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. 3343 _____ Жучков О.Д.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Изучить основные структуры данных и научиться работать с ними. Реализовать линейный односвязный список на языке Python с использованием ООП.

Задание

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_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 — структура данных, состоящая из узлов, содержащих некоторые данные и ссылку на следующий узел. В отличие от массива, который последовательно хранит данные в памяти, связный список хранит элементы не последовательно. В связном списке для доступа к отдельному элементу необходимо пройти все элементы до него, начиная с “головы”. В массиве доступ к элементу осуществляется по индексу, что быстрее связного списка, но для добавления или удаления элемента посреди массива необходимо переместить все последующие элементы, что занимает гораздо больше времени, чем добавление или удаление элемента в связном списке.

Сложности методов:

Класс Node:

1. __init__ – O(1);
2. __str__ – O(1);
3. get_data – O(1).

Класс LinkedList:

1. __init__ – O(1);
2. __str__ – O(n);
3. __len__ – O(n);
4. append – O(n);
5. pop – O(n);
6. change_on_start - O(n);
7. clear – O(1).

Анализ:

Для реализации алгоритма бинарного поиска в связном списке достаточно сохранять ссылки на левую и правую границу рассматриваемой части списка и вычислять необходимый по счету элемент, пользуясь значением длины. Для каждого последующего шага потребуется O(n) времени, чтобы получить элемент посередине, а всего нужно будет получить элементы log(n) раз. Поэтому бинарный поиск неэффективен в связном списке, в отличие от массива. Для

поиска элемента в связном списке эффективнее пройтись подряд по всем элемента.

Тестирование

Результаты тестирования содержатся в таблице 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	l_l = LinkedList() print(l_l) print(len(l_l)) l_l.append(111) l_l.append(222) l_l.append(333) print(l_l) print(len(l_l)) l_l.pop() print(l_l) l_l.pop() print(l_l) l_l.pop() print(l_l) l_l.append(111) l_l.append(222) l_l.append(333) l_l.change_on_start(1, 1) print(l_l)	LinkedList[] 0 LinkedList[length = 3, [data: 111, next: 222; data: 222, next: 333; data: 333, next: None]] 3 LinkedList[length = 2, [data: 111, next: 222; data: 222, next: None]] LinkedList[length = 1, [data: 111, next: None]] LinkedList[] LinkedList[length = 3, [data: 1, next: 222; data: 222, next: 333; data: 333, next: None]]	Программа работает корректно.

Выводы

В ходе выполнения лабораторной работы изучены и применены на практике алгоритмы и структуры данных. На языке Python с помощью ООП реализован связный однонаправленный список и методы работы с ним.

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

    def __str__(self):
        return f"data: {self.data}, next: {self.next.data if
self.next is not None else None}"\

class LinkedList:
    def __init__(self, head=None):
        self.head = head
        self.length = 0
        if head is not None:
            self.length = 1

    def __len__(self):
        return self.length

    def append(self,element):
        new = Node(element)
        if self.head is None:
            self.head = new
            self.length = 1
            return
        temp = self.head
        while temp.next is not None:
            temp = temp.next
        temp.next = new
```

```
        self.length += 1

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

    def pop(self):
        if self.length == 0:
            raise IndexError("LinkedList is empty!")
        temp = self.head
        if self.length == 1:
            self.head = None
            self.length = 0
            return
        while temp.next.next is not None:
            temp = temp.next
        temp.next = None
        self.length -= 1

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

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