**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. 3343 |  | Пухов А. Д. |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург

2024

## Цель работы

Изучение и применение на практике однонаправленных списков в 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:

* 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.

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:

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.
  + Если значение переменной head равна None, метод должен создавать пустой список.
  + Если значение head не равно None, необходимо создать список из одного элемента.
* \_\_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.

Пример того, как должен выглядеть результат реализации см. ниже.

* pop(self) - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.
* 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

Связный список – это структура данных, в которой элементы (узлы) содержат данные и ссылку на следующий элемент в списке. Основное отличие связного списка от массива заключается в следующем:

**1.** Хранение памяти: узлы связного списка разбросаны в памяти, тогда как элементы массива располагаются последовательно.

**2.** Размер: связный список может динамически меняться по размеру, массив чаще всего имеет фиксированный размер.

**3.** Доступ к элементам: в массиве доступ по индексу, что обеспечивает быстрое обращение к элементам; в связном списке для доступа к элементу необходимо последовательное прохождение списка.

**4.** Изменения: добавление и удаление элементов в связном списке происходит быстрее, так как не требуется перемещать другие элементы, в отличие от массива, где последующие элементы сдвигаются.

**Сложность каждого метода:**

1. **class Node:**

**O(1):**

* + - \_\_init\_\_(self, data, next=None)
    - get\_data(self)
    - change\_data(self, new\_data)
    - \_\_str\_\_(self)

1. **Class LinkedList:**

**O(1):**

O(1):

* + - \_\_init\_\_(self, head=None)
    - \_\_len\_\_(self)
    - clear(self)

**O(n):**

* + - append(self, element)
    - \_\_str\_\_(self)
    - pop(self)
    - change\_on\_end(self, n, new\_data)

Бинарный поиск в связном списке неэффективен, потому что он требует последовательного прохода для доступа к элементам, в отличие от массива, где доступ возможен за константное время. В классическом массиве или списке Python бинарный поиск быстро находит середину, так как может обращаться к любому индексу напрямую. Реализовать бинарный поиск в связном списке технически возможно, но это потребует порядка O(n) операций для обращения к серединному элементу и будет значительно менее эффективно по сравнению с O(1) для массива. Обычно, если необходим бинарный поиск, предпочтительнее использовать массивы или другие структуры данных, которые поддерживают прямой доступ к элементам.

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()  linked\_list.append(10)  linked\_list.append(20)  linked\_list.append(30)  print(linked\_list)  linked\_list.change\_on\_end(1, 40)  print(linked\_list)  linked\_list.clear()  print(linked\_list) | LinkedList[]  LinkedList[length = 3, [data: 10, next: 20; data: 20, next: 30; data: 30, next: 40; data: 40, next: 50; data: 50, next: None]] | ОК |

## Выводы

В данной лабораторной работе была изучена работа со списками, и был реализован односвязный список.

# Приложение А Исходный код программы

Название файла: lb2.py

class Node:

    def \_\_init\_\_(self, data, next=None):

        self.\_\_data = data

        self.next = None

    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 head is None else 1

    def \_\_len\_\_(self):

        return self.length

    def append(self, element):

        new\_node = Node(element)

        if self.head is None:

            self.head = new\_node

        else:

            tmp = self.head

            while tmp.next:

                tmp = tmp.next

            tmp.next = new\_node

        self.length += 1

    def \_\_str\_\_(self):

        if self.length == 0:

            return "LinkedList[]"

        else:

            nodes = []

            current = self.head

            while current:

                nodes.append(str(current))

                current = current.next

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

    def pop(self):

        if self.length == 0:

            raise IndexError("LinkedList is empty!")

        tmp = self.head

        if tmp.next is None:

            self.head = None

        else:

            while tmp.next.next:

                tmp = tmp.next

            tmp.next = None

        self.length -= 1

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

        tmp = self.head

        if self.length < n or n <= 0:

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

        for i in range(self.length - n):

            tmp = tmp.next

        tmp.change\_data(new\_data)

    def clear(self):

        self.head = None

        self.length = 0