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 в Python

Студент гр. 3342 _____ Гончаров С. А.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург

2024

Цель работы

Создание линейного списка и реализация изменений в нем на языке Python.

Задание

Вариант 4

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).

• **__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.

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.

Выполнение работы

Были созданы классы Node и LinkedList. Реализованы методы согласно условию.

Связный список — это структура данных, используемая для хранения коллекции элементов, где каждый элемент содержит ссылку на следующий элемент в списке.

Связный список отличается от массива по следующим пунктам:

• **Хранение данных:** В массиве элементы хранятся последовательно в памяти, что позволяет быстро получать доступ к элементам по индексу. В связном списке элементы могут храниться в разных областях памяти, и для доступа к конкретному элементу необходимо пройти через все предшествующие элементы.

• **Размер:** Размер массива фиксирован и определяется при создании. В связном списке увеличение размера списка не требует копирования всех элементов, как в массиве.

• **Вставка и удаление элементов:** В связном списке вставка и удаление элементов находятся на высоком уровне эффективности, так как требуют изменения только ссылок на элементы. В массиве вставка и удаление элементов может потребовать перекопированния всех последующих элементов.

• **Доступ к элементам:** В массиве доступ к элементам осуществляется по индексу за константное время (O(1)), в связном списке время доступа к элементам зависит от их позиции в списке и может быть линейным (O(n)) в худшем случае.

Сложность методов, которые использовались в программе:

• __init__: **O(1)** – создание (инициализация) объекта класса Node или LinkedList.

• __len__: **O(n)** – вычисление длины списка.

• append: **O(n)** – добавление элемента в конец списка.

• __str__: **O(n)** – создание строкового представления списка.

• pop: **O(n)** – удаление последнего элемента списка.

• change_on_start: **O(n)** – изменение поля data n-того элемента с НАЧАЛА списка на new_data.

• clear: **O(1)** – очистка списка.

Отличия в использовании бинарного поиска для линейного списка:

В связном списке требуется последовательный переход от начала списка к середине, так как нельзя прямо обращаться к элементам по индексу. Необходим определенный способ нахождения "середины" списка. Алгоритм бинарного поиска в связном списке может быть менее эффективным из-за необходимости последовательного прохода по элементам списка.

Тестирование

Результаты тестирования программы представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные

1.	node = Node(1)
	print(node) # data: 1, next: None
	node.next = Node(2, None)
	print(node) # data: 1, next: 2
	print(node.get_data())
	l_l = LinkedList()
	print(l_l)
	print(len(l_l))
	l_l.append(111)
	l_l.append(222)
	l_l.append(333)
	print(l_l)
	print(len(l_l))
	l_l.pop()
	print(l_l)
	l_l.pop()
	print(l_l)
	l_l.pop()
	print(l_l)
	l_l.append(111)
	l_l.append(222)
	l_l.append(333)
	l_l.change_on_start(1, 1)
	print(l_l)
		data: 1, next: None
		data: 1, next: 2
		1
		LinkedList[]
		0
		LinkedList[length = 3, [data: 111, next: 222; data: 222, next: 333; data: 333, next: None]]
		3
		LinkedList[length = 2, [data: 111, next: 222; data: 222, next: None]]
		LinkedList[length = 1, [data: 111, next: None]]
		LinkedList[]
		LinkedList[length = 3, [data: 1, next: 222; data: 222, next: 333; data: 333, next: None]]

Выводы

Был создан односвязный список на языке python для хранения элементов. Были изучены способы реализации структур.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

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

    def get_data(self):
        return self._data

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

    def __str__(self):
        if self.next is None:
            return f"data: {self._data}, next: None"
        else:
            return       f"data:       {self._data},       next:
{self.next.get_data()}"

class LinkedList:
    def __init__(self, head=None):
        if head is None:        # Если переменная head равна None, с
оздаем пустой список
            self.head = None
            self.length = 0
        else:
            self.head = head    # Если head не равен None, создаем с
писок из 1 элемента
            self.length = 1

    def __len__(self):
        return self.length

    def append(self, element):
        if self.head is None:
            self.head = Node(element)
            self.length += 1
            return None

        current_element = self.head
        while current_element.next is not None:
            current_element = current_element.next

        current_element.next = Node(element)
        self.length += 1

    def __str__(self):
        if self.length == 0:
            return "LinkedList[]"
```

```
            current_element = self.head
            elements_array = [str(current_element)]

            while (current_element.next != None):
                current_element = current_element.next
                elements_array += [str(current_element)]

            elements_array = "[" + '; '.join(elements_array) + "]"
            return      f"LinkedList[length      =      {len(self)},
{elements_array}]"

        def pop(self):
            if self.length == 0:
                raise IndexError("LinkedList is empty!")
            if self.length == 1:
                self.head = None
                self.length = 0
                return None

            current_element = self.head
            while current_element.next.next is not None:
                current_element = current_element.next
            current_element.next = None
            self.length -= 1

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

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