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

Студент гр. 3342 _____ Песчатский С. Д.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург
2024

Цель работы

Введение в алгоритмы и структуры данных. Освоение алгоритмов и структур данных на языке 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.

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.

o **pop(self)** - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.

o **clear(self)** - очищение списка.

o **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

Связный список - это структура данных, состоящая из узлов, где каждый узел содержит данные и ссылку (указатель) на следующий узел в списке. Основные отличия связного списка от массива:

Память: В массиве элементы хранятся в непрерывной области памяти, в то время как узлы связного списка могут быть разбросаны по памяти, связываясь друг с другом через указатели. Также связный список требует дополнительной памяти для хранения указателей на следующие узлы, в то время как массив требует память только для хранения элементов.

Размер: Размер массива фиксирован и определяется при создании, в то время как связный список может динамически изменять свой размер путем добавления или удаления узлов.

Вставка и удаление: Вставка и удаление элементов в массиве может быть затратной операцией, так как требуется сдвигать другие элементы. В связном списке вставка и удаление узлов более эффективны, так как требуется только изменить указатели.

Доступ к элементам: В массиве доступ к элементам осуществляется по индексу, что делает операцию доступа быстрой. В связном списке доступ к элементам может быть медленнее, так как требуется последовательно переходить от одного узла к другому.

Сложность методов:

O(n):

LinkedList.__str__

append

pop

delete_on_start

O(1):

__init__

get_data

Node.__str__

__len__

__clear__

Для связного списка, бинарный поиск может быть реализован следующим образом: начиная с головы списка, двигаемся по элементам, чтобы найти средний элемент. Для этого используем две переменные, одна из которых движется на 2 ссылки за итерацию, а другая на одну. Когда первая переменная достигнет конца списка, вторая будет указывать на середину. Затем сравниваем средний элемент с ключом. Если ключ найден, поиск завершается. Если нет, определяем, какая половина списка будет использоваться для следующего поиска: левая, если ключ меньше среднего элемента, и правая, если больше. Процесс продолжается до тех пор, пока ключ не будет найден или список не будет исчерпан. Основное отличие бинарного поиска для связного списка от классического заключается в том, что в связном списке используются указатели для перемещения, что делает поиск более медленным из-за необходимости просмотра всех элементов.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ пп	Входные данные	Выходные данные	Комментарии
	node = Node(1) print(node) node.next = Node(2, None) print(node)	data: 1, next: None data: 1, next: 2	-
	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]] LinkedList[length = 1, [data: 10, next: None]] 1	-

Выводы

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

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
        pass

    def __str__(self):
        if self.next == None:
            return "data: "+str(self.data)+", next: "+str(self.next)
        else:
            return "data: " + str(self.data) + ", next: " +
str(self.next.data)
        pass


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

    def __len__(self):
        return self.length
        pass

    def append(self, element):
```

```
        self.length = self.length + 1
        new_node=Node(element)
        if self.head is None:
            self.head = new_node
            return
        current_node=self.head
        while current_node.next:
            current_node = current_node.next
        current_node.next = new_node
        pass

    def __str__(self):
        if self.length == 0:
            return "LinkedList[]"
        pass
        curr = self.head
        ans = "LinkedList[length = "+str(self.length) + ", ["
        first = True
        while curr.next:
            if first:
                ans = ans+"data: " + str(curr.data) + ", next: " +
str(curr.next.data) + ";"
                first = False
            else: ans = ans+" data: " + str(curr.data) + ", next: " +
str(curr.next.data) + ";"
            curr = curr.next
        if self.length != 1:
            ans = ans+" data: " + str(curr.data) + ", next: " +
str(curr.next) + "]]"
        else:
            ans = ans + "data: " + str(curr.data) + ", next: " +
str(curr.next) + "]]"
        return ans
        pass

    def pop(self):
        if self.length == 0:
```

```
            #print("LinkedList is empty!")
            raise IndexError
        if self.length == 1:
            self.head=None
            self.length = 0
            return
        current_node = self.head
        while (current_node.next.next):
            current_node = current_node.next
        current_node.next = None
        self.length = self.length - 1
        pass

    def delete_on_start(self, n):
        if self.length < n or n <1:
            #print("Element doesn't exist!")
            raise KeyError
        if n == 1:
            self.head = self.head.next
        else:
            current_node = self.head
            pos=0
            while pos+1 != n-1 and current_node.next:
                pos = pos+1
                current_node = current_node.next
            if current_node.next:
                current_node.next = current_node.next.next
        self.length = self.length - 1
        pass

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