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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Информатика»

Тема: Алгоритмы и структуры данных.

Студент гр. 3344 _____ Клюкин А.В.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Научиться реализовывать односвязный список с использованием классов на языке программирования Python.

Задание.

В данной лабораторной работе Вам предстоит реализовать связный однонаправленный список. Для этого необходимо реализовать 2 зависимых класса:

Node

Класс, который описывает элемент списка.

Он должен иметь 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

Класс, который описывает связный однонаправленный список.

Он должен иметь 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 равна 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.

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

Выполнение работы

1) Связный список – структура, в которой каждый элемент содержит ссылку на следующий или даже предыдущий, если список двунаправленный. Отличие в том, что массив располагает свои данные последовательно в памяти, а список – нет. Так же массив хранит один тип данных, а список не обязательно.

2) get_data – O(1), change_data – O(1), append() – O(n) (“в худшем случае”), pop() – O(n) (“в худшем случае”), change_on_start() – O(n) (“в худшем случае”), clear() – O(1)

3) Берется серединный элемент списка и сравнивается с искомым. Если подходит, то конец, иначе и исходя из сравнения берется соседний справа или слева элемент и снова сравнивается.
   Отличие реализации для классического списка Python в том, что можно использовать индексы для доступа к элементам списка, а в связном списке нет.

Выводы

Получен навык реализации односвязного списка с использованием классов.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Klyukin_Aleksandr_lb2.py

```
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 != None:
            return f'data: {self.data}, next: {self.next.data}'
        else:
            return f'data: {self.data}, next: None'

class LinkedList:
    def __init__(self, head=None):
        self.head = head
        self.length = 0
        if head:
            self.length += 1

    def __len__(self):
        return self.length

    def append(self, element):
        if self.head:
            last = self.head
            while last.next != None:
                last = last.next
            last.next = Node(element)
        else:
            self.head = Node(element)
        self.length += 1

    def __str__(self):
        if self.length == 0:
            return "LinkedList[]"
        else:
            out_data = ''
            node = self.head
            while node != None:
                out_data += node.__str__() + '; '
                node = node.next
               return f"LinkedList[length = {self.length}, [{out_data[:-
2]}]]"
```

```
def pop(self):
    if not self.head:
        raise IndexError("LinkedList is empty!")
    if self.head:
        last = self.head
        if self.head.next:
            while last.next.next != None:
                last = last.next
            last.next = None
            self.length-=1
        else:
            self.head = None
            self.length = 0

def change_on_start(self, n, new_data):
    if self.length < n or n <= 0:
        raise KeyError("Element doesn't exist!")
    node = self.head
    i = 1
    while i < n:
        node = node.next
        i+=1
    node.data = new_data

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