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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Информатика»

Тема: Алгоритмы и структуры данных в Python. Вариант 3.

Студент гр. 3341 Ступак А.А.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Изучить принципы работы однонаправленного списка на языке Python и научиться применять его для хранения данных и работы с ними. Создать список на основе классов и реализовать методы для работы с ним.

Задание

Вариант 3.

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_end(self, n, new_data) - меняет значение поля data n-того элемента с конца 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

Ответы на вопросы:

1. Указать, то такое связный список. Основные отличия связного списка от массива.

Связный список — это структура данных, состоящая из элементов, которые хранят не только собственную информацию, но и указатели на следующий и /или предыдущий элемент списка.

В связном списке, в отличие от массива, элементы могут размещаться в памяти где угодно, нет необходимости задавать его размер заранее, память выделяется во время выполнения программы, используется больше памяти, т. к. связный список хранит помимо значений указатели на следующий и/ или предыдущий элемент, доступ к элементам осуществляется путем последовательного прохождения по связному списку, а операции добавления и удаления элементов осуществляются быстрее.

2. Указать сложность каждого метода.

1) class Node:

__init__(self, data, next=None) — O(1)

get_data(self) — O(1)

change_data(self, new_data) — O(1)

__str__(self) — O(1)

2) class LinkedList:

__init__(self, head=None) — O(1)

__len__(self) — O(1)

append(self, element) — O(n)

__str__(self) — O(n)

pop(self) — O(n)

change_on_end(self, n, new_data) — O(n)

clear(self) — O(1)

3. Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

Описание возможной реализации бинарного поиска в связном списке:

1. Найти элемент в середине списка.

2. Сравнить этот элемент с искомым значением.

3. Если искомое значение совпало с элементом из середины, то поиск завершен.

4. Если искомое значение не совпало с элементом из середины, то нужно выбрать левую или правую половины списка для дальнейшего поиска:

Если искомое значение меньше, чем значение элемента посередине, то поиск будет продолжен в левой половине списка.

Если искомое значение больше, чем значение элемента посередине, то поиск будет продолжен в правой половине списка.

5. Продолжать алгоритм до тех пор, пока не будет обнаружено искомое значение или пока не будет достигнут конец списка.

Реализация алгоритма бинарного поиска для связного списка отличается от реализации для классического списка тем, что нахождение элемента в середине связного списка будет труднее, т. к. в классическом списке можно получить доступ к элементу по его индексу.

Разработанный программный код см. в приложении А.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	linked_list = LinkedList() print(linked_list) linked_list.append(10) linked_list.append(20) linked_list.append(30) print(linked_list)	LinkedList[] LinkedList[length = 3, [data: 10, next: 20; data: 20, next: 30; data: 30, next: None]]	Выходные данные соответствуют ожиданиям.
2.	linked_list = LinkedList() linked_list.append(10) linked_list.append(20) linked_list.append(30) print(linked_list) linked_list.pop() linked_list.pop() print(linked_list)	LinkedList[length = 3, [data: 10, next: 20; data: 20, next: 30; data: 30, next: None]] LinkedList[length = 1, [data: 10, next: None]]	Выходные данные соответствуют ожиданиям.
3.	linked_list = LinkedList() linked_list.append(10) linked_list.append(20) linked_list.append(30) print(linked_list) linked_list.change_on_end (2, 100) print(linked_list) linked_list.clear() print(linked_list)	LinkedList[length = 3, [data: 10, next: 20; data: 20, next: 30; data: 30, next: None]] LinkedList[length = 3, [data: 10, next: 100; data: 100, next: 30; data: 30, next: None]] LinkedList[]	Выходные данные соответствуют ожиданиям.

Выводы

В ходе выполнения лабораторной работы были изучены и освоены необходимые навыки для создания однонаправленных связных списков на языке Python, а также методов для работы с ними.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.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 is None:
            return f"data: {self.get_data()}, next: {self.next}"
                        return   f"data:   {self.get_data()},   next:
{self.next.get_data()}"

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

    def __len__(self):
        return self.length

    def append(self, element):
        new_node = Node(element)
        self.length += 1

        if self.head is None:
            self.head = new_node
            return

        current_node = self.head

        while current_node.next is not None:
            current_node = current_node.next

        current_node.next = new_node


    def __str__(self):
        if self.head is None:
            return "LinkedList[]"

        current_node = self.head
        data = ""

        while current_node is not None:
```

```
            if current_node.next is not None:
                data += f"{current_node}; "
            else:
                data += f"{current_node}"
            current_node = current_node.next

        return f"LinkedList[length = {self.length}, [{data}]]"

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

            if current_node.next is None:
                self.head = None
                return

            while current_node.next.next is not None:
                current_node = current_node.next

            current_node.next = None

    def change_on_end(self, n, new_data):
        if n > self.length or n <= 0:
            raise KeyError("Element doesn't exist!")

        current_node = self.head
        for _ in range(self.length - n):
            current_node = current_node.next

        current_node.change_data(new_data)

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