**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 |  | Колесниченко М. А. |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург

2024

## Цель работы

Изучение работы с линейным списком и его реализации на языке Python.

## 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   **\_\_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:

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\_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.

## Выполнение работы

Были созданы классы Node и LinkedList. Реализованы методы согласно условию.

Связный список — это структура данных, представляющая собой последовательность элементов, в которой каждый элемент хранит ссылку на следующий элемент в последовательности. Каждый элемент списка называется узлом. Узлы могут содержать как данные (например, числа или объекты), так и ссылки на следующие узлы.

**Основные отличия связного списка от массива:**

1. **Хранение данных:** В массиве элементы хранятся в последовательной области памяти, в то время как в связном списке каждый элемент хранится отдельно, и ссылки соединяют их в список.
2. **Доступ к элементам:** В массиве доступ к элементам осуществляется по индексу за константное время (O(1)), тогда как в связном списке время доступа к элементам зависит от их позиции в списке и может быть линейным (O(n)) в худшем случае.
3. **Вставка и удаление элементов:** В связном списке вставка и удаление элементов могут быть выполнены за константное время (O(1)), даже в начале и середине списка, тогда как в массиве эти операции могут потребовать сдвиг всех последующих элементов, что может быть дорого с точки зрения времени (O(n)).

**Сложность каждого метода в реализованном коде:**

1. **\_\_init\_\_**: O(1) - создание объекта LinkedList или Node.
2. **\_\_len\_\_**: O(n) - вычисление длины списка.
3. **append**: O(n) - добавление элемента в конец списка.
4. **\_\_str\_\_**: O(n) - создание строкового представления списка.
5. **pop**: O(n) - удаление последнего элемента списка.
6. **delete\_on\_end**: O(n) - удаление элемента по его позиции в конце списка.
7. **clear**: O(1) - очистка списка.

Отличие реализации бинарного поиска для связного списка от классического списка Python заключается в том, что для связного списка нет прямого доступа к элементам по индексу, что делает невозможным применение такой же стратегии деления списка на две части, как в случае массива. Вместо этого бинарный поиск в связном списке будет осуществляться с использованием указателей на начало и конец текущего диапазона, итеративно сокращая этот диапазон вдвое, пока не будет найден искомый элемент или не определится его отсутствие.

## Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | node = Node(1)  print(node) # data: 1, next: None  node.next = Node(2, None)  print(node) # data: 1, next: 2  print(node.get\_data())  l\_l = LinkedList()  print(l\_l)  print(len(l\_l))  l\_l.append(10)  l\_l.append(20)  l\_l.append(30)  l\_l.append(40)  print(l\_l)  print(len(l\_l))  l\_l.delete\_on\_end(3)  print(l\_l) | data: 1, next: None  data: 1, next: 2  1  LinkedList[]  0  LinkedList[length = 4, [data: 10, next: 20; data: 20, next: 30; data: 30, next: 40; data: 40, next: None]]  4  LinkedList[length = 3, [data: 10, next: 30; data: 30, next: 40; data: 40, next: None]] | Верный вывод |

## Выводы

Был реализован связный список с помощью классов на языке Python. Исследована скорость работы методов созданного класса и возможность использования 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 \_\_str\_\_(self):

if self.next is not None:

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

else:

return f"data: {self.get\_data()}, next: None"

class LinkedList:

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

self.head = head

self.length = len(self)

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

c = 0

temp = self.head

while temp != None:

temp = temp.next

c += 1

return c

def append(self, element):

x = Node(element)

if self.head == None:

self.head = x

else:

temp = self.head

while temp.next != None:

temp = temp.next

temp.next = x

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

if len(self) == 0:

return "LinkedList[]"

else:

temp = self.head

arr = [str(temp)]

while temp.next != None:

temp = temp.next

arr += [str(temp)]

arr = "[" + '; '.join(arr) + "]"

return f"LinkedList[length = {len(self)}, {arr}]"

def pop(self):

if len(self) == 0:

raise IndexError("LinkedList is empty!")

elif len(self) == 1:

self.head = None

else:

temp = self.head

while temp.next.next != None:

temp = temp.next

temp.next = None

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

if n <= 0 or n > len(self):

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

elif n == len(self):

self.head = self.head.next

else:

temp = self.head

for \_ in range(len(self) - n - 1):

temp = temp.next

temp.next = temp.next.next

def clear(self):

self.head = None