Университет ИТМО

Мобильные и сетевые технологии

**Лабораторная работа № 4**

***«Алгоритмы и структуры данных»***

Работу подготовил:

Кононов Степан

Группа К3140

Преподаватель: Татьяна Александровна Харьковская

Санкт-Петербург, 2021

# Задача 1.

*'''Реализация очереди в Python'''*class Queue:  
 def \_\_init\_\_(self):  
 self.queue = []  
 self.queueStart = 0  
  
 def push(self, val):  
 self.queue.append(val)  
  
 def pop(self):  
 if len(self.queue) == 0:  
 print("ERROR: current queue is empty")  
 return None  
 '''Передвигаем начало очереди, если неиспользуемых элем более половины удаляем их'''  
 result = self.queue[self.queueStart]  
 self.queueStart += 1  
 if self.queueStart > len(self.queue) // 2:  
 self.queue[:self.queueStart] = []  
 self.queueStart = 0  
 return result  
  
 def top(self):  
 return self.queue[0]  
  
 def size(self):  
 return len(self.queue)  
  
 def isempty(self):  
 return len(self.queue) == 0  
  
 def clear(self):  
 self.queue[:] = []  
  
queue = Queue()  
ans = []  
colvo = int(input())  
  
for i in range(colvo):  
 command = input()  
 temp = command.split()  
 com\_type, num = temp[0], (temp[1] if len(temp) == 2 else None)  
 if com\_type == "+":  
 queue.push(num)  
 if com\_type == "-":  
 ans.append(queue.pop())  
if len(ans) != 0:  
 print(\*ans)

# Задача 3

*'''Скобочная последовательность'''*class Stack:  
 def \_\_init\_\_(self):  
 self.stack = []  
  
 def push(self, val):  
 self.stack.append(val)  
  
 def pop(self):  
 if len(self.stack) == 0:  
 print("ERROR: current stack is empty")  
 return None  
 return self.stack.pop()  
  
 def top(self):  
 if len(self.stack) == 0:  
 return None  
 return self.stack[-1]  
  
 def isempty(self):  
 return len(self.stack) == 0  
  
 def clear(self):  
 self.stack[:] = []  
  
  
num = int(input())  
  
for i in range(num):  
 subsequence = input()  
 stack = Stack()  
 for sym in subsequence:  
 if stack.isempty():  
 stack.push(sym)  
 else:  
 if stack.top() == '(' and sym == ')':  
 stack.pop()  
 if stack.top() == '[' and sym == ']':  
 stack.pop()  
 if (stack.top() == '(' or stack.top() == '[') and (sym == '(' or sym == '['):  
 stack.push(sym)  
 if stack.isempty():  
 print('YES')  
 else:  
 print('NO')

# Задача 6

*'''Реализация очереди c минимумом в Python'''*from math import inf  
  
class Queue:  
 def \_\_init\_\_(self):  
 self.queue = []  
 self.queueStart = 0  
 self.queueMin = inf  
  
 def push(self, val):  
 self.queue.append(val)  
  
 def pop(self):  
 if len(self.queue) == 0:  
 print("ERROR: current queue is empty")  
 return None  
  
 '''Передвигаем начало очереди, если неиспользуемых элем более половины удаляем их'''  
 result = self.queue[self.queueStart]  
 self.queueStart += 1  
 if self.queueStart > len(self.queue) // 2:  
 self.queue[:self.queueStart] = []  
 self.queueStart = 0  
  
 return result  
  
 def top(self):  
 return self.queue[0]  
  
 def size(self):  
 return len(self.queue)  
  
 def isempty(self):  
 return len(self.queue) == 0  
  
 def clear(self):  
 self.queue[:] = []  
 def min\_in\_queue(self):  
 if len(self.queue) > 0:  
 return min(self.queue[self.queueStart:])  
  
queue = Queue()  
ans = []  
colvo = 7  
c = ['+ 1', '?', '+ 10', '?', '-', '?', '-']  
for i in range(colvo):  
 command = c[i]  
 temp = command.split()  
 com\_type, num = temp[0], (temp[1] if len(temp) == 2 else None)  
 if com\_type == "+":  
 queue.push(int(num))  
 if com\_type == "-":  
 queue.pop()  
 if com\_type == "?":  
 ans.append(queue.min\_in\_queue())  
  
  
if len(ans) != 0:  
 print(\*ans)

# Задача 7

import gc  
  
  
class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
 self.prev = None  
class Doublo\_LinkedList:  
 def \_\_init\_\_(self):  
 *# Инициализируем головной указатель в начале он никуда не указывает* self.head = None  
 self.tail = None  
 self.lenght = 0  
  
 *# Поместить в конец списка* def push(self, new\_data):  
 *# Создаем новый элемент списка с указанным значением* new\_node = Node(new\_data)  
  
 *# Так-как это последний элемент* new\_node.next = None  
  
 *# Если список был пустым* if self.head is None:  
 new\_node.prev = None  
 self.tail = new\_node  
 else:  
 self.head.next = new\_node  
 new\_node.prev = self.head  
 *# Перемещаем указатель на текущий элемент* self.head = new\_node  
 self.lenght += 1  
  
 *# Вставить после определенного эхлемента* def insertAfter(self, prev\_node, new\_data):  
  
 *# Проверим что предыдущий элемент сущствует* if prev\_node is None:  
 print("This node doesn't exist in DLL")  
 return  
  
 *# Создаём новый элемент списка* new\_node = Node(new\_data)  
  
 *# Задаём след элем как след после prev\_node* new\_node.next = prev\_node.next  
 *# Задаем предыдущий элемент для текущего* new\_node.prev = prev\_node  
  
 *# Задаем предыдущему элементу след как текущий* prev\_node.next = new\_node  
  
 *# Если мы не добавляли в конец тогда указываем след элементу указатель на текущий* if new\_node.next is not None:  
 new\_node.next.prev = new\_node  
 else:  
 self.head = new\_node  
 self.lenght += 1  
  
 *# Вставить перед определенного эхлемента* def insertBefore(self, next\_node, new\_data):  
 *# Проверим что "след" элемент сущствует* if next\_node is None:  
 print("This node doesn't exist in DLL")  
 return  
  
 *# Создаём новый элемент списка* new\_node = Node(new\_data)  
  
 *# Задаём след элем как след после prev\_node* new\_node.next = next\_node  
 *# Задаем предыдущий элемент для текущего* new\_node.prev = next\_node.prev  
  
 next\_node.prev = new\_node  
  
 *# Подвязывваем пердыдущий элемент если он есть к текущему* if new\_node.prev is not None:  
 new\_node.prev.next = new\_node  
 *# Обновляем хвост* if new\_node.prev is None:  
 self.tail = new\_node  
 self.lenght += 1  
  
 *# Печать списка (развернут)* def printList(self):  
 if self.is\_empty():  
 return  
 cur\_node = self.head  
 it\_first\_iter = True  
 while cur\_node.prev != None:  
 if it\_first\_iter:  
 print(cur\_node.data, '<-- head')  
 it\_first\_iter= False  
 else:  
 print(cur\_node.data)  
 cur\_node = cur\_node.prev  
 print(cur\_node.data, '<-- tail')  
  
 *# Получение элемнта по индексу* def take\_elem(self, index):  
 if index >= self.lenght:  
 print("List index out of range")  
 return  
 cur\_node = self.head  
 iterables = self.lenght - 1 - index  
 for i in range(iterables):  
 cur\_node = cur\_node.prev  
 return cur\_node  
 *# Удаление элемента* def deleteNode(self, dele):  
  
 *# Base Case* if self.head is None or dele is None:  
 return  
 *# If node to be deleted is head node* if self.head == dele:  
 *# Если удаляем последний элемент из списка* if self.head == self.tail:  
 self.tail = None  
 self.head = self.head.prev  
  
 elif dele.prev == None:  
 self.tail = dele.next  
 dele.next.prev = None  
  
 else:  
 dele.prev.next = dele.next  
 dele.next.prev = dele.prev  
  
 self.lenght -= 1  
 gc.collect()  
 *# Поиск элемента по значению (возвращает первое правое вхождение)* def rfind\_elem(self, elem\_data):  
 cur\_node = self.head  
 while cur\_node.data != elem\_data:  
 cur\_node = cur\_node.prev  
 return cur\_node  
  
 def is\_empty(self):  
 return self.head == None  
  
 *# Получение верхнего элемнта* def top(self):  
 return self.head  
 def down(self):  
 return self.tail  
  
stack = Doublo\_LinkedList()  
*#temp = input()*commands = '89+17-\*'  
for elem in commands:  
 if elem in '+-\*':  
 a = stack.head.prev.data  
 b = stack.head.data  
 stack.deleteNode(stack.head)  
 stack.deleteNode(stack.head)  
 if elem == '+':  
 result = a + b  
 elif elem == '-':  
 result = a - b  
 else:  
 result = a \* b  
 stack.push(result)  
 else:  
 stack.push(int(elem))  
print(stack.tail.data)

# Задача 12

for i in range(iterables):  
 cur\_node = cur\_node.prev  
 return cur\_node  
 *# Удаление элемента* def deleteNode(self, dele):  
  
 *# Base Case* if self.head is None or dele is None:  
 return  
 *# If node to be deleted is head node* if self.head == dele:  
 self.head = self.head.prev  
 elif dele.prev == None:  
 dele.next.prev = None  
 else:  
 dele.prev.next = dele.next  
 dele.next.prev = dele.prev  
 self.lenght -= 1  
 gc.collect()  
 *# Поиск элемента по значению (возвращает первое правое вхождение)* def rfind\_elem(self, elem\_data):  
 cur\_node = self.head  
 while cur\_node.data != elem\_data:  
 cur\_node = cur\_node.prev  
 return cur\_node  
  
  
 *# Получение верхнего элемнта* def top(self):  
 return self.head  
  
  
line = Doublo\_LinkedList()  
crowd = None  
'''  
for i in range (line.lenght):  
 print(line.take\_elem(i).data, ' i = ', i)'''  
  
  
*#n, m = map(int, input().split())*n, m = 3, 4  
com = ['left 2 1', 'right 3 1', 'leave 1', 'name 2']  
  
line.push(1) *# 0*for x in range(m):  
 command = com[x]  
 temp = command.split()  
 com\_type, I, J = temp[0], (int(temp[1]) if len(temp) > 1 else None), (int(temp[2]) if len(temp) > 2 else None)  
 if (com\_type == 'left'):  
 line.insertBefore(line.rfind\_elem(J), I)  
 if (com\_type == 'right'):  
 line.insertAfter(line.rfind\_elem(J), I)  
 if (com\_type == 'leave'):  
  
 line.deleteNode(line.rfind\_elem(I))  
  
 if (com\_type == 'name'):  
  
 if line.rfind\_elem(I).prev == None:  
 left = 0  
 else:  
 left = line.rfind\_elem(I).prev.data  
 if line.rfind\_elem(I).next == None:  
 right = 0  
 else:  
 right = line.rfind\_elem(I).next.data  
 print(left, right)  
for i in range (line.lenght):  
 print(line.take\_elem(i).data, ' i = ', i)

# Задача 13

import gc  
from time import perf\_counter  
  
  
class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
 self.prev = None  
  
class Doublo\_LinkedList:  
 def \_\_init\_\_(self):  
 *# Инициализируем головной указатель в начале он никуда не указывает* self.head = None  
 self.tail = None  
 self.lenght = 0  
  
 *# Поместить в конец списка* def push(self, new\_data):  
 *# Создаем новый элемент списка с указанным значением* new\_node = Node(new\_data)  
  
 *# Так-как это последний элемент* new\_node.next = None  
  
 *# Если список был пустым* if self.head is None:  
 new\_node.prev = None  
 self.tail = new\_node  
 else:  
 self.head.next = new\_node  
 new\_node.prev = self.head  
 *# Перемещаем указатель на текущий элемент* self.head = new\_node  
 self.lenght += 1  
  
 *# Вставить после определенного эхлемента* def insertAfter(self, prev\_node, new\_data):  
  
 *# Проверим что предыдущий элемент сущствует* if prev\_node is None:  
 print("This node doesn't exist in DLL")  
 return  
  
 *# Создаём новый элемент списка* new\_node = Node(new\_data)  
  
 *# Задаём след элем как след после prev\_node* new\_node.next = prev\_node.next  
 *# Задаем предыдущий элемент для текущего* new\_node.prev = prev\_node  
  
 *# Задаем предыдущему элементу след как текущий* prev\_node.next = new\_node  
  
 *# Если мы не добавляли в конец тогда указываем след элементу указатель на текущий* if new\_node.next is not None:  
 new\_node.next.prev = new\_node  
 else:  
 self.head = new\_node  
 self.lenght += 1  
  
 *# Вставить перед определенного эхлемента* def insertBefore(self, next\_node, new\_data):  
 *# Проверим что "след" элемент сущствует* if next\_node is None:  
 print("This node doesn't exist in DLL")  
 return  
  
 *# Создаём новый элемент списка* new\_node = Node(new\_data)  
  
 *# Задаём след элем как след после prev\_node* new\_node.next = next\_node  
 *# Задаем предыдущий элемент для текущего* new\_node.prev = next\_node.prev  
  
 next\_node.prev = new\_node  
  
 *# Подвязывваем пердыдущий элемент если он есть к текущему* if new\_node.prev is not None:  
 new\_node.prev.next = new\_node  
 *# Обновляем хвост* if new\_node.prev is None:  
 self.tail = new\_node  
 self.lenght += 1  
  
 *# Печать списка (развернут)* def printList(self):  
 if self.is\_empty():  
 return  
 cur\_node = self.head  
 it\_first\_iter = True  
 while cur\_node.prev != None:  
 if it\_first\_iter:  
 print(cur\_node.data, '<-- head')  
 it\_first\_iter= False  
 else:  
 print(cur\_node.data)  
 cur\_node = cur\_node.prev  
 print(cur\_node.data, '<-- tail')  
  
 *# Получение элемнта по индексу* def take\_elem(self, index):  
 if index >= self.lenght:  
 print("List index out of range")  
 return  
 cur\_node = self.head  
 iterables = self.lenght - 1 - index  
 for i in range(iterables):  
 cur\_node = cur\_node.prev  
 return cur\_node  
 *# Удаление элемента* def deleteNode(self, dele):  
  
 *# Base Case* if self.head is None or dele is None:  
 return  
 *# If node to be deleted is head node* if self.head == dele:  
 *# Если удаляем последний элемент из списка* if self.head == self.tail:  
 self.tail = None  
 self.head = self.head.prev  
  
 elif dele.prev == None:  
 self.tail = dele.next  
 dele.next.prev = None  
  
 else:  
 dele.prev.next = dele.next  
 dele.next.prev = dele.prev  
  
 self.lenght -= 1  
 gc.collect()  
 *# Поиск элемента по значению (возвращает первое правое вхождение)* def rfind\_elem(self, elem\_data):  
 cur\_node = self.head  
 while cur\_node.data != elem\_data:  
 cur\_node = cur\_node.prev  
 return cur\_node  
  
 def is\_empty(self):  
 return self.head == None  
  
 *# Получение верхнего элемнта* def top(self):  
 return self.head  
 def down(self):  
 return self.tail  
  
class Doublo\_LinkedList\_Fast\_Stepa\_edition:  
 def \_\_init\_\_(self):  
 *# Инициализируем головной указатель в начале он никуда не указывает* self.head = None  
 self.tail = None  
 self.lenght = 0  
 *# Словарь значение - элемент | при обращении возвращает последнее вхождение* self.elem\_dict = dict()  
  
 *# Поместить в конец списка* def push(self, new\_data):  
 *# Создаем новый элемент списка с указанным значением* new\_node = Node(new\_data)  
  
 *# Так-как это последний элемент* new\_node.next = None  
  
 *# Если список был пустым* if self.head is None:  
 new\_node.prev = None  
 self.tail = new\_node  
 else:  
 self.head.next = new\_node  
 new\_node.prev = self.head  
 *# Перемещаем указатель на текущий элемент* self.head = new\_node  
 self.lenght += 1  
 self.elem\_dict[new\_node.data] = new\_node  
  
 *# Вставить после определенного эхлемента* def insertAfter(self, prev\_node, new\_data):  
  
 *# Проверим что предыдущий элемент сущствует* if prev\_node is None:  
 print("This node doesn't exist in DLL")  
 return  
  
 *# Создаём новый элемент списка* new\_node = Node(new\_data)  
  
 *# Задаём след элем как след после prev\_node* new\_node.next = prev\_node.next  
 *# Задаем предыдущий элемент для текущего* new\_node.prev = prev\_node  
  
 *# Задаем предыдущему элементу след как текущий* prev\_node.next = new\_node  
  
 *# Если мы не добавляли в конец тогда указываем след элементу указатель на текущий* if new\_node.next is not None:  
 new\_node.next.prev = new\_node  
 else:  
 self.head = new\_node  
 self.lenght += 1  
 self.elem\_dict[new\_node.data] = new\_node  
  
 *# Вставить перед определенного эхлемента* def insertBefore(self, next\_node, new\_data):  
 *# Проверим что "след" элемент сущствует* if next\_node is None:  
 print("This node doesn't exist in DLL")  
 return  
  
 *# Создаём новый элемент списка* new\_node = Node(new\_data)  
  
 *# Задаём след элем как след после prev\_node* new\_node.next = next\_node  
 *# Задаем предыдущий элемент для текущего* new\_node.prev = next\_node.prev  
  
 next\_node.prev = new\_node  
  
 *# Подвязывваем пердыдущий элемент если он есть к текущему* if new\_node.prev is not None:  
 new\_node.prev.next = new\_node  
 *# Обновляем хвост* if new\_node.prev is None:  
 self.tail = new\_node  
 self.lenght += 1  
 self.elem\_dict[new\_node.data] = new\_node  
  
 *# Печать списка (развернут)* def printList(self):  
 if self.is\_empty():  
 return  
 cur\_node = self.head  
 it\_first\_iter = True  
 while cur\_node.prev != None:  
 if it\_first\_iter:  
 print(cur\_node.data, '<-- head')  
 it\_first\_iter= False  
 else:  
 print(cur\_node.data)  
 cur\_node = cur\_node.prev  
 print(cur\_node.data, '<-- tail')  
  
 *# Получение элемнта по индексу* def take\_elem(self, index):  
 if index >= self.lenght:  
 print("List index out of range")  
 return  
 cur\_node = self.head  
 iterables = self.lenght - 1 - index  
 for i in range(iterables):  
 cur\_node = cur\_node.prev  
 return cur\_node  
 *# Удаление элемента* def deleteNode(self, dele):  
  
 *# Base Case* if self.head is None or dele is None:  
 return  
 *# Если это не сработает то плохо* self.elem\_dict.pop(dele.data)  
 *# If node to be deleted is head node* if self.head == dele:  
 *# Если удаляем последний элемент из списка* if self.head == self.tail:  
 self.tail = None  
 self.head = self.head.prev  
  
 elif dele.prev == None:  
 self.tail = dele.next  
 dele.next.prev = None  
  
 else:  
 dele.prev.next = dele.next  
 dele.next.prev = dele.prev  
  
 self.lenght -= 1  
 gc.collect()  
 *# Поиск элемента по значению (возвращает первое правое вхождение)* def rfind\_elem(self, elem\_data):  
 cur\_node = self.head  
 while cur\_node.data != elem\_data:  
 cur\_node = cur\_node.prev  
 return cur\_node  
  
 def is\_empty(self):  
 return self.head == None  
  
 *# Получение верхнего элемнта* def top(self):  
 return self.head  
 def down(self):  
 return self.tail  
  
line = Doublo\_LinkedList()  
fast\_line = Doublo\_LinkedList\_Fast\_Stepa\_edition()  
  
col\_vo = 10\*\*4  
t\_start = perf\_counter()  
for i in range(col\_vo//2):  
 line.push(i)  
for i in range(col\_vo//2, col\_vo):  
 line.insertBefore(line.rfind\_elem(0), i)  
  
time\_1 = perf\_counter() - t\_start  
print(time\_1)  
  
t\_start = perf\_counter()  
for i in range(col\_vo//2):  
 fast\_line.push(i)  
for i in range(col\_vo//2, col\_vo):  
 fast\_line.insertBefore(fast\_line.elem\_dict[0], i)  
  
time\_2 = perf\_counter() - t\_start  
print(time\_2)