**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 в Python»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3342 |  | Пушко К. Д. |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург

2024

## Цель работы

Изучить алгоритмы и структуры данных в Python, научиться использовать их для решения практических задач. Реализовать связный однонаправленный список и реализовать функционал для работы с ним.

## 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:

* data # Данные элемента списка, приватное поле.
* next # Ссылка на следующий элемент списка.

И следующие методы:

* \_\_init\_\_(self, data, next) - конструктор, у которого значения по умолчанию для аргумента next равно None.
* get\_data(self) - метод возвращает значение поля data (это необходимо, потому что в идеале пользователь класса не должен трогать поля класса Node).
* change\_data(self, new\_data) - метод меняет значение поля data объекта Node.
* \_\_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:

Linked List

Класс, который описывает связный однонаправленный список.

Он должен иметь 2 поля:

* head # Данные первого элемента списка.
* length # Количество элементов в списке.

И следующие методы:

* \_\_init\_\_(self, head) - конструктор, у которого значения по умолчанию для аргумента head равно None.
* \_\_len\_\_(self) - перегрузка метода \_\_len\_\_, он должен возвращать длину списка (этот стандартный метод, например, используется в функции 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 стандартного метода \_\_str\_\_, который преобразует объект в строковое представление. Для данной лабораторной необходимо реализовать следующий формат перевода объекта класса однонаправленного списка в строку:
* pop(self) - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.
* clear(self) - очищение списка.
* 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

Для выполнения лабораторной работы было необходимо создать 2 класса. Класс Node и класс Linked List.

В классе Node представлены поля: data и next, которые инициализируются при создании экземпляра класса. Поле data отвечает за хранение информации, которую содержит узел, поле next – ссылка на следующий узел. Так же в классе представлены методы: init(self,data,next) – инициализирует экземпляр класса, get\_data(self) – возвращает значение поля data, change\_data(self, new\_data) – меняет значение поля data, str(self) - перегрузка стандартного метода, который преобразует объект в строковое представление.

В классе Linked List представлены поля: head – первый элемент списка и length – количество элементов в списке.

Так же в классе представлены методы: init(self, head) – инициализирует список, len(self) - перегрузка метода len, возвращает длину списка, append(self, element) – добавляет новый узел в конец списка, str(self) - перегрузка стандартного метода str, который преобразует объект в строковое представление, pop(self) – удаляет последний элемент списка, clear(self) – очищает список, change\_on\_start(self, n, new\_data) - изменяет поля data n-того элемента с НАЧАЛА списка на new\_data.

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

Связный список — это базовая структура данных для хранения набора значений. Элементы связного списка могут быть расположены, как угодно, в памяти, не обязательно последовательно. Связь между элементами осуществляется с помощью ссылок (связок) в каждом из элементов на следующий элемент (None, если элемент последний). Такое непоследовательное хранение элементов и упорядоченная связь между ними – принципиальное отличие и преимущество связанных списков в сравнении с массивами.

2. Указать сложность каждого метода.

Класс Node:

• \_\_init\_\_ — O(1);

• get\_data — O(1);

• \_\_str\_\_ — O(1).

Класс LinkedList:

• \_\_init\_\_ — O(n);

• \_\_len\_\_ — O(1);

• append — O(n);

• \_\_str\_\_ — O(n);

• pop — O(n);

• change\_on\_start — O(n);

• clear — O(1).

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

В отсортированном списке находим середину с помощью деления длины списка на два, запоминая ссылку на средний элемент. После перемещения в середину очередной половины необходимо будет вновь совершать проход по списку, что является неэффективным действием (сложность взятия элемента по индексу – О(n)). В массиве же бинарный поиск работает с помощью сравнения элементов и вычисления нужных индексов. В нем не обязательно проходить по всему списку.

Разработанный программный код см. в приложении А.

## Выводы

Были изучены принципы работы с линейными списками в Python. Для выполнения работы был реализован класс LinkedList, представляющий линейный односвязный список, и методы данного класса.

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 is None):

return f'data: {self.\_\_data}, next: None'

return f'data: {self.\_\_data}, next: {self.next.get\_data()}'

class LinkedList:

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

if head is None:

self.head = None

self.length = 0

else:

self.head = head

self.length = 1

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

return self.length

def append(self, element):

if self.head is None:

self.head = Node(element)

self.length = 1

return

temp = self.head

while temp.next is not None:

temp = temp.next

temp.next = Node(element)

self.length +=1

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

if (self.length == 0):

return 'LinkedList[]'

message = 'LinkedList[length = ' + str(self.length) + ', ['

temp = self.head

while temp.next is not None:

message += str(temp) + '; '

temp = temp.next

message += str(temp)

message += ']]'

return message

def pop(self):

if self.length == 0:

raise IndexError("LinkedList is empty!")

if self.length == 1:

self.head = None

self.length = 0

return

temp = self.head

while temp.next.next is not None:

temp = temp.next

temp.next = 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!")

temp = self.head

for i in range(n-1):

temp = temp.next

temp.change\_data(new\_data)

def clear(self):

self.head = None

self.length = 0