**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 О(1).

**Метод *\_\_str\_\_(self):*** возвращает строку в формате *“data: <значение объекта>, prev: <значение предшествующего объекта>, next: <значение следующего объекта>”.* Сложность О(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 О(1).

**Метод *\_\_str\_\_(self):***возвращает строку в формате *“LinkedList[length = <длина списка>, [<элемент1; элемент2; … >]]”*. Сложность О(n).

**Метод *\_\_clear\_\_(self):***обнуляет приватное поле *\_\_length* и полям *\_\_first\_\_* и *\_\_last\_\_* присваивает значение *None*. Сложность О(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 О(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 О(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 О(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!")