МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Информационные технологии»

ТЕМА: АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ В PYTHON

Студент гр. 3341 Байрам Э.

Преподаватель Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Изучите основные алгоритмы и структуры данных в Python, освойте основные методы работы со списками в этом языке, затем создайте программу и реализуйте в ней односвязный список для хранения числовых данных.

Задание

В этой лабораторной работе вам предстоит реализовать односвязный список. Для этого необходимо создать два зависимых класса:

Node

Этот класс описывает элемент списка.

У него должно быть 2 поля:

- data # Данные элемента списка, приватное поле.
- next # Ссылка на следующий элемент списка.

И следующие методы:

- __init__(self, data, next) - конструктор, у которого значения по умолчанию для аргумента next равно None.
- get_data(self) - метод возвращает значение поля data (это необходимо, потому что в идеале пользователь класса не должен трогать поля класса Node).
- change_data(self, new_data) - метод меняет значение поля data объекта Node.
- __str__(self) - перегрузка стандартного метода __str__, который преобразует объект в строковое представление. Для данной лабораторной необходимо реализовать следующий формат перевода объекта класса Node в строку:

“data: <node_data>, next: <node_next>”,

где <node_data> - это значение поля data объекта Node, <node_next> - это значение поля next объекта, на который мы ссылаемся, если он есть, иначе None.

Пример того, как должен выглядеть результат реализации __str__ см. ниже.

Пример того, как должен выглядеть вывод объекта:

```
node = Node(1)
print(node) # data: 1, next: None
node.next = Node(2, None)
print(node) # data: 1, next: 2
```

Linked List

Этот класс описывает односвязный список.

У него должно быть 2 поля:

- head # Данные первого элемента списка.
- length # Количество элементов в списке.

И следующие методы:

- __init__(self, head) - конструктор, у которого значения по умолчанию для аргумента head равно None.
  - Если значение переменной head равно None, метод должен создавать пустой список.
  - Если значение head не равно None, необходимо создать список из одного элемента.
- __len__(self) - перегрузка метода __len__, он должен возвращать длину списка (этот стандартный метод, например, используется в функции len).
- append(self, element) - добавление элемента в конец списка. Метод должен создать объект класса Node, у которого значение поля data будет равно element и добавить этот объект в конец списка.
- __str__(self) - перегрузка стандартного метода __str__, который преобразует объект в строковое представление. Для данной лабораторной необходимо реализовать следующий формат перевода объекта класса однонаправленного списка в строку:
  - Если список пустой, то строковое представление:

“LinkedList[]”

· Если не пустой, то формат представления следующий:

“LinkedList[length = <len>, [data:<first_node>.data, next: <first_node>.data; data:<second_node>.data, next:<second_node>.data; … ; data:<last_node>.data, next: <last_node>.data]”,

где <len> - длина связного списка, <first_node>, <second_node>, <third_node>, … , <last_node> - элементы однонаправленного списка.

Пример того, как должен выглядеть результат реализации см. ниже.

o pop(self) - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.

o clear(self) - очищение списка.

o change_on_start(self, n, new_data) - изменение поля data n-того элемента с НАЧАЛА списка на new_data. Метод должен выбрасывать исключение KeyError, с сообщением "Element doesn't exist!", если количество элементов меньше n.

Пример того, как должно выглядеть взаимодействие с Вашим связным списком:

```
linked_list = LinkedList()
print(linked_list) # LinkedList[]
print(len(linked_list)) # 0
linked_list.append(10)
print(linked_list) # LinkedList[length = 1, [data: 10, next: None]]
print(len(linked_list)) # 1
linked_list.append(20)
print(linked_list) # LinkedList[length = 2, [data: 10, next:20; data: 20, next: None]]
```

```
print(len(linked_list)) # 2
linked_list.pop()
print(linked_list)
print(linked_list) # LinkedList[length = 1, [data: 10, next: None]]
print(len(linked_list)) # 1
```

Вам не требуется реализовывать создание экземпляров ваших классов и вызов методов, это сделает проверяющая система.

Выполнение работы

Класс Node создается с приватным полем data (хранящим данные узла) и публичным полем next (ссылкой на следующий узел). Метод __init__() инициализирует экземпляр класса Node, где значение next по умолчанию равно None. Метод get_data() возвращает значение приватного поля data, а change_data() заменяет данные узла на новые. Метод __str__() создает строковое представление объекта класса Node с использованием форматирования строк.

Затем создается класс LinkedList, содержащий два поля: head (данные первого узла списка) и length (количество элементов в списке). Метод __init__() инициализирует объект класса LinkedList в зависимости от содержимого head. Метод __len__() возвращает значение поля length. Метод append() добавляет узел с данными element в конец списка. Метод __str__() возвращает представление списка в виде строки. Метод pop() удаляет последний узел из списка, вызывая исключение IndexError, если список пуст. Метод clear() сбрасывает значение head на None и length на 0. Метод change_on_start() заменяет данные n-го узла с начала списка, выбрасывая исключение KeyError, если узла с индексом n нет в списке.

Код программы представлен в приложении А.

Выводы

В процессе выполнения работы были освоены основные алгоритмы и структуры данных в языке Python, а также освоены базовые методы работы со списками. С использованием этих знаний была создана программа, в рамках которой был реализован односвязный список для хранения численных данных. Этот опыт позволил лучше понять принципы работы структур данных и их применение в реальных задачах.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
    class Node:
        def __init__(self, data, next = None):
            self.__data = data
            self.next = next

        def get_data(self):
            return self.__data

        def change_data(self, new_data):
            self.__data = new_data

        def __str__(self):
            if self.next is not None:
                return     f"data:     {self.get_data()},     next:
{self.next.get_data()}"
            else:
                return f"data: {self.get_data()}, next: None"

    class LinkedList:
        def __init__(self, head = None):
            self.head = head
            if head is not None:
                self.length = 1
            else:
                self.length = 0

        def __len__(self):
            return self.length

        def append(self, element):
            new_node = Node(element)
            if self.head is not None:
                current = self.head
                while current.next is not None:
                    current = current.next
                current.next = new_node
            else:
                self.head = new_node
            self.length += 1

        def __str__(self):
            if self.head is None:
                result = f"LinkedList[]"
            else:
                result = f"LinkedList[length = {self.length}, ["
                current = self.head
                while current is not None:
                    if current.next is None:
                        result   +=   f"data:   {current.get_data()},
next: None]]"
```

```
                    else:
                        result += f"data: {current.get_data()},
next: {current.next.get_data()}; "
                    current = current.next
            return result

        def pop(self):
            if self.head is None:
                raise IndexError("LinkedList is empty!")
            elif self.head.next is None:
                self.head = None
            else:
                current = self.head
                while current.next.next is not None:
                    current = current.next
                current.next = None
            self.length -= 1

        def clear(self):
            self.head = None
            self.length = 0

        def change_on_start(self, n, new_data):
            if n > self.length or n <= 0:
                raise KeyError("Element doesn't exist!")
            current = self.head
            for i in range(n-1):
                current = current.next
            current.change_data(new_data)
```