1. Radix Sort

class ArrayQueue:  
 *""" FIFO queue implementation using a Python list as underlying storage. """* DEFAULT\_CAPACITY = 10 # moderate capacity for all new queues  
  
 def \_\_init\_\_(self):  
 *"""Create an empty queue."""* self.\_data = [None] \* ArrayQueue.DEFAULT\_CAPACITY  
 self.\_size = 0  
 self.\_front = 0  
  
 def \_\_len\_\_(self):  
 *"""Return the number of elements in the queue."""* return self.\_size  
  
 def is\_empty(self):  
 *"""Return True if the queue is empty."""* return self.\_size == 0  
  
 def first(self):  
 *"""Return (but do not remove) the element at the front of the queue.  
  
 Raise Empty exception if the queue is empty.  
 """* if self.is\_empty():  
 raise Exception('Queue is empty')  
 return self.\_data[self.\_front]  
  
 def dequeue(self):  
 *"""Remove and return the first element of the queue (i.e., FIFO).  
  
 Raise Empty exception if the queue is empty.  
 """* if self.is\_empty():  
 raise Exception('Queue is empty')  
 answer = self.\_data[self.\_front]  
 self.\_data[self.\_front] = None # help garbage collection  
 self.\_front = (self.\_front + 1) % len(self.\_data)  
 self.\_size -= 1  
 return answer  
  
 def enqueue(self, e):  
 *"""Add an element to the back of queue."""* if self.\_size == len(self.\_data):  
 self.\_resize(2 \* len(self.data)) # double the array size  
 avail = (self.\_front + self.\_size) % len(self.\_data)  
 self.\_data[avail] = e  
 self.\_size += 1  
  
 def \_resize(self, cap): # we assume cap >= len(self)  
 *"""Resize to a new list of capacity >= len(self)."""* old = self.\_data # keep track of existing list  
 self.\_data = [None] \* cap # allocate list with new capacity  
 walk = self.\_front  
 for k in range(self.\_size): # only consider existing elements  
 self.\_data[k] = old[walk] # intentionally shift indices  
 walk = (1 + walk) % len(old) # use old size as modulus  
 self.\_front = 0 # front has been realigned  
  
  
def radix\_sort(l):  
 *""" do sorting the list l using radix algorithm and return back the list after sorted  
  
 init division and mod"""* division = 1  
 mod = 10  
  
 start = True  
 while start:  
 start = False  
 """ Create array bucket with 10 queue. Each queue will contain all number have digit   
 (each loop while will from least significant digit to the most significant digit) respective 0 - 9.  
 each digit in number will be in range [0-9], so we need 10 queues """  
 bucket = []  
 for i in range(0, 10):  
 bucket.append(ArrayQueue()) # init queue for each bucket element  
  
 """ Now, go though all number in list l, and put it respective to queue """  
 for num in l:  
 position = num % mod // division # find the position of queue in bucket array for this number  
 bucket[position].enqueue(num) # put this number into position  
 if not start and position > 0: # check if no more digit to stop while loop  
 start = True  
  
 l = [] # init list l to put numbers back in order  
 """ Now, push back element to list l by sorting the digit"""  
 for bucknum in bucket:  
 while not bucknum.is\_empty():  
 l.append(bucknum.dequeue())  
  
 """ Increase the mod and division to do for next digit"""  
 mod = mod \* 10  
 division = division \* 10  
  
 return l # final, return the list l after sorted  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 l = [35, 53, 55, 33, 52, 32, 25]  
 print("Before sort is: ", l) # output: [35, 53, 55, 33, 52, 32, 25]  
 l = radix\_sort(l)  
 print("After sort is: ", l) # output: [25, 32, 33, 35, 52, 53, 55]

1. Postfix Notation

*"""Basic example of an adapter class to provide a stack interface to Python's list."""*#from ..exceptions import Empty  
  
class ArrayStack:  
 *"""LIFO Stack implementation using a Python list as underlying storage."""* def \_\_init\_\_(self):  
 *"""Create an empty stack."""* self.\_data = [] # nonpublic list instance  
  
 def \_\_len\_\_(self):  
 *"""Return the number of elements in the stack."""* return len(self.\_data)  
  
 def is\_empty(self):  
 *"""Return True if the stack is empty."""* return len(self.\_data) == 0  
  
 def push(self, e):  
 *"""Add element e to the top of the stack."""* self.\_data.append(e) # new item stored at end of list  
  
 def top(self):  
 *"""Return (but do not remove) the element at the top of the stack.  
  
 Raise Empty exception if the stack is empty.  
 """* if self.is\_empty():  
 raise Exception('Stack is empty')  
 return self.\_data[-1] # the last item in the list  
  
 def pop(self):  
 *"""Remove and return the element from the top of the stack (i.e., LIFO).  
  
 Raise Empty exception if the stack is empty.  
 """* if self.is\_empty():  
 raise Exception('Stack is empty')  
 return self.\_data.pop() # remove last item from list  
  
  
def evalPostFix(expr):  
 *""" using stack to eval the expression expr, then return this value  
  
 Create stack S"""* S = ArrayStack()  
  
 """ Loop each character in expression. We have problem with the number greater than 9, it should be in () """  
  
 big\_num = "" # this to make the number in all cases: 1 digit or more digits  
 """ if the character in expression is '(' mean that the number has more than 1 digit, it end with ')'"""  
 find\_big\_num = False  
 for c in expr:  
 """ Check if this character is digit or not.   
 If digit, then push it to stack S.  
 If '(', then enable the flag find\_big\_num, and begin create this big num  
 If ')', end of create big num, and disable the find\_big\_num  
 If not digit pop two numbers and do calculate the value"""  
 if c.isdigit():  
 big\_num += c  
 if not find\_big\_num:  
 S.push(big\_num) # push number into stack  
 big\_num = "" # reset the big\_num to get next number in expression  
  
 elif c == '(':  
 find\_big\_num = True  
 elif c == ')':  
 find\_big\_num = False  
 S.push(big\_num) # push number into stack  
 big\_num = "" # reset the big\_num to get next number in expression  
 else:  
 n1 = int(S.pop()) # pop first number  
 n2 = int(S.pop()) # pop second number  
 if c == '+': # check if this is addition  
 n3 = n1 + n2  
 elif c == '-': # check if this is subtraction  
 n3 = n2 - n1  
 elif c == '\*': # check if this is multi  
 n3 = n2 \* n1  
 elif c == '/': # check if this is div  
 n3 = n2 / n1  
  
 """ Push the value back to stack"""  
 S.push(str(n3))  
  
 """ Return the last results. It stored in stack"""  
 return S.pop()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 """ Open file data.txt and read each expression in it, then cal the function evalPostFix to get the value  
 Final, print the output this result"""  
 with open('data.txt', 'r') as f:  
 for line in f:  
 print("The value of expression ", line.rstrip(), " is ", evalPostFix(line.rstrip()))