

**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. 3344

Вердин К.К.

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, которая состоит из узлов, где каждый узел содержит значение и ссылку на следующий узел в списке. Это позволяет хранить данные в линейной последовательности, где каждый элемент может быть расположен в памяти независимо от других элементов.

Отличия между связным списком и массивом:

1. Связный список не требует непрерывного блока памяти для хранения элементов, в отличие от массива, который требует непрерывной области памяти.
2. В связном списке доступ к элементам осуществляется последовательно, начиная с начала списка и переходя к следующему элементу по ссылке. В массиве доступ к элементам осуществляется по индексу, что позволяет быстро получить доступ к любому элементу.
3. Связный список более гибок в изменении размера, так как не требует копирования всех элементов при добавлении или удалении элементов. В массиве при добавлении или удалении элементов может потребоваться перераспределение памяти и копирование элементов.
4. Связный список может быть более эффективным при частых операциях вставки или удаления элементов в середине списка, так как не требуется сдвигать остальные элементы, как в массиве.

Сложности методов.

$O(n)$ :

```
LinkedList.__str__  
append  
pop  
delete_on_start
```

$O(1)$ :

```
__init__  
get_data  
Node.__str__
```

\_\_len\_\_

\_\_clear\_\_

append(если добавляем head)

pop(если список пуст)

delete\_on\_start(если удаляем первый элемент)

Для связного списка, реализация бинарного поиска может выглядеть следующим образом:

- 1) Найдем средний элемент связанного 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 связанный 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.	<pre>node = Node(1) print(node) node.next = Node(2, None) print(node)</pre>	<pre>data: 1, next: None data: 1, next: 2</pre>	-
2.	<pre>linked_list = LinkedList() print(linked_list) print(len(linked_list)) linked_list.append(10) print(linked_list) print(len(linked_list)) linked_list.append(20) print(linked_list) print(len(linked_list)) linked_list.pop() print(linked_list) print(linked_list) print(len(linked_list))</pre>	<pre>LinkedList[] 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</pre>	-



## **Выводы**

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

## 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 __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 self.head:
            current = self.head
            while current.next: current = current.next
            current.next = Node(element)
        else: self.head = Node(element)
        self.length += 1

    def __str__(self):
        if not self.head:
            return "LinkedList[]"
```

```

        current = self.head
        array = [self.head]
        for i in range(self.length - 1):
            current = current.next
            if current: array.append(current)
        return f"LinkedList[length = {self.length}, [{"';
''.join(map(str,array))}]]"

```

```

def pop(self):
    if not self.head:
        raise IndexError("LinkedList is empty!")
    elif not self.head.next:
        self.head = None
        self.length -=1
    else:
        current = self.head
        while current.next.next:
            current = current.next
        current.next = None
        self.length -= 1

```

```

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

```

```

def delete_on_start(self, n):
    if self.length < n or n<=0:
        raise KeyError("Element doesn't exist!")
        return
    elif n == 1:
        self.head = self.head.next
    else:
        current = self.head
        for i in range(n-2):
            current = current.next
        if not current.next.next: current.next = None
        else: current.next = current.next.next
    self.length-=1

```