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

# ОТЧЕТ

**по лабораторной работе №2**

**по дисциплине «Информатика»**

**Тема: Введение в алгоритмы и структуры данных**

Студент гр. 3344 _____ Охрименко Д.И.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

**Цель работы**

Целью работы создание структуры данных и методов работы с ней на языке Python.

**Задание**

Вариант 1. 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:

1) *data* # Данные элемента списка, приватное поле.
2) *next* # Ссылка на следующий элемент списка.

И следующие методы:

1) *__init__(self, data, next)* - конструктор, у которого значения по умолчанию для аргумента *next* равно *None*.
2) *get_data(self)* - метод возвращает значение поля *data* (это необходимо, потому что в идеале пользователь класса не должен трогать поля класса *Node*).
3) *__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*.

*Linked List:* класс, который описывает связный однонаправленный список. Он должен иметь 2 поля:

1) *head* # Данные первого элемента списка.
2) *length* # Количество элементов в списке.

И следующие методы:

1) *__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.

2) *__len__(self)* - перегрузка метода *__len__*, он должен возвращать длину списка (этот стандартный метод, например, используется в функции *len*).
3) *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.
4) *__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:
    a. Если список пустой, то строковое представление: *“LinkedList[]”*
    b. 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]”,*
5) *pop(self)* - удаление последнего элемента. Метод должен выбрасывать исключение *IndexError* с сообщением *"LinkedList is empty!"*, если список пустой.
6) *clear(self)* - очищение списка.
7) *delete_on_end(self, n)* - удаление n-того элемента с конца списка. Метод должен выбрасывать исключение *KeyError*, с сообщением *"Element doesn't exist!"*, если количество элементов меньше n.

**Выполнение работы**

Связный список — это структура данных, состоящая из узлов, каждый из которых содержит данные и ссылку на следующий узел в списке.

Основные отличия связного списка от массива:

Размер: размер массива фиксирован и определяется при создании, а связный список может динамически расти или уменьшаться во время выполнения программы.

Хранение в памяти: в массивах элементы хранятся в последовательных блоках памяти, а в связном списке каждый узел может находиться в любом месте памяти, и его связь с другими узлами обеспечивается указателями.

Доступ к элементам: в массивах доступ к элементам осуществляется по индексу за константное время O(1), а в связных списках — путём последовательного прохода от начала до нужного элемента.

Вставка и удаление: в массивах вставка и удаление элементов в середину списка требуют сдвига всех элементов после изменяемого, что может быть дорогостоящим. В связных списках эти операции эффективны, так как требуют изменения только указателей на узлы.

Сложности методов:

O(1): __init__, get_data, Node. __str__, __len__, __clear__, append (если добавляем голову), pop (если список пуст), delete_on_end (если удаляем голову).

O(n): LinkedList.__str__, append, pop, delete_on_end.

Реализация бинарного поиска для связного списка:

Для нахождения середины связного списка используются два указателя: один движется по списку на одну позицию за каждую итерацию, а другой — на две позиции. Когда быстрый указатель достигает конца списка, медленный указывает на середину.

На каждом шаге бинарного поиска находится средний элемент списка. Если он равен искомому значению, возвращаем его.

Если ключ не совпадает со средним элементом, выбираем, какую половину списка использовать для следующего поиска: 1) если ключ меньше среднего узла,

для следующего поиска используем левую часть списка; 2) если ключ больше среднего узла, для следующего поиска используем правую часть списка.

Продолжаем делить интервал поиска, пока не найдём искомый элемент или не исследуем весь список.

Основное отличие реализации алгоритма бинарного поиска для связного списка заключается в необходимости использования алгоритма для нахождения среднего элемента, так как прямого доступа по индексу нет. Это делает алгоритм менее эффективным.

Разработанный программный код см. в приложении А.

**Тестирование**

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

| № п/п | Тест | Выходные данные | Комментарии |
|---|---|---|---|
| 1. | `a = LinkedList()`<br>`a.append(4)`<br>`a.append(2)`<br>`a.append(1)`<br>`a.pop()`<br>`print(a)` | `LinkedList[length = 2, [data: 4, next: 2; data: 2, next: None]]` | Верный результат |
| 2. | `a = LinkedList()`<br>`a.append(4)`<br>`a.append(2)`<br>`a.append(1)`<br>`a.delete_on_end(1)`<br>`a.delete_on_end(2)`<br>`print(a)`<br>`a.clear()`<br>`print(a)` | `LinkedList[length = 1, [data: 2, next: None]]`<br>`LinkedList[]` | Верный результат |

**Выводы**

Написана программа реализующая однонаправленный список. Получены навыки работы с ним, изучена сложность O(n) для алгоритмов работы со списком.

## ПРИЛОЖЕНИЕ А

## ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lb2.py

```
class Node:
    data = None
    next = None

    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.data}, next: {None if not self.next else
self.next.data}"


class LinkedList:
    head = None
    length = 0

    def __init__(self, head=None):
        self.head = Node(head)

    def __len__(self):
        i = 0
        node = self.head
        if(not node or node.data == None):
            return i
        while(node.next):
            i += 1
            node = node.next
        return i + 1

    def append(self, element):
        node = self.head
        if(not node or node.data == None):
            if(not node):
                self.__init__(element)
                return element
            node.data = element
            self.length += 1
            return element

        while(node.next):
            node = node.next
        node.next = Node(element)
        self.length += 1
        return element

    def __str__(self):
        line = f"LinkedList[length = {len(self)}, ["
        node = self.head
```

```
        if(not node or node.data == None):
            return "LinkedList[]"
        while(node.next):
            line += f"data: {node.data}, next: {node.next.data}; "
            node = node.next
        line += f"data: {node.data}, next: {node.next}]]"
        return line

    def pop(self):
        node = self.head
        prev_node = node
        if(not node or node.data == None):
            raise IndexError("LinkedList is empty!")
        while(node.next):
            prev_node = node
            node = node.next
        if prev_node.next == None:
            self.head = None
        prev_node.next = None
        return node.data

    def delete_on_end(self, n):
        node = self.head
        prev_node = node
        if(len(self) < n or n <= 0):
            raise KeyError("Element doesn't exist!")
        ind = len(self) - n
        for i in range(ind):
            prev_node = node
            node = node.next
        if prev_node == node:
            self.head = self.head.next
            return node.data
        prev_node.next = node.next
        return node.data

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