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. 3343 _____ Лобова Е. И.

Преподаватель _____ Иванов Д. И.

Санкт-Петербург
2024

Цель работы

Целью работы является знакомство с различными структурами данных, алгоритмами и их сложностью, а также реализация связанного однонаправленного списка на языке Python.

Задание

Вариант 3

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).
- **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.

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.
- **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)
print(linked_list) # LinkedList[length = 1, [data: 10, next: None]]
print(len(linked_list)) # 1
```

Выполнение работы

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

Связный список - структура данных, каждый из элементов которой содержит как собственные данные, так и некоторое количество ссылок на следующий и/или предыдущий узел списка. Его главное отличие от массива в том, что в списке могут хранится объекты любого типа, а в массиве только одинакового, также он не хранится в памяти последовательно, как массив. Также сложность некоторых методов для массива и связанных списков будет отличаться.

2. Указать сложность каждого метода.

Класс Node:

- __init__(self) - O(1)
- get_data(self) – O(1)
- change_data(self, new_data) – O(1)
- __str__(self) – O(1)

Класс LinkedList:

- __init__(self) - O(1)
- __len__(self) – O(1)
- append(self, element) – O(n)
- __str__(self) – O(n)
- pop(self) – O(n)
- change_on_end(self, n, new_data) – O(n)
- clear(self) – O(1)

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

Бинарный поиск в связном списке отличается от бинарного поиска в классическом списке Python (например, в виде списка или массива) из-за различной структуры данных.

Для классического списка Python можно использовать индексы для доступа к элементам, что упрощает бинарный поиск, в то время как в связном списке необходимо выполнять обход узлов для доступа к элементам.

Из-за необходимости последовательного обхода элементов в связном списке, сложность выполнения бинарного поиска увеличивается по сравнению с классическим списком.

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.	node = Node(1) print(node) print(node) print(node.get_data()) l_l = LinkedList() print(l_l) print(len(l_l)) l_l.append(111) l_l.append(222) l_l.append(333) print(l_l) print(len(l_l)) l_l.pop() print(l_l) l_l.pop() print(l_l) l_l.pop() print(l_l) l_l.append(111) l_l.append(222) l_l.append(333) l_l.change_on_end(1, 3) print(l_l)	data: 1, next: None 1 LinkedList[] 0 LinkedList[length = 3, [data: 111, next: 222; data: 222, next: 333; data: 333, next: None]] 3 LinkedList[length = 2, [data: 111, next: 222; data: 222, next: None]] LinkedList[length = 1, [data: 111, next: None]] LinkedList[] LinkedList[length = 3, [data: 111, next: 222; data: 222, next: 3; data: 3, next: None]]	Методы обоих классов работают корректно при корректных поданных данных.
2.	try: l_l = LinkedList() l_l.append(10) l_l.append(20) l_l.append(30) l_l.append(40)	OK	При некорректных введенных данных срабатывает исключение.

	l_l.change_on_end(-1, 2) print(l_l) except (KeyError, ValueError): print('OK')		

Выводы

Были изучены различных структуры данных и сложности их основных методов. Также была написана программа, в соответствии с заданным вариантом, в которой с помощью классов реализован однонаправленный связанный список с различными методами.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
class Node:
    def __init__(self, data, next = None):
        self.next = next
        self.__data = data

    def get_data(self):
        return self.__data

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

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

class LinkedList:
    def __init__(self, head = None):
        self.head = head
        if self.head == None:
            self.length = 0
        else: 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
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
        self.length += 1

    def __str__(self):
        if self.length == 0:
            return "LinkedList[]"
        else:
            s = f"LinkedList[length = {self.length}, ["
            current = self.head
            while current.next:
                s    +=    f"data:    {current.get_data()},    next:
{current.next.get_data()}; "
                current = current.next
            s += f"data: {current.get_data()}, next: None]]"
        return s
```

```
def pop(self):
    if self.length == 0:
        raise IndexError("LinkedList is empty!")
    elif self.length == 1:
        self.head = None
        self.length -= 1
    else:
        current = self.head
        while current.next.next:
            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("Element doesn't exist!")
    else:
        current = self.head
        for i in range(self.length - n):
            current = current.next
        current.change_data(new_data)

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