**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. 3341 |  | Байрам Э. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Изучите основные алгоритмы и структуры данных в Python, освойте основные методы работы со списками в этом языке, затем создайте программу и реализуйте в ней односвязный список для хранения числовых данных.

## Задание

В этой лабораторной работе вам предстоит реализовать односвязный список. Для этого необходимо создать два зависимых класса:

Node

Этот класс описывает элемент списка.

У него должно быть 2 поля:

o data # Данные элемента списка, приватное поле.

o next # Ссылка на следующий элемент списка.

И следующие методы:

o \_\_init\_\_(self, data, next) - конструктор, у которого значения по умолчанию для аргумента next равно None.

o get\_data(self) - метод возвращает значение поля data (это необходимо, потому что в идеале пользователь класса не должен трогать поля класса Node).

o change\_data(self, new\_data) - метод меняет значение поля data объекта Node.

o \_\_str\_\_(self) - перегрузка стандартного метода \_\_str\_\_, который преобразует объект в строковое представление. Для данной лабораторной необходимо реализовать следующий формат перевода объекта класса Node в строку:

“data: <node\_data>, next: <node\_next>”,

где <node\_data> - это значение поля data объекта Node, <node\_next> - это значение поля next объекта, на который мы ссылаемся, если он есть, иначе None.

Пример того, как должен выглядеть результат реализации \_\_str\_\_ см. ниже.

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

Этот класс описывает односвязный список.

У него должно быть 2 поля:

o head # Данные первого элемента списка.

o length # Количество элементов в списке.

И следующие методы:

o \_\_init\_\_(self, head) - конструктор, у которого значения по умолчанию для аргумента head равно None.

· Если значение переменной head равно None, метод должен создавать пустой список.

· Если значение head не равно None, необходимо создать список из одного элемента.

o \_\_len\_\_(self) - перегрузка метода \_\_len\_\_, он должен возвращать длину списка (этот стандартный метод, например, используется в функции len).

o 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.

o \_\_str\_\_(self) - перегрузка стандартного метода \_\_str\_\_, который преобразует объект в строковое представление. Для данной лабораторной необходимо реализовать следующий формат перевода объекта класса однонаправленного списка в строку:

· Если список пустой, то строковое представление:

“LinkedList[]”

· Если не пустой, то формат представления следующий:

“LinkedList[length = <len>, [data:<first\_node>.data, next: <first\_node>.data; data:<second\_node>.data, next:<second\_node>.data; … ; data:<last\_node>.data, next: <last\_node>.data]”,

где <len> - длина связного списка, <first\_node>, <second\_node>, <third\_node>, … , <last\_node> - элементы однонаправленного списка.

Пример того, как должен выглядеть результат реализации см. ниже.

o pop(self) - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.

o clear(self) - очищение списка.

o change\_on\_start(self, n, new\_data) - изменение поля data n-того элемента с НАЧАЛА списка на new\_data. 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.

## Выполнение работы

Класс Node создается с приватным полем data (хранящим данные узла) и публичным полем next (ссылкой на следующий узел). Метод \_\_init\_\_() инициализирует экземпляр класса Node, где значение next по умолчанию равно None. Метод get\_data() возвращает значение приватного поля data, а change\_data() заменяет данные узла на новые. Метод \_\_str\_\_() создает строковое представление объекта класса Node с использованием форматирования строк.

Затем создается класс LinkedList, содержащий два поля: head (данные первого узла списка) и length (количество элементов в списке). Метод \_\_init\_\_() инициализирует объект класса LinkedList в зависимости от содержимого head. Метод \_\_len\_\_() возвращает значение поля length. Метод append() добавляет узел с данными element в конец списка. Метод \_\_str\_\_() возвращает представление списка в виде строки. Метод pop() удаляет последний узел из списка, вызывая исключение IndexError, если список пуст. Метод clear() сбрасывает значение head на None и length на 0. Метод change\_on\_start() заменяет данные n-го узла с начала списка, выбрасывая исключение KeyError, если узла с индексом n нет в списке.

Код программы представлен в приложении А.

## Выводы

В процессе выполнения работы были освоены основные алгоритмы и структуры данных в языке Python, а также освоены базовые методы работы со списками. С использованием этих знаний была создана программа, в рамках которой был реализован односвязный список для хранения численных данных. Этот опыт позволил лучше понять принципы работы структур данных и их применение в реальных задачах.

# **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 change\_data(self, new\_data):

self.\_\_data = new\_data

def \_\_str\_\_(self):

if self.next is not None:

return f"data: {self.get\_data()}, next: {self.next.get\_data()}"

else:

return f"data: {self.get\_data()}, next: None"

class LinkedList:

def \_\_init\_\_(self, head = None):

self.head = head

if head is not None:

self.length = 1

else:

self.length = 0

def \_\_len\_\_(self):

return self.length

def append(self, element):

new\_node = Node(element)

if self.head is not None:

current = self.head

while current.next is not None:

current = current.next

current.next = new\_node

else:

self.head = new\_node

self.length += 1

def \_\_str\_\_(self):

if self.head is None:

result = f"LinkedList[]"

else:

result = f"LinkedList[length = {self.length}, ["

current = self.head

while current is not None:

if current.next is None:

result += f"data: {current.get\_data()}, next: None]]"

else:

result += f"data: {current.get\_data()}, next: {current.next.get\_data()}; "

current = current.next

return result

def pop(self):

if self.head is None:

raise IndexError("LinkedList is empty!")

elif self.head.next is None:

self.head = None

else:

current = self.head

while current.next.next is not None:

current = current.next

current.next = None

self.length -= 1

def clear(self):

self.head = None

self.length = 0

def change\_on\_start(self, n, new\_data):

if n > self.length or n <= 0:

raise KeyError("Element doesn't exist!")

current = self.head

for i in range(n-1):

current = current.next

current.change\_data(new\_data)