**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. Вариант 1

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Коршков А.А. |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург

2024

## Цель работы

Научиться создавать класс односвязного списка на языке программирования Python, описывать элемент списка, методы для изменения элементов в односвязном списке.

## 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.

**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.
* **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.

## Выполнение работы

Связный список - это структура данных, которая состоит из узлов, каждый из которых содержит данные и ссылку (или указатель) на следующий узел в списке. Последний узел обычно указывает на NULL или None, что означает конец списка.

Основные отличия связного списка:

* Связный список может динамически изменяться, добавляя или удаляя элементы без необходимости перемещения всех элементов, в отличие от массива, который требует перекопирования при изменении размера.
* Вставка и удаление элементов в середине связного списка более эффективны, так как не требуется сдвиг всех последующих элементов, в отличие от массива, где это может быть затратно по ресурсам.

Первым делом создаётся класс Node, в функции\_\_init\_\_ создаётся приватное поле data и поле next, которое будет содержать указатель на следующий элемент. Чтобы получить значение элемента списка используется метод get\_data. При выводе объекта данного класса в консоль будет выведена информация о полях data и next. Сложность данных методов \_\_init\_\_, get\_data, \_\_str\_\_ O(1).

Затем создаётся LinkedList с полями head и length. Поле head будет содержать в себе первый объект класса Node (первый элемент списка), length – длину всего списка. Сложность метода \_\_init\_\_ O(1).

Чтобы добавить элемент в связный список создадим объект класса Node. Если поле head пустое (список пустой), то в head запишем созданный объект, иначе создадим переменную current и с помощью поля next будем совершать сдвиг, до тех пор, пока next не будет равно None, после чего в поле next записываем добавляемый элемент, длину всего списка увеличиваем на 1. Сложность данного метода append O(n), потому что необходимо пройтись по всем элементам списка.

Чтобы вывести информацию о списке, необходимо пока в поле next есть элемент выводить поле data у текущего и у следующего элемента (с помощью get\_data()). Если элемент последний, то в выводе следующего элемента будет None. Сложность данного метода O(n).

Чтобы удалить последний элемент из списка, нужно, чтобы у предпоследнего элемента поле next стало None, длина уменьшается на 1. Сложность данного метода pop O(n), потому что необходимо пройтись по всем элементам списка.

Для удаления определённого элемента с конца, необходимо создать переменную end, чтобы определить, какой элемент нужно удалить сначала (список односвязный, храниться первый элемент списка), с помощь переменной position можно определить индекс элемента, который рассматривается в данный момент. Если end и position совпадают, то в поле head помещается следующий элемент списка, длина уменьшается на 1, в противном случае необходимо пройтись по списку, пока position не равен end, после чего в поле элемента next помещается элемент, который стоит после удалемого элемента (т.е. current.next = current.next.next). Сложность данного метода delete\_on\_end O(n), т.к. придётся пройтись по списку до необходимого элемента.

Чтобы полностью очистить список необходимо очистить поле head, отвечающий за первый элемент списка, и поле length обнулить. Сложность метода clear O(1).

Реализация бинарного поиска для связного списка. Предположим, что элементы уже отсортированы по возрастанию. Создадим переменные start = 0 и end = self.length - 1 для определения границ искомого элемента, position = 0, чтобы понимать, на каком месте элемент, mid = (start + end) / 2 – средний элемент и current – текущий элемент связного списка. Пока start <= end, с помощью переменных position и current, определяем средний элемент, если искомый элемент равен ему, то возвращаем значение (self.current), в противном случае если искомый элемент больше среднего, то start = mid + 1, иначе end = mid - 1, позиция и сам элемент возвращаются в исходную позицию position = 0, current = self.head. Если элемент не был найден, то функция бинарного поиска вернёт -1.

Основное отличие бинарного поиска связного списка от классического списка python, в том, что необходимы дополнительные переменные для нахождения среднего элемента, с которым идёт сравнение искомого элемента. В классическом списке достаточно обратиться по индексу к элементу списка, а связном списке нужно пройтись по всем элементам, пока не будет найден средний.

## Выводы

Была написана программа, которая содержит в себе реализацию односвязного линейного списка. В ней можно создавать список, добавлять и удалять элементы в различных позициях, а также выводить информацию о каждом элементе 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 \_\_str\_\_(self):

if self.next is None:

return f"data: {self.\_\_data}, next: None"

return f"data: {self.\_\_data}, next: {self.next.get\_data()}"

class LinkedList:

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

self.head = head

self.length = 0

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

return self.length

def append(self, element):

new = Node(element)

if self.head is None:

self.head = new

self.length += 1

return

current = self.head

while current.next:

current = current.next

current.next = new

self.length += 1

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

if self.length == 0:

return "LinkedList[]"

res = ""

current = self.head

while current.next:

res += f"data: {current.get\_data()}, next: {current.next.get\_data()}; "

current = current.next

res += f"data: {current.get\_data()}, next: None"

return f"LinkedList[length = {self.length}, [{res}]]"

def pop(self):

if self.head is None:

raise IndexError("LinkedList is empty!")

if self.length == 1:

self.head = None

self.length -= 1

return

current = self.head

while current.next.next:

current = current.next

current.next = None

self.length -= 1

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

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

raise KeyError("Element doesn't exist!")

current = self.head

end = self.length - n

position = 0

if position == end:

self.head = self.head.next

self.length -= 1

return

while current is not None and position + 1 != end:

position = position + 1

current = current.next

if current is not None:

current.next = current.next.next

self.length -= 1

def clear(self):

self.head = None

self.length = 0

# Приложение Б тЕСТИРОВАНИЕ

**Тест №1**

Входные данные (программа):

node = Node(1)

print(node)

node.next = Node(2, None)

print(node)

Выходные данные:

data: 1, next: None

data: 1, next: 2

Комментарий:

Тест показывает, что элемент однонаправленного списка инициализируется корректно.

**Тест №2**

Входные данные (программа):

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))

Выходные данные:

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

Комментарий:

Проверка инициализации списка, затем добавляется элемент, проверяется, что вывод корректен, добавляется и удаляется следующий элемент, результаты этих преобразований видны.

**Тест №3**

Входные данные (программа):

linked\_list = LinkedList()

linked\_list.append(10)

print(linked\_list, len(linked\_list))

linked\_list.append(20)

linked\_list.append(30)

linked\_list.pop()

print(linked\_list, len(linked\_list))

linked\_list.append(40)

linked\_list.append(50)

linked\_list.append(60)

linked\_list.delete\_on\_end(2)

print(linked\_list, len(linked\_list))

linked\_list.clear()

print(linked\_list, len(linked\_list))

Выходные данные:

LinkedList[length = 1, [data: 10, next: None]] 1

LinkedList[length = 2, [data: 10, next: 20; data: 20, next: None]] 2

LinkedList[length = 4, [data: 10, next: 20; data: 20, next: 40; data: 40, next: 60; data: 60, next: None]] 4

LinkedList[] 0

Комментарий:

Инициализирован список, добавлен один элемент и вывод результата, затем добавляется два элемента и последний удаляется, затем добавляется 3 элемента и удаляется второй с конца, затем список очищается.