**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Информационные технологии»**

Тема: Алгоритмы и структуры данных в Python

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3343 |  | Лобова Е. И. |
| Преподаватель |  | Иванов Д. И. |

Санкт-Петербург

2024

## Цель работы

Целью работы является знакомство с различными структурами данных, алгоритмами и их сложностью, а также реализация связанного однонаправленного списка на языке Python.

## Задание

Вариант 3

​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\_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. *Пример того, как должно выглядеть взаимодействие с Вашим связным списком:*

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

## Выполнение работы

1. Указать, что такое связный список. Основные отличия связного списка от массива.

Связный список - структура данных, каждый из элементов которой содержит как собственные данные, так и некоторое количество ссылок на следующий и/или предыдущий узел списка. Его главное отличие от массива в том, что в списке могут хранится объекты любого типа, а в массиве только одинакового, также он не хранится в памяти последовательно, как массив. Также сложность некоторых методов для массива и связанных списков будет отличаться.

1. Указать сложность каждого метода.

Класс Node:

* \_\_init\_\_(self) - O(1)
* get\_data(self) – O(1)
* change\_data(self, new\_data) – O(1)
* \_\_str\_\_(self) – O(1)

Класс LinkedList:

* \_\_init\_\_(self) - O(1)
* \_\_len\_\_(self) – O(1)
* append(self, element) – O(n)
* \_\_str\_\_(self) – O(n)
* pop(self) – O(n)
* change\_on\_end(self, n, new\_data) – O(n)
* clear(self) – O(1)

1. Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python.

Бинарный поиск в связном списке отличается от бинарного поиска в классическом списке Python (например, в виде списка или массива) из-за различной структуры данных.

Для классического списка Python можно использовать индексы для доступа к элементам, что упрощает бинарный поиск, в то время как в связном списке необходимо выполнять обход узлов для доступа к элементам.

Из-за необходимости последовательного обхода элементов в связном списке, сложность выполнения бинарного поиска увеличивается по сравнению с классическим списком.

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 |
|  | node = Node(1)  print(node)  print(node) print(node.get\_data())  l\_l = LinkedList()  print(l\_l)  print(len(l\_l))  l\_l.append(111)  l\_l.append(222)  l\_l.append(333)  print(l\_l)  print(len(l\_l))  l\_l.pop()  print(l\_l)  l\_l.pop()  print(l\_l)  l\_l.pop()  print(l\_l)  l\_l.append(111)  l\_l.append(222)  l\_l.append(333)  l\_l.change\_on\_end(1, 3)  print(l\_l) | data: 1, next: None  1  LinkedList[]  0  LinkedList[length = 3, [data: 111, next: 222; data: 222, next: 333; data: 333, next: None]]  3  LinkedList[length = 2, [data: 111, next: 222; data: 222, next: None]]  LinkedList[length = 1, [data: 111, next: None]]  LinkedList[]  LinkedList[length = 3, [data: 111, next: 222; data: 222, next: 3; data: 3, next: None]] | Методы обоих классов работают корректно при корректных поданных данных. |
|  | try:  l\_l = LinkedList()  l\_l.append(10)  l\_l.append(20)  l\_l.append(30)  l\_l.append(40)  l\_l.change\_on\_end(-1, 2)  print(l\_l)  except (KeyError, ValueError):  print('OK') | ОК | При некорректных введенных данных срабатывает исключение. |

## Выводы

Были изучены различных структуры данных и сложности их основных методов. Также была написана программа, в соответствии с заданным вариантом, в которой с помощью классов реализован однонаправленный связанный список с различными методами.

# Приложение А Исходный код программы

Название файла: main.py

class Node:

def \_\_init\_\_(self, data, next = None):

self.next = next

self.\_\_data = data

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}"

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

class LinkedList:

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

self.head = head

if self.head == None:

self.length = 0

else: self.length = 1

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

return self.length

def append(self, element):

new\_node = Node(element)

if self.length == 0:

self.head = new\_node

self.length += 1

return

current = self.head

while current.next:

current = current.next

current.next = new\_node

self.length += 1

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

if self.length == 0:

return "LinkedList[]"

else:

s = f"LinkedList[length = {self.length}, ["

current = self.head

while current.next:

s += f"{str(current)}; "

current = current.next

s += f"{str(current)}]]"

return s

def pop(self):

if self.length == 0:

raise IndexError("LinkedList is empty!")

elif self.length == 1:

self.head = None

self.length -= 1

else:

current = self.head

while current.next.next:

current = current.next

current.next = 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:

current = self.head

for i in range(self.length - n):

current = current.next

current.change\_data(new\_data)

def clear(self):

self.head = None

self.length = 0