**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 |  | Малахов А.И. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Изучить алгоритмы и структуры данных в Python, освоить основы работы с линейными списками и их практическая реализация на языке программирования 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:

* **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:

“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.

**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:

· 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.

* **pop(self)** - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.максимальная скорость (в км/ч, положительное целое число)
* **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.

## Выполнение работы

Были разработаны классы Node и LinkedList, в которых реализованы соответствующие методы в соответствии с поставленной задачей.

Класс Node:

Имеет два переопределенных метода: инициализация и строковый вид (выводится значение текущего элемента, далее, если есть следующий элемент в списке, его значение выводится после «next», иначе выводится только значение текущего элемента и None после «next»), а также методы возврата значения элемента класса и изменения значения элемента.

Класс LinkedList:

Метод \_\_init\_\_ — если аргумент head равен None, то создается пустой список, иначе список из одного элемента.

Метод \_\_len\_\_ — возвращает поле length элемента класса.

Метод append — создает элемент класса Node, если список был пустым, то это становится первым элементом, иначе в цикле доходит до последнего элемента списка и новый элемент добавляется в конец.

Метод \_\_str\_\_ — выводится форматная строка согласно условию задачи.

Метод pop — удаляется последний элемент в списке, если список пустой, то срабатывает исключение IndexError с сообщением "LinkedList is empty!".

Метод change\_on\_end — значение n-того элемента с конца списка меняется на new\_data.

Метод clear — очищение списка.

Сложность каждого метода в реализованном коде:

* \_\_init\_\_: O (1) - создание объекта LinkedList или Node.
* \_\_len\_\_: O (n) - вычисление длины списка.
* append: O (n) - добавление элемента в конец списка.
* \_\_str\_\_: O (n) – создание строкового представления списка.
* pop: O (n) - удаление последнего элемента списка.
* change\_on\_end: O (n) - изменение элемента по позиции с конца списка.
* clear: O (1) - очистка списка.

Связанный список — это структура данных, состоящая из последовательности элементов, называемых узлами, где каждый узел содержит данные и ссылку на следующий элемент в списке. Каждый узел в связанном списке соединен с последующим узлом, образуя цепочку из элементов.

**Основные отличия связного списка от массива:**

* В массиве элементы хранятся в последовательной области памяти, в то время как в связном списке каждый элемент хранится отдельно и может быть расположен в памяти как угодно (не последовательно), ссылки соединяют их в список.
* В случае связного списка вставка и удаление элементов могут быть выполнены за постоянное время (O(1)), включая начало и середину списка, в то время как в массиве эти операции могут потребовать сдвиг всех последующих элементов, что может быть затратным по времени (O(n)).
* В массиве элементы доступны по индексу за постоянное время (O(1)), в то время как в связном списке время доступа к элементам зависит от их позиции и может быть линейным (O(n)) в худшем случае.

Реализация бинарного поиска в связном списке отличается от классического списка Python тем, что в связном списке отсутствует прямой доступ к элементам по индексу, что мешает разделению списка на две части, как в массиве. Вместо этого в бинарном поиске для связного списка используются указатели на начало и конец текущего диапазона, которые итеративно сокращаются вдвое, пока не будет найден искомый элемент. В целом это неэффективно, так как в любом случае придется проходить по всему списку. Проще сразу пройти по всему списку либо конвертировать его в массив.

## Выводы

Был создан связный список с использованием классов на языке Python, после чего была проанализирована производительность методов этого класса и рассмотрена возможность применения бинарного поиска в связном списке.

# **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.\_\_data}, next: {self.next.get\_data() if self.next else None}"

class LinkedList:

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

self.head = head

self.length = 0 if not head else 1

if head is not None:

self.append(head)

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

return self.length

def append(self, element):

new\_node = Node(element)

if not self.head:

self.head = new\_node

else:

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:

nodes = []

current = self.head

while current:

nodes.append(current.\_\_str\_\_())

current = current.next

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

def pop(self):

if self.length == 0:

raise IndexError('LinkedList is empty!')

if self.length == 1:

self.clear()

return

else:

current = self.head

while current.next.next is not None:

current = current.next

current.next = None

self.length -= 1

def clear(self):

self.head = None

self.length = 0

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

if n > self.length or n <= 0:

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

current = self.head

for \_ in range(self.length - n):

current = current.next

current.change\_data(new\_data)h = 0