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. 3342 _____ Хайруллов Д.Л.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Целью работы является изучение основ работы с алгоритмами и структурами данных в языке программирования Pyhon.

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:

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_end(self, n) - удаление n-того элемента с конца списка. 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
```

Выполнение работы

В ходе выполнения работы необходимо было реализовать классы элементов списка и сам связанный список. Связный список - это структура данных, состоящая из узлов, каждый из которых содержит данные и ссылку на следующий узел в списке. В отличие от массива, в связном списке каждый элемент может быть размещен в любом месте в памяти, поскольку каждый элемент имеет ссылку на следующий элемент. Вставка и удаление элементов: Добавление и удаление элементов в связном списке требует только изменения ссылок на следующие элементы, что делает их более эффективными, чем массивы, где вставка и удаление элементов может требовать перестановки других элементов. Доступ к элементам: Для доступа к элементам в массиве можно использовать индекс, а в связном списке нужно перебирать элементы, начиная с первого элемента.

Сложности реализованных методов в коде:

Класс Node:

1. Метод __init__ имеет сложность O(1);
2. Метод get_data имеет сложность O(1);
3. Метод __str__ имеет сложность O(1).

Класс LinkedList:

1. Метод __init__ имеет сложность O(1);
2. Метод len имеет сложность O(1);
3. Метод append имеет сложность O(n);
4. Метод __str__ имеет сложность O(1);
5. Метод pop имеет сложность O(n);
6. Метод delete_on_end имеет сложность O(n);
7. Метод clear имеет сложность O(1).

При реализации бинарного поиска в связном списке отличие заключается в том, что бинарный поиск по сути требует доступа к элементам по индексу или по среднему элементу массива. В связном списке доступ к элементам по индексу не такой эффективный, как в классическом списке Python, так как для

доступа к элементу по индексу в связном списке нужно последовательно обходить все предыдущие элементы.

Для реализации бинарного поиска в связном списке можно использовать два подхода:

1. Преобразование связного списка в массив

- Обходить связный список и создать массив, в котором каждому элементу будет соответствовать свой индекс. Затем выполнить бинарный поиск по этому массиву.

- Сложность преобразования связного списка в массив: O(n) (где n - количество элементов в списке), сложность бинарного поиска: O(log n). Общая сложность: O(n + log n).

2. Бинарный поиск в самом связном списке

- Осуществлять бинарный поиск в самом связном списке, но зная только начальный элемент. Для этого можно использовать указатель на начало и на конец интервала поиска.

- Сложность бинарного поиска в связном списке: O(log n).

Основное отличие реализации бинарного поиска для связного списка от классического списка Python заключается в эффективности доступа к элементам по индексу и необходимости преобразования структуры списка для удобства применения алгоритма бинарного поиска.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные 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	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 изучены основы работы с алгоритмами и структурами данных в языке программирования Python. Были реализованы классы элементов списка и самого списка, методы для работы с ними.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
class Node:
    def __init__(self, data, next=None):
        self.__data = data
        self.next = next
        pass

    def get_data(self):
        return self.__data

    def __str__(self):
        next_data = "None" if self.next is None else self.next.get_data()
        return f"data: {self.get_data()}, next: {next_data}"
        pass


class LinkedList:
    def __init__(self, head=None):
        self.head = head
        self.length = 0 if head is None else 1
        pass

    def __len__(self):
        return self.length
        pass

    def append(self, element):
        new_node = Node(element)

        if self.head is None:
            self.head = new_node
        else:
            current_node = self.head
            while(current_node.next is not None):
                current_node = current_node.next
            current_node.next = new_node

        self.length +=1
        pass

    def __str__(self):
        if self.head is None:
            return "LinkedList[]"
        else:
            current_node = self.head
            str_list    =       f"LinkedList[length    =    {self.length},
[{str(current_node)}"
            current_node = current_node.next

            while(current_node is not None):
                str_list += f"; {str(current_node)}"
                current_node = current_node.next
```

```
            str_list += "]]"
            return str_list
        pass

    def pop(self):
        if self.head is None:
            raise IndexError("LinkedList is empty!")
        elif self.head.next is None:
            self.length = 0
            self.head = None
        else:
            current_node = self.head
            while(current_node.next.next is not None):
                current_node = current_node.next
            current_node.next = None
            self.length -= 1
        pass

    def delete_on_end(self, n):

        if self.length < n or n <= 0  or self.head is None:
            raise KeyError("Element doesn't exist!")

        elif self.length == n:
            self.head = self.head.next
            self.length -= 1

        else:
            current_node = self.head
            for i in range(self.length - n - 1):
                current_node = current_node.next
            current_node.next = current_node.next.next

            self.length -= 1
        pass

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