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. 3344 _____ Пачев Д.К.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Написать программу на языке Python, реализующую связный однонаправленный список.

Задание

Вариант 2.

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:

- **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.

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.

o **clear(self)** - очищение списка.

o **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.

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

Класс Node – класс для представления элементов связного списка. Внутри реализованы:

- метод __init__() для инициализации полей __data и next
- метод get_data() для получения поля __data
- метод change_data() для изменения поля __data
- а также переопределен метод __str__()

Класс LinkedList – класс для реализации связного однонаправленного списка.

Внутри реализованы:

- метод __init__() для инициализации полей head и length
- переопределен метод __len__(), который возвращает поле length
- метод append() – добавляет элемент в список с помощью цикла while
- метод pop() – удаляет последний элемент с помощью цикла for
- метод clear() – очищает список, устанавливая нулевую длину списка и присваивая голове списка значения None
- метод delete_on_start() – удаляет n-тый элемент с начала списка с помощью цикла for
- переопределен метод __str__() для представления списка в строковом формате

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

Связный список - это линейная структура данных, где элементы, или узлы, связаны с помощью указателей. Основные отличия между связным списком и массивом включают:

1. Выделение памяти: Для связных списков не требуется выделение непрерывной области памяти, в отличие от массивов, которым нужны смежные блоки памяти.

2. Динамический размер: Связные списки могут легко менять размер во время выполнения программы, в то время как у массивов размер фиксирован после выделения.
3. Вставка и удаление: Вставка или удаление элементов в связном списке эффективна, особенно в середине, т.к. она включает обновление указателей, в отличие от массивов, где такие операции могут быть затратными.
4. Время доступа: Массивы обеспечивают доступ к элементам по индексам за постоянное время, в то время как в связных списках для доступа к конкретному элементу нужно пройти от головы, что приводит к медленным временам доступа для произвольного доступа.

2) Указать сложность каждого метода.

   O(1):

   __init__()

   get_data()

   change_data()

   Node.__str__()

   __len__

   clear()

   O(n):

   append()

   pop()

   delete_on_start()

   LinkedList.__str__()

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

   1) Установим указатели на начало и конец интервала поиска (начало - голова списка, конец - последний элемент).

2) На каждом шаге бинарного поиска вычисляем среднюю точку текущего интервала.
3) Сравниваем значение средней точки с ключом поиска и соответственно сужаем интервал, двигая указатели.
4) Повторяем шаги 3-4 до тех пор, пока ключ не будет найден или интервал сужен до одного элемента.

Отличие от реализации бинарного поиска в классическом списке Python заключается в том, что для связного списка нет прямого доступа по индексам, и поэтому каждый шаг поиска требует перехода от головы списка к середине с учетом указателей. Это делает реализацию более сложной и требующей дополнительных операций по сравнению с обычным списком Python, где можно просто обращаться к элементам по индексу для бинарного поиска

Тестирование

Результаты тестирования представлены в Таблице 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(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	верно

Выводы

В ходе выполнения лабораторной работы была разработана программа на языке Python, которая реализует связный однонаправленный список.

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, data):
        self.__data = data

    def __str__(self):
        if self.next is not None:
            return f'data: {self.__data}, next:
{self.next.get_data()}'
        return f'data: {self.__data}, next: {self.next}'


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

    def __len__(self):
        return self.length

    def append(self, element):
        node = Node(element)
        if self.head is None:
            self.head = node
            self.length += 1
        else:
            item = self.head
            while item:
                if item.next is None:
                    item.next = node
                    self.length += 1
                    break
                item = item.next

    def pop(self):
        if self.length == 0:
            raise IndexError('LinkedList is empty!')
        self.length -= 1
        if self.length == 0:
            self.head = None
        else:
            current = self.head
            for i in range(self.length - 1):
                current = current.next
            current.next = None
```

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

    def delete_on_start(self,n):
        if n > self.length or n < 1:
            raise KeyError("Element doesn't exist!")
        if n == 1:
            self.head = self.head.next
        else:
            current = self.head
            for i in range(n - 2):
                current = current.next
            current.next = current.next.next
        self.length -= 1

    def __str__(self):
        if self.length == 0:
            return 'LinkedList[]'
        items = []
        item = self.head
        while item:
            items.append(f'data: {item.get_data()}, next:
{item.next.get_data() if item.next else None}')
            item = item.next
        return f"LinkedList[length = {self.length}, [{';
'.join(items)}]]"
```