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.

Выполнил:
Мезенцев Богдан
К 3139
Проверил:
Афанасьев Антон Владимирович

Санкт-Петербург
2024 г.

Содержание отчёта

Задачи по варианту

Задание 1. Стек

Реализация функции stack_commands():

```python
def stack_commands(M, commands):
    """Возвращает элементы, которые были удалены из стека."""
    stack = []  # стек
    deleted_elements = []  # удалённые элементы
    for command in commands:
        if command[0] == '+':
            _, number = command.split('+')
            stack.append(int(number))  # добавление элемента в стек
        elif command == '-':
            deleted_elements.append(stack.pop())
    return deleted_elements
```

Данная функция создает список stack и добавляет в него элемент, если получает команду “+” и удаляет элемент, если получает команду “-” . После удаления элемента функция возвращает удаленный элемент.

Задание 2. Очередь

Реализация функции queue_commands():

```python
def queue_commands(M, commands):
    """Возвращает элементы текущей очереди после каждой команды удаления элемента."""
    queue = deque()
    current_queue = []

    for command in commands:
        if command[0] == '+':
            _, number = command.split('+')
            queue.append(int(number))  # Добавляем число в очередь
        elif command == '-':
            if queue:
                current_queue.append(queue.popleft())  # Извлекаем элемент из начала очереди
            else:
                current_queue.append("Queue is empty")  # На случай, если очередь пуста
    return current_queue
```

Данная функция создает список объект(саму очередь) и список для хранения элементов в текущей очереди.
Функция добавляет элемент в очередь, если получает команду “+” и удаляет элемент, если получает команду “-” .
После удаления элемента функция возвращает удаленный элемент.

Задание 3. Скобочная последовательность. Версия 1

Реализация функции check_correct_pairs(), которая проверяет на правильность полученную последовательность скобок.

```
def check_correct_pairs(sequence):
    """Проверяет на правильность последовательность скобок."""
    stack = []
    pairs = {')': '(', ']': '['}  # Правильное соответствие закрывающих и открывающих скобок
    for simbol in sequence:
        if simbol in "([":  # Если проверяем открывающюю скобку
            stack.append(simbol)
        elif simbol in ")]":  # Если проверяем закрывающюю скобку
            if not stack or stack[-1] != pairs[simbol]:  # Если стек пуст или есть несоответствие
                return "NO"
            stack.pop()  # Удаляем соответствующую открывающую скобку
    return "YES" if not stack else "NO"  # Если стек пуст, последовательность правильная


def main(sequences):
    """Проверяет правильность каждой последовательности."""
    return [check_correct_pairs(sequence) for sequence in sequences]
```

Данная функция проверяет соответствие скобок с помощью словаря, то есть каждой открытой скобке соответствует закрытая скобка.
Если все скобки прошли проверку, то функция удаляет их из списка и выводит “YES”, иначе “NO”.
Функция main() вызывает функцию check_correct_pairs() для каждой полученной последовательности.

Задание 6. Очередь с минимумом

Реализация функции queue_comands_min():

```python
def queue_commands_min(M, commands):
    """Возвращает элементы текущей очереди и мимнимальный элемент"""
    queue = deque()  # основаная очередь
    current_queue = []  # очередь для хранения текущего результата
    min_queue = []  # очередь для хранения минимального элемента

    for command in commands:
        if command[0] == '+':
            _, number = command.split("+")
            number = int(number)
            queue.append(number)  # Добавляем число в очередь

            while min_queue and min_queue[-1] > number:  # Обновляем очередь минимальных элементов
                min_queue.pop()
            min_queue.append(number)

        elif command == '-':
            if queue:
                removed = queue.popleft()  # Удаляем элемент из начала очереди
                if removed == min_queue[0]:
                    min_queue.pop(0)

        elif command == "?":
            if min_queue:  # Запрос минимального элемент
                current_queue.append(min_queue[0])
            else:
                current_queue.append("Queue is empty")  # Случай, если очередь пуста
    return current_queue
```

Данная функция делает почти все то же самое, что и функция из Задания 2. см. страницу 2, только тут мы добавляем очередь для хранения текущего минимального элемента. Также добавляется, если функция получает команду “?”, то она возвращает текущую очередь и текущий минимальный элемент.

Задание 8. Постфиксная запись

Реализация функции Postfix_calculate():

```python
def Postfix_calculate(expression):
    """Вычисляет значение выражения в постфиксной записи."""
    stack = []
    actions = ['+', '-', '*', '/']

    for simbol in expression.split():
        if simbol not in actions:
            stack.append(int(simbol))  # Если символ является числом, то добавляем его в стек
        else:
            b = stack.pop()
            a = stack.pop()

            # Выполняем соответственную операцию
            if simbol == '+':
                stack.append(a + b)
            elif simbol == '-':
                stack.append(a - b)
            elif simbol == '*':
                stack.append(a * b)
            elif simbol == '/':
                stack.append(int(a / b))

    return stack[0]  # Возвращаем значение выражения
```

Данная функция разделяет числа и символы математических действий.
Далее она производит соответствующую математическую операцию и добавляет результат в стек.
По итогу функция возвращает результат со значением выражения.

Задание 13. Реализация стека, очереди и связанных списков.

1) **Реализация стека на основе связанного списка с функциями isEmpty, push, pop и вывода данных:**

```
class Node:
    """Класс, описывающий новый узел в связанном списке"""
    def __init__(self, value):
        """Инициализация данных узла"""
        self.value = value  # Значение, которое содержит узел
        self.next = None  # Указатель на следующий узел. Если узел последний, next = None
```

Данный класс представляет создание нового узла связанного списка.

```
class Stack:
    """Класс реализует стек на основе связанного списка и описывает методы для работы с ним"""
    def __init__(self):
        """Инициализация верхнего элемента стека"""
        self.top = None  # Указатель на верхний элемент стека. Если стек пуст, top = None
```

Данный класс представляет следующие методы для работы со стеком:

```
def isEmpty(self):
    """Проверяет, стек пустой или нет"""
    return self.top is None

def push(self, value):
    """Добавляет новый элемент в стек"""
    new_node = Node(value)  # Создание нового узела(элемента стека)
    new_node.next = self.top  # Для нового узла передаём текущий верхний элемент стека
    self.top = new_node  # Новый узел становится верхним элементом стека

def pop(self):
    """Удаляет верхний элемент стека и возвращает его данные"""
    deleted_value = self.top.value  # Сохраняем значение верхнего узла(удаляемого элемента)
    self.top = self.top.next  # Меняем указатель верхеного элемента на следующий узел
    return deleted_value  # Возвращаем значение удалённого узла

def display_stack(self):
    """Выводит элементы всего стека"""
    current_top = self.top  # Первый элемент стека
    elements = []  # Список для хранения элементов
    while current_top:  # Пока не закончатся все узлы, будем добавлять их в список
        elements.append(current_top.value)
        current_top = current_top.next  # Меняем указатель верхеного элемента на следующий узе
    return ", ".join(map(str, elements))
```

isEmpty - Проверяет верхний элемент стека и проверяет стека на пустоту
push - Добавляет новый элемент в стек. Сначала создается новый узел. Затем переопределяется верхний элемент стека.

pop - Удаляет верхний элемент стека и возвращает его значение. Сначала сохраняется значение удаляемого элемента в отдельную переменную. Затем переопределяется верхний элемент стека.
display_stack - Выводит содержимое стека. Данный метод поочередно обращается в значениям элементов стека и добавляет их в список.

2) **Реализация очереди на основе связанного списка с функциями Enqueue, Dequeue с проверкой на переполнение и опустошения очереди:**

```
class Node:
    """Класс, описывающий новый узел в связанном списке"""
    def __init__(self, value):
        self.value = value  # Данные, которые содержит узел
        self.next = None  # Указатель на следующий узел. Если узел последний, next = None
```

Данный класс представляет создание нового узла связанного списка.

```
class Queue:
    """Класс реализует очередь на основе связанного списка и описывает методы для работы с ней"""
    def __init__(self, max_size):
        self.first = None  # Указатель на начало очереди
        self.last = None  # Указатель на конец очереди
        self.size = 0  # Текущий размер очереди
        self.max_size = max_size  # Максимальная вместимость очереди
```

Данный класс представляет следующие методы для работы с очередью:

```
def isEmpty(self):
    """Проверяет, очередь пуста или нет"""
    return self.size == 0
```

isEmpty - проверяет очередь на пустоту, проверяя ее размер.

```
def isFull(self):
    """Проверка на переполнение"""
    return self.size == self.max_size
```

isFull - проверяет очередь на переполнение, сравнивая текущий размер очереди с максимальным размером.

```
def enQueue(self, value):
    """Добавляет элемент в очередь"""
    if self.isFull():
        return print("Очередь переполнена!")

    new_node = Node(value)  # Создаём новый узел
    if self.last is None: # Если очередь пуста
        self.first = self.last = new_node
    else:
        self.last.next = new_node  # Добавляем новый узел в конец очереди
        self.last = new_node  # Меняем указатель для последнего элемента
    self.size += 1
```

enQueue - добавляет элемент в очередь. Сначала создается новый узел. Затем переопределяется последний элемент очереди. После добавления размер очереди увеличивается на 1.

```
def deQueue(self):
    """Удаляет первый элемент очереди"""
    self.first = self.first.next  # Меняем указатель для первого элемента

    if self.first is None:  # Если очередь стала пустой
        self.last = None

    self.size -= 1
```

deQueue - удаляет элемент из очереди. Сначала переопределяется первый элемент очереди. Затем размер очереди уменьшается на 1.

```
def display_queue(self):
    """Выводит элементы всей очереди"""
    elements = []  # Список для хранения элементов
    current_first = self.first  # Первый элемент очереди

    while current_first:  # Пока есть узлы
        elements.append(current_first.value)
        current_first = current_first.next

    return ", ".join(map(str, elements))
```

display_queue - Выводит содержимое очереди, поочередно обращаясь к значениям элементов.