**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 |  | Сербиновский Ю.М. |
| 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. Был реализован связной список, то есть список, в котором элементы связаны с помощью указателей друг на друга. В отличие от массива, в связном списке могут храниться элементы разного размера и типа, однако эти элементы хранятся в памяти неупорядоченно.
2. Сложность методов класса Node:

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

Сложность методов класса LinkedList:

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

1. Бинарный поиск в связном списке:

При выполнении бинарного поиска в связном списке функция ищет средний элемент, сравнивает его с входным значением. Если значение совпадает, функция завершается; в противном случае, она вызывает саму себя для левой или правой части связного списка в зависимости от результата сравнения среднего элемента и входного значения.

В классическом списке для поиска элементов можно использовать индексы, что значительно ускоряет процесс выполнения поиска. В случае связного списка, такая возможность отсутствует, что делает выполнение бинарного поиска более время затратным по сравнению с классическим списком, где доступ к элементам осуществляется напрямую по индексам.

## Выводы

На основе двух классов был реализован связной список на языке Python. Был реализован API для связного 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 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.\_\_data}, next: None"

class LinkedList:

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

self.\_head = \_head

self.\_length = 0

tmp = self.\_head

while tmp:

self.\_length += 1

tmp = tmp.next

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, None)

else:

self.\_head = Node(element, None)

self.\_length += 1

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

if self.\_length == 0:

return "LinkedList[]"

else:

rtr\_str = f"LinkedList[length = {self.\_length}, ["

tmp = self.\_head

while tmp.next:

rtr\_str += tmp.\_\_str\_\_() + "; "

tmp = tmp.next

rtr\_str += tmp.\_\_str\_\_() + "]]"

return rtr\_str

def pop(self):

if self.\_length == 0:

raise IndexError("LinkedList is empty!")

elif self.\_length == 1:

self.\_head = None

self.\_length -= 1

else:

tmp = self.\_head

while tmp.next.next:

tmp = tmp.next

tmp.next = None

self.\_length -= 1

def clear(self):

self.\_head = None

self.\_length = 0

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

if self.\_length < n or n <= 0:

raise KeyError("Index out of the range")

else:

tmp = self.\_head

for i in range(n-1):

tmp = tmp.next

tmp.change\_data(new\_data)