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: Алгоритмы и структуры данных в Python. Вариант 1

Студент гр. 3343 _____ Гребнев Е. Д.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург

2024

Цель работы

Научиться создавать класс односвязного списка на языке программирования Python, описывать элемент списка, методы для изменения элементов в односвязном списке.

Задание

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).
- **__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:

- **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.
- **__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.
- **pop(self)** - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.
- **clear(self)** - очищение списка.
- **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 представляет собой структуру данных, состоящую из узлов, каждый из которых содержит данные и ссылку на следующий узел в списке. Обычно последний узел указывает на NULL или None, что сигнализирует о конце списка. Основные отличия связного списка от массива заключаются в его динамической природе: связный список позволяет добавлять или удалять элементы без перекопирования всех элементов, в отличие от массива, который требует переопределения при изменении размера. Кроме того, вставка и удаление элементов в середине связного списка более эффективны, так как не требуется сдвиг всех последующих элементов, что является затратным по ресурсам для массивов.

Методы связного списка имеют следующую сложность:

__init__, get_data, __str__: O(1)
append: O(n)
pop: O(n)
delete_on_end: O(n)
clear: O(1)

Реализация бинарного поиска в связном списке требует определения границ искомого элемента (start и end), а также переменных для определения среднего элемента (mid) и текущего элемента (current). Пока начальная граница не превышает конечную, средний элемент определяется и сравнивается с искомым. Если они совпадают, возвращается значение. В противном случае границы сужаются в зависимости от того, больше или меньше искомый элемент среднего. Если элемент не найден, функция возвращает -1.

Отличие реализации алгоритма бинарного поиска для связного списка от классического списка Python заключается в необходимости дополнительных переменных для определения среднего элемента и текущего положения в списке.

В классическом списке Python достаточно обратиться по индексу к элементу, в то время как в связном списке необходимо пройти по всем элементам до нахождения среднего.

Выводы

Была написана программа, которая содержит в себе реализацию односвязного линейного списка. В ней можно создавать список, добавлять и удалять элементы в различных позициях, а также выводить информацию о каждом элементе списка.

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.__data}, next: {self.next.get_data() if
self.next else None}"


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

    def __len__(self):
        return self.length

    def append(self, element):
        node = Node(element)
        if not self.head:
            self.head = node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = node
        self.length += 1

    def __str__(self):
        if not self.head:
            return "LinkedList[]"
        current = self.head
        node_strings = []
        while current:
            node_strings.append(f"data: {current.get_data()}, next:
{current.next.get_data() if current.next else None}")
            current = current.next
        return f"LinkedList[length = {len(node_strings)}, [{';
'.join(node_strings)}]]"

    def pop(self):
        if not self.head:
            raise IndexError("LinkedList is empty!")
        elif not self.head.next:
            data = self.head.get_data()
            self.head = None
```

```
            self.length = 0
            return data
        else:
            current = self.head
            while current.next.next:
                current = current.next
            data = current.next.get_data()
            current.next = None
            self.length -= 1
            return data

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

    def delete_on_end(self, n):
        if not self.head:
            raise KeyError("Element doesn't exist!")

        length = 0
        current = self.head
        while current:
            length += 1
            current = current.next

        if n > length or n < 1:
            raise KeyError("Element doesn't exist!")

        position = length - n

        if position == 0:
            self.head = self.head.next
            self.length -= 1
            return
        current = self.head
        for _ in range(position - 1):
            current = current.next

        current.next = current.next.next
        self.length -= 1
```