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 _____ Бажуков С.В.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Введение в алгоритмы и структуры данных. Освоение алгоритмов и структур данных на языке Python.

Задание.

Вариант 2. 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).
- __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.

- delete_on_start(self, n) - удаление n-того элемента с НАЧАЛА списка. Метод должен выбрасывать исключение KeyError, с сообщением "Element doesn't exist!", если количество элементов меньше n.

Выполнение работы

Связный список - это структура данных, состоящая из узлов, где каждый узел содержит данные и ссылку (указатель) на следующий узел в списке. Основные отличия связного списка от массива:

Память: В массиве элементы хранятся в непрерывной области памяти, в то время как узлы связного списка могут быть разбросаны по памяти, связываясь друг с другом через указатели. Также связный список требует дополнительной памяти для хранения указателей на следующие узлы, в то время как массив требует память только для хранения элементов.

Размер: Размер массива фиксирован и определяется при создании, в то время как связный список может динамически изменять свой размер путем добавления или удаления узлов.

Вставка и удаление: Вставка и удаление элементов в массиве может быть затратной операцией, так как требуется сдвигать другие элементы. В связном списке вставка и удаление узлов более эффективны, так как требуется только изменить указатели.

Доступ к элементам: В массиве доступ к элементам осуществляется по индексу, что делает операцию доступа быстрой. В связном списке доступ к элементам может быть медленнее, так как требуется последовательно переходить от одного узла к другому.

Сложность методов:

O(n):

LinkedList.__str__

append

pop

delete_on_start

O(1):

__init__

get_data

Node.__str__

__len__

__clear__

Для связного списка бинарный поиск может быть реализован следующим образом: начиная с головы списка, двигаемся по элементам, чтобы найти средний элемент. Для этого используем две переменные, одна из которых движется на 2 ссылки за итерацию, а другая на одну. Когда первая переменная достигнет конца списка, вторая будет указывать на середину. Затем сравниваем средний элемент с ключом. Если ключ найден, поиск завершается. Если нет, определяем, какая половина списка будет использоваться для следующего поиска: левая, если ключ меньше среднего элемента, и правая, если больше. Процесс продолжается до тех пор, пока ключ не будет найден или список не будет исчерпан. Основное отличие бинарного поиска для связного списка от классического заключается в том, что в связном списке используются указатели для перемещения, что делает поиск более медленным из-за необходимости просмотра всех элементов.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ пп	Входные данные	Выходные данные	Комментарии
	node = Node(1) print(node) node.next = Node(2, None) print(node)	data: 1, next: None data: 1, next: 2	-
	linked_list = LinkedList() print(linked_list) print(len(linked_list)) linked_list.append(10) print(linked_list) print(len(linked_list)) linked_list.append(20) print(linked_list) print(len(linked_list)) linked_list.pop() print(linked_list) print(linked_list) print(len(linked_list))	LinkedList[] 0 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]] LinkedList[length = 1, [data: 10, next: None]] 1	-

Выводы

Были получены базовые знания об алгоритмах и структурах данных и их применении в языке 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 __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
        if head:
            self.length = 1
        else:
            self.length = 0

    def __len__(self):
        return self.length

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

    def __str__(self):
```

```
        boof = self.head
        if not boof:
            return "LinkedList[]"
        Arr = []
        while boof:
            Arr.append(boof)
            boof = boof.next
         return f"LinkedList[length = {self.length}, [{'; '.join(map(str,
Arr))}]]"

    def pop(self):
        if not self.head:
            raise IndexError("LinkedList is empty!")
        elif not self.head.next:
            self.head = None
            self.length -= 1
        else:
            boof = self.head
            while boof.next.next:
                boof = boof.next
            boof.next = None
            self.length -= 1

    def clear(self):
        self.head = None
        self.length = 0

    def delete_on_start(self, n):
        if self.length < n or n <= 0:
            raise KeyError("Element doesn't exist!")
        elif n == 1:
            self.head = self.head.next
        else:
            boof = self.head
            for _ in range(n-2):
                boof = boof.next
            boof.next = boof.next.next
        self.length -= 1
```