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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Информатика»

Тема: Алгоритмы и структуры данных.

Студент гр. 3344 _____ Ханнанов А.Ф.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Получить опыт реализации структур данных и алгоритмов работы с ними.

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).
- change_data(self, new_data) - метод меняет значение поля data объекта 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.

o __len__(self) - перегрузка метода __len__, он должен возвращать длину списка (этот стандартный метод, например, используется в функции len).

o 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.

o __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.

- change_on_start(self, n, new_data) - изменение поля data n-того элемента с НАЧАЛА списка на new_data. Метод должен выбрасывать исключение KeyError, с сообщением "Element doesn't exist!", если количество элементов меньше n.

Выполнение работы

1) Связный список — это структура данных, в которой элементы отсортированы по порядку с помощью указателей. Отличие от массива в том, что массив может хранить только данные одного типа; также массив хранится в одном участке памяти, а элементы списка в различных.
2) Для класса Node:
    - get_data() – O(1)
    - change_data – O(1)

Для класса LinkedList:

- append() – O(n)
- pop() – O(n)
- change_on_start() – O(n)
- clear() – O(1)

3) Реализация бинарного поиска:

Функция находит средний элемент и сравнивает его с входным. Если элемент ему равен, то функция завершается, иначе функция вызывает себя с указанием левой или правой части связного списка (выбор зависит от сравнения среднего элемента с входным).

Разница бинарного поиска в связном списке и классическом списке состоит в том, что для поиска элементов в классическом списке можно использовать их индексы, что ускоряет работы функции. Для связного списка выполнение функции будет дольше.

Выводы

Получен опыт в реализации связного списка в языке Python при помощи классов. Изучены способы создания алгоритмов для работы списка.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Khannanov_Artem_lb2.py

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

    def get_data(self):
        return self.data

    def change_data(self, new_data):
        self.data = new_data

    def __str__(self):
        if self.next:
            return f"data: {self.data}, next: {self.next.data}"
        return f"data: {self.get_data()}, next: None"


class LinkedList:
    def __init__(self, head=None):
        self.head = head
        self.length = 0
        while head:
            head = head.next
            self.length += 1

    def __len__(self):
        return self.length

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

    def __str__(self):
        if self.length:
            s = ''
            tmp = self.head
            while tmp:
                s += tmp.__str__() + '; '
                tmp = tmp.next
            return f"LinkedList[length = {self.length}, [{s[:-2]}]]"
        else:
            return "LinkedList[]"

    def pop(self):
```

```
    if not self.head:
        raise IndexError("LinkedList is empty!")
    if self.head.next:
        tmp = self.head
        while tmp.next.next:
            tmp = tmp.next
        tmp.next = None
    else:
        self.head = None
    self.length -= 1

def change_on_start(self, n, new_data):
    if self.length < n or n <= 0:
        raise KeyError("Element doesn't exist!")
    tmp = self.head
    for i in range(1, n):
        tmp = tmp.next
    tmp.data = new_data

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