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. 3343 _____ Силяев Р.А.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Изучить особенности работы с односвязным списком на языке Python. Написать программу и сравнить список и массив.

Задание

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.

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.

Пример того, как должен выглядеть результат реализации см. ниже.

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

- **clear(self)** - очищение списка.

- **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.

Выполнение работы

LinkedList или связный список – это структура данных. Связный список обеспечивает возможность создать двунаправленную очередь из каких-либо элементов. Каждый элемент такого списка считается узлом. По факту в узле есть его значение, а также две ссылки – на предыдущий и на последующий узлы. То есть список «связывается» узлами, которые помогают двигаться вверх или вниз по списку. Из-за таких особенностей строения из связного списка можно организовать стек, очередь или двойную очередь.

Сложности методов:

Класс Node:

1. __init__ – O(1);
2. get_data – O(1);
3. change_data – O(1);
4. __str__ – O(1).

Класс LinkedList:

1. __init__ – O(n);
2. __len__ – O(1);
3. append – O(n);
4. __str__ – O(n);
5. pop – O(n);
6. clear – O(1);
7. change_on_start – O(n).

Для бинарного поиска требуется log(n) раз получать элементы каждый такой запрос займет O(n). В связном списке же за O(n) уже будет найден элемент.

Тестирование

Результаты тестирования содержатся в таблице 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	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	0 LinkedList[length = 1, [data: 10, next: None]] 1 LinkedList[length = 2, [data: 10, next: 20; data: 20, next: None]] 2 LinkedList[length = 1, [data: 10, next: None]] LinkedList[length = 1, [data: 10, next: None]] 1	□

Выводы

В ходе выполнения лабораторной работы были изучены особенности работы с односвязным списком на языке Python. Также была написана программа, изучены сложности списка и массива.

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 change_data(self, new_data):
        self.__data = new_data

    def __str__(self):
        return f'data: {self.__data}, next: {None if self.next is None else self.next.get_data()}'

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

    def __len__(self):
        return self.length

    def append(self, element):
        temp = Node(element)
        if self.length == 0:
            self.length = 1
            self.head = temp
            return
        cur = self.head
        while cur.next:
            cur = cur.next
```

```
            self.length += 1
            cur.next = temp


    def __str__(self):
        temp = self.head
        if self.head is None:
            return 'LinkedList[]'
        else:
            value = []
            while temp != None:
                value.append(temp.__str__())
                temp = temp.next
            return    f'LinkedList[length    =    {len(self)},    [{";
".join(value)}]]'

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

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

```
            prev = temp
            temp = temp.next
        cur.next = temp.next
        prev.next = cur

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