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 в Python

Студентка гр. 3342 _____ Епонишникова А.И

Преподаватель _____ Иванов Д.В

Санкт-Петербург

2024

Цель работы

Целью работы является на практике изучить работу с линейным списком

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.

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.

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

Выполнение работы

Был создан класс Node, здесь были представлены поля data и next. Также были представлены методы: init, get_data, str.

Был создан класс LinkedList. Поля – head, length. Методы – init, len, append, str, pop, cleat, delete_on_end.

Связный список — это структура данных, в которой элементы линейно упорядочены, но порядок определяется не номерами элементов (как в массивах), а указателями, входящих в состав элементов списка и указывают на следующий элемент.

Основные отличия связного списка от массива:

Доступ к элементам в связном списке происходит последовательно, начиная с первого элемента и переходя к следующему по ссылке. В массиве же доступ к элементам осуществляется напрямую по индексу.

Вставка и удаление элементов в середине связного списка происходит быстрее, чем в массиве, так как не требуется сдвигать все последующие элементы.

Каждый элемент содержит ссылку на следующий элемент. В отличие от массива, где элементы хранятся в памяти последовательно, в связном списке элементы могут быть разбросаны по памяти.

Сложность каждого метода:

Класс Node:

__init__ - O(1)

get_data – O(1)

__str__ - O(1)

Класс LinkedList:

__init__ - O(1)

__len__ - O(n)

append – O(n)

__str__ - O(n)

pop – O(n)

delete_on_end – O(n)

clear – O(1)

Для реализации бинарного поиска в связном списке требуется учитывать особенности структуры данных. В связном списке нет прямого доступа к элементам по индексу, поэтому для реализации бинарного поиска необходимо использовать итеративный подход, начиная с головы списка и двигаясь к нужному элементу.

Разработанный программный код см. в приложении А.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные
	node = Node(1) print(node) # data: 1, next: None node.next = Node(2, None) print(node) # data: 1, next: 2 print(node.get_data()) l_l = LinkedList() print(l_l) print(len(l_l)) l_l.append(10) l_l.append(20) l_l.append(30) l_l.append(40) print(l_l) print(len(l_l)) l_l.delete_on_end(3) print(l_l)	data: 1, next: None data: 1, next: 2 1 LinkedList[] 0 LinkedList[length = 4, [data: 10, next: 20; data: 20, next: 30; data: 30, next: 40; data: 40, next: None]] 4 LinkedList[length = 3, [data: 10, next: 30; data: 30, next: 40; data: 40, next: None]]

Выводы

Был реализован связный список на языке Python. Были определены различия между связным списком и массивом, исследована скорость работы методов и описана возможная реализация бинарного списка.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lab2.c

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

    def get_data(self):
        return self.data

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


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

    def __len__(self):
        return self.length

    def append(self, element):
        new_element = Node(element)
        if self.length == 0:
            self.head = new_element
        else:
            tmp = self.head
            while tmp.next is not None:
                tmp = tmp.next
            tmp.next = new_element
        self.length += 1


    def __str__(self):
        if self.length == 0:
            return f"LinkedList[]"
        tmp = self.head
        arr = []
        while tmp.next is not None:
            arr.append(str(tmp))
            tmp = tmp.next
        arr.append(str(tmp))
                return  f'LinkedList[length  =  {self.length},  [{";
".join(arr)}]]'


    def pop(self):
```

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


def delete_on_end(self, n):
    if self.length < n or n <= 0:
        raise KeyError (f"Element doesn't exist!")
    elif self.length == n:
        self.head = self.head.next
    else:
        tmp = self.head
        i = 2
        idx = self.length - n + 1
        while i < idx:
            tmp = tmp.next
            i += 1
        tmp.next = tmp.next.next

    self.length -=1



def clear(self):
    while self.length != 0:
        self.pop()
```