

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

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

Студентка гр. 2384

Валеева А.А.

Преподаватель

Иванов Д.В.

Санкт-Петербург

2023

## **Цель работы.**

Целью данной лабораторной работы является создание структур данных на примере связного списка в Python.

## **Задание.**

Вариант 2.

**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**.

**\_\_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, next: None

node.__next__ = Node(2, None)

print(node) # data: 1, next: 2
```

### **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.

### **`delete_on_start(self, n)`**

удаление `n`-того элемента с НАЧАЛА списка. 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

```

### **Выполнение работы.**

Первый класс — класс *Node*.

Описание конструктора: заполняем приватное поле `__data` значением переданной переменной *data*, а в метод `__next__` передаем переменную *next*.

Метод `get_data()`: возвращает значение поля `__data`. Сложность  $O(1)$ .

Метод `__str__()`: если не последний элемент, то выводим результат в виде: значение текущего элемента, значение следующего элемента. Если элемент последний, то вид вывода: значение текущего элемента, *None*. Сложность  $O(1)$ .

Следующий класс — класс *LinkedList*.

Описание конструктора: заполняем поле `__head__` значением переданной переменной *head*, если значение переменной *None*, то длина списка = 0 (`self.__length`), иначе 1.

Метод `__len__()`: возвращает значение переменной *length*. Сложность  $O(1)$ .

Метод *append*: увеличиваем длину списка на 1, если список пустой (значение поля `__head__` *None*), тогда поле `__head__` заполняется значением переменной *element*. Иначе мы заводим дополнительную переменную *tmp*, с помощью которой мы будем проходить по элементам списка, вначале она равна значению *head*. Пока элемент не последний (*tmp.\_\_next\_\_ is not None*), текущий элемент равен следующему. Создаем элемент класса *Node* и добавляем его в конец списка через `__next__()`. Сложность  $O(n)$ , где *n* — длина списка.

Метод `__str__()`: Для вывода информации о всех элементах списка проходим по списку, информация о каждом элементе записывается в массив, после чего при помощи форматной строки формируется результат и возвращается строка. Сложность  $O(n)$ , где *n* — длина списка.

Метод *pop()*: если первый элемент *None*, тогда «выбрасываем» ошибку, уменьшаем длину на 1 (так как убираем один элемент), если изначально он и был один (то есть длина после уменьшения стала равна 0), тогда начальный элемент *head* приравниваем *None*. Иначе, проверяем не будет ли следующий элемент *None*, доходя до предпоследнего элемента указатель на последний делаем *None*. Сложность  $O(n)$ , где *n* — длина списка.

Метод *delete\_on\_start()*: сначала проверяем, не меньше ли длина нашего списка, чем номер удаляемого элемента и не меньше ли номер, чем 1. Если одно из условий выполняется, в таком случае «выбрасываем ошибку». Если номер равен 1, то есть удалить нужно первый элемент, в таком случае значение переменной *head* = *None*. Иначе идем по элементам списка, пока следующий элемент не будет удаляемым (проверяем через переменную *count*, изначально равную 1 из-за индексации), заменяем указатель на следующий элемент на указатель на следующий. Уменьшаем длину на 1. Сложность  $O(n)$ , где *n* — длина списка, так как может быть случай с удалением последнего элемента.

Метод *clear()*: обнуляем длину, *head* = *None*. Сложность  $O(1)$ .

Ответы на вопросы:

1) Связный список — динамическая структура данных, которая состоит из узлов, каждый узел хранит некоторые данные и указатель на следующий узел (в случае двухсвязного списка ещё и на предыдущую).

Основным отличием списка от массива является непоследовательное хранение данных (т.е. нельзя обратиться к элементу по индексу), расходуется дополнительная память для хранения указателей на следующий(предыдущие) элемент(ы).

2) Сложности методов указаны в тексте выполнения работы.

3) Возможная реализация бинарного поиска в связном списке: тот же алгоритм, как и в обычном — сравнение середины с искомым элементом, изменение границ поиска. Но бинарный поиск в односвязном списке неэффективен, т.к. нельзя обращаться к элементу по индексу. Поэтому даже при отсортированном списке сложность поиска будет  $O(n * \log(n))$ , т.к. доступ к элементу осуществляется за  $O(n)$ . Поэтому вместо сортировки и бинарного поиска следует использовать обычный линейный поиск, тогда сложность алгоритма будет  $O(n)$ , где  $n$  - это длина связного списка.

Отличием от стандартного списка Python является то, то в стандартном списке алгоритм двоичного поиска более эффективен и имеет смысл применения, так как есть доступ к элементу по индексу за сложность  $O(1)$  и поиск займёт  $O(\log(n))$ .

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) print(node.get_data())</pre>	<pre>data: 1, next: None data: 1, next: 2 1</pre>	Проверка конструктора и др. методов класса Node.
2.	<pre>l_1= LinkedList() l_1.append(10) l_1.append(20) l_1.pop() print(l_1)</pre>	<pre>LinkedList[lengt h= 1, [data: 10, next: None]]</pre>	Создание класса экземпляра LinkedList и проверка методов удаления, добавления, взятия длины.
3.	<pre>try: l_1 = LinkedList() l_1.pop() except IndexError: print('ok')</pre>	<pre>ok</pre>	Проверка работы исключений.

## Выводы.

Были изучены односвязные списки, а также создание своих классов и перезагрузка методов в языке программирования Python.



## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Valeeva\_Alina\_lb2.py

```
class Node:
    def __init__(self, data, next = None):
        self.__data = data
        self.__next__ = next

    def get_data(self):
        return self.__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 None:
            self.__length = 0
        else:
            self.__length = 1

    def __len__(self):
        return self.__length

    def append(self, element):
        self.__length += 1
        if self.__head__ is None:
            self.__head__ = Node(element)
        else:
            tmp = self.__head__
            while tmp.__next__ is not None:
                tmp = tmp.__next__
            last_elem = Node(element)
            tmp.__next__ = last_elem

    def __str__(self):
        data = []
        tmp = self.__head__
        while tmp is not None:
```

```

        data.append(str(tmp))
        tmp = tmp.__next__
    return f"LinkedList[length = {self.__length}, [{'; '.join(data)}]]" if data
else "LinkedList[]"

```

```

def pop(self):
    if self.__head__ is None:
        raise IndexError("LinkedList is empty!")
    self.__length -= 1
    if self.__length == 0:
        self.__head__ = None
    else:
        temp = self.__head__
        while temp.__next__ is not None:
            temp = temp.__next__
        temp.__next__ = None

```

```

def delete_on_start(self, n):
    tmp = self.__head__

    if self.__length < n or n < 1:
        raise KeyError(f'{n} doesn't exist!')

    elif n == 1:
        self.__head__ = tmp.__next__
    else:
        count = 1
        while count + 1 < n:
            tmp = tmp.__next__
            count += 1
        tmp.__next__ = tmp.__next__.__next__
        self.__length -= 1

```

```

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

```