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. 2384 _____ Кузьминых Е.М

Преподаватель _____ Шевская Н.В.

Санкт-Петербург

2023

Цель работы.

Научиться использовать основные алгоритмы и структуры данных на примере реализации связного списка на языке программирования Python.

Задачи.

Вариант №1

В данной лабораторной работе Вам предстоит реализовать связный однонаправленный список.

Node

Класс, который описывает элемент списка.

Класс *Node* должен иметь 2 поля:

__data # данные, приватное поле

__next__ # ссылка на следующий элемент списка

Вам необходимо реализовать следующие методы в классе *Node*:

__init__(self, data, next)

конструктор, у которого значения по умолчанию для аргумента *next* равно *None*.

get_data(self)

метод возвращает значение поля *__data*.

__str__(self)

перегрузка метода *__str__*. Описание того, как должен выглядеть результат вызова метода смотрите ниже в примере взаимодействия с *Node*.

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.

Класс *LinkedList* должен иметь 2 поля:

__head__ # данные первого элемента списка

__length # количество элементов в списке

Вам необходимо реализовать конструктор:

__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.

и следующие методы в классе *LinkedList*:

__len__(self)

перегрузка метода *__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 метода *__str__*. Описание того, как должен выглядеть результат вызова метода смотрите ниже в примере взаимодействия с *LinkedList*.

pop(self)

удаление последнего элемента. Метод должен выбрасывать исключение *IndexError* с сообщением "*LinkedList is empty!",* если список пустой.

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.

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.

В отчете вам требуется:

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

Указать сложность каждого метода.

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

Выполнение работы.

Данный код содержит два класса: *Node* и *LinkedList*. Класс *Node* представляет элемент связного списка, который содержит данные и указатель на следующий элемент. Класс *LinkedList* представляет сам связный список, который содержит указатель на головной элемент и длину списка.

Класс *Node*:

метод *init* создает новый элемент, устанавливая значение данных и указатель на следующий элемент (по умолчанию *None*)

метод *get_data* возвращает данные узла

метод *str* возвращает строку с данными узла и ссылкой на следующий элемент

Класс *LinkedList*:

метод *init* создает новый связный список, устанавливая головной элемент (по умолчанию *None*) и длину списка

метод *len* возвращает длину списка

метод *append* добавляет новый элемент в конец списка, обновляя указатель последнего узла на новый элемент

метод *pop* удаляет последний элемент из списка и возвращает его данные, обновляя указатель предпоследнего узла на *None*

метод *clear* очищает список, устанавливая головной элемент в *None* и длину списка в 0

метод *delete_on_end* удаляет элемент на n-ой позиции с конца списка, обновляя указатель на предыдущий элемент на следующий элемент

метод *str* возвращает строку с длиной списка и данными всех узлов

Связный список - это структура данных, которая представляет собой последовательность узлов, где каждый элемент содержит данные и указатель на следующий элемент. Основное отличие связного списка от массива заключается в том, что для доступа к элементам массива можно использовать индекс, а для

доступа к элементам связного списка необходимо последовательно перебирать узлы, начиная с головного.

Сложность методов *LinkedList*:

метод *init* - O(1)

метод *len* - O(1)

метод *append* - O(n)

метод *pop* - O(n)

метод *clear* - O(1)

метод *delete_on_end* - O(n)

метод *str* - O(n)

Для реализации бинарного поиска в связном списке можно использовать следующий алгоритм:

Найти длину связного списка.

Установить два указателя на голову списка: *left* и *right*.

Вычислить индекс середины списка: *mid = (left + right) // 2*.

Пройти по списку от начала до середины и переместить указатель *left* на середину списка.

Если искомый элемент меньше элемента, на который указывает *left*, продолжать поиск в левой части списка (установив rig*ht = mid*), иначе продолжать поиск в правой части списка (установив *left = mid*).

Повторять шаги 3-5 до тех пор, пока элемент не будет найден.

Отличие реализации бинарного поиска для связного списка от реализации для классического списка Python заключается в том, что для связного списка требуется проходить по каждому элементу, в отличие от массива, где можно обратиться к элементу по индексу. Это приводит к тому, что бинарный поиск в связном списке имеет линейную сложность *O(n)*, в то время как в обычном списке сложность составляет *O(log n)*. Поэтому, если нужно выполнить множество операций поиска, лучше использовать стандартные списки.

Тестирование.

№	Входные данные	Выходные данные	Комментарий
1	ll = LinkedList() ll.append(1) ll.append(2) ll.append(3) print(ll) ll.pop() print(ll)	LinkedList[length = 3, [data: 1, next: 2; data: 2, next: 3; data: 3, next: None]] LinkedList[length = 2, [data: 1, next: 2; data: 2, next: None]]	Ответ верный

Вывод.

В ходе выполнения лабораторной работы была написана программа, представляющая из себя реализацию однонаправленного односвязного списка, для изучения структур данных и алгоритмов.

Приложение А.

Исходный код программы.

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

    def get_data(self):
        return self.__data

    def __str__(self):
        next_node = 'None' if self.__next__ is None else
self.__next__.__data
        return f"data: {self.__data}, next: {next_node}"


class LinkedList:
    def __init__(self, __head__=None):
        self.__head__ = __head__
        if __head__ == None:
            self.__length = 0
        else:
            self.__length = 1

    def __len__(self):
        return self.__length

    def append(self, element):
        if self.__length > 0:

            n = self.__head__
            while n.__next__ is not None:
                n = n.__next__
            n.__next__ = Node(element)
        else:
            self.__head__ = Node(element)
        self.__length += 1
```

```
def pop(self):
    if self.__length != 0:
        if self.__length==1:
            self.__head__=self.__head__.__next__
        else:
            tmp1 = self.__head__
            tmp2=tmp1.__next__
            while tmp2.__next__ is not None:
                tmp1=tmp2
                tmp2=tmp2.__next__
            tmp1.__next__=None
        self.__length-=1
    else:
        raise IndexError("LinkedList is empty!")


def clear(self):
    self.__length = 0
    self.__head__ = None

def delete_on_end(self, n):
    if 0<n<=self.__length:
        if 0 < n <= self.__length:
            if n == self.__length:
                self.__head__ = self.__head__.__next__
            else:
                tmp = self.__head__
                for i in range(self.__length - n - 1):
                    tmp = tmp.__next__
                tmp.__next__ = tmp.__next__.__next__
            self.__length -= 1
    else:
        raise KeyError("<element> doesn't exist!");
def __str__(self):
    result = 'LinkedList[]'
    if self.__length != 0:
        array = []
```

```
                tmp = self.__head__
                while tmp is not None:
                    array.append(tmp.__str__())
                    tmp = tmp.__next__
                str = '; '.join(array)
                result   =   f'LinkedList[length   =   {self.__length},
[{str}]]'

            return result
```