**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

|  |  |  |
| --- | --- | --- |
| Студент гр. 3341 |  | Перевалов.П.И. |
| 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, а также методами для инициализации объекта, получения данных и перегрузки метода str для удобного вывода информации объекта.

2. Создание класса LinkedList, который описывает связанный однонаправленный список с полями головного элемента и количеством элементов списка. Также требуется реализовать методы инициализации объекта, получения длины списка, добавления элемента в конец списка, перегрузки метода str для вывода списка в строковом представлении, удаления последнего элемента, очистки списка и удаления n-того элемента с начала 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:

o data # Данные элемента списка, приватное поле.

o next # Ссылка на следующий элемент списка.

И следующие методы:

o \_\_init\_\_(self, data, next) - конструктор, у которого значения по умолчанию для аргумента next равно None.

o get\_data(self) - метод возвращает значение поля data (это необходимо, потому что в идеале пользователь класса не должен трогать поля класса Node).

o \_\_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:

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.

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.

## Основные теоретические положения

Связанный список (LinkedList) - это структура данных, которая состоит из узлов, каждый из которых содержит данные и ссылку на следующий узел в списке. Основные положения о LinkedList включают:

1. Динамичность: LinkedList динамически растет и сжимается по мере добавления или удаления элементов, так как каждый узел содержит ссылку на следующий узел.

2. Быстрое добавление и удаление: Вставка или удаление элементов в LinkedList имеет константную сложность O(1), в отличие от массива, где эти операции могут быть более сложными (в зависимости от позиции элемента).

3. Медленный доступ к элементам: Для доступа к элементам LinkedList нужно итерироваться с начала списка до нужного элемента, что делает доступ к элементу в LinkedList медленнее, чем в массиве.

4. Не непрерывная память: Узлы в LinkedList распределены в памяти не последовательно, поэтому нет гарантии, что они будут храниться в соседних ячейках памяти.

5. Внедрение: LinkedList обычно используется, когда необходимо часто добавлять и удалять элементы, и доступ по индексу не так важен.

Это основные положения о LinkedList как структуре данных.

## Выполнение работы

Ход работы к данному коду можно описать следующим образом:

1. Определение класса Node:

- Создание класса Node с атрибутами data (данные элемента) и next (ссылка на следующий элемент).

- Определение метода \_\_init\_\_() для инициализации объекта класса Node с передачей данных и ссылки на следующий элемент.

- Определение метода get\_data(), который возвращает данные элемента.

- Определение метода \_\_str\_\_(), который возвращает строковое представление данных и ссылки на следующий элемент объекта Node.

2. Определение класса LinkedList:

- Создание класса LinkedList с атрибутами head (головной элемент списка) и length (длина списка).

- Определение метода \_\_init\_\_() для инициализации объекта класса LinkedList, устанавливающего головной элемент и длину списка.

- Определение метода len(), который возвращает длину списка.

- Определение метода \_\_str\_\_(), который возвращает строковое представление списка.

- Определение метода append(), добавляющего новый элемент в конец списка.

- Определение метода pop(), удаляющего последний элемент из списка.

- Определение метода delete\_on\_start(), удаляющего n-тый элемент с начала списка.

- Определение метода clear(), очищающего список (удаляющего все элементы).

3. Дополнительные пояснения:

- Метод \_\_str\_\_() выводит список в красивом виде, обходя все элементы списка.

- Метод append() добавляет новый элемент, обходя список до конца.

- Метод pop() удаляет последний элемент, обходя список до предпоследнего.

- Метод delete\_on\_start() удаляет n-тый элемент, пересчитывая указатели.

- Метод clear() устанавливает head в None и длину в 0.

## Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 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 | 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 | Проверка работы основных методов класса |

## Выводы

В рамках данной задачи была реализован связанный однонаправленный список через два класса: Node и LinkedList.

# Приложение А Исходный код программы

Название файла: main.py

class Node:

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: {self.next.data if (self.next != None) else None}"

class LinkedList:

head = []

length = 0

def \_\_init\_\_(self, head = None):

if head != None:

self.head = head

self.length += 1

def \_\_len\_\_(self):

return self.length

def append(self, element):

if self.head == []:

self.head = Node(element)

else:

current = self.head

while current.next:

current = current.next

current.next = Node(element)

self.length += 1

def \_\_str\_\_(self):

if self.length == 0:

return "LinkedList[]"

str = f"LinkedList[length = {self.length}, [{self.head.\_\_str\_\_()}"

current = self.head

while current.next:

current = current.next

str += f"; {current.\_\_str\_\_()}"

str += "]]"

return str

def pop(self):

if self.length == 0:

raise IndexError("LinkedList is empty!")

if self.length == 1:

self.head = []

self.length = 0

return

counter = 1

current = self.head

while counter != self.length - 1:

current = current.next

counter += 1

current.next = None

self.length -= 1

def delete\_on\_start(self, n):

n -= 1

if (n < 0 or self.length <= n):

raise KeyError("<element> doesn't exist!")

if (n == 0):

self.head = self.head.next

self.length -= 1

return

current = self.head

for i in range(n-1):

current = current.next

current.next = current.next.next

self.length -= 1

def clear(self):

self.head = None

self.length = 0