САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ

ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ

ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе № 4 по курсу «Алгоритмы и структуры данных»

Тема: Стек, Очередь, Связанный список

Вариант 23

Выполнил:

Хабиби Ясер

Группа К3140

Проверил: Афанасьев А. В.

Санкт-Петербург 2024 г.

# Введение

Лабораторная работа посвящена разбору элементарных структур данных, та- ких как стек, очередь и связанный список

Задачи по варианту: 1, 4, 5 и 11, также были взяты и решены задания 8,13

Задача 1: Стек

# В данной задаче требовалось реализовать работу стека, поддерживающего две команды: "+ N" — добавление числа в стек, и "-" — изъятие элемента из стека. Для каждой операции извлечения элемента необходимо выводить результат. Команды гарантируют, что стек не пуст при извлечении элемента, а его размер не превысит 10^6 элементов в процессе выполнения. Решение задачи предполагает использование стандартной структуры данных "стек", где добавление и извлечение элементов выполняются за константное время.

# Код программы

class Stack:  
 def \_\_init\_\_(self):  
 self.stack = []  
  
 def push(self, value):  
 self.stack.append(value)  
  
 def pop(self):  
 return self.stack.pop() if self.stack else None  
  
  
def process\_stack\_commands(commands):  
 stack = Stack() # Инициализация стека  
 results = []  
  
 for command in commands:  
 if command.startswith('+'):  
 \_, value = command.split()  
 stack.push(int(value))  
 elif command == '-':  
 results.append(stack.pop())  
  
 return results

# Тесты и анализ

Алгоритм и Стек был протестирован на различных тестах указанных в условии

def test\_stack\_operations(self):  
 # given  
 commands = [  
 "+ 10",  
 "+ 20",  
 "+ 30",  
 "-",  
 "-",  
 "-",  
 ]  
 expected = [30, 20, 10]  
  
 # when  
 result, elapsed\_time, peak\_memory = measure\_performance(process\_stack\_commands, commands)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory, 256, "Memory usage exceeded 256 MB")  
  
def test\_empty\_stack(self):  
 # given  
 commands = ["-"]  
 expected = [None]  
  
 # when  
 result, elapsed\_time, peak\_memory = measure\_performance(process\_stack\_commands, commands)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory, 256, "Memory usage exceeded 256 MB")  
  
def test\_mixed\_operations(self):  
 # given  
 commands = [  
 "+ 5",  
 "+ 15",  
 "-",  
 "+ 25",  
 "-",  
 "-",  
 ]  
 expected = [15, 25, 5]  
  
 # when  
 result, elapsed\_time, peak\_memory = measure\_performance(process\_stack\_commands, commands)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory, 256, "Memory usage exceeded 256 MB")

Задача 4: Скобочная Последовательность. Версия 2

В данном задании была реализована программа для проверки правильности скобочной последовательности. Программа проходила по строке, проверяя, что каждая закрывающая скобка имеет соответствующую открывающую скобку, и учитывались все возможные типы скобок: круглые, квадратные и фигурные. Реализация продемонстрировала основы алгоритмической проверки на корректность последовательности символов.

Код программы

def check\_brackets(sequence):  
 stack = []  
 bracket\_pairs = {'(': ')', '[': ']', '{': '}'}  
  
 for i, char in enumerate(sequence):  
 if char in bracket\_pairs: # Открывающая скобка  
 stack.append((char, i + 1))  
 elif char in bracket\_pairs.values(): # Закрывающая скобка  
 if not stack:  
 return i + 1  
 top, pos = stack.pop()  
 if bracket\_pairs[top] != char:  
 return i + 1  
  
 if stack:  
 \_, pos = stack.pop()  
 return pos  
  
 return "Success"  
  
  
def process\_brackets():  
 data = file\_read()  
 result = check\_brackets(data[0])  
 file\_write([result])

# Тесты и анализ

Алгоритм был испытан на различных сетах данных, и на всех он показал себя отлично.

def test\_success\_cases(self):  
 # given  
 cases = [  
 ("[]", "Success"),  
 ("{}", "Success"),  
 ("()", "Success"),  
 ("{[()]}", "Success"),  
 ("foo(bar);", "Success"),  
 ("", "Success")  
 ]  
  
 for sequence, expected in cases:  
 # when  
 result = check\_brackets(sequence)  
  
 # then  
 self.assertEqual(result, expected)  
  
def test\_error\_cases(self):  
 # given  
 cases = [  
 ("{", 1),  
 ("}", 1),  
 ("{[}", 3),  
 ("foo(bar[i);", 10),  
 ("(}", 2)  
 ]  
  
 for sequence, expected in cases:  
 # when  
 result = check\_brackets(sequence)  
  
 # then  
 self.assertEqual(result, expected)

Задача 5: Стек с Максимумом

В данном задании требовалось реализовать стек, который поддерживает операции Push(), Pop() и Max() таким образом, чтобы каждая операция выполнялась за константное время. Для решения задачи был использован дополнительный стек, который хранит текущие максимальные значения на каждом шаге.

# Код программы

def \_\_init\_\_(self):  
 self.stack = [] # Основной стек  
 self.max\_stack = [] # Стек для хранения максимумов  
  
 def push(self, value):  
 self.stack.append(value)  
 # Добавляем новый максимум в max\_stack  
 if not self.max\_stack or value >= self.max\_stack[-1]:  
 self.max\_stack.append(value)  
  
 def pop(self):  
 if not self.stack:  
 return None  
 value = self.stack.pop()  
 # Убираем из max\_stack, если удаляемое значение было максимумом  
 if value == self.max\_stack[-1]:  
 self.max\_stack.pop()  
 return value  
  
 def max(self):  
 # Возвращаем текущий максимум  
 return self.max\_stack[-1] if self.max\_stack else None  
  
  
def process\_max\_stack():  
 commands = file\_read() # Читаем команды из input.txt  
 max\_stack = MaxStack()  
 results = []  
  
 for command in commands:  
 if command.startswith("push"):  
 \_, value = command.split()  
 max\_stack.push(int(value))  
 elif command == "pop":  
 max\_stack.pop()  
 elif command == "max":  
 results.append(max\_stack.max())

# Тесты и анализ

Алгоритм был протестирован на различных сетах данных, Во всех случаях алгоритм отработал быстро и безошибочно, что доказывает, что тот был написан верно.

def test\_operations(self):  
 # given  
 stack = MaxStack()  
  
 # when  
 stack.push(2)  
 stack.push(1)  
 max1 = stack.max()  
 stack.pop()  
 max2 = stack.max()  
 stack.push(7)  
 max3 = stack.max()  
  
 # then  
 self.assertEqual(max1, 2)  
 self.assertEqual(max2, 2)  
 self.assertEqual(max3, 7)  
  
def test\_empty\_stack(self):  
 # given  
 stack = MaxStack()  
  
 # when  
 max\_value = stack.max()  
  
 # then  
 self.assertIsNone(max\_value)

Задача 11: Бюрократия

В данной задаче необходимо было смоделировать процесс получения справок в министерстве бюрократии с учетом очереди посетителей, где каждый посетитель хочет получить несколько справок. За один прием выдается только одна справка, и если у посетителя остаются невыданные справки, он встает в конец очереди. Задача заключалась в том, чтобы посчитать, сколько еще справок нужно каждому оставшемуся в очереди посетителю, когда прием завершится, или вывести -1, если все посетители уйдут до конца дня.

# Код программы

def process\_queue(n, m, demands):  
 queue = list(demands) # Очередь только с требованиями справок  
  
 while m > 0 and queue:  
 current = queue.pop(0) # Берем первого в очереди  
 if current > 1:  
 queue.append(current - 1)  
 m -= 1  
  
 # Если очередь пуста, возвращаем -1  
 if not queue:  
 return ["-1"]  
 else:  
 return [str(len(queue))] + [" ".join(map(str, queue))]  
  
  
def process\_bureaucracy():  
 data = file\_read()  
 n, m = map(int, data[0].split())  
 demands = list(map(int, data[1].split()))  
 result = process\_queue(n, m, demands)  
 file\_write(result)

# Тесты и анализ

Для испытания данной программы были загружены различные сеты данных. Во всех случаях программа отработала точно и быстро

def test\_example\_cases(self):  
 # given  
 n1, m1, demands1 = 3, 2, [1, 2, 3]  
 n2, m2, demands2 = 4, 5, [2, 5, 2, 3]  
  
 # when  
 result1 = process\_queue(n1, m1, demands1)  
 result2 = process\_queue(n2, m2, demands2)  
  
 # then  
 self.assertEqual(result1, ["2", "3 1"]) # Пример 1  
 self.assertEqual(result2, ["3", "4 1 2"]) # Пример 2  
  
def test\_no\_people\_left(self):  
 # given  
 n, m, demands = 2, 10, [1, 1]  
  
 # when  
 result = process\_queue(n, m, demands)  
  
 # then  
 self.assertEqual(result, ["-1"]) # Все справки выданы, очередь пуста.  
  
def test\_partial\_completion(self):  
 # given  
 n, m, demands = 4, 3, [1, 1, 1, 1]  
  
 # when  
 result = process\_queue(n, m, demands)  
  
 # then  
 self.assertEqual(result, ["1", "1"]) # Остаётся только 1 посетитель.  
  
def test\_large\_demand(self):  
 # given  
 n, m, demands = 5, 5, [5, 5, 5, 5, 5]  
  
 # when  
 result = process\_queue(n, m, demands)  
  
 # then  
 self.assertEqual(result, ["5", "4 4 4 4 4"]) # Все остаются, но с уменьшенным требованием.  
  
def test\_edge\_case\_no\_demands(self):  
 # given  
 n, m, demands = 3, 10, [0, 0, 0]  
  
 # when  
 result = process\_queue(n, m, demands)  
  
 # then  
 self.assertEqual(result, ["-1"]) # Никто ничего не требует, очередь пуста.  
  
def test\_one\_person(self):  
 # given  
 n, m, demands = 1, 1, [10]  
  
 # when  
 result = process\_queue(n, m, demands)  
  
 # then  
 self.assertEqual(result, ["1", "9"]) # Один человек остаётся с уменьшенным требованием.

Задача 8 : постфиксная запись

задача посвящена вычислению арифметических выражений, заданных в постфиксной записи (также известной как польская обратная запись). В постфиксной нотации операции следуют за их операндами, что исключает необходимость в скобках для определения порядка операций. Основное преимущество этого подхода заключается в упрощении вычислений с использованием стека.

# Код программы

def evaluate\_postfix(expression):  
 stack = []  
 operators = {'+', '-', '\*'}  
  
 for token in expression:  
 if token.isdigit() or (token[0] == '-' and token[1:].isdigit()): # Если это число  
 stack.append(int(token))  
 elif token in operators: # Если это оператор  
 b = stack.pop()  
 a = stack.pop()  
 if token == '+':  
 result = a + b  
 elif token == '-':  
 result = a - b  
 elif token == '\*':  
 result = a \* b  
 stack.append(result)  
  
  
 return stack.pop()  
  
  
def process\_postfix():  
 data = file\_read()  
 n = int(data[0])  
 expression = data[1].split()  
  
 result = evaluate\_postfix(expression)  
 file\_write([result])

# Тесты и анализ

Для испытания данной программы были загружены различные сеты данных. Во всех случаях программа отработала точно и быстро

def test\_simple\_expression(self):  
 # given  
 expression = ["8", "9", "+", "1", "7", "-", "\*"]  
  
 # when  
 result = evaluate\_postfix(expression)  
  
 # then  
 self.assertEqual(result, -102)  
  
def test\_single\_number(self):  
 # given  
 expression = ["5"]  
  
 # when  
 result = evaluate\_postfix(expression)  
  
 # then  
 self.assertEqual(result, 5)  
  
def test\_multiple\_operations(self):  
 # given  
 expression = ["2", "3", "\*", "5", "+"]  
  
 # when  
 result = evaluate\_postfix(expression)  
  
 # then  
 self.assertEqual(result, 11)  
  
def test\_large\_expression(self):  
 # given  
 expression = ["10", "5", "\*", "2", "3", "+", "-"]  
  
 # when  
 result = evaluate\_postfix(expression)  
  
 # then  
 self.assertEqual(result, 45)

Задача 13 Реализация Стека Очереди Связных списков :

Описание

Эта задача фокусируется на реализации и использовании связанных структур данных, таких как однонаправленные и двунаправленные списки. Основной целью является понимание их работы, практическая реализация и выполнение операций, таких как добавление, удаление и поиск элементов.

# Код программы

class Node:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.next = None  
  
  
class LinkedQueue:  
 def \_\_init\_\_(self):  
 self.front = None  
 self.rear = None  
  
 def isEmpty(self):  
 return self.front is None  
  
 def enqueue(self, value):  
 new\_node = Node(value)  
 if self.isEmpty():  
 self.front = self.rear = new\_node  
 else:  
 self.rear.next = new\_node  
 self.rear = new\_node  
  
 def dequeue(self):  
 if self.isEmpty():  
 raise IndexError("Queue is empty")  
 value = self.front.value  
 self.front = self.front.next  
 if self.front is None:  
 self.rear = None  
 return value  
  
 def \_\_str\_\_(self):  
 elements = []  
 current = self.front  
 while current:  
 elements.append(current.value)  
 current = current.next  
 return " <- ".join(map(str, elements))  
  
  
def process\_queue():  
 commands = read\_queue\_input()  
 queue = LinkedQueue()  
 results = []  
  
 for command in commands:  
 if command.startswith("enqueue"):  
 \_, value = command.split()  
 queue.enqueue(int(value))  
 elif command == "dequeue":  
 try:  
 results.append(queue.dequeue())  
 except IndexError:  
 results.append("error")  
 elif command == "isEmpty":  
 results.append("true" if queue.isEmpty() else "false")  
  
 write\_output(results)

# и также

class Node:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.next = None  
  
  
class LinkedStack:  
 def \_\_init\_\_(self):  
 self.top = None  
  
 def isEmpty(self):  
 return self.top is None  
  
 def push(self, value):  
 new\_node = Node(value)  
 new\_node.next = self.top  
 self.top = new\_node  
  
 def pop(self):  
 if self.isEmpty():  
 raise IndexError("Stack is empty")  
 value = self.top.value  
 self.top = self.top.next  
 return value  
  
 def \_\_str\_\_(self):  
 elements = []  
 current = self.top  
 while current:  
 elements.append(current.value)  
 current = current.next  
 return " -> ".join(map(str, elements))  
  
  
def process\_stack():  
 commands = read\_stack\_input()  
 stack = LinkedStack()  
 results = []  
  
 for command in commands:  
 if command.startswith("push"):  
 \_, value = command.split()  
 stack.push(int(value))  
 elif command == "pop":  
 try:  
 results.append(stack.pop())  
 except IndexError:  
 results.append("error")  
 elif command == "isEmpty":  
 results.append("true" if stack.isEmpty() else "false")  
  
 write\_output(results)

# 

# Тесты и анализ

Для испытания данной программы были загружены различные сеты данных. Во всех случаях программа отработала точно и быстро

def test\_queue\_operations(self):  
 # given  
 queue = LinkedQueue()  
  
 # when  
 queue.enqueue(1)  
 queue.enqueue(2)  
 queue.enqueue(3)  
  
 # then  
 self.assertEqual(str(queue), "1 <- 2 <- 3")  
 self.assertEqual(queue.dequeue(), 1)  
 self.assertEqual(queue.dequeue(), 2)  
 self.assertEqual(queue.dequeue(), 3)  
 self.assertTrue(queue.isEmpty())  
  
def test\_dequeue\_empty\_queue(self):  
 # given  
 queue = LinkedQueue()  
  
 # when / then  
 with self.assertRaises(IndexError):  
 queue.dequeue()

и также

def test\_stack\_operations(self):  
 # given  
 stack = LinkedStack()  
  
 # when  
 stack.push(1)  
 stack.push(2)  
 stack.push(3)  
  
 # then  
 self.assertEqual(str(stack), "3 -> 2 -> 1")  
 self.assertEqual(stack.pop(), 3)  
 self.assertEqual(stack.pop(), 2)  
 self.assertEqual(stack.pop(), 1)  
 self.assertTrue(stack.isEmpty())  
  
def test\_pop\_empty\_stack(self):  
 # given  
 stack = LinkedStack()  
  
 # when / then  
 with self.assertRaises(IndexError):  
 stack.pop()

# Заключение

В процессе выполнения лабораторной работы, посвящённой быстрой сортировке и её различным вариантам, были разработаны и протестированы несколько алгоритмов сортировки и алгоритмов сортировки за линейное время. Каждый из методов, включая различные способы использования алгоритма быстрой сортировки и алгоритмов сортировки за линейное время, был подробно изучен, реализован на Python и проверен с помощью модуля unittest. Были проанализированы принципы их работы, выявлены преимущества и недостатки, а также выполнена оценка их временной сложности.

Выполнение этой лабораторной работы позволило углубить знания об основных алгоритмах сортировки и алгоритмов сортировки за линейное время, улучшить навыки программирования на Python, а также развить умение проводить модульное тестирование программ.