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

Студент гр. 3344		**Сьомак Д.А.**
Преподаватель		**Иванов Д.В.**

Санкт-Петербург

2024

Цель работы

Получение практических навыков работы с алгоритмами и структурами данных на языке Python. Написание программы, основанной на реализации связного однонаправленного списка.

Задание

Вариант 1

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:

o __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.

o __len__(self) - перегрузка метода __len__, он должен возвращать длину списка (этот стандартный метод, например, используется в функции len).

o 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.

o __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.

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

o clear(self) - очищение списка.

- delete_on_end(self, n) - удаление n-того элемента с конца списка. Метод должен выбрасывать исключение KeyError, с сообщением "Element doesn't exist!", если количество элементов меньше n.

Выполнение работы

1. Связный список – это динамическая структура данных, список, в котором каждый элемент хранит в себе данные и ссылку на следующий элемент (если список односвязный) или ссылку на следующий и на предыдущий (если список двусвязный). Преимущество связного списка перед массивом заключается в том, что порядок элементов может не совпадать с порядком их расположения в памяти компьютера, однако из-за этого для обращения к определённому элементу нужно проходить по всему списку до него.

2. Сложность реализованных методов:

Класс Node:

__init__ - O(1);

get_data - O(1);

__str__ - O(1).

Класс LinkedList:

__init__ - O(n);

__len__ - O(1);

append - O(n);

__str__ - O(n);

pop - O(n);

clear - O(1);

delete_on_end - O(n).

3. Бинарный поиск в связном списке может быть реализован следующим образом:

1. Найти длину списка.

2. Найти середину списка.

3. Сравнить искомое значение со значением в середине списка.

4. Если искомое значение меньше значения в середине списка, повторить шаги 2–3 для левой половины списка.

5. Если искомое значение больше значения в середине списка, повторить шаги 2–3 для правой половины списка.

6. Если искомое значение равно значению в середине списка, вернуть индекс этого элемента.

Отличия от реализации бинарного поиска для классического списка Python заключаются в том, что для связного списка нельзя получить доступ к элементам по индексу за конкретное время, как в случае с классическим списком Python. Реализация бинарного поиска в односвязном списке неэффективна и не целесообразна потому, что операция взятия по индексу имеет сложность O(N), вследствие чего быстрее просто пройти по каждому элементу списка.

Исходный код см. в приложении А.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	node = Node(1) print(node) node.next = Node(2, None) print(node) 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(len(linked_list))	 data: 1, next: None data: 1, next: None 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]] 1	-

Выводы

Был получен практический опыт работы с алгоритмами и структурами данных на языке Python. Была написана программа, внутри которой была реализован связный однонаправленный список, а также методы для работы с ним.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Somak_Demid_lb2.py

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

    def get_data(self):
        return self.__data

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


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

    def __len__(self):
        return self.length

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

    def __str__(self):
        if self.head != None:
            inf = f'LinkedList[length = {self.length}, ['
            temp = self.head
            while temp:
                inf += str(temp) + '; '
                temp = temp.next
            inf = inf[:-2] + ']]'
            return inf
        else:
            return 'LinkedList[]'

    def pop(self):
        if self.head is None:
            raise IndexError("LinkedList is empty!")
```

```
    elif self.length == 1:
        self.__init__()
    else:
        temp = self.head
        while temp.next.next:
            temp = temp.next
        temp.next = None
        self.length -= 1


def delete_on_end(self, n):
    if self.length < n or n <= 0:
        raise KeyError("Element doesn't exist!")
    else:
        index = self.length - n
        temp = self.head
        if index == 0:
            self.head = temp.next
        else:
            for i in range(index-1):
                temp = temp.next
            temp.next = temp.next.next
        self.length -= 1


def clear(self):
    self.__init__()
```