

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

Колесниченко М. А.

Преподаватель

Иванов Д. В.

Санкт-Петербург

2024

Цель работы

Изучение работы с линейным списком и его реализации на языке 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:

- о **data** # Данные элемента списка, приватное поле.
- о **next** # Ссылка на следующий элемент списка.

И следующие методы:

- о **__init__(self, data, next)** - конструктор, у которого значения по умолчанию для аргумента next равно None.

- о **get_data(self)** - метод возвращает значение поля data (это необходимо, потому что *в идеале* пользователь класса не должен трогать поля класса Node).

- о **__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:

- о **head** # Данные первого элемента списка.
- о **length** # Количество элементов в списке.

И следующие методы:

- о **__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.

- о **__len__(self)** - перегрузка метода __len__, он должен возвращать длину списка (этот стандартный метод, например, используется в функции len).

- о **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.

- о **__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.

- о **pop(self)** - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.

- о **clear(self)** - очищение списка.

- о **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. Реализованы методы согласно условию.

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

Отличие реализации бинарного поиска для связного списка от классического списка Python заключается в том, что для связного списка нет прямого доступа к элементам по индексу, что делает невозможным применение такой же стратегии деления списка на две части, как в случае массива. Вместо этого 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.	<pre>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_1 = LinkedList() print(l_1) print(len(l_1)) l_1.append(10) l_1.append(20) l_1.append(30) l_1.append(40) print(l_1) print(len(l_1)) l_1.delete_on_end(3) print(l_1)</pre>	<pre>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]]</pre>	Верный вывод

Выводы

Был реализован связный список с помощью классов 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):
        if self.next is not None:
            return f"data: {self.get_data()}, next: {self.next.get_data()}"
        else:
            return f"data: {self.get_data()}, next: None"

class LinkedList:
    def __init__(self, head=None):
        self.head = head
        self.length = len(self)

    def __len__(self):
        c = 0
        temp = self.head
        while temp != None:
            temp = temp.next
            c += 1
        return c

    def append(self, element):
        x = Node(element)
        if self.head == None:
            self.head = x
        else:
            temp = self.head
            while temp.next != None:
                temp = temp.next
            temp.next = x

    def __str__(self):
        if len(self) == 0:
            return "LinkedList[]"
        else:
            temp = self.head
            arr = [str(temp)]
            while temp.next != None:
                temp = temp.next
                arr += [str(temp)]
            arr = "[" + '; '.join(arr) + "]"
            return f"LinkedList[length = {len(self)}, {arr}]"
```

```

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

def delete_on_end(self, n):
    if n <= 0 or n > len(self):
        raise KeyError("Element doesn't exist!")
    elif n == len(self):
        self.head = self.head.next
    else:
        temp = self.head
        for _ in range(len(self) - n - 1):
            temp = temp.next
        temp.next = temp.next.next

def clear(self):
    self.head = None

```