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. 3344 _____ Кузнецов Р. А.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург

2024

Цель работы

Изучение алгоритмов и структур данных на языке программирования Python, реализация программы при помощи односвязного списка.

Задание

Вариант 3

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.

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.
    1. Если значение переменной head равна None, метод должен создавать пустой список.
    2. 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:
    1. Если список пустой, то строковое представление: “LinkedList[]”
    2. 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_end(self, n, new_data) - меняет значение поля data n-того элемента с конца списка на new_data. Метод должен выбрасывать исключение KeyError, с сообщением "Element doesn't exist!", если количество элементов меньше n.

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

Выполнение работы

Связный список - это структура данных, которая состоит из узлов, каждый из которых содержит какое-то значение и ссылку на следующий узел в списке. Связные списки бывают односвязные и двусвязные(отличаются тем, что во втором случае есть ссылка еще и на предыдущий узел). Отличие такой структуры данных от массива состоит в том, что в связном списке элементы хранятся в различных участках памяти, а в массиве в одном, также массив не может хранить различные типы данных в отличии от списка.

Класс Node представлен односвязным списком, в котором присутствуют такие поля как data(элемент) и next(ссылка). Методы в этом классе: __init__, который инициализирует экземпляр класса, get_data, который возвращает содержимое какого либо элемента списка, change_data, который меняет содержимое элемента на то, которое предоставит пользователь и __str__, который возвращает строку вида “data: <node_data>, next: <node_next>”. Реализация этих методов достаточно проста.

Следующий класс LinkedList, который имеет поля length(длина списка) и head(первый элемент). Здесь присутствуют такие методы как: __init__, который инициализирует голову и длину списка, посредством перебора до конца списка циклом, __len__, который возвращает длину списка, append, который добавляет element и увеличивает длину. Происходит проверка на заполненность списка и если он пустой, то элемент становится head, иначе же добавляется в конец списка. Следующий метод это __str__, который выводит некоторые данные о списке. В начале выводится длина, затем циклом программа вставляет в строку очередной элемент списка, ставя между ними точку с запятой. Если же список пустой выводится пустая строка, иначе готовая. Метод pop удаляет последний элемент списка и отнимает единицу, если список не пустой. Если есть более 2 элементов, с помощью цикла программа проходит до предпоследнего элемента списка и заменяет его поле next значением None. Если список пустой выводится

ошибка. Еще, метод change_on_end, который меняет n-й элемент с конца на нужный. Если положительное n не превосходит длину списка, программа идет до (длина списка – индекс) элемента и меняет его значение. Если n не удовлетворяет условиям, выводится ошибка. Последний метод clear очищает список путем нулевой инициализации. get_data, clear и change_data — O(1); append, pop и change_on_start — O(n).

Реализация бинарного поиска в односвязном списке затруднительна, но в двусвязном – вполне возможна. Алгоритм поиска будет такой же как и в обычном списке, но вместо индексов придется использовать ссылки на элементы. Также необходимо чтобы список был упорядочен. Сложность заключается как раз в проблеме доступа к элементу. Реализация состоит из: определить длину списка, создать два указателя для отслеживания текущих границ списка, найти средний элемент между границами, сравнить с целевым значением и обновить какую то переменную с границы.

Тестирование

Результаты тестирования представлены в таблице 1.

Таблица 1 – Результаты тестирования

Тест	Выходные данные	Комментарии

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_l = LinkedList()
print(l_l)
print(len(l_l))
l_l.append(10)
l_l.append(20)
l_l.append(30)
l_l.append(40)
print(l_l)
print(len(l_l))
l_l.change_on_end(2, 3)
print(l_l)
	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 = 4, [data: 10, next: 20; data: 20, next: 3; data: 3, next: 40; data: 40, next: None]]
		Корректно

Выводы

Были изучены алгоритмы и структуры данных на языке программирования Python, реализована программы при помощи односвязного списка.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
class Node:
    def __init__(self, data, next=None):
        self.data, self.next = data, 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.lenght, self.head = 0, head

        tmp = self.head
        while tmp:
            tmp, self.lenght = self.next, self.lenght + 1

    def __len__(self):
        return self.lenght

    def append(self, element):
        self.lenght, tmp = self.lenght + 1, self.head

        if self.head:
            while tmp.next:
                tmp = tmp.next
            tmp.next = Node(element)
        else:
            self.head = Node(element)

    def __str__(self):
        tmp, link = self.head, "LinkedList["

        if self.head:
            link += f"length = {self.lenght}, ["

            while tmp:
                                link  +=  f"data:  {tmp.get_data()},  next:
{tmp.next.get_data() if tmp.next else None}"
                if tmp.next:
                    link += "; "
                tmp = tmp.next

            link += "]]"
        else:
            link += "]"
```

```
        return link


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

        self.lenght -= 1
        if self.head.next:
            tmp = self.head

            while tmp.next.next:
                tmp = tmp.next

            tmp.next = None
        else:
            self.head = None


    def change_on_end(self, n, new_data):
        if not 0 < n <= self.lenght:
            raise KeyError("Element doesn't exist!")

        cnt, tmp = 0, self.head

        while tmp.next and cnt != self.lenght - n:
            tmp, cnt = tmp.next, cnt + 1

        tmp.change_data(new_data)


    def clear(self):
        self.__init__()
```