# МИНОБРНАУКИ РОССИИ

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

# «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

**Кафедра МОЭВМ**

# ОТЧЕТ

**по лабораторной работе №2 по дисциплине «Информатика»**

# Тема: «Алгоритмы и структуры данных в Python»

|  |  |  |
| --- | --- | --- |
| Студент гр. 3342 |  | Белаид Фарук |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург 2024

# Цель работы

Изучить алгоритмы и структуры данных в Python, способы их реализации. Написать согласно заданию программу, которая реализует структуру однонаправленного связного списка в виде класса.

# 

# Задание

В данной лабораторной работе Вам предстоит реализовать связный

***однонаправленный*** список.

*Node*

Класс, который описывает элемент списка. Класс Node должен иметь 2 поля:

**data** - данные, приватное поле

**next** - ссылка на следующий элемент списка

Вам необходимо реализовать следующие методы в классе Node:

**init (self, data, next)** - конструктор, у которого значения по умолчанию для аргумента next равно None.

**get\_data(self)** - метод возвращает значение поля data.

**change\_data(self, new\_data)** - метод меняет значение поля data.

**str (self)** - перегрузка метода str . Описание того, как должен выглядеть результат вызова метода смотрите ниже в примере взаимодействия с Node.

*Linked List*

Класс, который описывает связный однонаправленный список. Класс LinkedList должен иметь 2 поля:

**head** - данные первого элемента списка

**length** - количество элементов в списке Вам необходимо реализовать конструктор:

**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.

и следующие методы в классе LinkedList:

**len (self)** - перегрузка метода 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 метода str . Описание того, как должен выглядеть результат вызова метода смотрите ниже в примере взаимодействия с LinkedList.

**pop(self)** - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.

**clear(self)** - очищение списка.

**change\_on\_end(self, n, new\_data)** - меняет значение поля data n- того элемента с конца списка на new\_data. Метод должен выбрасывать исключение KeyError, с сообщением "<element> doesn't exist!", если количество элементов меньше n.

# Основные теоретические положения

Связный список – это структура данных, которая состоит из узлов, включающих какое-нибудь содержимое и указатель на следующий элемент. Отличием связного списка от массива заключается в том, что у списка элементы в памяти не упорядочены, он может хранить содержимое разных типов данных, обладает более быстрыми вставкой и удалением элемента, о при этом обращение по индексу происходит медленнее.

**Выполнение работы**

В ходе выполнения работы были реализованы следующие классы и их методы:

*Node*

Поля:

* *data* – содержимое элемента списка
* *next* – ссылка на следующий элемент списка Методы:
* *init (self, data, next=None)* – конструктор.
* *get\_data(self)* – возвращает значение поля *data*. O(1)
* *change\_data(self, new\_data)* – меняет значение поля *data*. O(1)
* *str (self)* – перегрузка метода, возвращает строку с информацией об элементе. O(1)

*Linked List*

Поля:

* *head* – ссылка на первый элемент списка
* *length* – количество элементов в списке Методы:
* *init (self, head=None)* – создаёт пустой список или состоящий из одного элемента в зависимости от того, передано ли head.
* *len (self)* – перегрузка метода, возвращает значение *length*. O(1)
* *append(self, element)* – добавляет элемент в конец списка, при помощи цикла while доходит до последнего элемента и его полю *next* присваивает ссылку на следующий элемент. O(n)
* *str (self)* – перегрузка метода, возвращает строку с информацией о списке (получает информацию, проходя по списку). O(n)
* *pop(self)* – удаляет последний элемент, а в случае, если список пустой, выбрасывает исключение IndexError с сообщением "LinkedList is empty!". O(n)
* *change\_on\_end(self, n, new\_data)* – изменяет содержимое n-того с конца элемента, доходя до него в цикле while и изменяя значение про помощи метода элемента *change\_data*, а в случае неверного индекса выбрасывает исключение KeyError, с сообщением "<element> doesn't exist!". O(n)
* *clear(self*) – очищает список, присваивая первому элементу None и изменяя длину на 0. O(1)

Алгоритм бинарного поиска работает с отсортированными структурами данных, что означает то, что связный список для начала нужно отсортировать. Двоичный поиск заключается в том, что находится середина списка и сравнивается элемент в середине с искомым, в случае, если искомый больше то находим середину уже правой части, иначе левой. Так продолжается, пока не сойдётся всё до одного элемента, который и будет являться тем, который искали. Сложность бинарного поиска в списке заключается в том, что не можем быстро найти серединный элемент, потому что не можем брать по индексу, нам придётся каждый раз проходить половину рабочей области. В итоге сложность бинарного поиска будет больше O(n), что делает его бесполезным в работе со списками, так как пройти по нему будет проще и быстрее.

Разработанный программный код см. в приложении А.

# 

# Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | node = Node(1) print(node)  node. next = Node(2, None) print(node) | data: 1, next: None data: 1, next: 2 | Верно |

|  |  |  |  |
| --- | --- | --- | --- |
| 2. | l\_l = LinkedList() print(l\_l) l\_l.append(10) l\_l.append(20) l\_l.append(30) l\_l.append(40) print(l\_l) print(len(l\_l))  l\_l.change\_on\_end(2, 3)  l\_l.change\_on\_end(4, 100) print(l\_l) | LinkedList[]  LinkedList[length = 4, [data: 10,  next: 20; data: 20, next: 30; data:  30, next: 40; data: 40, next: None]]  4  LinkedList[length = 4, [data: 100,  next: 20; data: 20, next: 3; data: 3,  next: 40; data: 40, next: None]] | Верно |

# 

**Выводы**

Были изучены алгоритмы и структуры данных в Python и способы их реализации. Согласно заданию была разработана программа, которая реализует однонаправленный связный список, представляющий собой класс с соответствующими полями и методами.

# ПРИЛОЖЕНИЕ А

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

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 != 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):

if head is None:

self.\_\_head\_\_ = None

self.\_\_length = 0

else:

self.\_\_head\_\_ = Node(head)

self.\_\_length = 1

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

return self.\_\_length

def append(self, element):

self.\_\_length += 1

if self.\_\_head\_\_ == None:

self.\_\_head\_\_ = Node(element)

else:

tmp = self.\_\_head\_\_

while tmp.next!= None:

tmp = tmp.next

tmp.next = Node(element)

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

if self.\_\_length == 0:

return "LinkedList[]"

else:

tmp = self.\_\_head\_\_

string = tmp.\_\_str\_\_()

while tmp.next != None:

tmp = tmp.next

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

return f'LinkedList[length = {self.\_\_length}, [{string}]]'

def pop(self):

if self.\_\_length == 0:

raise IndexError("LinkedList is empty!")

tmp = self.\_\_head\_\_

if tmp.next != None:

while tmp.next.next != None:

tmp = tmp.next

tmp.next= None

else:

self.\_\_head\_\_ = None

self.\_\_length -= 1

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

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

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

else:

num = self.\_\_length - n

if num == 0:

self.\_\_head\_\_.change\_data(new\_data)

else:

tmp = self.\_\_head\_\_

for i in range(num):

tmp = tmp.next

tmp.change\_data(new\_data)

def clear(self):

self.\_\_length = 0

self.\_\_head\_\_ = None