**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе №2**

**по дисциплине «Информатика»**

**Тема: Алгоритмы и структуры данных в Python.**

| Студент гр. 3343 |  | Волох И.О. |
| --- | --- | --- |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург

2024

## Цель работы

Научиться создавать класс односвязного списка на языке программирования Python, описывать элемент списка, методы для изменения элементов в односвязном списке.

## Задание

В данной лабораторной работе Вам предстоит реализовать связный **однонаправленный** список. Для этого необходимо реализовать 2 зависимых класса:

**Node**

Класс, который описывает элемент списка.

Он должен иметь 2 поля:

* **data** - Данные элемента списка, приватное поле.
* **next** - Ссылка на следующий элемент списка.

И следующие методы:

* **\_\_init\_\_(self, data, next)** - конструктор, у которого значения по умолчанию для аргумента next равно None.
* **get\_data(self)** - метод возвращает значение поля data (это необходимо, потому что в идеале пользователь класса не должен трогать поля класса Node).
* **\_\_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:

* **head** - Данные первого элемента списка.
* **length** - Количество элементов в списке.

И следующие методы:

* **\_\_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.
* **\_\_len\_\_(self)** - перегрузка метода \_\_len\_\_, он должен возвращать длину списка (этот стандартный метод, например, используется в функции 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 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.
* **pop(self)** - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.
* **clear(self)** - очищение списка.
* **delete\_on\_end(self, n)** - удаление n-того элемента с конца списка. Метод должен выбрасывать исключение KeyError, с сообщением "Element doesn't exist!", если количество элементов меньше n.

## Выполнение работы

Связный список — это структура данных, состоящая из узлов, каждый из которых содержит какое-то значение и ссылку на следующий узел в списке. Последний узел указывает на нуль.

Методы реализованные в работе имеют следующую сложность:

O(1) : \_\_init\_\_ , get\_data, \_\_len\_\_, clear, \_\_iter\_\_ , \_\_next\_\_ .

O(n): append, \_\_str\_\_, pop, delete\_on\_end .

Основные отличия между связным списком и массивом:

1. В массиве доступ к элементам осуществляется по индексу. В связном списке доступ к элементам происходит последовательно, начиная с начала списка и переходя от узла к узлу.
2. Память: В массиве память выделяется непрерывным блоком. Связный список использует динамическое выделение памяти для каждого узла, что позволяет его размер списка.
3. Вставка и удаление элементов: В массиве вставка и удаление элементов требует сдвигать остальные элементы. В связном списке вставка и удаление элементов происходит быстро, так как требуется только изменение ссылок на узлы.
4. Размер: Размер массива фиксирован и определяется при его создании, в то время как связный список может быть динамический.

Для бинарного поиска в связном списке: найти середину списка и сравнить искомый элемент с элементом в середине. Если он меньше, то меняем указатель на конец списка( ставим на узел, предшествующий середине ) и ищем элемент в середине левой части затем сравниваем с ним, иначе делаем в правой части тоже самое, но меняем указатель на начало списка( указывал на узел, следующий за серединой ). Продолжаем поиск до нахождения искомого элемента.

Основные различия: нужно дополнительно обновлять указатель на конец и начало списка.

**Вывод.**

Была написана программа, которая содержит в себе реализацию односвязного линейного списка. В ней можно создавать список, добавлять и удалять элементы в различных позициях, а также выводить информацию о каждом элементе списка.

# Приложение А Исходный код программы

Название файла: main.py

class Node:

def \_\_init\_\_(self, data = None, next = None):

self.data = data

self.next = next

def get\_data(self):

return self.data

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

if self.next != None:

return f'data: {self.data}, next: {self.next.data}'

else:

return f'data: {self.data}, next: {self.next}'

class LinkedList(list):

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

self.head = head

self.length = len(self)

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

count = 0

for i in self:

count += 1

return count

def append(self, element):

addition = Node(element)

if self.head is None:

self.head = addition

else:

current = self.head

while current.next is not None:

current = current.next

current.next = addition

def \_\_iter\_\_(self):

current = self.head

while current:

yield current

current = current.next

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

if len(self) == 0:

return "LinkedList[]"

else:

current = self.head

nodes\_info = []

while current:

if current.next is not None:

nodes\_info.append("data: {}, next: {}".format(current.data, current.next.data))

else:

nodes\_info.append("data: {}, next: {}".format(current.data, current.next))

current = current.next

nodes\_info\_str = "; ".join(nodes\_info)

return 'LinkedList[length = {}, [{}]]'.format(len(self), nodes\_info\_str)

def pop(self):

if self.head is None:

raise IndexError("LinkedList is empty!")

elif self.head.next == None:

self.head = None

else:

current = self.head

while current.next.next is not None:

current = current.next

current.next = None

def clear(self):

self.head = None

def delete\_on\_end(self, n):

if n > len(self) or n <= 0:

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

n = len(self) - n

if n == len(self) - 1:

self.pop()

return self

current = self.head

second\_part = []

i, j = 0, 0

if n == 0:

self.head = self.head.next

return self

while current != None:

if j > n:

second\_part.append(current)

current = current.next

j += 1

current = self.head

while i != n - 1:

current = current.next

i += 1

current.next = second\_part[0]