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. 3343 Пухов А. Д.

Преподаватель Иванов Д. В.

Санкт-Петербург

2024

Цель работы

Изучение и применение на практике однонаправленных списков в языке 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_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.

Выполнение работы

Связный список – это структура данных, в которой элементы (узлы) содержат данные и ссылку на следующий элемент в списке. Основное отличие связного списка от массива заключается в следующем:

1. Хранение памяти: узлы связного списка разбросаны в памяти, тогда как элементы массива располагаются последовательно.

2. Размер: связный список может динамически меняться по размеру, массив чаще всего имеет фиксированный размер.

3. Доступ к элементам: в массиве доступ по индексу, что обеспечивает быстрое обращение к элементам; в связном списке для доступа к элементу необходимо последовательное прохождение списка.

4. Изменения: добавление и удаление элементов в связном списке происходит быстрее, так как не требуется перемещать другие элементы, в отличие от массива, где последующие элементы сдвигаются.

Сложность каждого метода:

1) **class Node:**

O(1):

- __init__(self, data, next=None)
- get_data(self)
- change_data(self, new_data)
- __str__(self)

2) **Class LinkedList:**

O(1):

O(1):

- __init__(self, head=None)
- __len__(self)
- clear(self)

O(n):

- append(self, element)
- __str__(self)
- pop(self)
- change_on_end(self, n, new_data)

Бинарный поиск в связном списке неэффективен, потому что он требует последовательного прохода для доступа к элементам, в отличие от массива, где доступ возможен за константное время. В классическом массиве или списке Python бинарный поиск быстро находит середину, так как может обращаться к любому индексу напрямую. Реализовать бинарный поиск в связном списке технически возможно, но это потребует порядка O(n) операций для обращения к серединному элементу и будет значительно менее эффективно по сравнению с O(1) для массива. Обычно, если необходим бинарный поиск, предпочтительнее использовать массивы или другие структуры данных, которые поддерживают прямой доступ к элементам.

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() linked_list.append(10) linked_list.append(20) linked_list.append(30) print(linked_list) linked_list.change_on_end(1, 40) print(linked_list) linked_list.clear() print(linked_list)	LinkedList[] LinkedList[length = 3, [data: 10, next: 20; data: 20, next: 30; data: 30, next: 40; data: 40, next: 50; data: 50, next: None]]	ОК

Выводы

В данной лабораторной работе была изучена работа со списками, и был реализован односвязный список.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lb2.py

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

    def get_data(self):
        return self.__data

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

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

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)
        if self.head is None:
            self.head = new_node
        else:
            tmp = self.head
            while tmp.next:
                tmp = tmp.next
            tmp.next = new_node
        self.length += 1


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

```
        current = self.head
        while current:
            nodes.append(str(current))
            current = current.next
        return f"LinkedList[length = {self.length}, [{'; '.join(nodes)}]]"

def pop(self):
    if self.length == 0:
        raise IndexError("LinkedList is empty!")
    tmp = self.head
    if tmp.next is None:
        self.head = None
    else:
        while tmp.next.next:
            tmp = tmp.next
        tmp.next = None
    self.length -= 1

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

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