**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.

|  |  |  |
| --- | --- | --- |
| Студент гр. 3344 |  | Бубякина Ю.В. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Получить опыт реализации структур данных и алгоритмов работы с ними.

## 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   change\_data(self, new\_data) - метод меняет значение поля data объекта 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:

        “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   change\_on\_start(self, n, new\_data) - изменение поля data n-того элемента с НАЧАЛА списка на new\_data. Метод должен выбрасывать исключение KeyError,  с сообщением "Element doesn't exist!", если количество элементов меньше n.

## Выполнение работы

1. Связный список — это структура данных, в которой элементы отсортированы по порядку с помощью указателей. Отличие от массива в том, что массив может хранить только данные одного типа; также массив хранится в одном участке памяти, а элементы списка в различных.
2. Для класса Node:

* get\_data() – O(1)
* change\_data – O(1)

Для класса LinkedList:

* append() – O(n)
* pop() – O(n)
* change\_on\_start() – O(n)
* clear() – O(1)

1. Реализация бинарного поиска:

Функция находит средний элемент и сравнивает его с входным. Если элемент ему равен, то функция завершается, иначе функция вызывает себя с указанием левой или правой части связного списка (выбор зависит от сравнения среднего элемента с входным).

Разница бинарного поиска в связном списке и классическом списке состоит в том, что для поиска элементов в классическом списке можно использовать их индексы, что ускоряет работы функции. Для связного списка выполнение функции будет дольше.

## Выводы

Получен опыт в реализации связного списка в языке Python при помощи классов. Изучены способы создания алгоритмов для работы списка.

# Приложение А Исходный код программы

Название файла: Bubyakina\_Yuliya\_lb2.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):

if self.next:

return f"data: {self.data}, next: {self.next.data}"

return f"data: {self.get\_data()}, next: None"

class LinkedList:

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

self.head = head

self.length = 0

while head:

head = head.next

self.length += 1

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

return self.length

def append(self, element):

if self.head:

tmp = self.head

while tmp.next:

tmp = tmp.next

tmp.next = Node(element)

else:

self.head = Node(element)

self.length += 1

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

if self.length:

s = ''

tmp = self.head

while tmp:

s += tmp.\_\_str\_\_() + '; '

tmp = tmp.next

return f"LinkedList[length = {self.length}, [{s[:-2]}]]"

else:

return "LinkedList[]"

def pop(self):

if not self.head:

raise IndexError("LinkedList is empty!")

if self.head.next:

tmp = self.head

while tmp.next.next:

tmp = tmp.next

tmp.next = None

else:

self.head = None

self.length -= 1

def change\_on\_start(self, n, new\_data):

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

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

tmp = self.head

for i in range(1, n):

tmp = tmp.next

tmp.data = new\_data

def clear(self):

self.head = None

self.length = 0