**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 |  | Гельман П.Е. |
| Преподаватель |  | Иванов Д. И. |

Санкт-Петербург

2024

## Цель работы

Целью работы является освоение работы с односвязным списком в Python на базе ООП и создание программы на основе полученных знаний.

## Задание

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.

## Выполнение работы

Связный список - это структура данных, которая состоит из узлов, каждый из которых содержит какое-то значение и ссылку на следующий узел в списке. Последний узел может указывать на None, обозначая конец списка.

Основные отличия между связным списком и массивом:

1. Динамичность: связный список может изменять размер динамически, в отличие от массива, размер которого задается заранее.
2. Вставка и удаление элементов: в связном списке вставка и удаление элементов происходят быстрее и требуют меньше ресурсов, чем в массиве, так как не требуется сдвигать элементы.
3. Сложность доступа к элементам: в связном списке доступ к произвольному элементу осуществляется за O(n) времени, в то время как в массиве за O(1) времени.

Сложность всех методов:

class Node:

\_\_init\_\_(self, data, next=None) — O(1)

get\_data(self) — O(1)

change\_data(self, new\_data) — O(1)

\_\_str\_\_(self) — O(1)

class LinkedList:

\_\_init\_\_(self, head=None) — O(1)

\_\_len\_\_(self) — O(1)

append(self, element) — O(n)

\_\_str\_\_(self) — O(1)

pop(self) — O(n)

change\_on\_start(self, n, new\_data) — O(n)

clear(self) — O(1)

Алгоритм бинарного поиска предназначен для отсортированного набора значений, соответственно для решения задачи бинарного поиска в связанном списке сначала необходимо его отсортировать, а затем выполнять следующие действия:

1. Найти длину списка.
2. Установить два указателя - левый и правый (указывающие на начало и конец списка).
3. Найти середину списка, опираясь на длину.
4. Сравнить значение в середине с искомым значением.
5. Если значение равно искомому, элемент найден.
6. Если искомое значение меньше значения в середине, продолжить поиск в левой половине списка, обновив правый указатель.
7. Если искомое значение больше значения в середине, продолжить поиск в правой половине списка, обновив левый указатель.
8. Повторять шаги 3-7, пока не будет найден элемент или до тех пор, пока левый указатель не окажется правее правого указателя.

Отличия реализации алгоритма бинарного поиска для связного списка и для классического списка Python:

В классическом списке Python (например, списке list), бинарный поиск эффективен благодаря прямому доступу к элементам по индексу за O(1) времени.

Для отсортированного связного списка бинарный поиск требует гораздно больше времени на каждую итерацию поиска из-за необходимости последовательного прохода по элементам и первоначальной сортировки элементов. Таким образом, бинарный поиск в связном списке может быть менее эффективным по сравнению с классическим списком Python.

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 |
|  | linked\_list = LinkedList()  print(linked\_list)  print(len(linked\_list))  linked\_list.append(10)  print(linked\_list)  print(len(linked\_list))  linked\_list.append(20)  print(linked\_list)  print(len(linked\_list))  linked\_list.pop()  print(linked\_list)  print(linked\_list)  print(len(linked\_list)) | LinkedList[]  0  LinkedList[length = 1, [data: 10, next: None]]  1  LinkedList[length = 2, [data: 10, next: 20; data: 20, next: None]]  2  LinkedList[length = 1, [data: 10, next: None]]  LinkedList[length = 1, [data: 10, next: None]]  1 | Верно |
|  | linked\_list = LinkedList(10)  linked\_list.append(20)  linked\_list.append(30)  print("Связанный список до изменения:")  print(linked\_list)  linked\_list.change\_on\_start(2, 25)  print("\nСвязанный список после изменения:")  print(linked\_list)  linked\_list.pop()  print("\nСвязанный список после извлечения:")  print(linked\_list)  linked\_list.clear()  print("\nСвязанный список после очистки:")  print(linked\_list) | Связанный список до изменения:  LinkedList[length = 3, [data: 10, next: 20; data: 20, next: 30; data: 30, next: None]]  Связанный список после изменения:  LinkedList[length = 3, [data: 10, next: 25; data: 25, next: 30; data: 30, next: None]]  Связанный список после извлечения:  LinkedList[length = 2, [data: 10, next: 25; data: 25, next: None]]  Связанный список после очистки:  LinkedList[] | Верно |

## Выводы

Была освоена работа с связанным однонаправленным списком на языке Python с помощью ООП и реализована программа, позволяющая работать с таким списком.

# 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:

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

else:

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

class LinkedList:

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

if head:

self.head = Node(head)

self.length = 1

else:

self.head = None

self.length = 0

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

return self.length

def append(self, element):

new\_node = Node(element)

if self.head is None:

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 not hasattr(self, 'head') or self.head is None or self.length == 0:

return "LinkedList[]"

else:

current = self.head

nodes = []

while current:

nodes.append(f"data: {current.get\_data()}, next: {current.next.get\_data() if current.next else None}")

current = current.next

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

def pop(self):

if self.length == 0:

raise IndexError("LinkedList is empty!")

elif self.length == 1:

self.head = None

self.length = 0

return self

#raise IndexError("LinkedList is empty!")

else:

current = self.head

while current.next.next:

current = current.next

self.length -= 1

current.next = None

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

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

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

current = self.head

tmp = self.head

for \_ in range(n-1):

current = current.next

current.change\_data(new\_data)

def clear(self):

self.head = None

self.length = 0