МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Информатика»
Тема: Алгоритмы и структуры данных в Python

Студент гр. 3342 _____ Романов Е.А.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Изучение структуры данных линейный список и реализация этой структуры на языке программирования Python.

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_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 – динамическая структура данных, элементы которой ссылаются друг на друга (каждый элемент на следующий, каждый элемент на предыдущий, каждый элемент на предыдущий и следующий).

Работа состоит из описания двух классов.

Первый класс Node имеет три метода:

- __init__, который принимает в качестве аргументов данные, которые необходимо сохранить и ссылку на следующий элемент списка. Сложность - O(1).
- get_data – возвращает значение приватного поля data экземпляра класса Node
- str - выводит экземпляр класса в строковом представлении, согласно заданию

Второй класс LinkedList имеет семь методов:

- __init__ - принимает ссылку на головной элемент списка и задаёт поле length, хранящее количество элементов списка. Сложность O(1).
- len при помощи цикла while подсчитывает количество элементов в списке. Сложность O(n).
- append принимает в качестве аргумента элемент, который необходимо добавить в список и добавляется его в конец. Сложность O(n).
- str – выводит информацию о списке, согласно заданию
- pop - при помощи цикла while доходит до предпоследнего элемента списка и меняет значение его ссылки на следующий элемент на None, таким образом удаляя элемент. Сложность O(n).
- delete_on_start – принимает в качестве аргумента число – порядковый номер элемента в списке. При помощи цикла for находит

элемент, ссылающийся на тот, что необходимо удалить, и меняет значение его поля next на следующий за удаляемым элемент. Сложность O(n).

- clear –проходит по элементам списка и последовательно их удаляет. Сложность O(n).

Бинарный поиск возможен только в сортированном наборе данных. Поэтому сначала необходимо отсортировать список. Далее можно воспользоваться алгоритмом бинарного поиска:

1) Найдите средний элемент связанного списка
2) Сравните 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.
7) Этот процесс продолжается до тех пор, пока не будет найден ключ или не будет исчерпан весь связанный список.

Отличие бинарного поиска в связном списке и классическом списке языка Python заключается прежде всего в поиске среднего элемента для сравнения, так, в связном списке для нахождения среднего элемента необходимо проходить, по крайней мере половину исследуемой части списка, при каждой итерации поиска. При условии, что известна длина списка. В то же время в классических списках для этого можно использовать арифметику указателей, тогда поиск необходимого элемента будет выполнятся за константную сложность.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

Входные данные	Выходные данные	Комментарий
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]] 1	Вывод верный

Выводы

В ходе лабораторной работы была изучена структура данных связный список и написана реализация этой структуру на языке программирования 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 f'data: {self.get_data()}, next: {None}'


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

    def __len__(self):
        if self.length == 0: return 0
        amount = 0
        current_el = self.head
        while current_el != None:
            current_el = current_el.next
            amount+=1
        return amount

    def append(self, element):
        new_elem = Node(element)
        current_el = self.head
        if current_el != None:
            while current_el.next != None:
                current_el = current_el.next
            current_el.next = new_elem
        else: self.head = new_elem
        self.length += 1

    def __str__(self):
        if self.length <= 0: return "LinkedList[]"
        else:
            nodes_data = []
            current_el = self.head
            while current_el != None:
                nodes_data.append(str(current_el))
                current_el = current_el.next
            return f"LinkedList[length = {len(self)}, [{';
'.join(nodes_data)}]]"

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

```
        if self.length == 1:
                self.head = None

        else:
            current_el = self.head
            while current_el.next.next != None:
                current_el = current_el.next
            current_el.next = None
        self.length -=1


    def delete_on_start(self, n):
        current_el = self.head
        if (n > self.length or n<=0) :
            raise KeyError("Element doesn't exist!")
        else:
            if n == 1:
                self.head = current_el.next
            else:
                for i in range(n-2):
                    current_el = current_el.next
                current_el.next = current_el.next.next if current_el.next
!= None else None
            self.length -=1

    def clear(self):
        current_el = self.head
        tmp = self.head
        while current_el != None:
            tmp = current_el.next
            current_el.next = None
            current_el = tmp
        self.length = 0
        self.head = None
```