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 _____ Хангулян С. К.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург

2024

Цель работы

Целью работы является изучение алгоритмов и структур данных в Python, создание 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

В работе был реализован односвязный список. Односвязный список – структура данных, каждый элемент которой имеет 2 поля: указатель на следующий элемент next и содержимое элемента data. В таком списке не работает арифметика указателей и индексация, в односвязном списке можно двигаться только в одном направлении: вперед. Полем next конечного элемента является None, означающее конец списка. Список удобнее и быстрее для добавления и удаления элементов на низкоуровневых ЯП, но время, за которое можно получить доступ к элементу, растет с увеличением индекса элемента. В односвязном списке чем дальше элемента от начала head, тем дольше время получения доступа к нему. Частично эту проблему решает двусвязный список.

Класс Node

Экземпляры класса являются элементами односвязного списка, класс имеет поля data и next. Метод __init__ инициализирует экземпляр класса. Метод get_data возвращает содержимое данного элемента. Метод change_data меняет содержимое данного элемента на новое. Метод __str__ возвращает строку вида “data: <node_data>, next: <node_next>”. Все методы данного класса являются простыми в реализации.

Класс LinkedList

Экземпляр класса является односвязным списком и имеет поля длина length и первый элемент списка head. Метод __init__ инициализирует голову и длину списка. Длина вычисляется с помощью прохода до конца списка циклом. Метод __len__ возвращает длину списка.

Метод append добавляет элемент element в конец списка. Длина списка увеличивается на единицу. В случае ненулевого списка с помощью цикла в переменную temp записывается очередной элемент, пока не дойдет до последнего. Полем next последнего элемента объявляется данный элемент. В случае пустого списка головой объявляется данный элемент.

Метод __str__ выводит список и его длину. В начале выводится длина, затем, тем же самым циклом, программа вписывает очередной элемент списка в строку, ставя между ними точку с запятой. В конце возвращается созданная строка. В случае пустого списка выводится строка с пустым содержимым списка.

Метод pop удаляет последний элемент списка. В случае непустого списка от длины отнимается единица; если есть более 2 элементов, с помощью цикла программа проходит до предпоследнего элемента списка и заменяет его поле next значением None. В противном случае полем головы head сразу объявляется None. В случае пустого списка выводится соответствующая ошибка.

Метод change_on_end меняется n-й с конца элемент на данный. В случае, если положительное n не превосходит длину списка, программа идет до length – n элемента и меняет его значение. В случае, если n не удовлетворяет условиям, выводится соответствующая ошибка.

Метод clear очищает список путем нулевой инициализации.

Методы append, __str__, pop, change_on_end сложнее в реализации предыдущих методов, однако все строится на циклах и проходке до нужного элемента.

Сложности методов:

1. O(1)
    a. __init__
    b. get_data
    c. Node.__str__
    d. __len__
    e. __clear__
    f. append (если добавляем head)
    g. pop (если список пуст)
2. O(n)
    a. LinkedList.__str__

b. append
c. pop
d. change_on_end

Бинарный поиск в односвязном списке может быть реализован следующим образом:

1. Инициализируем переменные left и right, указывающие на граничные элементы.
2. На каждом шаге сравниваем значение в середине списка с искомым значением.
3. Если значение в середине списка равно искомому значению, возвращаем его индекс.
4. Если значение в середине списка больше искомого значения, продолжаем поиск в левой части списка, сдвигая указатель right на значение середины списка.
5. Если значение в середине списка меньше искомого значения, продолжаем поиск в правой части списка, сдвигая указатель left на значение середины списка.
6. Повторяем шаги 2-5, пока не найдем искомый элемент или пока left не окажется больше right.
7. Если элемент не был найден, возвращаем -1.

Таким образом, при реализации бинарного поиска в односвязном списке необходимо учитывать особенности работы с указателями на элементы списка и правильное обновление граничных указателей при каждом шаге поиска.

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 в Python. Был создан однонаправленный список, были созданы и определены методы для работы с ним.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Khangulyan_Sargis_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):
        return f"data: {self.get_data()}, next: {self.next.get_data()
if self.next else None}"


class LinkedList:
    def __init__(self, head = None):
        self.length = 0
        self.head = head

        temp = self.head
        while temp:
            temp = temp.next
            self.length += 1

    def __len__(self):
        return self.length

    def append(self, element):
        self.length += 1
        if self.head:
            temp = self.head
            while temp.next:
                temp = temp.next
            temp.next = Node(element)
        else:
            self.head = Node(element)

    def __str__(self):
        if self.head:
            LinkedList = f"LinkedList[length = {self.length}, ["
            temp = self.head
            while temp:
                LinkedList   +=   f"data:   {temp.get_data()},   next:
{temp.next.get_data() if temp.next else None}"
                if temp.next:
                    LinkedList += "; "
                temp = temp.next
            LinkedList += "]]"
            return LinkedList
        else:
```

```
        return "LinkedList[]"

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

    def change_on_end(self, n, new_data):
        if 0 < n <= self.length:
            count = 0
            temp = self.head
            while count != self.length - n and temp.next:
                temp = temp.next
                count += 1
            temp.change_data(new_data)
        else:
            raise KeyError("Element doesn't exist!")

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