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 информационные технологии»
Тема: «Алгоритмы и структуры данных в Python»

Студент гр. 9383 _____ Корсунов А.А.

Преподаватель _____ Розмочаева Н.В.

Санкт-Петербург
2019

Цель работы.

Научиться работать с алгоритмами и структурами данных.

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) Связаный список — базовая динамическая структура данных в информатике, состоящая из узлов, каждый из которых содержит как собственно

данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка.

Основные отличия от массива:

- Массив хранится в памяти как упорядоченная последовательность элементов (хранятся строго друг за другом), когда как в списке элементы хранятся неупорядоченно.
- В связанном списке помимо самого элемента хранятся указатели на следующий и предыдущий элементы, на что выделяется дополнительная память.
- В связанном списке доступ к элементу осуществляется путем сменой указателя, когда в массиве доступ к элементу осуществляется по его индексу.

2) Сложности методов:

- __inir__() - 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 связанном списке:

С помощью цикла while перемещаемся к следующему элементу связного списка, пока не встретится None, предварительно заводится счетчик длины списка. Таким образом будет известна длина списка. Далее заводится еще одно поле, с помощью которого можно будет определить «индекс» элемента. От начала списка происходит перемещение по следующим элементам, пока это поле не станет равно половины длины списка (первоначальное поле // 2). Если значение текущего элемента меньше искомого, то происходит смещение вправо, первое поле становится равно второму полю, второе поле обнуляется, если больше — смещаемся влево, первое поле равно второму, второе поле

обнуляется, если равно — возвращаем значение элемента. Все это происходит в цикле пока первое поле не станет равно 1 или пока рассматриваемый элемент не станет равен искомому.

Вывод.

В ходе проделанной работы была изучена работа с алгоритмами и структурами данных в Python. Были использованы возможности двусвязного списка путем переопределения требуемых по заданию методов.

Приложение А
Исходный код программ

```
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):
        if self.__next__ is None:
            next = None
        else:
            next = self.__next__.get_data()
        if self.__prev__ is None:
            prev = None
        else:
            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.__first__ = first
        self.__last__ = last
        self.__length = 0
        if first is None and last is not None:
            raise ValueError("invalid value for last")
        if first is not None and last is None:
            self.__length = 1
            self.__first__ = Node(first)
            self.__last__ = self.__first__
        if (first is not None) and (last is not None):
            self.__length = 2
            self.__first__ = Node(first)
            self.__last__ = Node(last)
            self.__first__.__next__ = self.__last__
            self.__last__.__prev__ = self.__first__
```

```
def __len__(self):
    return self.__length


def append(self, element):
    if self.__first__ is None:
        self.__length = self.__length + 1
        self.__init__(element)
    else:
        self.__length = self.__length + 1
        self.__last__.__next__ = Node(element, prev = self.__last__)
        self.__last__ = self.__last__.__next__

def __str__(self):
    if self.__first__ is None:
        return "LinkedList[]"
    if self.__first__ is not None:
        buffer = self.__first__
        list_r = str(buffer)
        while buffer.__next__ is not None:
            buffer = buffer.__next__
            list_r += "; " + str(buffer)
        return "LinkedList[length = {}, [{}]]".format(len(self), list_r)


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


def popitem(self, element):
    if self.__first__ is None:
        raise KeyError("{} doesn't exist!".format(element))
    if len(self) == 1 and self.__first__.get_data() == element:
        self.clear()
    elif self.__first__.get_data() == element:
        self.__first__ = self.__first__.__next__
        self.__first__.__prev__ = None
        self.__length = self.__length - 1
```

```
    elif self.__last__.get_data() == element:
        self.pop()
    else:
        bufer = self.__first__
        while bufer.get_data() != element and bufer.__next__ != None:
            bufer = bufer.__next__
        if bufer.__next__ == None:
            raise KeyError("{} doesn't exist!".format(element))
        bufer.__prev__.__next__ = bufer.__next__
        bufer.__next__.__prev__ = bufer.__prev__


def clear(self):
    self.__length = 0
    self.__init__()
```