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 _____ Иванов П.Д.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Изучить особенности работы с односвязными списками. Написать свою реализацию односвязного списка в Python, используя ООП. Получить асимптотическую сложность реализованных методов.

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_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

Описание работы кода:

Node класс представляет узел списка. Каждый узел содержит данные и ссылку на следующий элемент.

LinkedList класс представляет сам список. Он имеет ссылку на головной узел.

Метод append добавляет новый узел в конец списка.

Метод pop удаляет последний узел из списка.

Метод clear очищает список.

Метод delete_on_end удаляет узел на заданной позиции с конца списка.

__str__ -методы используются для удобного вывода списка.

Связный список - это структура данных, состоящая из узлов, каждый из которых содержит данные и ссылку на следующий узел.

Основное отличие от массива заключается в способе организации данных: в массиве элементы хранятся последовательно в памяти, а в связном списке каждый элемент может находиться в произвольном месте памяти, а ссылки между ними обеспечивают связь.

Сложность методов:

append: O(n), так как приходится пройти по всему списку, чтобы добавить новый элемент в конец.

pop: O(n), так как приходится пройти по всему списку, чтобы найти последний элемент.

clear: O(1), так как просто обнуляется ссылка на головной узел.

delete_on_end: O(n), так как в некоторых случаях придется пройти почти весь список.

Реализация бинарного поиска в связном списке отличается от классического списка Python из-за специфики связного списка. Поскольку элементы не расположены последовательно в памяти, бинарный поиск неэффективен. Традиционно бинарный поиск осуществляется на отсортированных массивах, где можно быстро вычислить середину и сравнить с искомым элементом. В связном списке необходимо последовательно проходить от начала к концу (или наоборот), что приводит к линейной сложности O(n).

Тестирование

Результаты тестирования содержатся в таблице 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	plinked_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(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	Проверка реализации работы со связным списком

Выводы

В результате работы был реализован односвязный список в языке Python с использованием ООП, а также была выведена асимптотическая сложность реализованных методов.

Приложение А

Исходный код программы

Название файла: main.py

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

    def get_data(self):
        return self.__data

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


class LinkedList:
    def __init__(self, head=None):
        self.head = head

    @property
    def length(self):
        ct = 0
        current = self.head
        if current is None:
            return 0

        while current.next is not None:
            ct += 1
            current = current.next
        ct += 1
        return ct

    def __len__(self):
        return self.length

    def append(self, element):
        current = self.head
```

```
        if current is None:
            self.head = Node(element)
        else:
            while current.next is not None:
                current = current.next

            current.next = Node(element)

    def get_list(self):
        res = []
        current = self.head
        while current.next is not None:
            res.append(str(current))
            current = current.next
        res.append(str(current))

        return res

    def __str__(self):
        if self.head is None:
            return "LinkedList[]"

                return  f"LinkedList[length  =  {self.length},  [{';
'.join(self.get_list())}]]"

    def pop(self):
        if self.head is None:
            raise IndexError('LinkedList is empty!')

        current = self.head
        if current.next is None:
            self.head = None
        else:
            while current.next.next is not None:
                current = current.next
            current.next = None

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

```
def delete_on_end(self, n):
    if (self.length < n) or (n <= 0):
        raise KeyError("Element doesn't exist!")

    idx = self.length - n
    current = self.head
    for _ in range(idx - 1):
        current = current.next

    if idx == 0:
        self.head = self.head.next
    if idx == self.length - 1:
        current.next = None
    else:
        current.next = current.next.next
```