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 _____ Старков С.А

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Изучить основные особенности структур данных и методов работы с ними. Написать собственную практическую реализацию линейного односвязного списка на Python, используя ООП. Сравнить асимптотическую сложность операций над списком и над массивом.

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.

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.

- __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.

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.
- 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 — это структура данных, которая состоит из узлов, содержащих данные и ссылки («связки») на следующий узел списка. Основное отличие связного списка от массива заключается в том, что массив — это структура данных, которая хранит элементы в памяти последовательно, а связный список — это структура данных, которая хранит элементы в памяти не последовательно, а связывая их между собой ссылками. В массиве доступ к элементам осуществляется по индексу, это позволяет быстро получать доступ к любому элементу массива. Но, при добавлении или удалении элементов в середине массива, необходимо перемещать все элементы после добавляемого или удаляемого, что занимает много времени и ресурсов. В связном списке доступ к элементам осуществляется последовательно, начиная с головы списка. Для доступа к элементу по индексу необходимо последовательно пройти все узлы до нужного. Однако, при добавлении или удалении элементов в середине списка, необходимо просто изменить ссылки на узлы, что занимает меньше времени и ресурсов, чем в массиве. Таким образом, связный список позволяет эффективно добавлять и удалять элементы в середине списка, а массив обеспечивает быстрый доступ к элементам по индексу.

Сложности методов:

Класс Node:

1. init – O(1);
2. get_data – O(1);
3. str – O(1).

Класс LinkedList:

1. init – O(1);
2. len – O(n);
3. append – O(n);
4. str – O(n);
5. pop – O(n);
6. delete_on_start - O(n);
7. clear – O(1).

Анализ:

Алгоритм бинарного поиска в связном списке не оптимален, поскольку требует log(n) раз получения элементов, и каждый запрос занимает O(n) времени. Вместо этого, в связном списке проход по элементам за O(n) приведет к нахождению элемента.

Тестирование

Результаты тестирования содержатся в таблице 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	linked_list = LinkedList() print(linked_list) # LinkedList[] print(len(linked_list)) # 0 linked_list.append(10) print(linked_list) # LinkedList[length = 1, [data: 10, next: None]] print(len(linked_list)) # 1 linked_list.append(20) print(linked_list) # LinkedList[length = 2, [data: 10, next:20; data: 20, next: None]] print(len(linked_list)) # 2 linked_list.pop() print(linked_list) print(linked_list) # LinkedList[length = 1, [data: 10, next: None]] print(len(linked_list)) # 1	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	Программа сработала корректно.

Выводы

В ходе выполнения лабораторной работы были изучены и применены на практике алгоритмы и структуры данных в Python. Разработан односвязный линейный список с применением полученных знаний, реализованы методы для 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 __str__(self):
        return f"data: {self.get_data()}, next: {None if self.next
is None else self.next.get_data()}"


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


    def __len__(self):
        iterator = self.head
        len = 0
        if iterator is None:
            return 0
        while iterator is not None:
            len+=1
            iterator = iterator.next
        return len

    def append(self, element):
        iter = self.head
        if iter is None:
            self.head = Node(element, None)
        else:
            while iter.next is not None:
                iter = iter.next
```

```
            iter.next = Node(element,None)

    def __str__(self):
        if self.head is None:
            return "LinkedList[]"
        else:
            li = []
            iterator = self.head
            while iterator is not None:
                li.append(str(iterator))
                iterator = iterator.next
            strd = '; '.join(li)
            return f"LinkedList[length = {len(self)}, [{strd}]]"

    def pop(self):
        if self.head is None:
            raise KeyError("LinkedList is empty!")
        else:
            iter = self.head
            if len(self)  > 1:
                for _ in range(len(self) - 1-1):
                    iter = iter.next
                iter.next = None
            else:
                self.head = None
        pass

    def delete_on_start(self, n):
        index = n-1
        if len(self)-1 < index or index < 0:
            raise KeyError("Element doesn't exist!")
        iterator = self.head
        for _ in range(index-1):
            iterator = iterator.next



        if index == 0:
            self.head = self.head.next
        if len(self)-1 == index:
```

```
            iterator.next = None
        else:
            iterator.next = iterator.next.next


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