Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Мегафакультет трансляционных информационных технологий

Факультет инфокоммуникационных технологий

Дисциплина: Алгоритмы и структуры данных

**Отчет по Лабораторной работе №4**

Выполнила: Микулина Алиса Романовна

Группа: K3143, 1 курс

Преподаватель: Харьковская Татьяна Александровна

Санкт-Петербург

25.11.2021

**Описание задания**

**Задание 2.**

Реализовать работу очереди. Для каждой операции изъятия элемента необходимо выводить результат.

**Задание 4.**

Написать код для определения, является ли последовательность из скобок ()[]{} правильной. В случае, если последовательность неправильная, вывести первую неправильную закрывающую скобку. Если таковой нет, то первую неправильную открывающую. Если неправильных скобок не найдено, то последовательность верная.

**Задание 5.**

Написать стек таким образом, чтобы он поддерживал 3 операции: push(), pop() и max(). Все три операции должны выполняться за константное время.

**Задание 8.**

Реализовать код, который бы считал результат любого выражения, записанного постфиксной записью.

**Задание 9.**

Реализовать работу очереди в поликлинике. Помимо добавления элемента в конец очереди и вывода элемента из начала очереди, необходимо иметь возможность добавлять элемент в середину очереди (да-да, те самые люди, которым “только спросить”).

**Задание 13.1.**

Реализовать стек на основе связного списка с функциями isEmpty(), push(), pop() и с возможностью вывода данных.

**Задание 13.3.**

Реализовать односвязный список с функциями вывода содердимого списка, добавления элемента в начало списка, удаления элемента с начала списка, добавления и удаления элемента после заданного элемента (key); поиска элемента в списке.

**Описание решения и исходный код**

**Задача 2.**

Код для решения достаточно прост. У нас есть изначально пустой массив для очереди, у которого также есть значения nose и tail. Nose — начало очереди, откуда удаляются элементы. Tail — хвост очереди, в который элементы добавляются. При добавлении нового элемента в конец массива, мы увеличиваем tail на 1, а при удалении элемента из начала, увеличиваем nose на 1. Таким образом, когда мы захотим удалить еще один элемент из очереди, это будет возможно сделать за O(1), так же, как и добавление элемента.

def add(my\_queue, number):  
 global tail  
 my\_queue.append(number)  
 tail += 1  
  
def delete(my\_queue, ans):  
 global nose  
 ans.append(my\_queue[nose])  
 nose += 1  
  
my\_queue = []  
ans = []  
nose, tail = 0, 1  
  
with open("input.txt", "r") as f:  
 iterations = int(f.readline())  
 for i in range(iterations):  
 line = f.readline().strip()  
 if len(line) > 2:  
 action, number = list(line.split(" "))  
 add(my\_queue, number)  
 else:  
 action = line  
 delete(my\_queue, ans)  
  
  
with open("output.txt", "w") as d:  
 for elem in ans:  
 d.write(str(elem) + '\n')

**Задача 4.**

Задача для изгнания демонов из тела несчастного выполняющего. Планировалось потратить на нее часик-другой, а в итоге это вылилось в 5 часов добрых слов о питоне и размышление о смысле жизни. Казалось бы, понял принцип того, верная ли скобочная последовательность или нет, и жизнь становится сказкой, НО…

Но тогда везение человека, получившего эту задачу как часть их варианта, было бы слишком велико (и да, я не очень понимаю, почему 3-я задача распределялась по вариантам наравне с 4, ведь они РЕАЛЬНО разной сложности). Короче говоря, возможность определить, верна ли последовательность, была мною реализована в течение где-то часа, а потом начались танцы с бубном…

Бубен штука интересная, сколько с ним не танцуй, лучше вряд ли станет, нужно думать, а это сложно (но я умею, честно). Для начала я прописала все так, чтобы программа могла определить, верна ли последовательность, для одного вида скобок: (). Логика, мною используемая, была очень проста. Пробегаться по строке начинаем с начала. Берем элемент строки, смотрим, является ли он скобкой. Если не является, просто его пропускаем, так как буковки и циферки нас не интересуют. Потом смотрим, является ли взятая нами скобка открывающей. Если да, то кладем ее в стек. Если скобка закрывающая, то снимаем предыдущую скобку со стека. Если первая скобка сразу закрывающая, выводим индекс этой скобки. Если после полной пробежки по строке в стеке все еще остаются какие-то скобки, мы с чистой душой выводим индекс первой из них.

Да. Было бы прекрасно, если бы оно продолжило работать при увеличении количества скобок. Было бы прекрасно. Слишком…

Создаем словарь с соответствующими скобочками, то есть ) соответствует (, ] соответствует [, а } соответствует {. Это поможет нам избежать огромного количества ifов, что значительно упрощает код. Теперь на первом этапе опять проверяем, скобка ли наш символ, потом, если символ находится в значениях словаря (то есть является открывающей скобкой), мы с чистой душой кладем его на стек. Как только мы встречаем закрывающую скобку, используем магию словарей. Смотрим, соответствует ли эта скобка значению, соответствующему ключу в словаре, при этом ключ — последний элемент на стеке. Если соответствует — ура! Снимаем последний элемент со стека, идем дальше. Если нет — ошибка найдена, и тут мы что делаем? Выводим индекс скобки? А вот и нет, добавляем скобку на стек! Почему? Хороший вопрос, над которым я думала в течение где-то трех часов (почему же не получается просто индекс-то вывести… какая жизнь сложная штука, да?). А секрет вот в чем: у нас же двухэтапная проверка. Если ошибка найдена, ее еще корректно вывести нужно. Если мы споткнулись где-то в конце строки, но при этом ошибка у нас только в открытых скобках, нельзя выводить последний элемент, а надо вывести индекс первой открывающей скобки. Для этого сооружаем цикл на проверку, есть ли у нас в стеке закрывающие скобки. При этом обходим стек с конца, чтобы поймать первую закрывающую скобочку. Если в стеке таковой нет, выводим индекс первой скобки (уже очевидно, что она открывающая, ведь закрывающих больше нет).

Дальше, по сути, все логично. Если наш стек внезапно оказывается пустым, а мы хотим добавить закрывающую скобку — выводим индекс. Если доходим до конца и в стеке все еще есть элементы, выводим индекс самой первой скобки, ведь закрывающих скобок в стеке уже нет (потому что если бы в стеке была еще одна закрывающая скобка, мы бы это отловили в нашем прекрасном цикле).

“Success” мы выводим только в одном случае — если проверка доходит до конца с пустым стеком. В таком случае все скобки нашли себе корректную пару и все прекрасно.

Демоны изгнаны, счастье пришло, мозги вышли из чата, потому что было уже пол четвертого ночи.

def is\_correct(line):  
 stack = []  
 equals\_back = {')': '(', ']': '[', '}': '{'}  
 for i in range(0, len(line)):  
 if line[i] not in ['(', ')', '[', ']', '{', '}']:  
 continue  
 if line[i] in equals\_back.values():  
 stack.append([line[i], i])  
 else:  
 if stack:  
 if stack[-1][0] == equals\_back[line[i]]:  
 stack.pop()  
 else:  
 stack.append([line[i], i])  
 for p in range(len(stack) - 1, -1, -1):  
 if stack[p][0] in equals\_back.keys():  
 return str(stack[p][-1] + 1)  
 return str(stack[0][-1] + 1)  
 else:  
 return str(i + 1)  
 if not stack:  
 return 'Success'  
 else:  
 return str(stack[0][-1] + 1)  
  
  
with open("input.txt", "r") as f:  
 line = f.readline().split()  
 ans = is\_correct(line[0])  
  
with open("output.txt", "w") as d:  
 d.write(ans)

**Задача 5.**

Стек с максимумом. Это же обычный стек, да? Ну, почти)

В обычном стеке мы добавляем наверх и убираем сверху элементы. Оба действия за O(1). Что же нужно сделать, чтобы за O(1) можно было еще и выводить максимум current стека. Первая идея — сделать так, чтобы первый элемент стека было нельзя удалить, и чтобы это именно он хранил в себе максимум. По логике, можно бы было при каждом добавлении нового элемента сравнивать значение этого элемента с первым элементом стека, но! Что делать, если мы будем удалять элементы сверху стека? Ведь не получится откатить значение максимума назад. Значит, ищем другой вариант.

Все еще храним максимум в первом элементе стека, но уже как вспомогательную вещь, с которой будем сравнивать поступающие элементы. Сначала кладем туда -1, ведь все значения, добавляемые в стек, будут положительны. Рядом с каждым элементом храним current максимум в формате [element, current\_maximum]. Current maximum вычисляем сравнивая первый вспомогательный элемент стека с новым элементом. Максимальное из двух значений записываем в первый элемент и рядом с элементом.

Если вдруг удаляем элемент с вершины стека, то и current\_maximum в первом элементе изменяем на current\_maximum рядом с последним элементом.

При вызове функции stack\_max просто выводим stack[-1][-1]. Вот и все :)

def add(number):  
 global stack  
 current\_maximum = max(number, stack[0][0])  
 stack[0][0] = current\_maximum  
 stack.append([number, current\_maximum])  
  
def delete():  
 global stack  
 stack.pop()  
 stack[0][0] = stack[-1][-1]

def stack\_max():  
 global stack, ans  
 ans.append(stack[-1][-1])  
  
stack = [[-1]]  
ans = []  
  
with open("input.txt", "r") as f:  
 iterations = int(f.readline())  
 for i in range(iterations):  
 line = list(f.readline().splitlines())  
 line = list(line[0].split(' '))  
 if len(line) > 1:  
 number = int(line[-1])  
 add(number)  
 else:  
 action = line[0]  
 if action == 'pop':  
 delete()  
 else:  
 stack\_max()  
  
  
with open("output.txt", "w") as d:  
 for elem in ans:  
 d.write(str(elem) + '\n')

**Задача 8.**

Принцип решения, опять-таки, крайне прост. Считываем строку с самого начала, через пробелы. Добавляем элементы в массив. Все элементы, являющиеся числами, преобразуем в int тип. Дальше добавляем по одному элементы в стек, пока не натыкаемся на какую-либо арифметическую операцию. Тогда снимаем со стека предыдущие два элемента и применяем к ним данную операцию, после чего помещаем результат на самый верх стека. Продолжаем идти вперед по выражению, пока не дойдем до конца и выводим результат.

def counter(expression, elements, stack):  
 for i in range(elements):  
 if expression[i] != '-' and expression[i] != '+' and expression[i] != '\*':  
 stack.append(expression[i])  
 else:  
 operation = expression[i]  
 b = stack.pop()  
 a = stack.pop()  
 if operation == '-':  
 stack.append(a - b)  
 elif operation == '+':  
 stack.append(a + b)  
 else:  
 stack.append(a \* b)  
  
  
stack = []  
  
with open("input.txt", "r") as f:  
 elements = int(f.readline())  
 expression = list(f.readline().split(' '))  
 for i in range(elements):  
 if expression[i] != '-' and expression[i] != '+' and expression[i] != '\*':  
 expression[i] = int(expression[i])  
 counter(expression, elements, stack)  
  
  
with open("output.txt", "w") as d:  
 d.write(str(stack[0]))

**Задача 9.**

Принцип задачи абсолютно аналогичен задаче №2, добавляется лишь одна функция наглого пациента. Мы просто математически вычисляем, куда вставлять элемент, и, собственно, insertим его на место с найденным индексом.

Время требовалось вычислить время выполнения алгоритма. Так как он, по сути, основан на координатах tail и nose, и циклы не используются, можно утверждать, что две операции, то есть добавление элемента в конец и удаление элемента с начала, выполняются каждая за O(1). Вставление в середину, однако, выполняется за O(n), так как индексы последующих элементов смещаются при вставлении нового элемента.

def add(my\_queue, number):  
 global tail  
 my\_queue.append(number)  
 tail += 1  
  
def naglo\_add(my\_queue, number):  
 global tail, nose  
 insertion = nose + (tail - 1 - nose) // 2 + (tail - 1 - nose) % 2  
 my\_queue.insert(insertion, number)  
 tail += 1  
  
def delete(my\_queue, ans):  
 global nose  
 ans.append(my\_queue[nose])  
 nose += 1  
  
my\_queue = []  
ans = []  
nose, tail = 0, 1  
  
with open("input.txt", "r") as f:  
 iterations = int(f.readline())  
 for i in range(iterations):  
 line = f.readline().strip()  
 if len(line) > 2:  
 action, number = list(line.split(" "))  
 if action == '+':  
 add(my\_queue, number)  
 else:  
 naglo\_add(my\_queue, number)  
 else:  
 action = line  
 delete(my\_queue, ans)  
  
  
with open("output.txt", "w") as d:  
 for elem in ans:  
 d.write(str(elem) + '\n')

**Задача 13.1.**

Сначала мною была написана задача 13.3, так что для реализации этой задачи мне всего лишь пришлось скопировать потребовалось скопировать часть кода оттуда и дописать крошечную функцию на проверку пустоты односвязного списка is\_empty(self), которая возвращает True, если список пуст и False, если нет.

class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
  
class LinkedList:  
 def \_\_init\_\_(self):  
 self.head = None  
  
 def is\_empty(self):  
 return self.head is None  
  
 def push(self, element):  
 if self.head == None:  
 self.head = Node(element)  
 return  
  
 new\_node = Node(element)  
 new\_node.next = self.head  
 self.head = new\_node  
  
 def pop(self):  
 if self.head is None:  
 return  
  
 self.head = self.head.next  
  
 def to\_array(self):  
 array = []  
 local\_head = self.head  
 while local\_head is not None:  
 array.append(local\_head.data)  
 local\_head = local\_head.next  
 return array

**Задача 13.3.**

Класс Node превращает новый элемент в вершинку, у которой есть значение и возможность иметь ссылку на какое-либо значение.

Класс LinkedList имеет множество функций, таких как push(), pop(), to\_array(), find(), insert\_at(), delete\_at().

Очень сложно было написать классы и понять, как они работают, потому что никогда раньше с ними не сталкивалась. Мозг немного поплавился, но у меня получилось.

class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
  
class LinkedList:  
 def \_\_init\_\_(self):  
 self.head = None  
  
 def push(self, element):  
 if self.head == None:  
 self.head = Node(element)  
 return  
  
 new\_node = Node(element)  
 new\_node.next = self.head  
 self.head = new\_node  
  
 def pop(self):  
 if self.head is None:  
 return  
  
 self.head = self.head.next  
  
 def to\_array(self):  
 array = []  
 local\_head = self.head  
 while local\_head is not None:  
 array.append(local\_head.data)  
 local\_head = local\_head.next  
 return array  
  
 def find(self, query):  
 local\_head = self.head  
 while local\_head is not None:  
 if local\_head.data == query:  
 return query  
 local\_head = local\_head.next  
  
 return None  
  
 def insert\_at(self, at, element):  
 local\_head = self.head  
 new = Node(element)  
  
 while local\_head is not None:  
 if local\_head.data == at:  
 new.next = local\_head.next  
 local\_head.next = new  
 return  
  
 local\_head = local\_head.next  
  
 def delete\_at(self, at):  
 local\_head = self.head  
  
 while local\_head.next is not None:  
 if local\_head.data == at:  
 next\_new = local\_head.next  
 local\_head.next = next\_new.next  
 return  
  
 local\_head = local\_head.next  
 return

**Описание проведенных тестов.**

**2**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Время выполнения** |
| 4 + 1 + 10 - - | 1 10 | 0.0006575000000000053 секунд |
| 12 + 1 + 10 - - + 23 + 12 - - + 14 + 16 + 1234 - | 1 10 23 12 14 | 0.0006630999999999998 секунд |
| 6 + 1 + 10 - + 2 + 1234 - | 1 10 | 0.0007013999999999909 секунд |

**4**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Время выполнения** |
| [] | Success | 0.0006407000000000079 секунд |
| {}[] | Success | 0.0006223999999999952 секунд |
| [()] | Success | 0.0009725000000000011 секунд |
| (()) | Success | 0.0008177999999999935 секунд |
| { | 1 | 0.0007802000000000087 секунд |
| {[} | 3 | 0.0008825999999999973 секунд |
| foo(bar) | Success | 0.0007248999999999867 секунд |
| foo(bar[i) | 10 | 0.0006049000000000054 секунд |
| ({[]()) | 7 | 0.0008305000000000118 секунд |
| ()[{}]} | 7 | 0.0007656999999999942 секунд |
| (()((){} | 1 | 0.0007948999999999873 секунд |
| ()[} | 4 | 0.000737599999999991 секунд |
| ([] | 1 | 0.0006126000000000048 секунд |

**5**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Время выполнения** |
| 5 push 2 push 1 max pop max | 2  2 | 0.0007206000000000018 секунд |
| 5 push 1 push 2 max pop max | 2  1 | 0.0006576999999999972 секунд |
| 6 push 1 push 7 max pop push 6 max | 7 6 | 0.0006682000000000077 секунд |

**8**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Время выполнения** |
| 7  8 9 + 1 7 - \* | -102 | 0.0009066999999999964 секунд |
| 3  3 4 + | 7 | 0.0009267999999999915 секунд |
| 5  10 15 - 3 \* | -15 | 0.0013473999999999986 секунд |
| 5  3 10 15 - \* | -15 | 0.0006203999999999932 секунд |

**9**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Время выполнения** |
| 7 + 1 + 2 - + 3 + 4 - - | 1 2 3 | 0.000880800000000001 секунд |
| 10  + 1  + 2  \* 3  -  + 4  \* 5  -  - | 1 3 2 5 4 | 0.0007447000000000009 секунд |
| 10  + 1  + 2  \* 3  -  \* 4  + 5  -  - | 1 3 4 2 5 | 0.0007409000000000027 секунд |

**13.1**

|  |  |
| --- | --- |
| **Input** | **Output** |
| a = LinkedList()  print(a.is\_empty())  a.push(1)  print(a.is\_empty())  a.push(2)  a.push(3)  a.push(4)  print(a.to\_array())  print(a.is\_empty())  a.pop()  a.pop()  a.pop()  a.pop()  print(a.is\_empty()) | True  False  [4, 3, 2, 1]  False  True |

**13.3**

|  |  |
| --- | --- |
| **Input** | **Output** |
| a = LinkedList()  a.push(1)  a.push(2)  a.push(3)  a.push(4)  print(a.to\_array())  a.pop()  print(a.to\_array())  print(a.find(3))  print(a.find(4))  a.insert\_at(2, 5)  print(a.to\_array())  a.insert\_at(5, 8)  print(a.to\_array())  a.delete\_at(None)  print(a.to\_array())  a.delete\_at(5)  print(a.to\_array()) | [4, 3, 2, 1]  [3, 2, 1]  3  None  [3, 2, 5, 1]  [3, 2, 5, 8, 1]  [3, 2, 5, 8, 1]  [3, 2, 5, 1] |

**Выводы по проделанной работе.**

О Господи, о Боже мой! Я добралась-таки до вывода по проделанной работе. Стеки, очереди и связные списки очень интересны по своей сути, особенно самостоятельно прописывать новые классы.

Очень расстроило распределение задач по вариантам. Хотелось бы, чтобы сложные задачки были у всех. Просто правда, когда на одинаковых условиях кому-то досталась 3-я задача, а кому-то, как мне, 4, это очень печально. Я решаю всю ночь и обогащаюсь, а невезунчики проспали самое интересное :(

А в основном, интересно было понять, как работают такие типы хранения информации и как их можно применять. На этом все, можно вроде-как защищаться 😊