

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра МОЭВМ**

**ОТЧЕТ**  
**по лабораторной работе №2**  
**по дисциплине «Информатика»**  
**Тема: «Алгоритмы и структуры данных в Python»**

Студент гр. 3342

Белаид Фарук

Преподаватель

Иванов Д. В.

Санкт-Петербург

2024

## **Цель работы**

Изучить алгоритмы и структуры данных в Python, способы их реализации. Написать согласно заданию программу, которая реализует структуру однонаправленного связного списка в виде класса.

## Задание

В данной лабораторной работе Вам предстоит реализовать связный *однонаправленный* список.

### Node

Класс, который описывает элемент списка.

Класс Node должен иметь 2 поля:

**\_\_data** - данные, приватное поле

**\_\_next\_\_** - ссылка на следующий элемент списка

Вам необходимо реализовать следующие методы в классе Node:

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

**get\_data(self)** - метод возвращает значение поля \_\_data.

**change\_data(self, new\_data)** - метод меняет значение поля \_\_data.

**\_\_str\_\_(self)** - перегрузка метода \_\_str\_\_. Описание того, как должен выглядеть результат вызова метода смотрите ниже в примере взаимодействия с Node.

### Linked List

Класс, который описывает связный однонаправленный список.

Класс LinkedList должен иметь 2 поля:

**\_\_head\_\_** - данные первого элемента списка

**\_\_length** - количество элементов в списке

Вам необходимо реализовать конструктор:

**\_\_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.

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.

**`clear(self)`** - очищение списка.

**`change_on_end(self, n, new_data)`** - меняет значение поля `_data` `n`- того элемента с конца списка на `new_data`. Метод должен выбрасывать исключение `KeyError`, с сообщением `"<element> doesn't exist!"`, если количество элементов меньше `n`.

## **Основные теоретические положения**

Связный список – это структура данных, которая состоит из узлов, включающих какое-нибудь содержимое и указатель на следующий элемент. Отличием связного списка от массива заключается в том, что у списка элементы в памяти не упорядочены, он может хранить содержимое разных типов данных, обладает более быстрыми вставкой и удалением элемента, о при этом обращение по индексу происходит медленнее.

## Выполнение работы

В ходе выполнения работы были реализованы следующие классы и их методы:

### Node

Поля:

- `__data` – содержимое элемента списка
- `__next__` – ссылка на следующий элемент списка

Методы:

- `__init__(self, data, next=None)` – конструктор.
- `get_data(self)` – возвращает значение поля `__data`.  $O(1)$
- `change_data(self, new_data)` – меняет значение поля `__data`.  $O(1)$
- `__str__(self)` – перегрузка метода, возвращает строку с информацией об элементе.  $O(1)$

### Linked List

Поля:

- `__head__` – ссылка на первый элемент списка
- `__length` – количество элементов в списке

Методы:

- `__init__(self, head=None)` – создаёт пустой список или состоящий из одного элемента в зависимости от того, передано ли `head`.
- `__len__(self)` – перегрузка метода, возвращает значение `__length`.  $O(1)$
- `append(self, element)` – добавляет элемент в конец списка, при помощи цикла `while` доходит до последнего элемента и его полю `__next` присваивает ссылку на следующий элемент.  $O(n)$
- `__str__(self)` – перегрузка метода, возвращает строку с информацией о списке (получает информацию, проходя по списку).  $O(n)$
- `pop(self)` – удаляет последний элемент, а в случае, если список пустой, выбрасывает исключение `IndexError` с сообщением "LinkedList is empty!".  $O(n)$

- *change\_on\_end(self, n, new\_data)* – изменяет содержимое n-того с конца элемента, доходя до него в цикле *while* и изменяя значение по помощи метода элемента *change\_data*, а в случае неверного индекса выбрасывает исключение *KeyError*, с сообщением "<element> doesn't exist!".  $O(n)$
- *clear(self)* – очищает список, присваивая первому элементу *None* и изменяя длину на 0.  $O(1)$

Алгоритм бинарного поиска работает с отсортированными структурами данных, что означает то, что связный список для начала нужно отсортировать. Двоичный поиск заключается в том, что находится середина списка и сравнивается элемент в середине с искомым, в случае, если искомый больше то мы находим середину уже правой части, иначе левой. Так продолжается, пока не сойдётся всё до одного элемента, который и будет являться тем, который мы искали. Сложность бинарного поиска в списке заключается в том, что мы не можем быстро найти серединный элемент, потому что мы не можем брать по индексу, нам придётся каждый раз проходить половину рабочей области. В итоге сложность бинарного поиска будет больше  $O(n)$ , что делает его бесполезным в работе со списками, так как пройти по нему будет проще и быстрее.

Разработанный программный код см. в приложении А.

## Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	<pre>node = Node(1) print(node) node.__next__ = Node(2, None) print(node)</pre>	<pre>data: 1, next: None data: 1, next: 2</pre>	Верно



2.	<pre> l_1 = LinkedList() print(l_1) l_1.append(10) l_1.append(20) l_1.append(30) l_1.append(40) print(l_1) print(len(l_1)) l_1.change_on_end(2, 3) l_1.change_on_end(4, 100) print(l_1) </pre>	<pre> LinkedList[] LinkedList[length = 4, [data: 10, next: 20; data: 20, next: 30; data: 30, next: 40; data: 40, next: None]] 4 LinkedList[length = 4, [data: 100, next: 20; data: 20, next: 3; data: 3, next: 40; data: 40, next: None]] </pre>	Верно
----	--	--	-------

## **Выводы**

Были изучены алгоритмы и структуры данных в Python и способы их реализации. Согласно заданию была разработана программа, которая реализует однонаправленный связный список, представляющий собой класс с соответствующими полями и методами.

## 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 != None:
            return f"data: {self.__data}, next: {self.next.get_data()}"
        return f"data: {self.__data}, next: {self.next}"

class LinkedList:
    def __init__(self, head = None):
        if head is None:
            self.__head__ = None
            self.__length = 0
        else:
            self.__head__ = Node(head)
            self.__length = 1

    def __len__(self):
        return self.__length

    def append(self, element):
        self.__length += 1
        if self.__head__ == None:
            self.__head__ = Node(element)
        else:
```

```

        tmp = self.__head__
        while tmp.next!= None:
            tmp = tmp.next
        tmp.next = Node(element)

def __str__(self):
    if self.__length == 0:
        return "LinkedList[]"
    else:
        tmp = self.__head__
        string = tmp.__str__()
        while tmp.next != None:
            tmp = tmp.next
            string += "; " + tmp.__str__()
        return f'LinkedList[length = {self.__length}, [{string}]]'

def pop(self):
    if self.__length == 0:
        raise IndexError("LinkedList is empty!")
    tmp = self.__head__
    if tmp.next != None:
        while tmp.next.next != None:
            tmp = tmp.next
        tmp.next= None
    else:
        self.__head__ = None
    self.__length -= 1

def change_on_end(self, n, new_data):
    if self.__length < n or n <= 0:
        raise KeyError("<element> doesn't exist!")
    else:
        num = self.__length - n
        if num == 0:
            self.__head__.change_data(new_data)
        else:
            tmp = self.__head__
            for i in range(num):
                tmp = tmp.next

```

```
tmp.change_data(new_data)

def clear(self):
    self.__length = 0
    self.__head__ = None
```