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. Вариант 2

Студент гр. 3343 _____ Поддубный В.А.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург

2024

Цель работы

Практическое применение объектно-ориентированного программирования (ООП) для создания собственной реализации односвязного списка на Python.

Задание

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.

o 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

1. Что такое связный список?

Односвязный список — это линейная структура данных, состоящая из узл ов, где каждый узел содержит данные (data) и ссылку (next) на следующий узел в последовательности.

Отличия от массива:

- **Организация в памяти:** Массив хранится в непрерывном блоке памяти, а узлы связного списка могут располагаться в произвольных местах памят и, связанных ссылками.
- **Доступ к элементам:** В массиве доступ к элементам осуществляется по и ндексу за O(1) время, в связном списке требуется пройти по ссылкам, что занимает O(n) времени в худшем случае.
- **Вставка/удаление:** Вставка/удаление в середине массива требует сдвига э лементов, что занимает O(n) времени. В связном списке достаточно измен ить ссылки, что занимает O(1) времени после нахождения нужного узла.

2. Сложность методов:

- __len__: O(n) - необходимо пройти по всем узлам, чтобы подсчитать коли чество элементов.
- append: O(n) - требуется найти последний узел, чтобы добавить новый.
- pop: O(n) - необходимо найти предпоследний узел, чтобы удалить последн ий.
- delete_on_start: O(n) - нужно пройти до n-1 узла, чтобы удалить n-ый.
- clear: O(1) - просто сбрасывает голову списка и длину.

3. Бинарный поиск в связном списке:

Классический бинарный поиск неэффективен для связного списка, так как он требует прямого доступа к середине списка, что занимает O(n) времени.

Возможное решение:

- **Использование сортированного списка:** Если отсортировать список, то каждый поиск будет занимать O(log n), однако сама сортировка займет O(n log n)

Отличия от бинарного поиска в классическом списке Python:

- В классическом списке Python бинарный поиск работает за O(log n) времени, так как доступ к элементам по индексу занимает O(1) время.
- В связном списке бинарный поиск в его классической форме не применим из-за необходимости последовательного доступа к элементам, но использование сортированного списка позволяет нам это сделать также за O(log n)

Тестирование

Результаты тестирования содержатся в таблице 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	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	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	Программа сработала корректно.

Выводы

Односвязные списки обладают своими преимуществами (эффективная вставка/удаление) и недостатками (медленный произвольный доступ) по сравнению с массивами. Выбор между ними зависит от конкретных требований задачи к эффективности операций и использованию памяти.

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.__data}, next: {self.next.__data if
self.next else None}"



class LinkedList:
    def __init__(self, head: Node = None):
        self.head = head
        if head is not None:
            self.length = 1
        else:
            self.length = 0

    def __len__(self):
        return self.length

    def append(self, element):
        self.length += 1
        current_node = self.head
        if current_node is None:
            self.head = Node(element)
            return
        while current_node.next:
            current_node = current_node.next
        current_node.next = Node(element)

    def __str__(self):
```

```
        if self.length == 0: return "LinkedList[]"
        current_node = self.head
        node_to_str_list = []
        while current_node:
            node_to_str_list.append(current_node.__str__())
            current_node = current_node.next
        linkedlist_elements = "; ".join(node_to_str_list)
        return f"LinkedList[length = {self.length},
[{linkedlist_elements}]]"

    def pop(self):
        if self.length == 0: raise IndexError("LinkedList is
empty!")
        current_node = self.head
        if self.length == 1:
            self.head = None
        else:
            while current_node.next.next:
                current_node = current_node.next
            current_node.next = None
        self.length -= 1

    def delete_on_start(self, n):
        if self.length < n or n <= 0: raise KeyError("Element
doesn't exist!")
        current_node = self.head
        count = 1
        if n == 1:
            self.head = current_node.next
        else:
            while count + 1 != n:
                count += 1
                current_node = current_node.next
            current_node.next = current_node.next.next
        self.length -= 1

    def clear(self):
        while self.head:
            self.pop()
```