**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. Вариант 2**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Атоян М. А. |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург

2024

## Цель работы

Изучить основные особенности структур данных и методов работы с ними. Написать собственную практическую реализацию линейного односвязного списка на Python, используя ООП. Сравнить асимптотическую сложность операций над списком и над массивом.

## Задание

**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 \_\_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 delete\_on\_start(self, n) - удаление n-того элемента с НАЧАЛА списка. Метод должен выбрасывать исключение KeyError, с сообщением "Element doesn't exist!", если количество элементов меньше n.

## Выполнение работы

Связный список - это структура данных, которая состоит из узлов, где каждый узел содержит данные и ссылку на следующий узел в списке. Основное отличие связного списка от массива заключается в том, что связный список не требует непрерывной памяти для хранения элементов, в отличие от массива. Кроме того, связный список позволяет эффективно добавлять и удалять элементы из середины списка, так как не требует переноса всех элементов при изменении размера. Также связный список не обладает прямым доступом к произвольному элементу по индексу, в отличие от массива, где это возможно.

Сложности методов:

Класс Node:

1. init – O(1);
2. get\_data – O(1);
3. str – O(1).

Класс LinkedList:

1. init – O(1);
2. len – O(1);
3. append – O(n);
4. str – O(n);
5. pop – O(n);
6. delete\_on\_start - O(n);
7. clear – O(1).

Анализ:

В случае классического списка Python, бинарный поиск выполняется путем разделения списка на две части и последующего сравнения искомого элемента с элементом в середине списка. Затем выполняется поиск в соответствующей половине списка в зависимости от результата сравнения.

В связном списке бинарный поиск немного сложнее из-за того, что к элементам списка можно получить доступ только последовательно, начиная с головы списка. Поэтому в бинарном поиске для связного списка сначала необходимо определить размер списка и затем выполнить поиск с учетом разделения списка наполовину. Однако такой подход не эффективен, так как он требует прохода по всему списку для нахождения размера.

## Тестирование

Результаты тестирования содержатся в таблице 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 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 | 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 | Программа сработала корректно. |

## Выводы

В ходе выполнения лабораторной работы были изучены и применены на практике алгоритмы и структуры данных в Python. Разработан односвязный линейный список с применением полученных знаний, реализованы методы для работы с ним.

# 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 \_\_str\_\_(self):

return f"data: {self.data}, next: {self.next.data if self.next else None}"

class LinkedList:

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

self.head = head

self.length = 1 if head else 0

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

return self.length

def append(self, data):

if (self.head):

current = self.head

while (current.next):

current = current.next

current.next = Node(data)

else:

self.head = Node(data)

self.length += 1

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

current = self.head

if (current == None):

return "LinkedList[]"

list\_of\_nodes = []

while current:

list\_of\_nodes.append(str(current))

current = current.next

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

def pop(self):

if (len(self) == 0):

raise IndexError("LinkedList is empty!")

current = self.head

if (len(self) == 1):

self.head = None

else:

while(current.next.next):

current = current.next

current.next = None

self.length -= 1

def clear(self):

self.head = None

self.length = 0

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

if (n > len(self) or n < 1):

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

current = self.head

if (n == 1):

self.head = self.head.next

else:

for \_ in range(n-2):

current = current.next

current.next = current.next.next

self.length -= 1