**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. 3343 |  | Силяев Р.А. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Изучить особенности работы с односвязным списком на языке Python. Написать программу и сравнить список и массив.

## Задание

**Node**

Класс, который описывает элемент списка.

Он должен иметь 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.

**Linked List**

Класс, который описывает связный однонаправленный список.

Он должен иметь 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 равна 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.

## Выполнение работы

LinkedList или связный список – это структура данных. Связный список обеспечивает возможность создать двунаправленную очередь из каких-либо элементов. Каждый элемент такого списка считается узлом. По факту в узле есть его значение, а также две ссылки – на предыдущий и на последующий узлы. То есть список «связывается» узлами, которые помогают двигаться вверх или вниз по списку. Из-за таких особенностей строения из связного списка можно организовать стек, очередь или двойную очередь. 0w0

Сложности методов:

Класс Node:

1. \_\_init\_\_ – O(1);
2. get\_data – O(1);
3. change\_data – O(1);
4. \_\_str\_\_ – O(1).

Класс LinkedList:

1. \_\_init\_\_ – O(n);
2. \_\_len\_\_ – O(1);
3. append – O(n);
4. \_\_str\_\_ – O(n);
5. pop – O(n);
6. clear – O(1);
7. change\_on\_start – O(n).

Для бинарного поиска требуется log(n) раз получать элементы каждый такой запрос займет O(n). В связном списке же за O(n) уже будет найден элемент.

## Тестирование

Результаты тестирования содержатся в таблице 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 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 | 0  LinkedList[length = 1, [data: 10, next: None]]  1  LinkedList[length = 2, [data: 10, next: 20; data: 20, next: None]]  2  LinkedList[length = 1, [data: 10, next: None]]  LinkedList[length = 1, [data: 10, next: None]]  1 | 👍 |

## Выводы

В ходе выполнения лабораторной работы были изучены особенности работы с односвязным списком на языке Python. Также была написана программа, изучены сложности списка и массива.

# Приложение А Исходный код программы

Название файла: main.py

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):

return f'data: {self.\_\_data}, next: {None if self.next is None else self.next.get\_data()}'

class LinkedList():

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

if head is None:

self.length = 0

self.head = None

else:

self.length = 1

self.head = Node(head)

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

return self.length

def append(self, element):

temp = Node(element)

if self.length == 0:

self.length = 1

self.head = temp

return

cur = self.head

while cur.next:

cur = cur.next

self.length += 1

cur.next = temp

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

temp = self.head

if self.head is None:

return 'LinkedList[]'

else:

value = []

while temp != None:

value.append(temp.\_\_str\_\_())

temp = temp.next

return f'LinkedList[length = {len(self)}, [{"; ".join(value)}]]'

def pop(self):

if self.length == 0:

raise IndexError('LinkedList is empty!')

elif self.length == 1:

self.clear()

else:

temp = self.head

while temp.next.next:

temp = temp.next

self.length -= 1

temp.next = None

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

temp = self.head

prev = temp

cur = Node(new\_data)

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

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

if n == 1:

cur.next = self.head.next

self.head = cur

else:

for i in range(n - 1):

prev = temp

temp = temp.next

cur.next = temp.next

prev.next = cur

def clear(self):

self.head = None

self.length = 0