uc3m | Universidad Carlos III de Madrid

Data Structures and Algorithms.
Author: Isabel Segura Bedmar

Unit 2 – Linear ADT

Problem – Balanced arithmetic expression.

Implement a Python class to guess if the delimiters (,),{,},[,] in an arithmetic expression (e.j. [(5+x)-(y+z)]) are balanced.

- Correct expression example: ()(()){([()])}
- Incorrect expression example: ({[])}

Use a stack to implement your solution. Consider the following hints:

- If an opening symbol is found **[,(,{** it must be pushed.
- If a closing symbol is found **],},)** the element at the top of the stack must be queried. If both symbols belong to the same type, the element must be removed.
- The arithmetic expression is balanced if at the end of the process the stack is empty.

Solution:

```
class Stack:
  """LIFO Stack implementation using a Python list as storage.
  The top of the stack stored at the end of the list."""

  def __init__(self):
    """Create an empty stack"""
    self.items=[]

  def __str__(self):
    #print the elements of the list
    return self.items


  def push(self,e):
```

```
    """Add the element e to the top of the stack"""
    self.items.append(e)

  def pop(self):
    """Remove and return the element from the top of the stack"""
    if self.isEmpty():
      print('Error: Stack is empty')
      return None

    return self.items.pop() #remove last item from the list

  def top(self):
    """Return the element from the top of the stack"""
    if self.isEmpty():
      print('Error: Stack is empty')
      return None

    #returns last element in the list
    return self.items[-1]


  def __len__(self):
    """Return the number of elements in the stack"""
    return len(self.items)

  def isEmpty(self):
    """Return True if the stack is empty"""
    return len(self.items)==0


def balanced(exp):
    """Checks if the parenthesis in exp are balanced"""

    s=Stack()
    for c in exp:
      if c=='(':
        s.push(c)
      elif c==')':
        if s.isEmpty():
          return False
        else:
          s.pop()
      else:
        #ignore any other character
        pass

    return s.isEmpty()


print('(((((((())',balanced('(((((((())'))
print('(()()()())',balanced('(()()()())'))
```

```
print('(((())))',balanced('(((())))'))
print('()))',balanced('()))'))
print('(()()(()',balanced('(()()(()')      )
print('(()((()))))',balanced('(()((()))))')      )

"""The previous function only works for parenthesis. Extend it in order to
deal also with:

    Brace: `{` and `}
    Brackets: `[` and `]'
"""

def sameType(a,b):
  if a=='(' and b==')':
    return True
  if a=='{' and b=='}':
    return True
  if a=='[' and b==']':
    return True

  return False


def sameType1(a, b):
  opening=['(','{','[']
  closing=[')','}',']']
  pos=opening.index(a)
  return b==closing[pos]



def balanced_ext(exp):
    """Checks if the parenthesis in the expression, exp, are balanced"""

    s=Stack()
    for c in exp:
      if c=='(' or c=='{' or c=='[':
        s.push(c)
      elif c==')' or c=='}' or c==']':

        if s.isEmpty():
          return False

        top=s.pop()

        if not sameType(c,top):
          return False
```

```
    return s.isEmpty()


print('()(()){([()])}',balanced_ext('()(()){([()])}'))
print('(((()(())){([()])}))',balanced_ext('(((()(())){([()])}))'))
print(')(()){([()])}',balanced_ext(')(()){([()])}'))
print('(',balanced_ext('('))
print('({[]})',balanced_ext('({[]})'))
```

Problem 2 – Josephus problem.

In the Jewish revolt against Rome, Josephus and 39 of his mates were holding out against the Romans in a cave. With defeat imminent, they resolved that they would rather die than be slaves to the Romans. They decided to arrange themselves in a circle. One man was designated as number one, and they proceeded clockwise killing every seventh man (step). Josephus was among other things an accomplished mathematician; so he instantly figured out where he ought to sit in order to be the last to go. But when the time came, instead of killing himself he joined the Roman side.

Implement a method to find out what position Josephus should sit in order to not be killed. The solution should generalizate for any number of jewish soldiers and any step. The solution should use a queue of integers (each soldier is represented with a number from 1 to n).

In the following video, you can find a nice explanation of this problem.
https://www.youtube.com/watch?v=uCsD3ZGzMgE

Solution:

```
class Queue:
  """FIFO Queue implementation using a Python list as storage.
  We add new elements at the tail of the list (enqueue)
  and remove elements from the head of the list (dequeue)."""

  def __init__(self):
    """Create an empty queue"""
    self.items=[]

  def enqueue(self,e):
    """Add the element e to the tail of the queue"""
    self.items.append(e)

  def dequeue(self):
    """Remove and return the first element in the queue"""
    if self.isEmpty():
      print('Error: Queue is empty')
      return None
```

```
    #remove first item from the list
    return self.items.pop(0)

  def front(self):
    """Return the first element in the queue"""
    if self.isEmpty():
      print('Error: Queue is empty')
      return None

    #returns first element in the list
    return self.items[0]

  def __len__(self):
    """Return the number of elements in the queue"""
    return len(self.items)

  def isEmpty(self):
    """Return True if the queue is empty"""
    return len(self.items)==0

  def __str__(self):
    strQ=''
    for x in self.items:
      strQ=strQ+','+str(x)
    #print the elements of the list
    return strQ[1:]


q=Queue()
print('isEmpty()',q.isEmpty())
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print('Content of queue',str(q))
print('front (first) element',q.front())
print('isEmpty()',q.isEmpty())
print('dequeue():',q.dequeue())
print('Content of queue',str(q))
print('front element:',q.front())
print('size:',len(q))

"""Now, we implement the function for the Josephus problem:"""

def josephus(num, k):
  q=Queue()
  #saved soldiers into the queue.
  for i in range(1,num+1):
    q.enqueue(i)

  while len(q)>1:
    count=1
```

```
    #k-1 dequeue/enqueue operations
    while count<k:
      q.enqueue(q.dequeue())
      count=count+1
    #kill the kth soldier
    print(str(q.dequeue()) + ' was killed')


  print('Surviving position: ' + str(q.front()))


josephus(40,5)
```

Problem – Printer Queue

Implement a Python class, PrinterQueue, to manage a network printer. The printer can receive requests from different machines in a network. The requests should be printed by entry order. Each request includes the following information: id (String) of the machine that performs it (for example "I3493") and the name of the file to print (for example "unit2.pdf"). Please, write a class, named Request, to represent a request.
The class, PrinterQueue, must implement the following operations:

- addRequest: takes a request as input and adds it to the set of requests.
- printWork: gets the first request and shows its data (id and name file) by console (it only simulates the imprension of the request) . The request has to be removed from the set of requests.
- getNumRequest(): returns the total number of requests.
- showAll(): shows all the requests that have have been not printed.
- printAll(): print all the requests. After processing a request, this must be removed.
- Include the needed instructions to test all the methods explained above.

Solution:

```
class Queue:
  """FIFO Queue implementation using a Python list as storage.
  We add new elements at the tail of the list (enqueue)
  and remove elements from the head of the list (dequeue)."""

  def __init__(self):
    """Create an empty queue"""
    self.items=[]

  def enqueue(self,e):
    """Add the element e to the tail of the queue"""
    self.items.append(e)
```

```
  def dequeue(self):
    """Remove and return the first element in the queue"""
    if self.isEmpty():
      print('Error: Queue is empty')
      return None
    #remove first item from the list
    return self.items.pop(0)

  def front(self):
    """Return the first element in the queue"""
    if self.isEmpty():
      print('Error: Queue is empty')
      return None

    #returns first element in the list
    return self.items[0]

  def __len__(self):
    """Return the number of elements in the queue"""
    return len(self.items)

  def isEmpty(self):
    """Return True if the queue is empty"""
    return len(self.items)==0



class Request:
  """This class represent a request to be printed"""
  def __init__(self,idMachine,nameFile):
    self.idMachine=idMachine
    self.nameFile=nameFile

  def __str__(self):
    return self.idMachine+"\t"+self.nameFile

class Printer:
  """This class simulates a network printer"""
  def __init__(self):
    self.q=Queue()

  def addRequest(self, request):
    self.q.enqueue(request)

  def getNumRequest(self):
    return len(self.q)

  def showAll(self):
    for r in self.q.items:
      print(str(r))
```

```
  def printWork(self):
    if self.q.isEmpty():
      print('There is no work to print')
      return

    r=self.q.dequeue()
    print("printing...",str(r))

  def printAll(self):
    while not self.q.isEmpty():
      self.printWork()


##main
p=Printer()
p.addRequest(Request("293939","Unit2.pdf"))
p.addRequest(Request("111","Unit1.pdf"))
p.addRequest(Request("333","Unit3.pdf"))
p.showAll()
print('Num works', p.getNumRequest())

print()
print('print the first work')
p.printWork()
print('showing all')
p.showAll()
print('printing all')
p.printAll()

print('Num works', p.getNumRequest())
```

Problem – Implementation of singly linked list using head.

Solution:

```
class Node:
  def __init__(self, e):
    self.element = e
    self.next = None

"""Now, we can implement the class for a singly linked list. Our class only
uses a refererence, head, for storing the first node, respectively. Moreover,
it includes an atributte, named size, which stores the number of elements in
the list."""

class SinglyLinkedList:
  """This is the implementation of a singly linked list. We only use
  a reference to the first node, named head"""
  def __init__(self):
    self.head=None
    self.size=0
```

```
  def addFirst(self,e):
    """Add a new element, e, at the beginning of the list"""
    #create the new node
    newNode=Node(e)
    #the new node must point to the current head
    newNode.next=self.head
    #update the reference of head to point the new node
    self.head=newNode
    #increase the size of the list
    self.size=self.size+1




  def addLast(self,e):
    """Add a new element at the end of the list"""
    if self.isEmpty():
      self.addFirst(e)
      return

    """Adds a new element, e, at the end of the list"""
    newNode=Node(e)

    #we move throught the list until to reach the last node
    current=self.head
    while current.next is not None:
      current=current.next

    #now, current is the last node
     #the last node must point to the new node (which will be the new last
node)
    current.next=newNode
    self.size=self.size+1




  def isEmpty(self):
    return self.head is None

  def removeFirst(self):
    """Removes the first element of the list"""
    if self.head is None:
      print('Error: list is empty!')
      return None

    #gets the first element, which we will return later
```

```
    first=self.head.element
    #updates head to point to the new head (the next node)
    self.head=self.head.next
    self.size=self.size-1

    return first

  def removeLast(self):
    """Removes and returns the last element of the list"""
    if self.head is None:
      print('Error: list is empty!')
      return None

    #we need to reach the penultimate node
    previous=None
    current=self.head
    while current.next is not None:
        previous=current
        current=current.next

    #here, current is the penultimate node, while current is the last node.
    #gest the element at the last node
    last=current.element
    #now, previous with next must point to None
    previous.next=None

    self.size=self.size-1

    return last

  def __str__(self):
    """Returns a string with the elements of the list"""
    temp=self.head
    strList=''
    while temp is not None:
      strList=strList+','+str(temp.element)
      temp=temp.next
    return strList[1:]


  def getAt(self,index):
    """Returns the element at the index position in the list"""

    #first, check the index is a right position in the list
    if index<0 or index>=self.size:
      print(index,'error: index out of range')
      return None

    #we need to reach the node at the index position in the list
    i=0
    current=self.head
```

```
    while  i<index:
      current=current.next
      i+=1
    #here, current is the node at the index position in the list
    #we return its element
    return current.element


  def contains(self,e):
    """It returns the first position of e into the list. If the element
    does no exist, then it returns -1"""

    index=-1

    found=False

    current=self.head
    #we traverse the nodes while found is not True.
    while current is not None and found==False:
      if current.element==e:
        found=True   #the loop condition becomes False
      current=current.next
      index=index+1

    #Warning: if e does not exist,
    #index is the number of nodes in the list
    if found:
      return index
    else:
      return -1


  def insertAt(self,index,e):
    """This methods inserts a new node containing the element e at the index
    position in the list"""

       #first, we must check that index is a right position. Note that
index=size
    #is a right position for the insertAt method.
    if index<0 or index>self.size:
      print(index, 'Error: index out of range')
      return


    if index==0:
      self.addFirst(e)
    #elif index==self.size:
    #  self.addLast(e)
    else:
      #we need to reach the previous node (the node at the index-1 position)
      i=0
```

```
      previous=self.head
      while i<index-1:
        previous=previous.next
        i=i+1

      #now, previous is the node with index-1
      #create the new node
      newNode=Node(e)
      #newnode must point to the node after previous (which is previous.next)
      newNode.next = previous.next
      #previous must point with its next reference to the new node
      previous.next = newNode
      self.size += 1


  def removeAt(self,index):
    """This methods removes the node at the index position in the list"""

    #We must check that index is a right position in the list
    #Remember that the indexes in a list can range from 0 to size-1
    if index<0 or index>=self.size:
      print(index,'Error: index out of range')
      return

    if index==0:
      self.removeFirst()
    elif index==self.size-1:
      self.removeLast()
    else:
      #we must to reach the node before the node at the index position
      i=0
      previous=self.head
      while i<index-1:
        previous=previous.next
        i=i+1

      #previous is the node at index -1 position


      previous.next = previous.next.next
      self.size=self.size-1

"""Once you have implemented the two classes, you can use them in order to
create your own lists:"""

def test():
    L=SinglyLinkedList()
    print("list:",str(L))


    L.addFirst(5)
```

```
L.addFirst(3)
L.addFirst(2)
L.addFirst(1)


#it should returns 1,2,3,5
print("list:",str(L))
L.addLast(0)
#it should returns 1,2,3,5,0
print("list:",str(L))

L.removeFirst();
#it should returns 2,3,5,0
print("list:",str(L))

L.removeLast();
#it should returns 2,3,5
print("list:",str(L))


for i in range(L.size):
  print("L.getAt({})={}".format(i,L.getAt(i)))

L.getAt(7)

L.insertAt(0,1)
#it should returns 1,2,3,5
print("list:",str(L))

L.insertAt(L.size,6)
#it should returns 1,2,3,5,6
print("list:",str(L))
L.insertAt(L.size,7)
#it should returns 1,2,3,5,6,7
print("list:",str(L))

L.insertAt(-10,0)
L.insertAt(L.size+1,0)

L.removeAt(-1)
L.removeAt(L.size)

L.removeAt(0)
#it should returns 2,3,5,6,7
print("list:",str(L))
L.removeAt(L.size-1)
#it should returns 2,3,5,6
print("list:",str(L))
L.removeAt(2)
#it should returns 2,3,6
print("list:",str(L))
```

```
#main
test()
```

Problem - Implementation of doubly linked list (with head and tail references).

Solution:

```
class DoublyNode:
  def __init__(self, e, n=None, p=None ):
    self.element = e
    self.next = n
    self.prev = p


class DoublyLinkedList:
  def __init__(self):
    """creates an empty list"""
    self.head=None
    self.tail=None
    self.size=0

  def isEmpty(self):
    """Checks if the list is empty"""
    return self.head is None

  def addFirst(self,e):
    """Add a new element, e, at the beginning of the list"""
    #create the new node
    newNode=DoublyNode(e)
    #the new node must point to the current head

    if self.isEmpty():
      self.tail=newNode
    else:
      newNode.next=self.head
```

```
    self.head.prev=newNode

  #update the reference of head to point the new node
  self.head=newNode
  #increase the size of the list
  self.size=self.size+1


def addLast(self,e):
  """Add a new element, e, at the end of the list"""
  #create the new node
  newNode=DoublyNode(e)

  if self.isEmpty():
    self.head=newNode
  else:
    newNode.prev=self.tail
    self.tail.next=newNode

  #update the reference of head to point the new node
  self.tail=newNode
  #increase the size of the list
  self.size=self.size+1


def __str__(self):
  """Returns a string with the elements of the list"""
  temp=self.head
  strList=''
  while temp is not None:
    strList=strList+','+str(temp.element)
    temp=temp.next
  return strList[1:]
```

```
def removeFirst(self):
  """Returns and remove the first element of the list"""
  if self.isEmpty():
    print("Error: list is empty")
    return None

  result=self.head.element
  self.head= self.head.next
  if self.head is None:
    self.tail=None
  else:
    self.head.prev = None

  self.size=self.size-1
  return result

def removeLast(self):
  """Returns and remove the last element of the list"""
  if self.isEmpty():
    print("Error: list is empty")
    return None

  result=self.tail.element
  self.tail= self.tail.prev
  if self.tail is None:
    self.head=None
  else:
    self.tail.next = None

  self.size=self.size-1
  return result


def insertAt(self,index,e):
```

```
  """It inserts the element e at the index position of the list"""
  if index<0 or index>self.size:
    print('Error: index out of range')
    return

  if index==0:
    self.addFirst(e)
  elif index==self.size:
        self.addLast(e)
  else:
    i=0
    aux=self.head
    while i<index:
      aux=aux.next
      i=i+1
    #aux is the node at the index position
    previous=aux.prev
    newNode=DoublyNode(e)
    newNode.next=aux
    newNode.prev=previous
    aux.prev=newNode
    previous.next=newNode
    self.size= self.size+1


def getAt(self,index):
  """Returns the element at the index position in the list"""

  #first, check the index is a right position in the list
  if index<0 or index>=self.size:
    print(index,'error: index out of range')
    return None

  #we need to reach the node at the index position in the list
```

```
  i=0
  current=self.head
  while  i<index:
    current=current.next
    i+=1
  #here, current is the node at the index position in the list
  #we return its element
  return current.element


def contains(self,e):
  """It returns the first position of e into the list. If the element
  does no exist, then it returns -1"""

  index=-1

  found=False

  current=self.head
  #we traverse the nodes while found is not True.
  while current is not None and found==False:
    if current.element==e:
      found=True   #the loop condition becomes False
    current=current.next
    index=index+1

  #Warning: if e does not exist,
  #index is the number of nodes in the list
  if found:
    return index
  else:
    return -1

def removeAt(self,index):
```

```
    """This methods removes the node at the index position in the list"""

    #We must check that index is a right position in the list
    #Remember that the indexes in a list can range from 0 to size-1
    if index<0 or index>=self.size:
      print(index,'Error: index out of range')
      return

    if index==0:
      self.removeFirst()
    elif index==self.size-1:
      self.removeLast()
    else:
      #we must to reach the node at the index position
      i=0
      node=self.head
      while i<index:
        node=node.next
        i=i+1

      prevNode=node.prev
      nextNode=node.next

      prevNode.next=nextNode
      nextNode.prev=prevNode
      self.size=self.size-1

  def show(self,opc):
    result=''
    if opc==0:
      node=self.head
      while node:
        result += str(node.element)+ ' '
        node=node.next
```

```
    else:
      node=self.tail
      while node:
        result += str(node.element)+ ' '
        node=node.prev
    print(result[:-1])

#main

def test():
    """This functions helps us to assess each method of the class"""
    l=DoublyLinkedList()

    for i in range(10):
      l.addLast(i)

    l.show(0)
    l.show(1)
    print(str(l))

    l.removeAt(5)
    print('after removing at 5:', str(l))
    l.removeAt(0)
    print('after removing at 0:', str(l))
    l.removeAt(l.size-1)
    print('after removing at size-1:', str(l))

    l.insertAt(0,0)
    print('after inserting at 0, 0:', str(l))
    l.insertAt(l.size,9)
    print('after inserting at size, 9:', str(l))
    l.insertAt(5,1)
    print('after inserting at 5, 1:', str(l))
    l.insertAt(5,1)
```

```
    print('after inserting at 5, 1:', str(l))

    result=''
    for i in range(l.size):
      result += ' ' + str(l.getAt(i))

    print(result[1:])

    print('contains 10',l.contains(10))
    print('contains 4',l.contains(4))
    print('contains 1',l.contains(1))

    l.removeLast()
    print('after removing last',str(l))
    l.removeFirst()
    print('after removing first',str(l))
    while not l.isEmpty():
      print("removing",l.removeLast())

test()
```

Problem - Checking palindrome words by using a doubly linked list.

A palindrome word is one that reads the same backward as forward. Examples:

Anna, Level, Civic, Madam, Noon.

Implement a Python function that takes a word and returns true if it is palindrome, else false.

In your solution, you **have to use a doubly linked list** where each node contains only one character of the input word.

Solution:

```
from doublylinkedlist import DoublyLinkedList
```

```
def checkPalindrome(word):
  l=DoublyLinkedList()
  for c in word:
    l.addLast(c)

  left=l.head
  right=l.tail
  size=l.size
  i=0
  print(size)
  while i<size//2:
    if left.element!=right.element:
      return False
    i+=1
    left=left.next
    right=right.prev

  return True

word='a'
print(word,checkPalindrome(word))

word='ab'
print(word,checkPalindrome(word))

word='anna'
print(word,checkPalindrome(word))

word='level'
print(word,checkPalindrome(word))

word='12 3 21'
print(word,checkPalindrome(word))
```

Problem 2 - Extending the doubly linked list class.

Given the DoublyLinkedList class (which you can download from aulaglobal).
Implement the following methods:

- getAtRev(self,index): which takes an index and returns the element at the index position starting from the end. For example, given the following list: A,B,C,D,E.
  - ❏ l.getAtRev(0) = 'E'
  - ❏ l.getAtRev(1)= 'D'
  - ❏ l.getAtRev(2)= 'C'
- getAtEff(self,index): as the getAt method, this new version also returns the element at the index position of the list. However, you must try to implement a more efficient method than the original method getAt, taking advantage of the doubly linked lists can be traversed in both forward and backward direction.
- Moreover, you must implement more efficient versions for the methods removeAt and insertAt.
- removeAll(self,e): which removes all the occurrences of e from the list. You must implement two different versions of the list:
  - a method that traverses all the nodes of the list for searching all the occurrences of e.
  - a method that exploits other methods such as removeAt.

Solution:

```
from doublylinkedlist import DoublyLinkedList
from doublylinkedlist import DoublyNode

class DoublyLinkedListExt1(DoublyLinkedList):
  """This class is a subclass of DoublyLinkedList. In this subclass,
  we add new methods such as getAtRev, getAtEff, insertAtEff and removeAtEff,
removeAll"""

  def getAtRev(self,index):
    """Returns the element at index position of the list, starting from the
end"""
    aux=self.tail
    i=0
    while aux!=None:
      if i==index:
        return aux.element
      aux=aux.prev
      i+=1

    print(index,' is out of range')
```

```
  return None

def getAtEff(self, index):
  """Returns the element at the index position taking advantage of the
  reversing order"""
  if index<0 or index>=self.size:
    print('error: index out of range')
  if index <= self.size//2:
    print(index,'searching from the beginning')
    return self.getAt(index)
  else:
    print(index,'searching from the tail')

    aux=self.tail
    i=self.size-1
    while aux!=None:
      if i==index:
        return aux.element
      aux=aux.prev
      i-=1


def insertAtEff(self,index,elem):
  """It inserts the element e at the index position of the list,
  taking advantage of traversing the list backward"""
  if index<0 or index>self.size:
    print('Error: index out of range')
    return

  if index==0:
    self.addFirst(elem)
  elif index==self.size:
    self.addLast(elem)
  elif index<=self.size//2:
    print(index,'insert- starting from the head')
    self.insertAt(index,elem)
  else:
    print(index,'insert- starting from the end')
    i=self.size-1
    aux=self.tail
    while i>index:
      aux=aux.prev
      i-=1
    #aux is the node at the index position
    previous=aux.prev
    newNode=DoublyNode(elem)
    newNode.next=aux
    newNode.prev=previous
    aux.prev=newNode
    previous.next=newNode
    self.size= self.size+1
```

```
def removeAtEff(self,index):
  """It removes the element at the index position of the list,
  taking advantage of traversing the list backward"""
  if index<0 or index>self.size:
    print('Error: index out of range')
    return

  if index==0:
    return self.removeFirst()
  elif index==self.size-1:
    return self.removeLast()
  elif index<=self.size//2:
    print(index,'remove- starting from the head')
    return self.removeAt(index)
  else:
    #we must to reach the node at the index position
    print(index,'remove- starting from the tail...')
    i=self.size-1
    node=self.tail
    while i>index:
      node=node.prev
      i-=1

    #node is the node that we want to remove
    node.prev.next=node.next
    node.next.prev=node.prev

    self.size=self.size-1
    return node.element

 def removeAll(self,e):
  """This functions takes an element as parameter and removes all its
  occurrences from the list"""
  pos=self.contains(e)
  while pos!=-1:
    self.removeAt(pos)
    pos=self.contains(e)

def removeAll2(self,e):
  """This functions takes an element as parameter and removes all its
  occurrences from the list"""
  node=self.head
  while node:
      if node.element==e:
          #we must remove this node
          if node is self.head:
              self.removeFirst()
          elif node is self.tail:
              self.removeLast()
          else:
```

```
                prevNode=node.prev
                nextNode=node.next
                prevNode.next=nextNode
                nextNode.prev=prevNode
                self.size-=1
        node=node.next


#main
def test():
    l=DoublyLinkedListExt1()

    for i in range(10):
      l.addLast(i)

    print(str(l))

    print("l.getAtRev(0)",l.getAtRev(0))

    print("l.getAtRev(1)",l.getAtRev(1))
    print("l.getAtRev(8)",l.getAtRev(8))
    print("l.getAtRev(10)",l.getAtRev(10))



    #for i in range(l.size):
    #  print(l.getAtEff(i))



    l.removeAtEff(1)
    print("after l.removeAtEff(1)",str(l))

    l.removeAtEff(8)
    print("after l.removeAtEff(8)",str(l))

    l.removeAtEff(-1)
    print("after l.removeAtEff(-1)",str(l))


    l.insertAtEff(l.size-1,5)
    print("after l.insertAtEff(l.size-1,5)",str(l))
    l.insertAtEff(l.size-3,5)
    print("after l.insertAtEff(l.size-3,5)",str(l))
    l.insertAtEff(3,5)
    print("after l.insertAtEff(3,5)",str(l))

for i in range(10):
        #l.addLast(i)
      #we add a random number N, such as 0<=N<=10
      l.addLast(random.randint(0,5))
```

```
    print("original list: ", str(l))
    l.removeAll(0)
    print("after removing 0: ", str(l))
    l.removeAll(1)
    print("after removing 1: ", str(l))
    l.removeAll(2)
    print("after removing 2: ", str(l))
    l.removeAll(3)
    print("after removing 3: ", str(l))
    l.removeAll(6)
    print("after removing 6 (which does not exist!!): ", str(l))


test()
```