
```

```
// ...
```

Напишем несколько функций класса

```
public MyStack(T s) // Конструктор s – максимальный размер
 {
     ____________
     ____________
     ____________
 }

public void push(T v) // Размещение элемента на вершине стека
{
     ____________
     ____________
}

public void clear()
{
     ____________
}
```

Использовать класс очень просто.

Для использования надо везде в тексте класса заменить T на int.

```
MyStack theStack = new MyStack(5);

theStack.push(20);
theStack.push(40);
theStack.push(60);

while(!theStack.isEmpty())
{
     int value = theStack.pop();
     System.out.print(value);
     System.out.print(" ");
}
```

Как работает эта программа, что она выводит?

Полный текст класса можно посмотреть в Справочнике.

Очередь (Queue)

Структура данных «очередь» очень напоминает обычную очередь из людей. Люди становятся с очередь, а кассир обслуживает их… извините за каламбур, по очереди.

Пришедший первым покупатель будет обслужен первым. То есть, очередь получается «стеком наоборот».

FIFO – First Input First Output. «Первым вошел, последним вышел».

Как и у стека доступен только один элемент – первый. Добавление происходит в конец очереди.

Функции по сути те же, но мы назовем их так, как они называются в стандартном Java интерфейсе Queue (те из них, которые реализованы в интерфейсе).

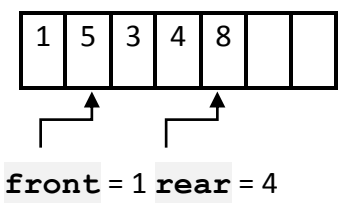
Операции работы с очередью:

Функция	Действие	Временная эффективность
void clear()	Очистка очереди	$O(1)$
boolean empty()	Проверка на пустоту. Важная операция, чтобы не брать из пустой очереди.	$O(1)$
boolean full()	Проверка на наполненность. Теоретически очередь бесконечна. Но на практике это не так.	$O(1)$
void offer(T v)	Добавить в очередь новое значение	$O(1)$
T peek()	Получить первый значение без удаления	$O(1)$
T poll(T v)	Взять первый элемент. Возвращается значение первого элемента, при этом он удалятся из очереди	$O(1)$

Будем хранить элементы очереди так же как и в реализации стека- в массиве. Только вместо одной вершины **`top`** будем хранить две переменных front и rear- начало и конец очереди. Для простоты кода будем также хранить и изменять nItems – текущее число элементов в очереди

Данные (поля) класса MyQueue

Массив **`queArray`** (**`maxSize`** = 7)



`front` = 1 **`rear`** = 4

В очереди на рисунке 4 элемента: 5, 3, 4 и 8, соответственно **`nItems`** = 4. Элемент со сначением 1 в нулевой ячейке когда-то был в очереди, но его уже изъяли («обслужили»)

```
class MyQueue
{
     private int maxSize;
     private T[] queArray;
     private int front;
     private int rear;
     private int nItems; // Для простоты кода будем хранить
                         // и изменять количество элементов
```

Добавлять элемент в очередь надо после rear и затем увеличить rear. Первый элемент имеет индекс front – при извлечении необходимо тоже его увеличить. Очередь будет при добавлении-извлечении по массиву как бы ехать, как змейка. Как однажды возмутился ученик: «Ну и где вы видели кассу с колесиками, которая едет вдоль очереди?!»

Проблема в том, что таким образом она быстро «доедет» до конца массива. И даже если в очереди будет всего один элемент (**`front`** = N-2, **`rear`** = N-1) мы ничего уже добавить в нее не сможем… Выйти из этого положения можно. **Нужно массив закольцевать!** То есть считать, что после N-1-го элемента следует нулевой. В коде это будет выглядеть так:

```
// ...
if(rear == maxSize - 1) // проверка на конец массива
     rear = -1;
rear++;
// ...
```

и

```
// ...
front++;
if(front == maxSize) // проверка на конец массива
     front = 0;
// ...
```

Напишем целиком функции добавления и удаления элемента

```
public void offer() // Добавление элемента в очередь
{
     _____________
     _____________
     _____________
     _____________
}

public T poll() // Извлечение элемента из очереди

     _____________
     _____________
```

```
        ____________
        ____________
        ____________
}
```

В таком случае – в закольцованном массиве **`rear`** может стать левее **`front`**. При этом возникнут трудности с подсчетом количества элементов (просто вычесть **`front`** из **`rear`** будет нельзя) и с проверкой на пустоту. Именно для того, чтобы их избежать мы и ввели переменную **`nItems`**!

Использование класса **`MyQueue`** полностью аналогично использованию класса **`MyStack`**.

Для проверки работоспособности этих классов обязательно решите задачи B и C.

Как всегда полный текст классов, реализующих стек и очередь вы можете посмотреть в Справочнике.

Обратите внимание: публичный класс с функцией main должен быть объявлен первым. То есть классы **`MyQueue`** и **`MyStack`** следует размещать после основного класса.

Структура данных «дек» (deque) – это очередь с двумя открытыми концами. Можно класть и брать элементы с обеих сторон. Пишется он абсолютно аналогично очереди.

Необходимо отметить, что мы не занимались защитой от ошибок. Что будет если добавить элемент в заполненный стек или взять из пустой очереди? В реальном программировании эти случаи надо предусмотреть обязательно. Но при решении олимпиадной задачи, для того чтобы сократить код, а значит выиграть время, их можно не писать. Просто быть очень аккуратным в использовании.

Стеки и очереди имеют очень много применений в программировании.

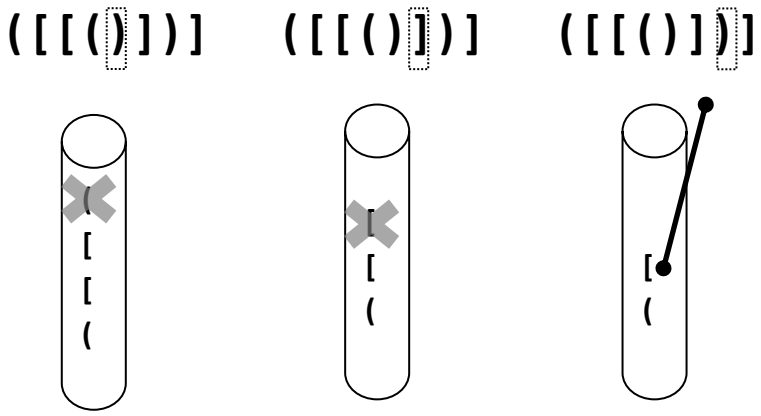
Вспомним задачу про правильную скобочную последовательность из первого ??? занятия. Решение простое: просто считаем «по пути» открывающие за 1, закрывающие за -1. Если в процессе мы ни разу «не ушли в минус» и в конце получили 0 – все хорошо, последовательность правильная. Нет – неправильная.

А что делать, если скобок два вида квадратные и круглые, как в задаче ??? На помощь приходит стек! Будем открывающие скобки складывать в стек, а как встретим закрывающую смотреть на вершину, если она парная закрывающей – вынимаем открывающую и продолжаем. Если нет – конец, последовательность некорректна!

Интересно, что **очередь можно реализовать при помощи… двух стеков!** Как?

Проверьте свои догадки по Подсказке 11.1. Попробуйте реализовать класс очереди, в котором будет из данных объявлено **только** два объекта MyStack и все операции с элементами

([[( )])] – скобки в правильном количестве, но баланс нарушен.



Парная – вынимаем. Парная – вынимаем. Непарная – конец!

В конце надо не забыть проверить стек на пустоту. Если он не пуст – открывающих оказалось больше, чем закрывающих – последовательность некорректна.

В задаче про шарики из Занятия №6 можно искать одинаковые, «удалять», снова искать. Решение идейно несложное, но технически трудное. И работает достаточно медленно. А если использовать похожую идею, то можно уложиться в один проход. Будем хранить в стеке номера шариков и их количество. Если новый шарик совпадает по цвету с верхним – добавлять единичку в количество верхних. Если нет – пробуем лопать. Смотрим количество верхних. Если оно больше трех – лопаем их (вынимаем все верхние). И добавляем новый шарик. Правда, при этом нужно хранить часть данных – заводить стек для решения все же необходимо. На этом занятии вам предлагается задача «Шарики - 2», ограничения в которой не позволяют делать в решении постоянные сдвиги массива.

Одно из применений очереди для обработки графов мы рассмотрим на следующем занятии.

В заключении о встроенных реализациях стека и очереди в Java

Стек представлен в стандартной библиотеке классом Stack. Очередь - интерфейсом Queue. Этот интерфейс реализуется, например, классом LinkedList (связанный список). Это означает, что класс LinkedList можно использовать в качестве очереди. Описание класса Stack и интерфейса Queue приведено в Справочнике. Надо сказать, что работают они только с настоящими объектами, поэтому для примитивных типов (int, double, char и т.д.) их использовать немного сложно, нужно использовать классы-обертки (см. Занятие №2 и пример по использованию стандартного класса Stack в Справочнике).

С практической точки зрения эти классы интересны прежде всего тем, что они реализовны не на базе массива и занимают переменный объем памяти в зависимости от реального количества хранимых в них элементов.

Пример использования класса Stack вы можете посмотреть в разборе задачи «Баржа».

Задачи

Задача A. Игра в пьяницу

В игре в пьяницу карточная колода раздается поровну двум игрокам. Далее они вскрывают по одной верхней карте, и тот, чья карта старше, забирает себе обе вскрытые карты, которые кладутся под низ его колоды. Тот, кто остается без карт – проигрывает.

Для простоты будем считать, что все карты различны по номиналу, а также, что самая младшая карта побеждает самую старшую карту ("шестерка берет туза").

Игрок, который забирает себе карты, сначала кладет под низ своей колоды карту первого игрока, затем карту второго игрока (то есть карта второго игрока оказывается внизу колоды).

Напишите программу, которая моделирует игру в пьяницу и определяет, кто выигрывает. В игре участвует 10 карт, имеющих значения от 0 до 9, большая карта побеждает меньшую, карта со значением 0 побеждает карту 9.

Формат входных данных

Программа получает на вход две строки: первая строка содержит 5 карт первого игрока, вторая – 5 карт второго игрока. Карты перечислены сверху вниз, то есть каждая строка начинается с той карты, которая будет открыта первой.

Формат выходных данных

Программа должна определить, кто выигрывает при данной раздаче, и вывести слово `first` или `second`, после чего вывести количество ходов, сделанных до выигрыша. Если на протяжении 10^6 ходов игра не заканчивается, программа должна вывести слово`botva`.

Пример

Входные данные	Выходные данные
1 3 5 7 9 2 4 6 8 0	second 5

Задача B. Простой стек

Реализуйте структуру данных "стек". Напишите программу, содержащую описание стека и моделирующую работу стека, реализовав все указанные здесь методы. Программа считывает последовательность команд и в зависимости от команды выполняет ту или иную операцию. После выполнения каждой команды программа должна вывести одну строчку. Возможные команды для программы:

push n

Добавить в стек число n (значение n задается после команды). Программа должна вывести `ok`.

pop

Удалить из стека последний элемент. Программа должна вывести его значение.

back

Программа должна вывести значение последнего элемента, не удаляя его из стека.

size

Программа должна вывести количество элементов в стеке.

clear

Программа должна очистить стек и вывести `ok`.

exit

Программа должна вывести `bye` и завершить работу.

Гарантируется, что набор входных команд удовлетворяет следующим требованиям: максимальное количество элементов в стеке в любой момент не превосходит 100, все команды `pop` и `back` корректны, то есть при их исполнении в стеке содержится хотя бы один элемент.

Пример протокола работы программы

Ввод	**Вывод**
push 2 push 3 push 5 back size pop size push 7 pop clear	ok ok ok 5 3 5 2 ok 7 ok

size exit	0 bye

Задача C. Очередь с защитой от ошибок

Реализуйте структуру данных "очередь". Напишите программу, содержащую описание очереди и моделирующую работу очереди, реализовав все указанные здесь методы. Программа считывает последовательность команд и в зависимости от команды выполняет ту или иную операцию. После выполнения каждой команды программа должна вывести одну строчку. Возможные команды для программы:

push n

Добавить в очередь число n (значение n задается после команды). Программа должна вывести `ok`.

pop

Удалить из очереди первый элемент. Программа должна вывести его значение.

front

Программа должна вывести значение первого элемента, не удаляя его из очереди.

size

Программа должна вывести количество элементов в очереди.

clear

Программа должна очистить очередь и вывести `ok`.

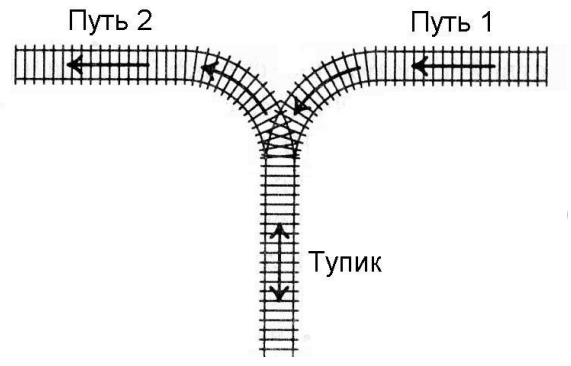
exit

Программа должна вывести `bye` и завершить работу.

Перед исполнением операций `front` и `pop` программа должна проверять, содержится ли в очереди хотя бы один элемент. Если во входных данных встречается операция `front` или `pop`, и при этом очередь пуста, то программа должна вместо числового значения вывести строку `error`.

Задача D. Сортировка вагонов

К тупику со стороны пути 1 (см. рисунок) подъехал поезд. Разрешается отцепить от поезда один или сразу несколько первых



вагонов и завезти их в тупик (при желании, можно даже завезти в тупик сразу весь поезд). После этого часть из этих вагонов вывезти в сторону пути 2. После этого можно завезти в тупик еще несколько вагонов и снова часть оказавшихся вагонов вывезти в сторону пути 2. И так далее (так, что 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. Нельзя с пути 1 попасть на путь 2, не заезжая в тупик.

Известно, в каком порядке изначально идут вагоны поезда. Требуется с помощью указанных операций сделать так, чтобы вагоны поезда шли по порядку (сначала первый, потом второй и т.д., считая от головы поезда, едущего по пути 2 в сторону от тупика).

Формат входных данных

Вводится число N — количество вагонов в поезде ($1 \le N \le 2000$). Дальше идут номера вагонов в порядке от головы поезда, едущего по пути 1 в сторону тупика. Вагоны пронумерованы натуральными числами от 1 до N, каждое из которых встречается ровно один раз.

Формат выходных данных

Если сделать так, чтобы вагоны шли в порядке от 1 до N, считая от головы поезда, когда поезд поедет по пути 2 из тупика, можно, выведите действия, которые нужно проделать с поездом. Каждое действие описывается двумя числами: типом и количеством вагонов:

- если нужно завезти с пути 1 в тупик K вагонов, должно быть выведено сначала число 1, а затем — число K ($K \ge 1$),
- если нужно вывезти из тупика на путь 2 K вагонов, должно быть выведено сначала число 2, а затем — число K ($K \ge 1$).
- Если возможно несколько последовательностей действий, приводящих к нужному результату, выведите любую из них.

Если выстроить вагоны по порядку невозможно, выведите одно число 0.

Примеры

Входные данные	**Выходные данные**
3 3 2 1	1 3 2 3
4 4 1 3 2	1 2 2 1 1 2 2 3
3 2 3 1	0

Задача E. Скобки

Дана последовательность из *N* круглых, квадратных и фигурных скобок. Выяснить, можно ли добавить в неё цифры и знаки арифметических действий так, чтобы получилось правильное арифметическое выражение.

Ограничения: 1 <= *N* <= 100 000.

Ввод: В первой строке находится число скобок *N*, во второй - *N* символов из набора (,), [,], {, }.

Вывод: Выводится слово "`Yes`", если получить правильное арифметическое выражение можно, или "`No`", если нельзя.

Примеры

Входные данные	Выходные данные
6 ([())]	No
24 {[()([]{})[]]({}{{}})}[]	Yes

В каждой строке сначала записан номер класса (число, равное 9, 10 или 11), затем (через пробел) – фамилия ученика.

Формат выходных данных

Необходимо вывести список школьников по классам: сначала всех учеников 9 класса, затем – 10, затем – 11. Внутри одного класса порядок вывода фамилий должен быть таким же, как на входе.

Пример

Входные данные	Выходные данные
9 Ivanov 10 Petrov 11 Sidorov 9 Grigoryev 9 Sergeev 10 Yakovlev	9 Ivanov 9 Grigoryev 9 Sergeev 10 Petrov 10 Yakovlev 11 Sidorov

Задача G. Обратная польская 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. Достоинство постфиксной записи в том, что она не требует скобок и дополнительных соглашений о приоритете операторов для своего чтения.

Дано выражение в постфиксой записи, содержащее однозначные числа, операции +, –, *. Вычислите значение записанного выражения.

Формат ввода

В единственной строке записано выражение в постфиксной записи, содержащее однозначные числа и операции +, –, *. Строка содержит не более 100 чисел и операций. Числа и операции отделяются друг от друга ровно одним пробелом.

Формат вывода

Необходимо вывести значение записанного выражения. Гарантируется, что результат выражения, а также результаты всех промежуточных вычислений по модулю меньше 2_{31}.

Пример ввода	Пример вывода
8 9 + 1 7 - *	-102

Задача H. Баржа

На барже располагается K грузовых отсеков. В каждый отсек можно поместить некоторое количество бочек с одним из 10 000 видов топлива. Причём извлечь бочку из отсека можно лишь в случае, если все бочки, помещённые в этот отсек после неё, уже были извлечены. Таким образом, в каждый момент времени в каждом непустом отсеке имеется ровно одна бочка, которую можно извлечь не трогая остальных. Будем называть такие бочки крайними.

Изначально баржа пуста. Затем она последовательно проплывает через N доков, причём в каждом доке на баржу либо погружается бочка с некоторым видом топлива в некоторый отсек, либо выгружается крайняя бочка из некоторого отсека. Однако, если указанный отсек пуст, либо если выгруженная бочка содержит не тот вид топлива, который ожидалось, следует зафиксировать ошибку. Если на баржу оказывается погружено более P бочек или если после прохождения всех доков она не стала пуста, следует также зафиксировать ошибку. От вас требуется либо указать максимальное количество бочек, которые одновременно пребывали на барже либо зафиксировать ошибку.

Входные данные

В первой строке три целых числа N, K и P ($1 \le N, K, P \le 100\,000$). Далее следует N строк с описанием действия, выполняемого в очередном доке. Если в нём происходит погрузка, то строка имеет вид «+ A B», где A — номер отсека, в который помещается бочка, а B — номер вида топлива в ней. Если же док занимается разгрузкой, то строка имеет вид «- A B», где A — номер отсека, из которого извлекается бочка, а B — номер ожидаемого вида топлива.

Выходные данные

Вывести либо одно число, равное искомому максимуму в случае безошибочного прохождения баржой маршрута, либо вывести слово «Error» в противном случае.

Примеры тестов

Пример ввода	Пример вывода

6 1 2 + 1 1 + 1 2 - 1 2 - 1 1 + 1 3 - 1 3	2

Задача I. Много шариков

В одной компьютерной игре игрок выставляет в линию шарики разных цветов. Их может быть очень много... Когда образуется непрерывная цепочка из трех и более шариков одного цвета, она удаляется из линии. Все шарики при этом сдвигаются друг к другу, и ситуация может повториться.

Напишите программу, которая по данной ситуации определяет, сколько шариков будет сейчас уничтожено. Естественно, непрерывных цепочек из трех и более одноцветных шаров в начальный момент может быть не более одной.

Входные данные

Даны количество шариков в цепочке (не более 100000) и цвета шариков (от 0 до 9, каждому цвету соответствует свое целое число).

Выходные данные

Требуется вывести количество шариков, которое будет уничтожено.

Примеры

Пример ввода	Пример вывода
5 1 3 3 3 2	3

Задача J. Контейнеры

На складе хранятся контейнеры с товарами N различных видов. Все контейнеры составлены в N стопок. В каждой стопке могут находиться контейнеры с товарами любых видов (стопка может быть изначально пустой).

Автопогрузчик может взять верхний контейнер из любой стопки и поставить его сверху в любую стопку. Необходимо расставить все контейнеры с товаром первого вида в первую стопку, второго вида — во вторую стопку и т.д.

Программа должна вывести последовательность действий автопогрузчика или сообщение о том, что задача решения не имеет.

Формат входных данных

В первой строке задается одно натуральное число N, не превосходящее 500. В следующих N строках описаны стопки контейнеров: сначала записано число ki — количество контейнеров в стопке, а затем ki чисел — виды товара в контейнерах в данной стопке, снизу вверх. В каждой стопке вначале не более 500 контейнеров (в процессе переноса контейнеров это ограничение может быть нарушено).

Формат выходных данных

Выведите описание действий автопогрузчика: для каждого действия укажите два числа — из какой стопки брать контейнер и в какую стопку класть. (Обратите внимание, что минимизировать количество операций автопогрузчика не требуется.) Если задача не имеет решения, выдайте одно число 0. Если контейнеры изначально правильно размещены по стопкам, выводить ничего не надо.

Пример

Входные данные	**Выходные данные**
3 4 1 2 3 2 0 0	1 2 1 3 1 2

Изначально в первой стопке лежат четыре контейнера — снизу контейнер с товаром первого вида, над ним — с товаром второго вида, над ним — третьего, и сверху — еще один контейнер с товаром второго вида.

Подсказки и решения. 11 Занятие.

11.1

Будем класть в первый стек, а брать из второго. Если на момент взятия второй стек пуст – переложим в него все из первого. Несмотря на то, что в функции взятия таким образом будет присутствовать цикл, каждый элемент либо сразу возьмется, либо переложится единственный раз. Временная эффективность всего алгоритма при этом сохранится!

11.2

```
if (c1 > c2 && ! (c2 == 0 && c1 == 9) || (c1 == 0 && c2 == 9))
```

Разбор. Занятие 11

11A. Игра в пьяницу

Колода каждого игрока – это очередь.

Рядом с основным классом решения разместим класс Очередь.

Заведем две очереди

```
MyQueue q1 = new MyQueue(), q2 = new MyQueue();// на все карты
```

И заполним их первоначальными колодами:

```
for (int i = 0; i < 5; i++) q1.offer(in.nextInt());
for (int i = 0; i < 5; i++) q2.offer(in.nextInt());
```

И будем моделировать. Берем по одному элементу из каждой и закладываем в ту, из которой вынули большую в правильном порядке.

```
int i = 0; // счетчик итераций
while (!q1.empty() && !q2.empty() && i < 1000*1000)
{
     int c1 = q1.poll();
     int c2 = q2.poll();
     if (_____________________) // Подсказка 11.2
     {
          q1.offer(c1);
          q1.offer(c2);
     }
     else
     {
          q2.offer(c1);
          q2.offer(c2);
     }
     i++;
}
```

Если за миллион итераций укладываемся (одна из очередей пустеет) – пишем ответ, иначе «botva». После цикла:

```
if (q1.empty()) out.print("second " + i);
else if (q2.empty()) out.print("first " + i);
else out.print("botva");
```

11B. Простой стек

Задача на проверку того, что стек написан правильно.

Достаточно завести объект стека в программе

```
MyStack st = new MyStack(100);
```

и считывая команды выполнять их, до того момента, пока не встретим “exit”:

```
String command = "";
while (!command.equals("exit"))
{
     command = in.next();
     if (command.equals("push"))
     {
          st.push(in.nextInt());
          out.println("ok");
     }
     if (command.equals("pop"))
     {
          out.println(st.pop());
     }
     // ...
```

Обратите внимание: если не делать out.flush(); после каждой команды, весь вывод появится после команды exit!

11C. Очередь с защитой от ошибок

Эта задача делается почти так же, как предыдущая.

Отличие в том, что надо изменить функции класса Очередь **`peek`** и **`poll`** так, чтобы они выводили ошибку – слово error. Для этого надо объявить объект **`out`** *статическим на уровне основного класса*:

```
class Main
{
     static PrintWriter out = new PrintWriter(System.out);
     public static void main...
     ...
```

Функция peek будет выглядеть так:

```
     public int peek() // Чтение первого элемента очереди
     {
          if (empty())
          {
               Main.out.print("error");
               return -1; // Ошибка!
          }
          return queArray[front];
```

```
    }
```

11D. Сортировка вагонов

Тупик – это стек. При помощи него можно перевернуть часть, идущую наоборот.

Заведем стек. Действовать будеим по одному вагону – правила этого не запрещают.

Помещать туда номер очередного вагона, в том случае, если нельзя вывезти из тупика – вынуть из стека – правильный очередной номер. Если можно вывозим один вагон.

Например, рассмотрим ситуацию

1	5	3	2	4

Нужен первый. Стек пуст – загоняем (первый) в тупик (1 1)

Опять нужен первый – он в стеке – выгоняем из тупика (2 1)

Нужен второй – стек пуст – (пятый) в тупик (1 1)

Опять нужен второй – в стеке пятый – (третий) в тупик (1 1)

Опять нужен второй – в стеке третий – загоняем (второй) в тупик (1 1)

Все еще нужен второй – он в стеке – выгоняем из тупика (2 1)

Нужен третий – он в стеке - выгоняем из тупика (2 1)

и т.д.

Если вагоны кончились, а стек оказался непуст – отсортировать нельзя.

Это можно понять только в самом конце, поэтому надо запоминать действия в массив, а потом его вывести, если отсортировать удалось.

Любопытно, что если было бы возможно отгонять поезд назад (то есть пути 1 и 2 дополнительно бы соединялись напрямую) и повторять операцию, отсортировать можно было бы всегда. Этот процесс очень похож на «блинную сортировку», описанную во врезке занятия по сортировкам.

11E. Скобки

«Теория» этой задачи подробно разбирается в тексте занятия.

Заведем стек с элементами типа char (заменим везде в тексте MyStack T на char).

Открывающие скобки будем закладывать в стек. Встретив закрывающую, смотрим в стек, если там правильная пара – вынимаем открывающую, иначе заканчиваем программу – «No».

В конце, если не закончили раньше по ошибке проверяем стек на пустоту. Если он не пуст – «No» - открывающих оказалось больше. Иначе выводим «Yes».

11F Списки по классам

Основное требование этой задачи, чтобы фамлилии оставались в том же порядке, что и в исходном файле.

Заведем три очереди (по одной для каждого класса):

```
MyQueue   q9  = new MyQueue(10000);
          q10 = new MyQueue(10000),
          q11 = new MyQueue(10000);
```

И «разложим» учеников по ним:

```
while(in.hasNext())
{
     int c = in.nextInt(); // читаем класс
     // и добавляем фамилию в соответствующую очередь
     if (c == 9)   q9.offer(in.next());
     if (c == 10) q10.offer(in.next());
     if (c == 11) q11.offer(in.next());
     if (c==0) break;
}
```

Обратите внимание на условие цикла.

А дальше выведем все три очереди с указанием класса:

```
while (!q9.empty())
{
     out.println (9 + " " + q9.poll());
}
// ...
```

11G Обратная польская запись

Эта форма записи очень удобна для вычислений. В ней нет скобок!

Числа помещаются в стек, а когда встречается знак, то два числа вынимаются из стека, над ними производится соответствующее действие и результат закладывается обратно в стек:

В цикле выполняем

```
String s = in.next();
if (s.charAt(0) == '+') stack.push(stack.pop() + stack.pop());
else if (s.charAt(0) == '-') stack.push(-stack.pop() +
                                                  stack.pop());
else if (s.charAt(0) == '*') stack.push(stack.pop()
                                                  stack.pop());
```

```
else stack.push(Integer.parseInt(s));
```

Оставшееся число в стеке – ответ.

11H Баржа

Эта задача тоже на моделирование. Нужно смоделировать работу множества стеков. Идейно это несложно. Ее особенность в том, что ограничения по памяти не позволяют делать массив стеков с реализацией массивах.

Мы будем использовать массив объектов класса Stack – встроенного в стандартную библиотеку классов Java. Внутренне они реализованы динамически и расширяются по мере необходимости (размер занимаемой памяти объектом-стеком зависит от того, сколько реально храниться элементов в нем). Эта задача включена в контест для того, чтобы показать, как пользоваться именно возможностями языка Java.

Создадим массив на K + 1 стеков (чтобы не думать о нумерации с единицы, «нулевой» док использоваться не будет):

```
Stack[] st = new Stack[K+1];
for (int i = 0; i <= K; i++)
{
          st[i] = new Stack();
}
```

И будем просто моделировать процесс:

```
int max = 0; // максимальное количество бочек на барже
int n = 0;   // текущее количество бочек

boolean error = false; // зафиксирована ошибка

for (int i = 0; i < N && !error ; i++)
{
     String command = in.next();
     int pod = in.nextInt();
     int type = in.nextInt();
     if (command.charAt(0)=='+')
     {
          // В стеках хранятся объекты Integer
          st[pod].push(new Integer(type)); // ставим бочку
          n++;
          if (st[pod].size() > P) {error = true; break;}
          if (n > max) max = n;
     }
     else
     {
          if (st[pod].size() == 0) {error = true; break;}
          // обратите внимание на приведение типа!
```

```
            Integer b = (Integer)st[pod].pop(); // снимаем бочку
            n--;
            if (b.intValue() != type) {error = true; break;}
        }
}
```

По окончании процесса выводим ответ:

```
if (error || n > 0) out.println("Error");
else out.print(max);
```

11I. Много шариков

Разбирается в тексте занятия

11J. Контейнеры

Проще всего для начала все скинуть в первую стопку.

Особые случаи это одна и две стопки. В первом ничего переносить не нужно. Во втором проще всего сделать так - переложить из нее верхние контейнеры №2 во вторую. Если при этом в ней не останется контейнеров №2 (все они были сверху и успешно перенесены) – все хорошо. Если нет – перенести невозможно.

Рассмотрим общий случай, когда контейнеров не меньше трех. В этом случае перенести всегда можно. Переложим все контейнеры из первой стопки в им соответствующие, а контейнеры №1 будем складывать во вторую стопку (вместе с контейнерами №2). Теперь разберем вторую. Контейнеры №2 в третью, а №1 на место – в первую. Теперь все контейнеры №2 во вторую с третьей. Заметим, что при этом достаточно завести всего три стека, а перекладки в контейнеры с большими номерами просто сразу выводить!

Занятие 11.Справочник

Шаблонные классы в Java (generics)

Начиная с версии Java 5 в языке появился механизм обобщённого программирования — шаблоны, внешне близкие к шаблонам C++. С помощью специального синтаксиса в описании классов и методов можно указать параметры-типы, которые внутри описания могут использоваться в качестве типов полей, параметров и возвращаемых значений методов.

Для создания класса, который мы потом хотим приспособливать под работу с разными типами, в угловых скобках указывается псевдо-имя класса. А при объявлении объекта в угловых скобках указывается имя реального типа:

```
// Объявление обобщённого класса
class GenericClass<T>
{
  T getFirst() { ... }
  void add(T obj) { ... }
}
```

Использование обобщённого класса в коде

```
// var – объект класса GenericClass работы со строками
GenericClass <String> var = new GenericClass<String>();
var1.add("qwerty");
String p = var1.getFirst();
```

Шаблоны конкретизируются именно классами, для примитивных типов (int, double и т.д.) их использовать нельзя!

Обратите внимание: публичный класс с функцией main должен быть объявлен первым. То есть классы **`MyQueue`** и **`MyStack`** следует размещать после основного класса.

Реализация стека на массиве (без защиты от ошибок)

Для использования следует заменить T везде в тексте на требуемый тип (например, int).

```
class MyStack
{
```

```
     private int maxSize; // Размер массива
     private T[] stackArray;
     private int top; // Вершина стека

public MyStack(T s) // Конструктор
{
     maxSize = s; // Определение размера стека
     stackArray = new T[maxSize]; // Создание массива
     top = -1; // Пока нет ни одного элемента
}

public void push(T v) // Размещение элемента на вершине стека
 {
     top++;
     stackArray[top] = v; // Увеличение top, вставка элемента
 }

public T pop() // Извлечение элемента с вершины стека
{    top--;
     return stackArray[top+1]; // Извлечение элемента
}
// ------------------------------------------------------------
-
 public long peek() // Чтение элемента с вершины стека
 {
     return stackArray[top];
 }
// ------------------------------------------------------------
-
 public boolean empty() // true, если стек пуст
{
     if (top == -1) return true;
     return false;
}
public boolean full() // true, если стек полон
{
     if (top == maxSize - 1) return true;
     return false;
}
public void clear()
{
     top = -1;
}
}
```

Обратите внимание: публичный класс с функцией main должен быть объявлен первым. То есть классы **`MyQueue`** и **`MyStack`** следует размещать после основного класса.

Реализация очереди на массиве (без защиты от ошибок)

Для использования следует заменить T везде в тексте на требуемый тип (например, int).

```
class MyQueue
{
     private int maxSize;
     private T[] queArray;
     private int front;
     private int rear;
     private int nItems; // Для простоты кода будем хранить
                         // и изменять количество элементов
     MyQueue(int s) // Конструктор
     {
          maxSize = s;
          queArray = new T[maxSize];
          front = 0;
          rear = -1;
          nItems = 0;
     }

     public void clear()
     {
          front = -1;
     }

     public void offer(T v) // Добавить элемент в очередь
     {
          if(rear == maxSize - 1) // проверка на конец массива
               rear = -1;
          rear++;
          queArray[rear] = v;
          nItems++; // увеличить счетчик
     }

     public T poll() // Извлечение первого элемента очереди
     {
          T temp = queArray[front]; // взять элемент
          front++;
          if(front == maxSize) // проверка на конец массива
               front = 0;
          nItems--; // уменьшить количество элементов
          return temp;
     }

     public T peek() // Чтение первого элемента очереди
     {
          return queArray[front];
     }
     public  boolean empty() // true если очередь пуста
```

```
    {
        if (nItems == 0) return true;
        return false;
    }

    public boolean isFull() // true если очередь полна
    {
        if (nItems == maxSize) return true;
        return false;
    }
}
```

Обратите внимание: публичный класс с функцией main должен быть объявлен первым. То есть классы `MyQueue` и `MyStack` следует размещать после основного класса.

Реализация очереди для элементов типа int на массиве шаблонным классом (без защиты от ошибок)

```
class MyQueue
{
    private int maxSize;
    private int[] queArray;
    private int front;
    private int rear;
    private int nItems; // Для простоты кода будем хранить
                        // и изменять количество элементов
    MyQueue(int s) // Конструктор
    {
        maxSize = s;
        queArray = new int[maxSize];
        front = 0;
        rear = -1;
        nItems = 0;
    }

    public void clear()
    {
        front = -1;
    }

    public void offer(int v) // Добавить элемент в очередь
    {
        if(rear == maxSize - 1) // проверка на конец массива
            rear = -1;
        rear++;
        queArray[rear] = v;
        nItems++; // увеличить счетчик
    }

    public int poll() // Извлечение первого элемента очереди
```

```
    {
        int temp = queArray[front]; // взять элемент
        front++;
        if(front == maxSize) // проверка на конец массива
            front = 0;
        nItems--; // уменьшить количество элементов
        return temp;
    }

    public int peek() // Чтение первого элемента очереди
    {
        return queArray[front];
    }
    public  boolean empty() // true если очередь пуста
    {
        if (nItems == 0) return true;
        return false;
    }

    public boolean isFull() // true если очередь полна
    {
        if (nItems == maxSize) return true;
        return false;
    }
}
```

Встроенный класс Stack

Для примитивных типов необходимо использовать классы обертки!

boolean	**empty**() Tests if this stack is empty.
E	**peek**() Looks at the object at the top of this stack without removing it from the stack.
E	**pop**() Removes the object at the top of this stack and returns that object as the value of this function.
E	**push**(E item) Pushes an item onto the top of this stack.
int	**search**(Object o)

	Returns the 1-based position where an object is on this stack.

Пример использования:

```
Stack q = new Stack();
q.push(new Integer(1)); // оборачиваем int в Integer
q.push(new Integer(2)); // оборачиваем int в Integer

System.out.println(q.pop() + ", " + q.pop());
```

Программа выведет 2, 1

Встроенный интерфейс Queue

Для примитивных типов необходимо использовать классы обертки!

boolean	**add**(E e) Inserts the specified element into this Queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.
E	**element**() Retrieves, but does not remove, the head of this Queue.
boolean	**offer**(E e) Inserts the specified element into this Queue if it is possible to do so immediately without violating capacity restrictions.
E	**peek**() Retrieves, but does not remove, the head of this Queue, or returns `null` if this Queue is empty.
E	**poll**() Retrieves and removes the head of this Queue, or returns `null` if this Queue is empty.
E	**remove**() Retrieves and removes the head of this Queue.

Реализуется, например, классом LinkedList.

Пример использования:

```
LinkedList q = new LinkedList();
q.offer("a");
```

```
q.offer("b");
System.out.println(q.poll() + ", " + q.poll());
```

Программа выведет a, b

Кстати, LinkedList не требует при создании указания максимального размера и расширяется по мере необходимости.