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

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Информационные технологии»
Тема: Алгоритмы и структуры данных в Python

Студент гр. 3343 _____ Пивоев Н. М.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург
2024

Цель работы

Ознакомиться с односвязным списком на языке Python и написать программу с его реализацией. Провести сравнение списка и массива, найти сходства и отличия.

Задание

Вариант 4.

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

Программа состоит из двух классов.

Первый – *class Node*, который является одним из элементов связанного списка. У него есть следующие методы:

- *__init__* - конструктор класса, заполняющий значение элемента и устанавливающий связь со следующим узлом.
- *get_data* – возвращает значение поля *__data*.
- *change_data* – обновляет поле *__data* в соответствием с полученным аргументом.
- *__str__* - возвращает информацию о узле списка.

Второй – *class LinkedList()* – связанный список. У него есть следующие методы:

- *__init__* - конструктор класса, создающий пустой список или состоящий из одного элемента.
- __len__ - возвращает длину списка.
- *append* – добавляет узел в список.
- *__str__* - возвращает информацию о списке.
- *pop* – удаляет последний элемент из списка, если он есть.
- *clear* – очищает список.
- *change_on_start* – изменяет n элемент, начиная отчёт со старта списка, если это возможно.

Связный список — структура данных, состоящая из узлов, содержащих данные и ссылки на следующий элемент списка. В памяти элементы хранятся не последовательно, как в массиве, а в разных участках памяти благодаря ссылочной связи между элементами. К преимуществам связанного списка можно отнести быстрое добавление и удаление в любой части списка, но доступ к элементам занимает много времени.

В массиве доступ к элементам осуществляется по индексу и можно получить нужную ячейку быстро, но при добавлении или удалении, придётся перемещать элементы после изменяемого, что довольно неэффективно и трудоёмко.

Сложности методов по времени:

Node:

- *__init__* – O(1);
- *get_data* – O(1);
- *change_data* – O(1);
- *__str__* – O(1);

LinkedList:

- *__init__* – O(1);
- *__len__* – O(1);
- *append* – O(n);
- *__str__* – O(n);
- *pop* – O(n);
- *clear* – O(1);
- *change_on_start* – O(n);

Бинарный поиск – алгоритм поиска значения среди отсортированных элементов. Ставится две границы и на каждой итерации граница сдвигается к середине, поэтому он имеет временную сложность O(log(n)). Для связанного списка очевидно бинарный поиск работать не будет, потому что его элементы не отсортированы. Можно преобразовать список в массив и отсортировать, но это займёт O(nlog(n)), что значительно сложнее, чем просто найти нужный элемент в списке.

Тестирование

Результаты тестирования содержатся в таблице 1.

Таблица 1.

№	Входные данные	Выходные данные	Комментарии
1.	linked_list = LinkedList() print(linked_list) linked_list.append(15) print(linked_list) linked_list.append(40) print(linked_list) linked_list.pop() print(linked_list)	LinkedList[] LinkedList[length = 1, [data: 15, next: None]] LinkedList[length = 2, [data: 15, next: 40; data: 40, next: None]] LinkedList[length = 1, [data: 15, next: None]]	Вывод соответствует ожиданиям.

Выводы

Таким образом, в ходе выполнения лабораторной работы были изучены особенности односвязного списка на языке Python и написана программа с его реализацией. Выявлены преимущества и недостатки связного 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):
        pt = Node(element)
        if self.length == 0:
            self.length = 1
            self.head = pt
            return
```

```
        current = self.head
        while current.next:
            current = current.next

        self.length += 1
        current.next = pt

    def __str__(self):
        pt = self.head
        if self.head is None:
            return 'LinkedList[]'

        else:
            value = []
            while pt != None:
                value.append(f'data: {pt.get_data()}, next: {None if
pt.next is None else pt.next.get_data()}')
                pt = pt.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:
            pt = self.head
            while pt.next.next:
                pt = pt.next

            self.length -= 1
            pt.next = None

    def clear(self):
        self.length = 0
```

```
        self.head = None

    def change_on_start(self, n, new_data):
        pt = self.head
        previous = pt
        current = Node(new_data)

        if n <= 0 or self.length < n:
            raise KeyError("Element doesn't exist!")

        if n == 1:
            current.next = self.head.next
            self.head = current

        else:
            for i in range(n - 1):
                previous = pt
                pt = pt.next

            current.next = pt.next
            previous.next = current
```