

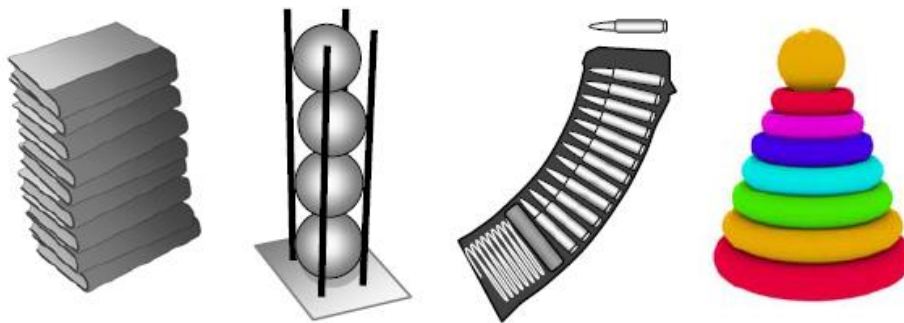
Стек, очередь, дек

Стек

Представьте себе стопку книг (подносов, кирпичей и т.п.). С точки зрения информатики её можно воспринимать как список элементов, расположенных в определенном порядке. Этот список имеет одну особенность – удалять и добавлять элементы можно только с одной («верхней») стороны. Действительно, для того, чтобы вытащить какую-то книгу из стопки, нужно сначала снять все те книги, которые находятся на ней. Положить книгу сразу в середину тоже нельзя.

Стек (англ. *stack* – стопка) – это линейный список, в котором элементы добавляются и удаляются только с одного конца («последним пришел – первым ушел»)

На рисунках показаны примеры стеков вокруг нас, в том числе автоматный магазин и детская пирамидка:



Стек используется при выполнении программ: в этой области оперативной памяти хранятся адреса возврата из подпрограмм; параметры, передаваемые функциям и процедурам, а также локальные переменные.

Задача 1. В файле записаны целые числа. Нужно вывести их в другой файл в обратном порядке. В этой задаче очень удобно использовать стек. Для стека определены две операции:

- добавить элемент на вершину стека (англ. *push* – втолкнуть);
- получить элемент с вершины стека и удалить его из стека (англ. *pop* – вытолкнуть)

Запишем алгоритм решения на псевдокоде. Сначала читаем данные и добавляем их в стек:

```
while файл не пуст:
    прочитать x
    добавить x в стек
```

Теперь верхний элемент стека – это последнее число, прочитанное из файла. Поэтому остается «вытолкнуть» все записанные в стек числа, они будут выходить в обратном порядке:

```
while стек не пуст:
    вытолкнуть число из стека в x
    записать x в файл
```

Использование списка

Поскольку стек – это линейная структура данных с переменным количеством элементов, для работы со стеком в программе на языке **Python** удобно использовать список. Вершина стека будет находиться в конце списка. Тогда для добавления элемента на вершину стека можно применить уже знакомый нам метод `append`:

```
stack.append ( x )
```

Чтобы снять элемент со стека, используется метод `pop`:

```
x = stack.pop ( )
```

Метод `pop` – это функция, которая выполняет две задачи:

1. удаляет последний элемент списка (если вызывается без параметров);
2. возвращает удалённый элемент как результат функции, так что его можно сохранить в какой-либо переменной.

Теперь несложно написать цикл ввода данных в стек из файла:

```
F = open ( "input.txt" )
stack = []
while True:
    s = F.readline()
    if not s: break
    stack.append( int(s) )
F.close()
```

или даже так:

```
stack = []
for s in open( "input.dat" ):
    stack.append( int(s) )
```

Затем выводим элементы массива в файл в обратном порядке:

```
F = open ( "output.txt", "w" )
while len(stack) > 0:
    x = stack.pop()
    F.write ( str(x) + "\n" )
F.close()
```

Заметим, что перед записью в файл с помощью метода **write** все данные нужно преобразовать в формат символьной строки, это делает функция **str**. Символ перехода на новую строку «\n» добавляется в конец строки вручную.

Вычисление арифметических выражений

Вы не задумывались, как компьютер вычисляет арифметические выражения, записанные в такой форме: $(5+15) / (4+7-1)$? Такая запись называется **инфиксной** – в ней знак операции расположен *между* операндами (данными, участвующими в операции). Инфиксная форма неудобна для автоматических вычислений, из-за того, что выражение содержит скобки и его нельзя вычислить за один проход слева направо.

В 1920 году польский математик Ян Лукашевич предложил **префиксную** форму, которую стали называть польской нотацией. В ней знак операции расположен перед операндами. Например, выражение $(5+15) / (4+7-1)$ может быть записано в виде $/ + 5 15 - + 4 7 1$. Скобок здесь не требуется, так как порядок операций строго определен: сначала выполняются два сложения $(+ 5 15$ и $+ 4 7)$, затем вычитание, и, наконец, деление. Первой стоит последняя операция.

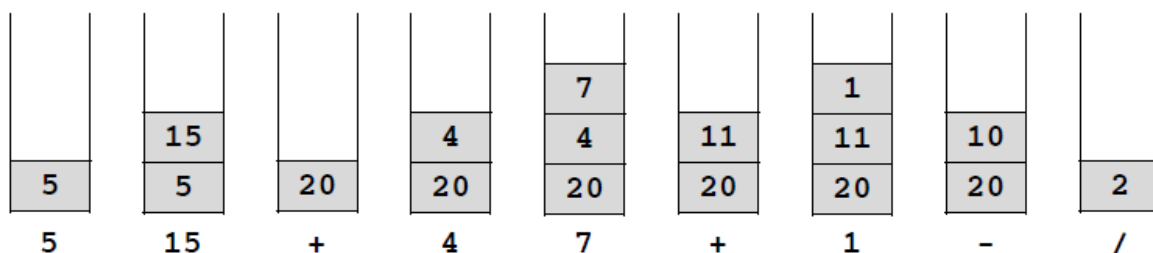
В середине 1950-х годов была предложена **обратная польская нотация** или **постфиксная форма** записи, в которой знак операции стоит *после* операндов:

$5 15 + 4 7 + 1 - /$

В этом случае также не нужны скобки, и выражение может быть вычислено за один просмотр с помощью стека следующим образом:

- если очередной элемент – число (или переменная), он записывается в стек;
- если очередной элемент – операция, то она выполняется с верхними элементами стека, и после этого в стек вталкивается результат выполнения этой операции.

Покажем, как работает этот алгоритм (стек «растёт» снизу-вверх):



В результате в стеке остается значение заданного выражения.

Приведём программу, которая вводит с клавиатуры выражение, записанное в постфиксной форме, и вычисляет его:

```

data = input().split()           # (1)
stack = []                       # (2)
for x in data:                   # (3)
    if x in "+-*/":              # (4)
        op2 = int(stack.pop())   # (5)
        op1 = int(stack.pop())   # (6)
        if x == "+": res = op1 + op2 # (7)
        elif x == "-": res = op1 - op2 # (8)
        elif x == "*": res = op1 * op2 # (9)
        else: res = op1 // op2    # (10)
        stack.append(res)        # (11)
    else:
        stack.append(x)          # (12)
print(stack[0])                  # (13)

```

В строке программы 1 результат ввода разбивается на части по пробелам с помощью метода `split`, в результате получается список `data`, содержащий отдельные элементы постфиксной записи – числа и знаки арифметических действий. В строке 2 создаём пустой стек, в строке 3 в цикле перебираем все элементы списка. Если очередной элемент, попавший в переменную `x` – это знак арифметической операции (строка 4), снимаем со стека два верхних элемента (строки 5-6), выполняем нужное действие (строки 7-10) и добавляем результат вычисления в стек (строка 11). Если же очередной элемент – это число (не знак операции), просто добавляем его в стек (строка 12). В конце программы в стеке должен остаться единственный элемент (результат), который выводится на экран (строка 13).

Скобочные выражения

Задача 2. Вводится символьная строка, в которой записано некоторое (арифметическое) выражение, использующее скобки трёх типов: `()`, `[]` и `{}`. Проверить, правильно ли расставлены скобки.

Например, выражение `() [{ () []]` – правильное, потому что каждой открывающей скобке соответствует закрывающая, и вложенность скобок не нарушается. Выражения

`[() [[[() [() }] ([]]`

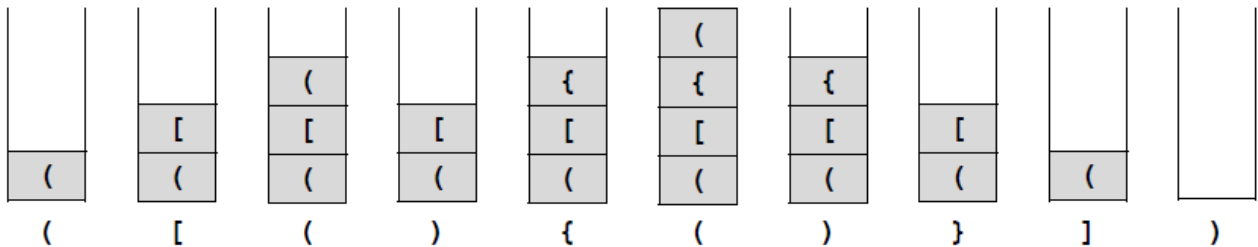
неправильные. В первых трёх есть непарные скобки, а в последних двух не соблюдается вложенность скобок.

Начнём с аналогичной задачи, в которой используется только один вид скобок. Её можно решить с помощью счётчика скобок. Сначала счётчик равен нулю. Строка просматривается слева направо, если очередной символ – открывающая скобка, то счётчик увеличивается на 1, если закрывающая – уменьшается на 1. В конце просмотра счётчик должен быть равен нулю (все скобки парные), кроме того, во время просмотра он не должен становиться отрицательным (должна соблюдаться вложенность скобок).

В исходной задаче (с тремя типами скобок) хочется завести три счётчика и работать с каждым отдельно. Однако, это решение неверное. Например, для выражения `{ ([]) }` условия «правильности» выполняются отдельно для каждого вида скобок, но не для выражения в целом.

Задачи, в которых важна вложенность объектов, удобно решать с помощью стека. Нас интересуют только открывающие и закрывающие скобки, на остальные символы можно не обращать внимания.

Строка просматривается слева направо. Если очередной символ – открывающая скобка, нужно втолкнуть её на вершину стека. Если это закрывающая скобка, то проверяем, что лежит на вершине стека: если там соответствующая открывающая скобка, то её нужно просто снять со стека. Если стек пуст или на вершине лежит открывающая скобка другого типа, выражение не верное и нужно закончить просмотр. В конце обработки правильной строки стек должен быть пуст. Кроме того, во время просмотра не должно быть ошибок. Работа такого алгоритма иллюстрируется на рисунке (для правильного выражения):



Введём строки **L** и **R**, которые содержат все виды открывающих и соответствующих закрывающих скобок:

```
L = "([{"
R = ")]}"
```

В основной программе создадим пустой стек

```
stack = []
```

Логическая переменная **err** будет сигнализировать об ошибке. Сначала ей присваивается значение **False** (ложь):

```
err = False
```

В основном цикле перебираем все символы строки **s**, в которой записано скобочное выражение:

```
for c in s:                                     # (1)
    if c in L:                                  # (2)
        stack.append(c)                        # (3)
        p = R.find(c)                          # (4)
        if p >= 0:                             # (5)
            if not stack: err = True            # (6)
        else:                                  # (7)
            top = stack.pop()                   # (8)
            if p != L.find(top):                # (9)
                err = True                     # (10)
        if err: break                          # (11)
```

Сначала мы ищем очередной символ строки **s** (который попал в переменную **c**) в строке **L**, то есть среди открывающих скобок (строка программы 2). Если это действительно открывающая скобка, вталкиваем ее в стек (строка 3).

Далее ищем символ среди закрывающих скобок (строка 4). Если нашли, то в первую очередь проверяем, не пуст ли стек. Если стек пуст, выражение неверное и переменная **err** принимает истинное значение (строка 6).

Если в стеке что-то есть, снимаем символ с вершины стека в переменную **top** (строка 8). В строке (9) сравнивается тип (номер) закрывающей скобки **p** и номер открывающей скобки, найденной на вершине стека. Если они не совпадают, выражение неправильное, и в переменную **err** записывается значение **True** (строка 10).

Если при обработке текущего символа обнаружено, что выражение неверное (переменная **err** установлена в **True**), нужно закончить цикл досрочно с помощью оператора **break** (строка 11).

После окончания цикла нужно проверить содержимое стека: если он не пуст, то в выражении есть незакрытые скобки, и оно ошибочно:

```
if len(stack) > 0: err = True
```

В конце программы остается вывести результат на экран:

```
if not err:
    print ( "Выражение правильное." )
else:
    print ( "Выражение неправильное." )
```

Очереди, деки

Все мы знакомы с принципом очереди: первым пришёл – первым обслужен (англ. *FIFO = First In – First Out*). Соответствующая структура данных в информатике тоже называется очередью.


Очередь – это линейный список, для которого введены две операции:

- добавление нового элемента в конец очереди;
- удаление первого элемента из очереди.

Очередь – это не просто теоретическая модель. Операционные системы используют очереди для организации сообщения между программами: каждая программа имеет свою очередь сообщений. Контроллеры жестких дисков формируют очереди запросов ввода и вывода данных. В сетевых маршрутизаторах создается очередь из пакетов данных, ожидающих отправки.

Задача 3. Рисунок задан в виде матрицы A , в которой элемент $A[y][x]$ определяет цвет пикселя на пересечении строки y и столбца x . Перекрасить в цвет 2 одноцветную область, начиная с пикселя (x_0, y_0) . На рисунке показан результат такой заливки для матрицы из 5 строк и 5 столбцов с начальной точкой $(1,0)$.

	0	1	2	3	4
0	0	1	0	1	1
1	1	1	1	2	2
2	0	1	0	2	2
3	3	3	1	2	2
4	0	1	1	0	0

$(1, 0)$


	0	1	2	4	5
0	0	2	0	1	1
1	2	2	2	2	2
2	0	2	0	2	2
3	3	3	1	2	2
4	0	1	1	0	0

Эта задача актуальна для графических программ. Один из возможных вариантов решения использует очередь, элементы которой – координаты пикселей (точек):

```

добавить в очередь точку  $(x_0, y_0)$ 
color = цвет начальной точки
while очередь не пуста:
    взять из очереди точку  $(x, y)$ 
    if  $A[y][x] == color$ :
         $A[y][x] = \text{новый цвет}$ 
        добавить в очередь точку  $(x-1, y)$ 
        добавить в очередь точку  $(x+1, y)$ 
        добавить в очередь точку  $(x, y-1)$ 
        добавить в очередь точку  $(x, y+1)$ 

```

Конечно, в очередь добавляются только те точки, которые находятся в пределах рисунка (матрицы A). Заметим, что в этом алгоритме некоторые точки могут быть добавлены в очередь несколько раз (подумайте, когда это может случиться). Поэтому решение можно несколько улучшить, как-то помечая точки, уже добавленные в очередь, чтобы не добавлять их повторно (попробуйте сделать это самостоятельно). Пусть изображение записано в виде матрицы A , которая на языке Python представлена как список списков (каждый внутренний список – отдельная строка матрицы). Тогда можно определить размеры матрицы так:

```

YMAX = len(A)
XMAX = len(A[0])

```

Значение **YMAX** – это число строк, а **XMAX** – число столбцов. Определим также цвет заливки:

```

NEW_COLOR = 2

```

Зададим координаты начальной точки, откуда начинается заливка:

```

x0 = 1
y0 = 0

```

и запомним её цвет в переменной **color**:

```

color = A[y0][x0]

```

Теперь создадим очередь (как список языка Python) и добавим в эту очередь точку с начальными координатами. Две координаты точки связаны между собой, поэтому в программелучше объединить их в единый блок, который в Python называется «кортеж» и заключается в круглые скобки. Таким образом, каждый элемент очереди – это кортеж из двух элементов:

```

Q = [ (x0, y0) ]

```

Кортеж очень похож на список (обращение к элементам также выполняется по индексу в квадратных скобках), но его, в отличие от списка, нельзя изменять. Остается написать основной цикл:

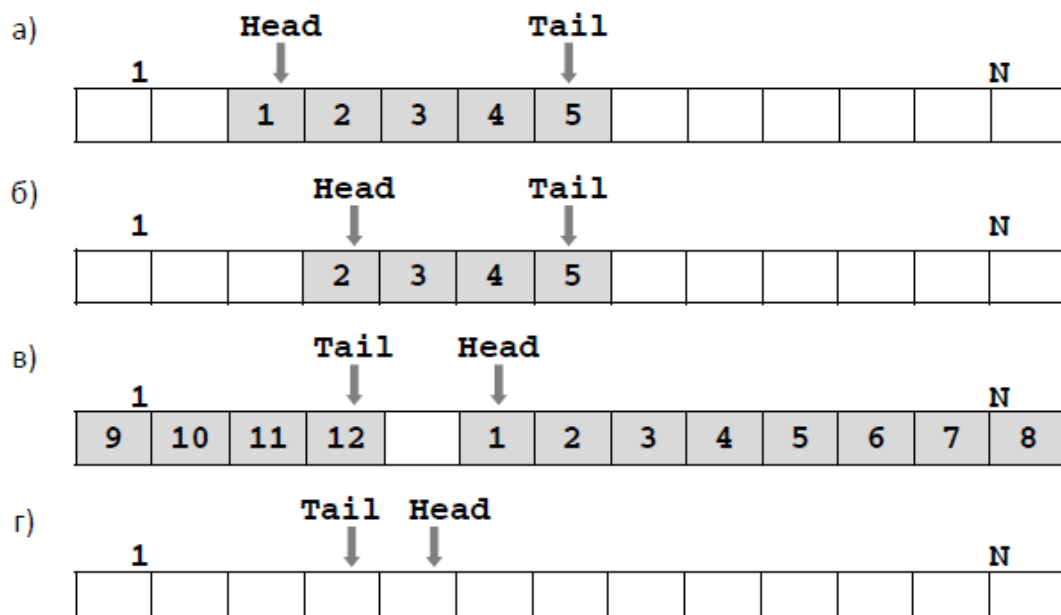
```
while len(Q) > 0:
    x, y = Q.pop(0)
    if A[y][x] == color:
        A[y][x] = NEW_COLOR
        if x > 0: Q.append( (x-1,y) )
        if x < XMAX-1: Q.append( (x+1,y) )
        if y > 0: Q.append( (x,y-1) )
        if y < YMAX-1: Q.append( (x,y+1) )
```

Начало очереди всегда совпадает с первым элементом списка (имеющим индекс 0). Цикл в строке 1 работает до тех пор, пока в очереди есть хоть один элемент (её длина больше нуля).

В строке 2 первый элемент удаляется из очереди. Как мы уже говорили, элемент очереди – это кортеж из двух элементов, поэтому мы сразу разбиваем его на отдельные координаты, используя множественное присваивание.

Если цвет текущей точки совпадает с цветом начальной точки, который хранится в переменной **color** (строка 3), эта точка закрашивается новым цветом (строка 4), и в очередь добавляются все точки, граничащие с текущей и попадающие на поле рисунка.

В некоторых языках программирования размер массива нельзя менять во время работы программы. В этом случае очередь моделируется иначе. Допустим, что мы знаем, что количество элементов в очереди всегда меньше **N**. Тогда можно выделить статический массив из **N** элементов и хранить в отдельных переменных номера первого элемента очереди («головы», англ. *head*) и последнего элемента («хвоста», англ. *tail*). На рисунке а показана очередь из 5 элементов. В этом случае удаление элемента из очереди сводится просто к увеличению переменной **Head** (рисунок б).



При добавление элемента в конец очереди переменная **Tail** увеличивается на 1. Если она перед этим указывала на последний элемент массива, то следующий элемент записывается в начало массива, а переменной **Tail** присваивается значение 1. Таким образом, массив оказывается замкнутым «в кольцо». На рисунке в показана полностью заполненная очередь, а на рисунке г – пустая очередь. Один элемент массива всегда остается незанятым, иначе невозможно будет различить состояния «очередь пуста» и «очередь заполнена».

Отметим, что приведенная здесь модель описывает работу кольцевого буфера клавиатуры, который может хранить до 15 двухбайтных слов.

Существует еще одна линейная динамическая структура данных, которая называется **дек**.

Дек (от англ. *deque* = *double ended queue*, двусторонняя очередь) – это линейный список, в котором можно добавлять и удалять элементы как с одного, так и с другого конца.

Из этого определения следует, что дек может работать и как стек, и как очередь. С помощью дека можно, например, моделировать колоду игральных карт. Для того, чтобы организовать дек в языке Python, также удобно использовать список. При этом основные операции с деком **d** выполняются так:

1. добавление элемента **x** в конец дека: **d.append(x)**
2. добавление элемента **x** в начало дека: **d.insert(0, x)** (добавляемый элемент будет иметь индекс 0)
3. удаление элемента с конца дека: **d.pop()**
4. удаление элемента с начала дека: **d.pop(0)**