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. 3342 _____ Русанов А.И.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Изучить алгоритмы и структуры данных в Python, научиться использовать их для решения практических задач. Реализовать связный однонаправленный список и реализовать функционал для работы с ним.

Задание

В данной лабораторной работе Вам предстоит реализовать связный однонаправленный список.

Node

Класс, который описывает элемент списка.

Класс Node должен иметь 2 поля:

__data # данные, приватное поле

__next__ # ссылка на следующий элемент списка

Вам необходимо реализовать следующие методы в классе Node:

__init__(self, data, next)

конструктор, у которого значения по умолчанию для аргумента next равно None.

get_data(self)

метод возвращает значение поля __data.

change_data(self, new_data)

метод меняет значение поля __data.

__str__(self)

перегрузка метода __str__. Описание того, как должен выглядеть результат вызова метода смотрите ниже в примере взаимодействия с Node.

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.

Класс LinkedList должен иметь 2 поля:

__head__ # данные первого элемента списка

__length # количество элементов в списке

Вам необходимо реализовать конструктор:

__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.

и следующие методы в классе LinkedList:

__len__(self)

перегрузка метода __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 метода __str__. Описание того, как должен выглядеть результат вызова метода смотрите ниже в примере взаимодействия с LinkedList.

pop(self)

удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.

clear(self)

очищение списка.

change_on_end(self, n, new_data)

меняет значение поля __data n-того элемента с конца списка на new_data. 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) # 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.

Выполнение работы

Класс Node. Имеет два переопределенных метода: инициализация и строковый вид (выводится значение текущего элемента, далее, если есть следующий элемент в списке, его значение выводится после «next», иначе выводится только значение текущего элемента и None после «next»), а также методы возврата значения элемента класса и изменения значения элемента.

Класс LinkedList.

Метод __init__ — если аргумент head равен None, то создается пустой список, иначе список из одного элемента.

Метод __len__ — возвращает поле __length элемента класса.

Метод append — создает элемент класса Node, если список был пустым, то это становится первым элементом, иначе в цикле доходит до последнего элемента списка и новый элемент добавляется в конец. Значение поля __length увеличивается.

Метод __str__ — переопределяется метод класса object, выводится форматная строка, при этом в методе в цикле используется метод __str__ класса Node.

Метод pop — удаляется последний элемент в списке, если список пустой — выбрасывается исключение. Если элемент в списке один, полю __head__ присваивается значение None, длина списка обнуляется. Если длина списка больше одного, полю __next__ предпоследнего элемента (до него доходим с помощью while) присваивается значение None.

Метод change_on_end — значение n-того элемента с конца списка меняется на new_data. Достигается это с помощью цикла while, проверяется, что переменная cnt равна разнице между длиной списка и значением n, и тогда вызывается метод change_data.

Метод clear — обнуляется длина списка и полю __head__ присвается None (список удаляется).

Сложности методов:

Класс Node:

• __init__ — O(1);

• get_data — O(1);

• __str__ — O(1).

Класс LinkedList:

• __init__ — O(n);

• __len__ — O(1);

• append — O(n);

• __str__ — O(n);

• pop — O(n);

• change_on_end — O(n);

• clear — O(1).

Ответы на вопросы:

1. Указать, что такое связный список. Основные отличия связного списка от массива.

Связный список — это структура данных для хранения набора значений. Элементы связного списка могут быть расположены как угодно в памяти, не последовательно. Связь между элементами осуществляется не посредством индексов, то есть «расстоянии» от первого элемента, как в массиве, а с помощью ссылок в каждом из элементов на следующий элемент (или None, если элемент последний). Если необходимо использовать индекс для обращения к элементу, то придется все равно прибегать к циклу и длине списка.

При использовании массива элементы хранятся в памяти непрерывно, то есть рядом друг с другом, а в связном списке — как угодно.

2. Указать сложность каждого метода.

Для массива необходимо, чтобы была свободная память подряд, в этом сложность. Сложность использования связного списка — если он односвязный, нельзя идти с конца, только с начала в сторону конца, но для этого можно использовать двусвязный список. Также в связном списке нужно хранить ссылки на элементы, а в массиве только значения элементов.

3. Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

В отсортированном списке находить середину с помощью деления длины списка на два. Далее запоминать ссылку на средний элемент. После перемещения в середину одной из «половин» (после сравнения искомого и среднего элементов ><=) считать, сколько элементов остается до предыдущей середины, если мы оказались в первой половине (делением количества элементов в половине на два) и не идти дальше, чем стоит предыдущий средний элемент. Также в каждом из узлов можно хранить индекс элемента, чтобы далее можно было вернуть его при нахождении совпадения с искомым элементом.

В целом это неэффективно, так как в любом случае придется проходить по всему списку (а то и не один раз). Проще сразу пройти по всему списку либо конвертировать его в массив.

В массиве бинарный поиск работает с помощью сравнения элементов и вычисления нужных индексов. В нем не обязательно проходить по всему списку.

Разработанный программный код см. в приложении А.

Выводы

Были изучены принципы работы с линейными списками в Python, создан такой список. Реализованы методы для добавления, удаления элементов из него. Также реализованы методы для подсчета количества элементов списка, замены значения элемента на новое по индексу и вывода значений элементов списка.

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 change_data(self, new_data):
        self.__data = new_data

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


class LinkedList:
    def __init__(self, head=None):
        self.__head__ = head
        self.__length = 1 if head is not None else 0

    def __len__(self):
        return self.__length

    def append(self, element):
        node = Node(element)
        current = self.__head__
        if current is None:
            self.__head__ = node
        else:
            while (current.next != None):
                current = current.next
            current.next = node
        self.__length += 1

    def __str__(self):
        if self.__length == 0:
            return "LinkedList[]"
        else:
            res = []
            current = self.__head__
            while (current is not None):
                res.append(current.__str__())
                current = current.next
            return  f"LinkedList[length  =  {self.__length},  [{';
'.join(res)}]]"

    def pop(self):
        if self.__length == 0:
            raise IndexError('LinkedList is empty!')
        elif self.__length == 1:
```

```
            self.__head__ = None
            self.__length = 0
        elif self.__length > 1:
            current = self.__head__
            while (current.next.next is not None):
                current = current.next
            current.next = None
            self.__length -= 1

    def change_on_end(self, n, new_data):
        if self.__length < n or n <= 0:
            raise KeyError(f"{n} doesn't exist!")
        else:
            current = self.__head__
            cnt = 0
            while cnt != self.__length - n:
                current = current.next
                cnt += 1
            current.change_data(new_data)

    def clear(self):
        self.__head__ = None
        self.__length = 0
```