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

Студент гр. 3342 _____ Роднов И.С.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург
2024

Цель работы

Реализация линейного списка на языке программирования Python.

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.

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_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

Были созданы классы Node и LinkedList, а так же реализованы все методы по условию задачи.

Связный список — это структура данных, представляющая собой последовательность элементов, в которой каждый элемент хранит ссылку на следующий элемент в последовательности. Каждый элемент списка называется узлом. Узлы могут содержать как данные (например, числа или объекты), так и ссылки на следующие узлы.

Основные отличия связного списка от массива:

1. **Доступ к элементам:** В массиве доступ к элементам осуществляется по индексу за константное время (O(1)), тогда как в связном списке время доступа к элементам зависит от их позиции в списке и может быть линейным (O(n)) в худшем случае.
2. **Хранение данных:** В массиве элементы хранятся в последовательной области памяти, в то время как в связном списке каждый элемент хранится отдельно, и ссылки соединяют их в список.
3. **Вставка и удаление элементов:** В связном списке вставка и удаление элементов могут быть выполнены за константное время (O(1)), даже в начале и середине списка, тогда как в массиве эти операции могут потребовать сдвиг всех последующих элементов, что может быть дорого с точки зрения времени (O(n)).

Сложность каждого метода в реализованном коде:

1. **__init__**: O(1) - создание объекта LinkedList или Node.
2. **append**: O(n) - добавление элемента в конец списка.
3. **__len__**: O(n) - вычисление длины списка.
4. **__str__**: O(n) - создание строкового представления списка.
5. **delete_on_end**: O(n) - удаление элемента по его позиции в конце списка.
6. **pop**: O(n) - удаление последнего элемента списка.
7. **clear**: O(1) - очистка списка.

В классическом списке Python бинарный поиск более эффективен благодаря быстрому доступу к элементам по индексу за O(1), в то время как в связном списке для доступа к элементам требуется O(n). Реализация бинарного поиска в связном списке менее эффективна из-за необходимости итерации по всему списку для доступа к элементам, что приводит к сложности O(n log n).. Бинарный поиск в связном списке будет осуществляться с использованием указателей на начало и конец текущего диапазона, итеративно сокращая этот диапазон вдвое, пока не будет найден искомый элемент или не определится его отсутствие.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	node = Node(1) print(node) # data: 1, next: None node.next = Node(2, None) print(node) # data: 1, next: 2 print(node.get_data()) l_l = LinkedList() print(l_l) print(len(l_l)) l_l.append(10) l_l.append(20) l_l.append(30) l_l.append(40) print(l_l) print(len(l_l)) l_l.delete_on_end(3) print(l_l)	data: 1, next: None data: 1, next: 2 1 LinkedList[] 0 LinkedList[length = 4, [data: 10, next: 20; data: 20, next: 30; data: 30, next: 40; data: 40, next: None]] 4 LinkedList[length = 3, [data: 10, next: 30; data: 30, next: 40; data: 40, next: None]]	Верный вывод

Выводы

Реализован связный список при помощи ООП на языке Python. Исследована скорость работы методов созданного класса и возможность использования бинарного поиска в связном списке.

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):
        next_data = self.next.get_data() if self.next else None
        return f"data: {self.__data}, next: {next_data}"


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

    def __len__(self):
        c = 0
        curr = self.head
        while curr:
            c += 1
            curr = curr.next
        return c

    def append(self, element):
        new_node = Node(element)
        if self.head is None:
            self.head = new_node
        else:
            curr = self.head
            while curr.next:
                curr = curr.next
            curr.next = new_node
        self.length += 1

    def __str__(self):
        if self.head is None:
            return "LinkedList[]"
        else:
            current = self.head
            elements = []
            while current is not None:
                next_data = current.next.get_data() if
current.next else None
                elements.append(f"data: {current.get_data()},
next: {next_data}")
```

```
                    current = current.next
                elements_str = "; ".join(elements)
                return     f"LinkedList[length     =     {len(self)},
[{elements_str}]]"

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

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

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