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

Студентка гр. 3343 _____ Гельман П.Е.

Преподаватель _____ Иванов Д. И.

Санкт-Петербург

2024

Цель работы

Целью работы является освоение работы с односвязным списком в Python на базе ООП и создание программы на основе полученных знаний.

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:

```
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

Связный список - это структура данных, которая состоит из узлов, каждый из которых содержит какое-то значение и ссылку на следующий узел в списке. Последний узел может указывать на None, обозначая конец списка.

Основные отличия между связным списком и массивом:

1. Динамичность: связный список может изменять размер динамически, в отличие от массива, размер которого задается заранее.
2. Вставка и удаление элементов: в связном списке вставка и удаление элементов происходят быстрее и требуют меньше ресурсов, чем в массиве, так как не требуется сдвигать элементы.
3. Сложность доступа к элементам: в связном списке доступ к произвольному элементу осуществляется за O(n) времени, в то время как в массиве за O(1) времени.

Сложность всех методов:

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)

class LinkedList:

__init__(self, head=None) — O(1)

__len__(self) — O(1)

append(self, element) — O(n)

__str__(self) — O(1)

pop(self) — O(n)

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

clear(self) — O(1)

Алгоритм бинарного поиска предназначен для отсортированного набора значений, соответственно для решения задачи бинарного поиска в связанном списке сначала необходимо его отсортировать, а затем выполнять следующие действия:

1. Найти длину списка.
2. Установить два указателя - левый и правый (указывающие на начало и конец списка).
3. Найти середину списка, опираясь на длину.
4. Сравнить значение в середине с искомым значением.
5. Если значение равно искомому, элемент найден.
6. Если искомое значение меньше значения в середине, продолжить поиск в левой половине списка, обновив правый указатель.
7. Если искомое значение больше значения в середине, продолжить поиск в правой половине списка, обновив левый указатель.
8. Повторять шаги 3-7, пока не будет найден элемент или до тех пор, пока левый указатель не окажется правее правого указателя.

Отличия реализации алгоритма бинарного поиска для связного списка и для классического списка Python:

В классическом списке Python (например, списке list), бинарный поиск эффективен благодаря прямому доступу к элементам по индексу за O(1) времени.

Для отсортированного связного списка бинарный поиск требует гораздно больше времени на каждую итерацию поиска из-за необходимости последовательного прохода по элементам и первоначальной сортировки элементов. Таким образом, бинарный поиск в связном списке может быть менее эффективным по сравнению с классическим списком Python.

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) print(len(linked_list)) linked_list.append(10) print(linked_list) print(len(linked_list)) linked_list.append(20) print(linked_list) print(len(linked_list)) linked_list.pop() print(linked_list) print(linked_list) print(len(linked_list))	LinkedList[] 0 LinkedList[length = 1, [data: 10, next: None]] 1 LinkedList[length = 2, [data: 10, next: 20; data: 20, next: None]] 2 LinkedList[length = 1, [data: 10, next: None]] LinkedList[length = 1, [data: 10, next: None]] 1	Верно
2.	linked_list = LinkedList(10) linked_list.append(20) linked_list.append(30) print("Связанный список до изменения:") print(linked_list) linked_list.change_on_start(2, 25) print("\nСвязанный список после изменения:") print(linked_list) linked_list.pop() print("\nСвязанный список после извлечения:") print(linked_list)	Связанный список до изменения: LinkedList[length = 3, [data: 10, next: 20; data: 20, next: 30; data: 30, next: None]] Связанный список после изменения: LinkedList[length = 3, [data: 10, next: 25; data: 25, next: 30; data: 30, next: None]] Связанный список после извлечения: LinkedList[length = 2, [data: 10,	Верно

	linked_list.clear() print("\nСвязанный список после очистки:") print(linked_list)	next: 25; data: 25, next: None]] Связанный список после очистки: LinkedList[]	

Выводы

Была освоена работа с связанным однонаправленным списком на языке Python с помощью ООП и реализована программа, позволяющая работать с таким списком.

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:
                        return  f"data:  {self.__data},  next:
{self.next.get_data()}"
        else:
            return f"data: {self.__data}, next: None"




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

    def __len__(self):
        return self.length

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

    def __str__(self):
         if not hasattr(self, 'head') or self.head is None or
self.length == 0:
            return "LinkedList[]"
        else:
            current = self.head
            nodes = []
            while current:
                    nodes.append(f"data: {current.get_data()},
next: {current.next.get_data() if current.next else None}")
                current = current.next

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

    def pop(self):
        if self.length == 0:
            raise IndexError("LinkedList is empty!")
```

```
        elif self.length == 1:
            self.head = None
            self.length = 0
            return self
            #raise IndexError("LinkedList is empty!")

        else:
            current = self.head
            while current.next.next:
                current = current.next
            self.length -= 1
            current.next = None

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

        current = self.head
        tmp = self.head
        for _ in range(n-1):
            current = current.next
        current.change_data(new_data)

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