**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

|  |  |  |
| --- | --- | --- |
| Студент гр. 2384 |  | Кузьминых Е.М |
| Преподаватель |  | Шевская Н.В. |

Санкт-Петербург

2023

**Цель работы.**

Научиться использовать основные алгоритмы и структуры данных на примере реализации связного списка на языке программирования Python.

**Задачи.**

Вариант №1

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*.

*\_\_str\_\_(self)*

перегрузка метода *\_\_str\_\_*. Описание того, как должен выглядеть результат вызова метода смотрите ниже в примере взаимодействия с *Node*.

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.

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.

*delete\_on\_end(self, n)*

удаление n-того элемента с конца списка. 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:

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

Указать сложность каждого метода.

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

**Выполнение работы.**

Данный код содержит два класса: *Node* и *LinkedList*. Класс *Node* представляет элемент связного списка, который содержит данные и указатель на следующий элемент. Класс *LinkedList* представляет сам связный список, который содержит указатель на головной элемент и длину списка.

Класс *Node*:

метод *init* создает новый элемент, устанавливая значение данных и указатель на следующий элемент (по умолчанию *None*)

метод *get\_data* возвращает данные узла

метод *str* возвращает строку с данными узла и ссылкой на следующий элемент

Класс *LinkedList*:

метод *init* создает новый связный список, устанавливая головной элемент (по умолчанию *None*) и длину списка

метод *len* возвращает длину списка

метод *append* добавляет новый элемент в конец списка, обновляя указатель последнего узла на новый элемент

метод *pop* удаляет последний элемент из списка и возвращает его данные, обновляя указатель предпоследнего узла на *None*

метод *clear* очищает список, устанавливая головной элемент в *None* и длину списка в 0

метод *delete\_on\_end* удаляет элемент на n-ой позиции с конца списка, обновляя указатель на предыдущий элемент на следующий элемент

метод *str* возвращает строку с длиной списка и данными всех узлов

Связный список - это структура данных, которая представляет собой последовательность узлов, где каждый элемент содержит данные и указатель на следующий элемент. Основное отличие связного списка от массива заключается в том, что для доступа к элементам массива можно использовать индекс, а для доступа к элементам связного списка необходимо последовательно перебирать узлы, начиная с головного.

Сложность методов *LinkedList*:

метод *init* - O(1)

метод *len* - O(1)

метод *append* - O(n)

метод *pop* - O(n)

метод *clear* - O(1)

метод *delete\_on\_end* - O(n)

метод *str* - O(n)

Для реализации бинарного поиска в связном списке можно использовать следующий алгоритм:

Найти длину связного списка.

Установить два указателя на голову списка: *left* и *right*.

Вычислить индекс середины списка: *mid = (left + right) // 2*.

Пройти по списку от начала до середины и переместить указатель *left* на середину списка.

Если искомый элемент меньше элемента, на который указывает *left*, продолжать поиск в левой части списка (установив rig*ht* = *mid*), иначе продолжать поиск в правой части списка (установив *left* = *mid*).

Повторять шаги 3-5 до тех пор, пока элемент не будет найден.

Отличие реализации бинарного поиска для связного списка от реализации для классического списка Python заключается в том, что для связного списка требуется проходить по каждому элементу, в отличие от массива, где можно обратиться к элементу по индексу. Это приводит к тому, что бинарный поиск в связном списке имеет линейную сложность *O(n)*, в то время как в обычном списке сложность составляет *O(log n)*. Поэтому, если нужно выполнить множество операций поиска, лучше использовать стандартные списки.

**Тестирование.**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Входные данные | Выходные данные | Комментарий |
| 1 | ll = LinkedList()  ll.append(1)  ll.append(2)  ll.append(3)  print(ll)  ll.pop()  print(ll) | LinkedList[length = 3, [data: 1, next: 2; data: 2, next: 3; data: 3, next: None]]  LinkedList[length = 2, [data: 1, next: 2; data: 2, next: None]] | Ответ верный |

**Вывод.**

В ходе выполнения лабораторной работы была написана программа, представляющая из себя реализацию однонаправленного односвязного списка, для изучения структур данных и алгоритмов.

**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 \_\_str\_\_(self):

next\_node = 'None' if self.\_\_next\_\_ is None else self.\_\_next\_\_.\_\_data

return f"data: {self.\_\_data}, next: {next\_node}"

class LinkedList:

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

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

if \_\_head\_\_ == None:

self.\_\_length = 0

else:

self.\_\_length = 1

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

return self.\_\_length

def append(self, element):

if self.\_\_length > 0:

n = self.\_\_head\_\_

while n.\_\_next\_\_ is not None:

n = n.\_\_next\_\_

n.\_\_next\_\_ = Node(element)

else:

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

self.\_\_length += 1

def pop(self):

if self.\_\_length != 0:

if self.\_\_length==1:

self.\_\_head\_\_=self.\_\_head\_\_.\_\_next\_\_

else:

tmp1 = self.\_\_head\_\_

tmp2=tmp1.\_\_next\_\_

while tmp2.\_\_next\_\_ is not None:

tmp1=tmp2

tmp2=tmp2.\_\_next\_\_

tmp1.\_\_next\_\_=None

self.\_\_length-=1

else:

raise IndexError("LinkedList is empty!")

def clear(self):

self.\_\_length = 0

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

def delete\_on\_end(self, n):

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

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

if n == self.\_\_length:

self.\_\_head\_\_ = self.\_\_head\_\_.\_\_next\_\_

else:

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

for i in range(self.\_\_length - n - 1):

tmp = tmp.\_\_next\_\_

tmp.\_\_next\_\_ = tmp.\_\_next\_\_.\_\_next\_\_

self.\_\_length -= 1

else:

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

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

result = 'LinkedList[]'

if self.\_\_length != 0:

array = []

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

while tmp is not None:

array.append(tmp.\_\_str\_\_())

tmp = tmp.\_\_next\_\_

str = '; '.join(array)

result = f'LinkedList[length = {self.\_\_length}, [{str}]]'

return result