

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. 3384

Рудаков А.Л.

Преподаватель

Иванов Д.В.

Санкт-Петербург

2024

Цель работы.

Изучить теорию об алгоритмах и структурах в Python. Применить знания на практике реализовав программный код 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 **change_data(self, new_data)** - метод меняет значение поля data объекта 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.

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:

- 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 связанного 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 **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.

Выполнение работы.

В начале работы был описан класс *Node*, в котором описаны элементы связного списка. В конструкторе описаны 2 поля: *__data* — приветное поле данных, и *next* — ссылка на следующий элемент. Далее идет метод *get_data()*, который возвращает *__data*. После него метод *change_data()*, в котором изменяется поле *__data*. Далее идет метод *__str__()*, в котором возвращается строка формата “*data: <node_data>, next: <node_next>*”.

Далее был реализован класс *LinkedList*, который описывает связный список. Конструктор инициализирует поле *head*, которое равно переданному значению или пустому списку, и поле *length()*, которое хранит длину списка, данное поле инициализируется при помощи метода *len()*. Метод *__len__()* возвращает длину списка. Принцип его работы состоит в том, что он проходит по всем элементам списка до момента, пока ссылка на следующий элемент не будет равна *None*, попутно считая количество. Далее идет метод *append()*, в котором новый элемент добавляется в конец списка. В цикле происходит итерация по всему списку до момента, пока ссылка на следующий элемент не равна *None*. После чего ссылка на последний элемент меняется на элемент, который нужно добавить. Обновляется длина списка. Далее идет метод *__str__()*, в котором происходит возврат строки в формате “*LinkedList[]*”, если список пустой и “*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]*” в противном случае. Далее реализован метод *pop()*, который удаляет последний элемент списка. Если длина списка равна 0, то при помощи *raise* выкидывается ошибка *IndexError("LinkedList is empty!")*, если длина равна 1, то ссылка на первый элемент изменяется на *None*, обновляется длина списка. Если длина больше 1, то идет итерация по списку до предпоследнего элемента, его поле *next* становится равным *None*, после чего

обновляется длина списка. Далее написан метод *change_on_start()*, который меняет поле *__data* *n*-ого элемента. Если длина списка меньше чем переданное число *n*, то выкидывается ошибка *KeyError("Element doesn't exist!")* при помощи *raise*, в противном случае происходит итерация по списку до нужного элемента, после чего вызывается метод *change_data()* класса *Node*. Далее написан метод *clear()*, в котором происходит очистка списка. В цикле происходит итерация по списку до момента, пока не встретится элемент с ссылкой на *None* (последний элемент). При этом ссылки на предыдущие элементы становятся равны 0. Длина равна 0.

Подводя итог:

Связный список — это структура данных, которая состоит из элементов, называемых узлами. В узлах хранятся данные, а между собой узлы соединены связями, которые являются ссылками на следующий или предыдущий элемент списка.

Связные списки используются для динамического доступа и хранения произвольного количества данных.

Основное отличие связных списков от массивов заключается в следующем:

- ⑩ Массивы хранятся в непрерывном блоке памяти, в то время как для узлов списка порядок расположения их в памяти не обязан совпадать с текущим внутренним.
- ⑩ Благодаря гибкости ссылочных связей скорость добавления или удаления элементов в связном списке выше, чем в динамическом массиве.
- ⑩ Из-за непоследовательного расположения узлов в памяти усложняется прямой доступ к элементу и определение его фактического адреса по индексу.

Сложность написанных методов:

- ⑩ $__len__() = O(n)$
- ⑩ $append() = O(n)$
- ⑩ $pop() = O(n)$
- ⑩ $change_on_start() = O(n)$

⑩ $clear = O(n)$

Возможный алгоритм бинарного поиска в связном списке:

Создать функцию которой поступает ссылка на начальный и последний элемент, посчитать количество элементов, вычислить среднее и вернуть ссылку на средний элемент.

Передавать ссылку на средний элемент, смотреть его значение, если оно больше, то текущему значению начального элемента передавать среднее, если оно меньше, то последнему. Передавать эти значения в предыдущую функцию, пока не будет найден ответ или начальный и конечный элемент не станут равными.

Разработанный программный код см. в приложении А.

Тестирование см. в приложении Б.

Выводы.

Была изучена теория об алгоритмах и структурах в Python. Был создан программный код внутри которого был создан класс Node — элементы списка и класс LinkedList — связный список. Внутри них были созданы требуемые по заданию методы.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main_lb2.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):
        if self.next != None:
            return f"data: {self.__data}, next: {self.next.__data}"
        else:
            return f"data: {self.__data}, next: {self.next}"

class LinkedList:
    def __init__(self, head = None):
        if head != None:
            self.head = [head]
        else:
            self.head = []
        self.length = len(self)

    def __len__(self):
        counts = 0
        if self.head != [] and self.head != None:
            counts = 1
            item = self.head
            while item.next != None:
                item = item.next
                counts += 1
            self.length = counts
        return counts

    def append(self, element):
        new_node = Node(element)
        if self.length > 0 :
            item = self.head
            while item.next != None:
                item = item.next
            item.next = new_node
            len(self)
        else:
            self.head = new_node
            len(self)

    def __str__(self):
        if self.length == 0:
            return "LinkedList[]"
```

```

else:
    s = f"LinkedList[length = {self.length}, ["
    item = self.head
    while item != None:
        s += str(item) + '; '
        item = item.next
    s = s[:-2] + ']'
    return s

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

def change_on_start(self, n, new_data):
    if len(self) < n or n <= 0:
        raise KeyError("Element doesn't exist!")
    else:
        item = self.head
        counted = 1
        while counted != n:
            item = item.next
            counted += 1
        item.change_data(new_data)

def clear(self):
    self.length = 0
    while self.head != None:
        item = self.head
        self.head = self.head.next
        item = None

```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

Таблица Б.1 - Примеры тестовых случаев

№ 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</pre>	<pre>data: 1, next: None data: 1, next: 2</pre>	Верно
2.	<pre>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</pre>	<pre>LinkedList[] 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]] 1</pre>	Верно