**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 _____ Мурдасов М.К.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

**Цель работы**

Изучение алгоритмов и структур данных в языке Python.

**Задание**

Вариант 4

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).
- **change_data(self, new_data)** - метод меняет значение поля data объекта 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.

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:**

- **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.

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 **change_on_start(self, n, new_data)** - изменение поля data n-того элемента с НАЧАЛА списка на new_data. 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
```

***Вам не требуется реализовывать создание экземпляров ваших классов и вызов методов, это сделает проверяющая система.***

*В отчете вам требуется:*

1. Указать, что такое связный список. Основные отличия связного списка от массива.
2. Указать сложность каждого метода.
3. Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

**Выполнение работы**

Связный список - это линейная структура данных, в которой элементы (узлы) хранятся в отдельных областях памяти и связаны друг с другом указателями. Каждый узел содержит данные и ссылку на следующий (иногда и на предыдущий) узел в списке.

**Связный список**

- **Структура данных:** Линейная структура данных, состоящая из узлов, каждый из которых содержит данные и ссылку на следующий узел.
- **Доступ к элементам:** Доступ к элементам осуществляется путем последовательного перехода от одного узла к другому, следуя ссылкам.
- **Вставка и удаление:** Вставка и удаление элементов могут выполняться в любом месте списка, перенаправляя ссылки.
- **Память:** Связные списки не требуют непрерывного блока памяти, как массивы.
- **Производительность:** Доступ к элементам может быть медленным, особенно для больших списков, из-за необходимости последовательного перехода.
- **Использование:** Подходит для ситуаций, когда требуется частая вставка и удаление элементов в произвольных местах.

**Массив**

- **Структура данных:** Непрерывная область памяти, состоящая из элементов фиксированного типа данных.
- **Доступ к элементам:** Доступ к элементам осуществляется с помощью индексов, что обеспечивает быстрый прямой доступ.
- **Вставка и удаление:** Вставка и удаление элементов могут быть дорогостоящими операциями, поскольку они требуют сдвига других элементов.
- **Память:** Элементы массива хранятся в непрерывной области памяти, что обеспечивает эффективное использование памяти.

- **Производительность:** Доступ к элементам быстрый, особенно для больших массивов, благодаря прямому доступу.
- **Использование:** Подходит для ситуаций, когда требуется быстрый доступ к элементам по их индексам и когда частота вставок и удалений невелика.

**Сложности методов:**

- O(1):
  - __init__
  - get_data()
  - change_data()
  - Node.__str__
  - __len__
  - append() – при self.head == None
  - pop() - при self.head == None
  - change_on_start() – при self.length < n or n <= 0
  - __clear__

- O(n):
  - LinkedList.__str__
  - append()
  - pop()
  - change_on_start

**Реализация бинарного поиска в связном списке:**

1. Найти средний узел списка.
2. Если значение среднего узла равно искомому значению, вернуть узел.
3. Если значение среднего узла меньше искомого значения, перейти к правой половине списка.

4. Если значение среднего узла больше искомого значения, перейти к левой половине списка.
5. Повторять шаги 1-4, пока не будет найден узел с искомым значением или не будет достигнуто начало или конец списка.

**Отличия от бинарного поиска в классическом списке Python:**

- В связном списке нет прямого доступа к элементам по индексу, поэтому поиск среднего узла требует перебора списка.

**Тестирование**

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

| № п/п | Входные данные | Выходные данные | Комментарии |
|---|---|---|---|
| 1. | node = Node(1)<br>print(node)<br>node.next = Node(2, None)<br>print(node)<br>print(node.get_data())<br>l_l = LinkedList()<br>print(l_l)<br>print(len(l_l))<br>l_l.append(10)<br>l_l.append(20)<br>l_l.append(30)<br>l_l.append(40)<br>print(l_l)<br>print(len(l_l))<br>l_l.change_on_start(2, 2)<br>print(l_l) | data: 1, next: None<br>data: 1, next: 2<br>1<br>LinkedList[]<br>0<br>LinkedList[length = 4, [data: 10, next: 20; data: 20, next: 30; data: 30, next: 40; data: 40, next: None]]<br>4<br>LinkedList[length = 4, [data: 10, next: 2; data: 2, next: 30; data: 30, next: 40; data: 40, next: None]] | Корректно |

**Выводы**

Была написана программа на Python с применением алгоритмов и структур данных.

## ПРИЛОЖЕНИЕ А

## ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Murdasov_Mikhail_lb2.py

```
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.data}"
        return f"data: {self.data}, next: None"


class LinkedList:
    def __init__(self, head = None):
        if head == 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):
        if self.head == None:
            self.head = Node(element)
        else:
            node = self.head
            while node.next != None:
                node = node.next
            node.next = Node(element)
        self.length += 1


    def __str__(self):
        if self.head == None:
            return "LinkedList[]"
        res = f"LinkedList[length = {self.length}, ["
        node = self.head
        while node != None:
            if node.next == None:
                res += f"{str(node)}"
                break
```

```
            res += f"{str(node)}; "
            node = node.next
        res += "]]"
        return res

    def pop(self):
        if self.head == None:
            raise IndexError("LinkedList is empty!")
        elif self.head.next == None:
            self.head = None
            self.length = 0
        else:
            node = self.head
            while node.next.next != None:
                node = node.next
            node.next = None
            self.length -= 1


    def change_on_start(self, n, new_data):
        if  self.length < n or n <= 0:
            raise KeyError("Element doesn't exist!")
        else:
            node = self.head
            for i in range(n-1):
                node = node.next
            node.data = new_data

    def clear(self):
        self.head = None
```