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

Студентка гр. 3342 Антипина В.А.

Преподаватель Иванов Д.И.

Санкт-Петербург
2024

Цель работы

Изучение способов реализации структур данных на языке Python.

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 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.

1. Если значение переменной head равна None, метод должен создавать пустой список.
2. 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_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.

Пример того, как должно выглядеть взаимодействие с Вашим связным списком:

```
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
```

Вам не требуется реализовывать создание экземпляров ваших классов и вызов методов, это сделает проверяющая система.

В отчете вам требуется:

1. Указать, что такое связный список. Основные отличия связного списка от массива.
2. Указать сложность каждого метода.
3. Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

Основные теоретические положения

Связный список — это список, каждый элемент которого хранит указатель на следующий элемент списка. Ключевое отличие связного списка от массива заключается в том, что элементы массива хранятся в памяти последовательно (что делает возможной арифметику указателей), а элементы связного списка — нет.

Сложность методов __init__, __str__, get_data — o(1), методов __len__, append, __str__(списка), pop, delete_on_end, clear — n.

Как было указано ранее, в связном списке нельзя обращаться к элементу по индексу, поэтому сложность алгоритма бинарного поиска будет выше для такого списка. (Операция взятия элемента по индексу выполняется за константное время, в связном списке придётся последовательно переходить от одного элементу к другому n/2 раз, каждый раз). Алгоритм может быть реализован так: находится длина списка, определяется индекс среднего элемента. Сохраняется адрес элемента, от которого начинается прохождение массива. Последовательно переходя от первого элемента к элементу с нужным индексом, нужно изменять значение счётчика (заодно можно сравнивать значения, вдруг цикл можно завершить раньше). Если не считать этого отличия, алгоритм в целом аналогичен обычному. Только нужно контролировать значение счётчика, например, чтобы он обнулялся, если элемент лежит левее среднего.

Выполнение работы

Был реализован класс Node. В конструкторе класса определяется приватное поле data поле next для хранения указателя на следующий элемент списка. Метод get_data возвращает значение приватного поля data. Волшебный метод __str__ наследуется от класса-родителя. Осуществляется проверка на то, является ли переданный элемент последним в списке. В зависимости от этого меняется вывод: для обычных элементов выводится значения текущего и следующего элемента, для последнего его значение и NULL.

Был реализован класс LinkedList. Был реализован конструктор, в котором было определено поле head. Волшебный метод __len__ возвращает нуль, если в списке нет элементов. Иначе счётчику присваивается значение 1, в переменную elem копируется значение головы списка, в цикле до ex пор, пока следующий элемент — не указатель на ноль, увеличивается счётчик и переменной присваивается значение поля next. Метод возвращает счётчик.

В методе append создаётся новый экземпляр класса Node, если список пустой, то этот элемент становится головой списка. Иначе осуществляется поиск последнего элемента списка (аналогично тому, как это делалось в предыдущих методах), в поле next последнего элемента записывается указатель на новый элемент списка.

Метод __str__ описывает ожидаемый вывод элементов списка. Если список пустой, возвращается "LinkedList[]", иначе в строку с помощью метода str объекта класса Node записывается каждый элемент списка через разделитель. Возвращается полученная строка.

Метод pop выводит ошибку, если список пустой, если в нём один элемент, «обнуляет» голову списка. Если в списке больше элементов, то осуществляется поиск последнего элемента и ему присваивается значение None.

Был реализован метод delete_on_end. Возвращается ошибка, если нужно удалить элемент с индексом, большим, чем количество элементов в

списке. Иначе, если индекс равен количеству элементов, голова списка начинает указывать на следующий элемент (второй в списке).

Метод clear очищает массив.

Разработанный программный код см. в приложении А.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	linked_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(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	Корректно

Выводы

Были изучены способы реализации структур на языке Python. Был реализован односвязный список элементов класса.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Antipina_Veronika_lb2.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):
        super().__str__()
        if(self.next!=None):
            return "data: {}, next: {}".format(self.__data,self.next.__data)
        else:
            return "data: {}, next: {}".format(self.__data,self.next)


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

    def __len__(self):
        if(self.head==None):
            return 0
        count = 1
        elem = self.head
        while(elem.next!=None):
            count+=1
            elem = elem.next
        return count

    def append(self, element):
        new_elem = Node(element)
        if(self.head==None):
            self.head = new_elem
        else:
            elem = self.head
            while(elem.next!=None):
                elem = elem.next
            elem.next = new_elem

    def __str__(self):
        if(self.head == None):
            return "LinkedList[]"
        else:
            result = "LinkedList[length = {}, [".format(len(self))
            elem = self.head

            while(elem.next!=None):
```

```
            result+=str(elem)+'; '
            elem = elem.next

        result+=str(elem)
    return result+']]'

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


def delete_on_end(self, n):
    if(n>len(self) or n<=0):
        raise KeyError("Element doesn't exist!")
    else:
        length = len(self)
        idx = length - n
        counter = 0
        elem = self.head
        if(idx==0):
            self.head = self.head.next

        while(elem.next!=None):
            if(counter+1==idx):
                elem.next = elem.next.next
            if(elem.next!=None):
                elem = elem.next
                counter+=1

def clear(self):
    elem = self.head
    if(elem!=None):
        while(elem.next!=None):
            elem.date = None
            next_el = elem.next
            elem.next = None
            elem = next_el
        self.head = None
```