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

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Введение в ИТ»

Тема: «Алгоритмы и структуры данных в Python»

Студент гр. 9383 _____ Чумак М.А.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2019

Цель работы.

Необходимо реализовать двунаправленный связный список на ООП в Python.

Задание.

В данной лабораторной работе Вам предстоит реализовать связный двунаправленный список.

Node

Класс, который описывает элемент списка.

Класс Node должен иметь 3 поля:

__data # данные, приватное поле

__prev__ # ссылка на предыдущий элемент списка

__next__ # ссылка на следующий элемент списка

Вам необходимо реализовать следующие методы в классе Node:

__init__(self, data, prev, next)

конструктор, у которого значения по умолчанию для аргументов prev и next равны 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, prev: None, next: None

node.__prev__ = Node(2, None, None)

print(node) # data: 1, prev: 2, next: None

node.__next__ = Node(3, None, None)

print(node) # data: 1, prev: 2, next: 3

Linked List

Класс, который описывает связный двунаправленный список.

Класс LinkedList должен иметь 3 поля:

__length # длина списка

__first__ # данные первого элемента списка

__last__ # данные последнего элемента списка

Вам необходимо реализовать конструктор:

__init__(self, first, last)

конструктор, у которого значения по умолчанию для аргументов first и last равны None.

Если значение переменной first равно None, а переменной last не равно None, метод должен вызывать исключение ValueError с сообщением: "invalid value for last".

Если значение переменной first не равно None, а переменной last равна None, метод должен создавать список из одного элемента. В данном случае, first равен last, ссылки prev и next равны None, значение поля __data для элемента списка равно first.

Если значения переменных не равны None, необходимо создать список из двух элементов. В таком случае, значение поля __data для первого элемента списка равно first, значение поля __data для второго элемента списка равно last.

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.

popitem(self, element)

удаление элемента, у которого значение поля __data равно element. Метод должен выбрасывать исключение KeyError, с сообщением "<element> doesn't exist!", если элемента в списке нет.

clear(self)

очищение списка.

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, prev: None, next: None]]
print(len(linked_list)) # 1

linked_list.append(20)
print(linked_list)
# LinkedList[length = 2, [data: 10, prev: None, next: 20; data: 20, prev: 10, next: None]]
```

print(len(linked_list)) # 2

linked_list.pop()

print(linked_list)

print(linked_list) # LinkedList[length = 1, [data: 10, prev: None, 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:

1. Указать, что такое связный список. Основные отличия связного списка от массива.
2. Указать сложность каждого метода.
3. Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

Выполнение работы.

1. Связный список — это базовая динамическая структура данных, состоящая из узлов, каждый из которых содержит данные, а также одну или две ссылки на следующих и предыдущий элементы. Основное назначение связного списка — предоставление механизма для хранения и доступа к произвольному количеству данных. Как следует из названия, это достигается связыванием данных вместе в список.

Элементы в связном списке в отличии от элементов в массиве в памяти идут не по порядку.

2. Сложность методов

Сложность методов Node:

__init__ : - O(1)

get_data: - O(1)

__str__: - O(1)

Сложность методов LinkedList:

__init__ : - O(1)

__len__ : - O(1)

append: - O(1)

__str__: - O(n)

pop: - O(1)

popitem: - O(n)

clear: - O(1)

3. Возможная реализация бинарного поиска в связном списке.

Необходимо дойти до элемента списка находящегося по середине, длина списка известна, поэтому можно реализовать это с помощью цикла while.

Сравнить элемент до которого дошли с искомым элементом, если текущий больше искомого, то сместиться влево на половину номера текущего элемента , если текущий меньше искомого то вправо. В реализации бинарного поиска в классическом списке python перемещаться к элементам можно по индексу, в отличии от связного списка, где это происходит с помощью цикла while.

Выводы.

Был реализован двусвязный направленный список в ООП на языке программирования Python. Для этого были написаны два класса LinkedList и Node.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.py:

```
class Node():

    def __init__(self, data, prev = None, next = None):
        self.__data = data
        self.__prev__ = prev
        self.__next__ = next

    def get_data(self):
        return self.__data

    def __str__(self):
        next = None
        prev = None
        if self.__next__ is not None:
            next = self.__next__.get_data()
        if self.__prev__ is not None:
            prev = self.__prev__.get_data()
        return 'data: {}, prev: {}, next: {}'.format(self.get_data(), prev, next)


class LinkedList():

    def __init__(self, first = None, last = None):
        self.__length = 0
        self.__first__ = first
        self.__last__ = last
        if first is None and last is not None:
            raise ValueError("invalid value for last")
        elif last is None and first is not None:
            self.__first__= Node(first)
            self.__last__ = self.__first__
            self.__length = 1
        elif first is not None and last is not None:
            self.__first__ = Node(first)
            self.__last__ = Node(last)
            self.__first__.__next__ = self.__last__
            self.__last__.__prev__ = self.__first__
            self.__length = 2

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

```
            self.__length = 1
            self.__first__ = Node(element)
            self.__last__ = self.__first__
        else:
            self.__length += 1
            self.__last__.__next__ = Node(element, self.__last__)
            self.__last__ = self.__last__.__next__

    def __len__(self):
        return self.__length


    def pop(self):
        if self.__length == 0:
            raise IndexError('LinkedList is empty!')
        self.__last__ = self.__last__.__prev__
        self.__last__.__next__ = None
        self.__length -= 1

    def __str__(self):
        if self.__length == 0:
            return 'LinkedList[]'
        else:
            s = self.__first__
            result = str(s)
            while s.__next__  is not None:
                s = s.__next__
                result += '; '+str(s)
            return 'LinkedList[length = {}, [{}]]'.format(self.__length, result)


    def clear(self):
        self.__first__ = Node(None)
        self.__last__ = Node(None)
        self.__length = 0

    def popitem(self, element):
        mid = self.__first__
        num = True
        while mid is not None:
            if element == mid.get_data():
                if self.__length == 1:
                    self.__first__ = Node(None)
                    self.__length = 0
                    self.__last__ = self.__first__
```

```
            num = False
            break
        if mid.__prev__ is None:
            self.__first__ = self.__first__.__next__
            self.__first__.__prev__ = None
            self.__length -= 1
        elif mid.__next__ is None:
            self.__last__ = self.__last__.__prev__
            self.__last__.__next__ = None
            self.__length -= 1
        else:
            mid.__next__.__prev__ = mid.__prev__
            mid.__prev__.__next__ = mid.__next__
            self.__length -= 1
        num = False
        break
    mid = mid.__next__
if num:
    raise KeyError("{} doesn't exist!".format(element))
```