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 _____ Малиновский А.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).
- **change_data(self, new_data)** - метод меняет значение поля data объекта 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.

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.

- **change_on_start(self, n, new_data)** - изменение поля data n-того элемента с НАЧАЛА списка на new_data. 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. change_data – O(1);
4. __str__ – O(1).

Класс LinkedList:

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

Алгоритм бинарного поиска в связном списке не рационален, т.к. для его реализации требуется log(n) раз получать элементы каждый запрос займет O(n), тогда как в связном списке пройдя по элементам за O(n) будет найдет элемент.

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:
    __data=0
    next=0
    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: {None if self.next is
None else self.next.get_data()}'

class LinkedList():
    head=None
    length=0
    def __init__(self, head=None):
        if(head):
            self.head=Node(head)
            self.length=1
        else:
            self.head=None
    def __len__(self):
        return self.length
    def append(self, element):
        tmp = Node(element)
        if self.length == 0:
            self.head = tmp
            self.length = 1
            return
        curr = self.head
```

```
        while curr.next :
            curr = curr.next
        curr.next = tmp
        self.length += 1

    def __str__(self):
        tmp=self.head
        outp=[]
        if self.head:
            while(tmp):
                outp.append(f"data:     {tmp.get_data()},     next:
{tmp.next.get_data() if tmp.next is not None else None}")
                tmp=tmp.next
            return   f'LinkedList[length   =   {len(self)},   [{";
".join(outp)}]]'
        else:
            return 'LinkedList[]'

    def pop(self):
        if len(self)==0:
            raise IndexError("LinkedList is empty!")
        elif len(self)==1:
            self.length=0
            self.head=None
            return self
        else:
            tmp=self.head
            while(tmp.next.next):
                tmp=tmp.next
            tmp.next=None
            self.length-=1


    def change_on_start(self, n, new_data):
        if len(self)<n or n<=0:
            raise KeyError("Element doesn't exist!")
        temp=self.head
        curr=Node(new_data)
        prev=temp
```

```
    if n==1:
       curr.next=self.head.next
       self.head=curr
    else:
        i=1
        while(i<n):
            prev=temp
            temp=temp.next
            i+=1
        prev.next = curr
        curr.next=temp.next


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