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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Информатика»

Тема: Алгоритмы и структуры данных в Python

Студент гр. 3341 Гребенюк В.А.

Преподаватель Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Целью работы является освоение работы с алгоритмами и структурами данных в Python. И разработка кода с их применением.

Задание

Вариант 4

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).
- **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.

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

Пример того, как должен выглядеть вывод объекта:

```
node = Node(1)
print(node) # data: 1, next: None
node.next = Node(2, None)
print(node) # data: 1, next: 2
```

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.

Пример того, как должно выглядеть взаимодействие с Вашим связным списком:

```
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
```

Вам не требуется реализовывать создание экземпляров ваших классов и вызов методов, это сделает проверяющая система.

В отчете вам требуется:

1. Указать, что такое связный список. Основные отличия связного списка от массива.
2. Указать сложность каждого метода.
3. Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

Выполнение работы

Создан код в соответствии с заданием.

1. Связный список — это структура данных, состоящая из узлов, каждый из которых содержит данные и ссылку на следующий элемент. Основное отличие связного списка от массива заключается в том, что элементы связного списка не располагаются в памяти последовательно, что позволяет более эффективно добавлять и удалять элементы, не требуя перемещения остальных элементов.

2. Класс *Node*:

__init__ - O(1)

get_data - O(1)

change_data - O(1)

__str__ - O(1)

Класс LinkedList

__init__ - O(1)

__len__ - O(n) // TODO: возможно сократить до O(1) при сохранении длинны в переменную. Нужно больше информации

append - O(n)

__str__ - O(n)

pop - O(n)

change_on_start - O(n)

clear - O(1)

Разработанный программный код см. в приложении А.

Выводы

В ходе работы были рассмотрены основные концепции связного списка, его отличия от массива, а также сложность различных методов, применяемых в классе *LinkedList*. Было установлено, что бинарный поиск в связном списке не является эффективным из-за отсутствия прямого доступа к элементам, что является существенным недостатком по сравнению с бинарным поиском в классических списках *Python*.

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


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

    def __len__(self):
        _len = 0
        cursor = self.head
        while cursor:
            _len += 1
            cursor = cursor.next
        return _len

    def append(self, element):
        if self.head is None:
            self.head = Node(element)
        else:
            cursor = self.head
            while cursor.next:
                cursor = cursor.next
            cursor.next = Node(element)

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

    def pop(self):
        if not self.head:
            raise IndexError("LinkedList is empty!")
```

```
                # should be:
                # raise IndexError("pop from empty LinkedList")
            cursor = self.head
            if not cursor.next:
                self.head = None
            else:
                while cursor.next.next:
                    cursor = cursor.next
                cursor.next = None

        def change_on_start(self, n, new_data):
            if n > len(self) or n <= 0:
                raise KeyError("Element doesn't exist!")
                # should be:
                # raise IndexError("LinkedList assignment index out of
range")
            cursor = self.head
            for _ in range(n - 1):
                cursor = cursor.next
            cursor.change_data(new_data)

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