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.Тест

Студентка гр. 3341 Чинаева М. Р.

Преподаватель Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Цель работы к данному заданию заключается в реализации связанного однонаправленного списка через создание двух зависимых классов: Node и LinkedList. Цель включает следующие задачи:

1. Создание класса Node, который описывает элемент списка с полями данных и ссылкой на следующий элемент.

2. Создание класса LinkedList, который описывает связанный однонаправленный список с полями головного элемента и количеством элементов списка.

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.

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:

o head # Данные первого элемента списка.

o length # Количество элементов в списке.

И следующие методы:

o __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.

- 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.

Основные теоретические положения

Связный список — это структура данных, в которой элементы линейно упорядочены, но порядок определяется не номерами элементов (как в массивах), а указателями, входящих в состав элементов списка и указывают на следующий элемент.

Массив, в отличие от списка используется для обращения к определенному участку памяти, а так же может хранить данные только одного типа, тогда как связный список может хранить несколько типов данных, из-за этого время на поиск нужного элемента массива затрачивается меньше O(1), тогда как перебор элементов связного списка занимает O(N).

Возможная реализация бинарного поиска(считаем что список отсортирован): сначала вычислить длину связного списка (возможно, пройтись по списку и посчитать количество итераций) Вычислить середину списка, сравнить ее с искомым значением, если подходит то возвращаем индекс, если же искомое значение меньше, проделываем все заново в правой половине списка, если больше, то в левой.

Отличие связного списка от классического для Python в том, что обычный список – это массив по сути своей, то есть возможен быстрый доступ по индексу. В случае бинарного поиска это избавляет нас от дополнительного перемещения по списку. Связный список менее эффективен, чем обычный для бинарного поиска.

Выполнение работы

Определение класса Node:

1. Определение метода __init__() для инициализации объекта класса Node с передачей данных и ссылки на следующий элемент (по умолчанию None)

2. Определение метода get_data(), который возвращает данные элемента.

3. Перегрузка стандартного метода __str__, который преобразует объект в строковое представление.

Определение класса LinkedList:

1. Определение метода __init__() для инициализации объекта класса LinkedList, устанавливающего головной элемент и длину списка, равную единице, если голова не None, и нулю в обратном случае.

2. Перегрузка метода len(), который возвращает длину списка.

3. Перегрузка метода append(), добавляющего новый элемент в конец списка. Если длина списка равна нулю, то новый элемент становится головой списка. В обратном случае с помощью цикла while(present.next != None) находится последний элемент списка и после него добавляется элемент.

4. Перегрузка метода __str__(), который возвращает строковое представление списка. Если длина списка равна нулю возвращается строка "LinkedList[]". Иначе с помощью цикла while(present.next!=None) к нужной строке приписываются данные очередного элемента связного списка.

5. Определение метода pop(), удаляющего последний элемент из списка. Если длина списка равна нулю выбрасывается исключение "LinkedList is empty!". Если длина равна 1, список становится пустым. Далее с помощью while (present.next.next != None) находим предпоследний элемент списка, его поле next делаем равным None и уменьшаем длину списка на 1.

6. Определение метода delete_on_start(), удаляющего n-тый элемент с начала списка. Если n не натуральное число или больше длины массива выбрасывается исключение "Element doesn't exist!". Если n равно единице, то элемент следующий в изначальном списке после головы, становится головой и длина списка уменьшается на 1. В остальных случаях с помощью цикла while (n

!= present_index+1) находится элемент, стоящий перед тем который надо удалить. Элемент удаляется посредством присваивания полю next текущего элемента поля, которое было следующим для удаленного элемента. Длина списка уменьшается на 1.

7. Определение метода clear(), очищающего список. Голова становится равной None, а длина списка 0

Сложность методов:

O(1): все методы класса Node, конструктор списка, получение длины списка и очищение списка, так как не зависят от длины списка

O(n): добавление элемента в список, строковое представление списка, удаление последнего элемента списка, удаление н-го с начала элемента, так как они требуют прохода по всему списку

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	test_list = LinkedList() print (test_list) print (len(test_list)) test_list.append(1) test_list.append(2) test_list.append(3) test_list.append(4) print (test_list) test_list.pop() test_list.delete_on_start(2) print (test_list) test_list.clear() print (test_list)	LinkedList[] 0 LinkedList[length = 4, [data: 1, next: 2; data: 2, next: 3; data: 3, next: 4; data: 4, next: None]] LinkedList[length = 2, [data: 1, next: 3; data: 3, next: None]] LinkedList[]	Проверка работы основных методов класса
2.	test_list = LinkedList() print (test_list) print (len(test_list)) test_list.append(1) test_list.append(2) print (test_list) test_list.pop() test_list.pop() print (test_list) print (len(test_list))	LinkedList[] 0 LinkedList[length = 2, [data: 1, next: 2; data: 2, next: None]] LinkedList[] 0	Проверка граничных случаев для pop()
3.	test_list = LinkedList() print (test_list) print (len(test_list)) test_list.append(1) test_list.append(2) print (test_list)	LinkedList[] 0 LinkedList[length = 2, [data: 1, next: 2; data: 2, next: None]] LinkedList[length = 1, [data:	Проверка граничных случаев для delete_on_start

	test_list.delete_on_start(1) print (test_list) test_list.delete_on_start(1) print (test_list) print (len(test_list))	2, next: None]] LinkedList[] 0	
4.	test_list = LinkedList() try: test_list.pop() except IndexError: print('OK') test_list.append(1) test_list.append(2) try: test_list.delete_on_start(-1) except KeyError: print('OK') try: test_list.delete_on_start(3) except KeyError: print('OK')	OK OK OK	Проверка работы исключений

Выводы

В рамках данной задачи была реализован связанный однонаправленный список через два класса: Node и LinkedList. Вычислены сложности каждого метода. Выявлены отличия связного списка от массива.

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.__data}, next: {None if self.next==None
else self.next.__data}"


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

    def __len__(self):
        return self.length

    def append(self, element):
        new_node = Node(element)
        if self.length == 0:
            self.head = new_node
            self.length = 1
        else:
            present = self.head
            while(present.next != None):
                present = present.next
            present.next = new_node
            self.length += 1

    def __str__(self):
        if self.length == 0:
            return "LinkedList[]"
        else:
            str_list = f"LinkedList[length = {self.length}, ["
            present = self.head
            while(present.next!=None):
                str_list += str(present)
                str_list += '; '
                present = present.next
            str_list += str(present) + ']]'
            return str_list
```

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

def delete_on_start(self, n):
    if n < 1 or n > self.length:
        raise KeyError("Element doesn't exist!")
    elif n == 1:
        present = self.head
        self.head = present.next
        self.length -= 1
    else:
        present = self.head
        present_index = 1
        while (n != present_index+1):
            present = present.next
            present_index += 1
        present.next = present.next.next
        self.length -= 1

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