

**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**

Студентка гр. 3344

Якимова Ю.А.

Преподаватель

Иванов Д.В.

Санкт-Петербург

2024

## **Цель работы**

Введение в алгоритмы и структуры данных. Освоение алгоритмов и структур данных на языке Python.

### **Задание.**

Вариант 2. 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:

- 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 `__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

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` 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 `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`.

## Выполнение работы

Связный список – это структура данных, каждый элемент (узел) которой содержит собственные данные и ссылку на следующий элемент (и на предыдущий, если список двусвязный).

Основные отличия связного списка от массива.

Связный список:

1. Связные списки хранятся не в непрерывном расположении.
2. Динамический размер.
3. Память выделяется во время выполнения.
4. Использует больше памяти, поскольку в нем хранятся как данные, так и адрес следующего узла.
5. Для доступа к элементу требуется обход всего связанного списка.
6. Операции вставки и удаления обычно выполняются быстрее.

Массив:

1. Массивы хранятся в непрерывном расположении.
2. Фиксированный размер.
3. Память выделяется во время компиляции.
4. Использует меньше памяти, чем связанные списки.
5. К элементам можно легко получить доступ.
6. Операции вставки и удаления обычно требуют больше времени.

Сложности методов.

$O(1)$ :

```
__init__  
get_data  
Node.__str__  
__len__  
__clear__  
append(добавление head)
```

pop(если список пуст)  
delete\_on\_start(удаление первого элемента)

O(n):

LinkedList.\_\_str\_\_  
append  
pop  
delete\_on\_start

Для связного списка реализация бинарного поиска может выглядеть следующим образом:

- 1) Найдем средний элемент связного списка. Для его нахождения, начиная с головы списка, идем двумя переменными до его конца. Одна переменная за одну итерацию будет проходить вперед на 2 ссылке, а другая на одну. Тогда, когда первая переменная дойдет до конца, вторая будет в середине.
- 2) Сравним средний элемент с ключом.
- 3) Если ключ найден в среднем элементе, процесс завершается.
- 4) Если ключ не найден в среднем элементе, выберите, какая половина будет использоваться в качестве следующего пространства поиска.
- 5) Если ключ меньше среднего узла, то для следующего поиска используется левая сторона.
- 6) Если ключ больше среднего узла, то для следующего поиска используется правая сторона.

Этот процесс продолжается до тех пор, пока не будет найден ключ или не будет исчерпан весь связный список. Основное отличие реализации бинарного поиска для связного списка от реализации для классического списка заключается в том, что в связном списке поиск осуществляется путем перемещения указателей, а не прямого доступа к элементам по индексу. Это делает поиск более медленным, так как требуется пройти по всем элементам до нужного.

### Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	<code>n = Node(1)</code> <code>l = LinkedList(n)</code> <code>l.append(2)</code> <code>print(l)</code> <code>l.pop()</code> <code>print(l)</code>	<code>LinkedList = 2, [data: 1, next: 2; data: 2, next: None]</code> <code>LinkedList = 1, [data: 1, next: None]</code>	-
2.	<code>n = Node(1)</code> <code>l = LinkedList(n)</code> <code>l.append(2)</code> <code>l.append(3)</code> <code>print(l)</code> <code>l.delete_on_start(2)</code> <code>print(l)</code>	<code>LinkedList = 3, [data: 1, next: 2; data: 2, next: 3; data: 3, next: None]</code> <code>LinkedList = 2, [data: 1, next: 3; data: 3, next: None]</code>	-
3.	<code>n = Node(1)</code> <code>print(n)</code> <code>print(n.get_data())</code>	<code>data: 1, next: None</code> <code>1</code>	-



## **Выводы**

Были получены базовые знания об алгоритмах и структурах данных в Python.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Yakimova\_Yuliya\_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):
        return f"data: {self.get_data()}, next: {self.next.get_data() if self.next else None}"

class LinkedList:
    def __init__(self, head = None):
        self.head = head
        self.length = 1 if self.head else 0

    def __len__(self):
        return self.length

    def append(self, element):
        if not self.head:
            self.head = Node(element)
        else:
            cur_el = self.head
            while (cur_el.next != None):
                cur_el = cur_el.next
            cur_el.next = Node(element)
            self.length += 1

    def __str__(self):
```

```

        if not self.head:
            return "LinkedList[]"
        else:
            elements = [self.head]
            for i in range(self.length - 1):
                elements.append(elements[-1].next)
            return f"LinkedList[length = {self.length}, [{';
'.join(map(str,
            elements))}]]"

    def pop(self):
        if not self.head:
            raise IndexError("LinkedList is empty!")

        if not self.head.next:
            self.head = None
            self.length -= 1

        else:
            cur_el = self.head
            while(cur_el.next.next != None):
                cur_el = cur_el.next
            cur_el.next = None
            self.length -= 1

    def delete_on_start(self, n):
        if self.length < n or n < 1:
            raise KeyError("Element doesn't exist!")

        if n == 1:
            self.head = self.head.next
            self.length -= 1
            return

        index = 1
        cur_el = self.head
        while(cur_el.next and index <= n):
            if n == index + 1:
                cur_el.next = cur_el.next.next
                self.length -= 1

```

```

        break
    cur_el = cur_el.next
    index += 1

def clear(self):
    self.head = None
    self.length = 0

```