

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

Студентка гр. 9304

Каменская Е.К.

Преподаватель

Размочаева Н.В.

Санкт-Петербург

2019

Цель работы.

Реализовать двусвязный список на языке Python.

Задание.

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
```

Выполнение работы.

1. Связный список - структура данных, представляющая собой конечное множество упорядоченных элементов (узлов), связанных друг с другом посредством указателей. Каждый элемент связного списка содержит поле с данными, а также указатель (ссылку) на следующий и/или предыдущий элемент.

Основные отличия:

- В памяти элементы хранятся не обязательно по порядку
- Возможность добавлять и удалять элементы без затрагивания других элементов
- Невозможна работа по индексам

2. Класс Node:

Конструктор `__init__(self, data, prev=None, next=None)`: присваивает полученные значения `prev`, `next` и `data` соответствующим полям. Если `prev` и `next` не переданы, по умолчанию задает их равными `None`.

Метод `get_data(self)`: возвращает значение поля `__data`. Сложность $O(1)$.

Метод `__str__(self)`: возвращает строку в формате "`data: <значение объекта>`", `prev: <значение предшествующего объекта>`, `next: <значение следующего объекта>`". Сложность $O(1)$.

Класс *LinkedList*:

Конструктор `__init__(self, first=None, last=None)`: поля `__first__` и `__last__` по умолчанию равны *None*, `__length` – нулю. Проверяется корректность введенных значений *last* и *first*. Если они неверны, выбрасывается *ValueError*, иначе, в зависимости от значений, создаются один или два элемента класса *Node*, меняется поле `__length`.

Метод `__len__(self)`: возвращает значение поля `__length`. Сложность $O(1)$.

Метод `__str__(self)`: возвращает строку в формате “*LinkedList*[*length* = <длина списка>, [<элемент1; элемент2; ... >]]”. Сложность $O(n)$.

Метод `__clear__(self)`: обнуляет приватное поле `__length` и полям `__first__` и `__last__` присваивает значение *None*. Сложность $O(1)$.

Метод `append(self, element)`: если в списке нет элементов, неравных *None*, задает значение первого своего элемента равным переданному значению. Если элементы есть, меняет значение поля `__last__` у последнего элемента. В любом случае приравнивает поле `__last__` к введенному значению и увеличивает `__length` на 1. Сложность $O(1)$

Метод `pop(self)`: если длина списка нулевая, выбрасывает *IndexError*, иначе меняет значение поля `__next__` предпоследнего элемента на *None* и декрементирует `__length` на единицу. Сложность $O(1)$.

Метод `popitem(self, element)`: задает переменную *current*, равную первому элементу списка. Пока *current* не равна *None* или *element*, итерируется по списку. Если *current* не равна *None*, `__length` декрементируется на единицу, проверяются значения предыдущего и следующего за найденным элементов и по результатам проверки меняются значения их полей `__next__` и `__prev__` или полей списка `__first__` и `__last__`. Если элемент не найден, выбрасывается *KeyError*. Сложность $O(n)$.

Разработанный программный код см. в приложении А.

3. Возможная реализация бинарного поиска в двунаправленном связном списке:

Создается переменная `index`, равная половине длины списка. Если `index` равна нулю, сообщается, что элемент не найден. Перейдя на `index` элементов вперед в списке, сравниваем значение элемента с искомым и, если они равны, возвращаем `True`. В противном случае `index` делится на 2 и мы перемещаемся вперед или назад по списку на `index` элементов, в зависимости от того, больше или меньше оказался искомый элемент. Алгоритм повторяется пока элемент не найден или `index` не равна нулю.

Выводы.

По принципу ООП на языке Python реализован двусвязный список и ряд методов для работы с ним. Каждому методу дана оценка сложности.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lab4.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):
        s = f"data: {self.__data}, "
        if self.__prev__ != None:
            s += f"prev: {self.__prev__.__data}, "
        else:
            s += "prev: None, "
        if self.__next__ != None:
            s += f"next: {self.__next__.__data}"
        else:
            s += "next: None"
        return s

class LinkedList:

    def __init__(self, first=None, last=None):
        self.__first__ = None
        self.__last__ = None
        self.__length = 0

        if (first == None) and (last != None):
            raise ValueError("invalid value for last")
        elif (first != None) and (last == None):
            node = Node(first)
            self.__first__ = node
            self.__last__ = node
            self.__length = 1
        elif (first != None) and (last != None):
            node1 = Node(first)
            node2 = Node(last)
            node1.__next__ = node2
            node2.__prev__ = node1
            self.__first__ = node1
            self.__last__ = node2
            self.__length = 2

    def __len__(self):
        return self.__length

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



```

        current = self.__first__
        output = []
        while current != None:
            output.append(str(current))
            current = current.__next__
        return f"LinkedList[length = {self.__length}, [" + ";
".join(output) + "]"

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

    def append(self, element):
        node = Node(element)
        node.__prev__ = self.__last__
        if self.__first__ == None:
            self.__first__ = node
        if self.__last__ != None:
            self.__last__.__next__ = node
        self.__last__ = node
        self.__length += 1

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

    def popitem(self, element):
        current = self.__first__
        while (current != None) and (current.get_data() !=
element):
            current=current.__next__
        if current != None:
            self.__length -= 1
            if current.__prev__ != None:
                current.__prev__.__next__ = current.__next__
            else:
                self.__first__ = current.__next__
            if current.__next__ != None:
                current.__next__.__prev__ = current.__prev__
            else:
                self.__last__ = current.__prev__
        else:
            raise KeyError(element.__str__() + " doesn't exist!")

```