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 _____ Коршунов П.И.

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

Связанный список – это структура данных, которая состоит из элементов, узлов. В узлах хранятся данные, а между собой узлы связаны связями. Связь – ссылка на следующий или предыдущий элемент списка. Основные отличия связного списка от массива.

Связный список:

1. Связанные списки хранятся не в непрерывном расположении.

2. Динамический размер.

3. Память выделяется во время выполнения.

4. Использует больше памяти, поскольку в нем хранятся как данные, так и адрес следующего узла.

5. Для доступа к элементу требуется обход всего связанного списка.

6. Операции вставки и удаления выполняются быстрее.

Массив:

1. Массивы хранятся в непрерывном расположении.

2. Фиксированный размер.

3. Память выделяется во время компиляции.

4. Использует меньше памяти, чем связанные списки.

5. К элементам можно легко получить доступ.

6. Операции вставки и удаления требуют времени.

Сложности методов.

O(1):

__init__

get_data

Node.__str__

__len__

__clear__

append(если добавляем head)

pop(если список пуст)

delete_on_start(если удаляем первый элемент)

O(n):

LinkedList.__str__

append

pop

delete_on_start

Для связного списка, реализация бинарного поиска может выглядеть следующим образом:

1) Найдем средний элемент связанного списка. Для его нахождения, начиная с головы списка, идем двумя переменными до его конца. Одна переменная за одну итерацию будет проходить вперед на 2 ссылке, а другая на одну. Тогда, когда первая переменная дойдет до конца, вторая будет в середине.
2) Сравним средний элемент с ключом.
3) Если ключ найден в среднем элементе, процесс завершается.
4) Если ключ не найден в среднем элементе, выберите, какая половина будет использоваться в качестве следующего пространства поиска.
5) Если ключ меньше среднего узла, то для следующего поиска используется левая сторона.
6) Если ключ больше среднего узла, то для следующего поиска используется правая сторона.

Этот процесс продолжается до тех пор, пока не будет найден ключ или не будет исчерпан весь связанный список. Основное отличие реализации бинарного поиска для связного списка от реализации для классического списка заключается в том, что в связном списке поиск осуществляется путем перемещения указателей, а не прямого доступа к элементам по индексу. Это делает поиск более медленным, так как требуется пройти по всем элементам до нужного.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	n = Node(1) l = LinkedList(n) l.append(2) print(l) l.pop() print(l)	LinkedList = 2, [data: 1, next: 2; data: 2, next: None]] LinkedList = 1, [data: 1, next: None]]	-
2.	n = Node(1) l = LinkedList(n) l.append(2) l.append(3) print(l) l.delete_on_start(2) print(l)	LinkedList = 3, [data: 1, next: 2; data: 2, next: 3; data: 3, next: None]] LinkedList = 2, [data: 1, next: 3; data: 3, next: None]]	-
3.	n = Node(1) print(n) print(n.get_data())	data: 1, next: None 1	-

Выводы

Были получены базовые знания об алгоритмах и структурах данных и их применении в Python.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Korshunov_Petr_lb2.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
        self.length = 1 if self.head else 0

    def __len__(self):
        return self.length

    def append(self, element):
        if not self.head:
            self.head = Node(element)
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = Node(element)
        self.length += 1

    def __str__(self):
        if not self.head:
```

```
            return f"{self.__class__.__name__}[]"

        elements = [self.head]
        elements.extend(elements[-1].next for _ in range(self.length - 1))
        return   f"{self.__class__.__name__}   =   {self.length},   [{';
'.join(map(str, elements))}]]"

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

        if not self.head.next:
            self.head = None
            self.length -= 1
            return

        current = self.head
        while current.next.next:
            current = current.next
        current.next = None
        self.length -= 1

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

    def delete_on_start(self, n):
        if n > self.length or n < 1:
            raise KeyError("Element doesn't exist!")

        if n == 1:
            self.head = self.head.next
            self.length -= 1
            return

        position = 1
        current = self.head
        while current.next and position <= n:
            if position == n - 1:
```

```
            current.next = current.next.next
            self.length -= 1
            break
        current = current.next
        position += 1
```