**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. 3341 |  | Гребенюк В.А. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Целью работы является освоение работы с алгоритмами и структурами данных в Python. И разработка кода с их применением.

## Задание

Вариант 4

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.

*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

***Вам не требуется реализовывать создание экземпляров ваших классов и вызов методов, это сделает проверяющая система.***

*В отчете вам требуется:*

1. Указать, что такое связный список. Основные отличия связного списка от массива.
2. Указать сложность каждого метода.
3. Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

## Выполнение работы

Создан код в соответствии с заданием.

1. Связный список — это структура данных, состоящая из узлов, каждый из которых содержит данные и ссылку на следующий элемент. Основное отличие связного списка от массива заключается в том, что элементы связного списка не располагаются в памяти последовательно, что позволяет более эффективно добавлять и удалять элементы, не требуя перемещения остальных элементов.

2. Класс *Node*:

*\_\_init\_\_* - O(1)

*get\_data* - O(1)

*change\_data* - O(1)

*\_\_str\_\_* - O(1)

Класс LinkedList

*\_\_init\_\_* - O(1)

*\_\_len\_\_* - O(n) // TODO: возможно сократить до O(1) при сохранении длинны в переменную. Нужно больше информации

*append* - O(n)

*\_\_str\_\_* - O(n)

*pop* - O(n)

*change\_on\_start* - O(n)

*clear* - O(1)

Разработанный программный код см. в приложении А.

## Выводы

В ходе работы были рассмотрены основные концепции связного списка, его отличия от массива, а также сложность различных методов, применяемых в классе *LinkedList*. Было установлено, что бинарный поиск в связном списке не является эффективным из-за отсутствия прямого доступа к элементам, что является существенным недостатком по сравнению с бинарным поиском в классических списках *Python*.

# 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):

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

class LinkedList:

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

self.head = head

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

\_len = 0

cursor = self.head

while cursor:

\_len += 1

cursor = cursor.next

return \_len

def append(self, element):

if self.head is None:

self.head = Node(element)

else:

cursor = self.head

while cursor.next:

cursor = cursor.next

cursor.next = Node(element)

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

if self.head is None:

return "LinkedList[]"

\_nodes = []

cursor = self.head

while cursor:

\_nodes.append(str(cursor))

cursor = cursor.next

return f"LinkedList[length = {len(self)}, [{'; '.join(\_nodes)}]]"

def pop(self):

if not self.head:

raise IndexError("LinkedList is empty!")

# should be:

# raise IndexError("pop from empty LinkedList")

cursor = self.head

if not cursor.next:

self.head = None

else:

while cursor.next.next:

cursor = cursor.next

cursor.next = None

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

if n > len(self) or n <= 0:

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

# should be:

# raise IndexError("LinkedList assignment index out of range")

cursor = self.head

for \_ in range(n - 1):

cursor = cursor.next

cursor.change\_data(new\_data)

def clear(self):

self.head = None