**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Информационные технологии»**

**Тема: Алгоритмы и структуры данных в Python**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Пивоев Н. М. |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург

2024

## Цель работы

Ознакомиться с односвязным списком на языке Python и написать программу с его реализацией. Провести сравнение списка и массива, найти сходства и отличия.

## Задание

Вариант 4.

**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.

**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.

     o  Если значение переменной head равна None, метод должен создавать пустой список.

     o  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:

     o  Если список пустой, то строковое представление:

        “LinkedList[]”

     o  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.

## Выполнение работы

Программа состоит из двух классов.

Первый – *class Node*, который является одним из элементов связанного списка. У него есть следующие методы:

* *\_\_init\_\_* - конструктор класса, заполняющий значение элемента и устанавливающий связь со следующим узлом.
* *get\_data* – возвращает значение поля *\_\_data*.
* *change\_data* – обновляет поле *\_\_data* в соответствием с полученным аргументом.
* *\_\_str\_\_* - возвращает информацию о узле списка.

Второй – *class LinkedList()* – связанный список. У него есть следующие методы:

* *\_\_init\_\_* - конструктор класса, создающий пустой список или состоящий из одного элемента.
* \_\_len\_\_- возвращает длину списка.
* *append* – добавляет узел в список.
* *\_\_str\_\_* - возвращает информацию о списке.
* *pop* – удаляет последний элемент из списка, если он есть.
* *clear* – очищает список.
* *change\_on\_start* – изменяет n элемент, начиная отчёт со старта списка, если это возможно.

Связный список — структура данных, состоящая из узлов, содержащих данные и ссылки на следующий элемент списка. В памяти элементы хранятся не последовательно, как в массиве, а в разных участках памяти благодаря ссылочной связи между элементами. К преимуществам связанного списка можно отнести быстрое добавление и удаление в любой части списка, но доступ к элементам занимает много времени.

В массиве доступ к элементам осуществляется по индексу и можно получить нужную ячейку быстро, но при добавлении или удалении, придётся перемещать элементы после изменяемого, что довольно неэффективно и трудоёмко.

Сложности методов по времени:

Node:

* *\_\_init\_\_* – O(1);
* *get\_data* – O(1);
* *change\_data* – O(1);
* *\_\_str\_\_* – O(1);

LinkedList:

* *\_\_init\_\_* – O(1);
* *\_\_len\_\_* – O(1);
* *append* – O(n);
* *\_\_str\_\_* – O(n);
* *pop* – O(n);
* *clear* – O(1);
* *change\_on\_start* – O(n);

Бинарный поиск – алгоритм поиска значения среди отсортированных элементов. Ставится две границы и на каждой итерации граница сдвигается к середине, поэтому он имеет временную сложность O(log(n)). Для связанного списка очевидно бинарный поиск работать не будет, потому что его элементы не отсортированы. Можно преобразовать список в массив и отсортировать, но это займёт O(nlog(n)), что значительно сложнее, чем просто найти нужный элемент в списке.

## Тестирование

Результаты тестирования содержатся в таблице 1.

Таблица 1.

|  |  |  |  |
| --- | --- | --- | --- |
| № | Входные данные | Выходные данные | Комментарии |
| 1. | linked\_list = LinkedList()  print(linked\_list)  linked\_list.append(15)  print(linked\_list)  linked\_list.append(40)  print(linked\_list)  linked\_list.pop()  print(linked\_list) | LinkedList[]  LinkedList[length = 1, [data: 15, next: None]]  LinkedList[length = 2, [data: 15, next: 40; data: 40, next: None]]  LinkedList[length = 1, [data: 15, next: None]] | Вывод соответствует ожиданиям. |

## Выводы

Таким образом, в ходе выполнения лабораторной работы были изучены особенности односвязного списка на языке Python и написана программа с его реализацией. Выявлены преимущества и недостатки связного списка и массива.

# 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):

return f'data: {self.\_\_data}, next: {None if self.next is None else self.next.get\_data()}'

class LinkedList():

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

if head is None:

self.length = 0

self.head = None

else:

self.length = 1

self.head = Node(head)

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

return self.length

def append(self, element):

pt = Node(element)

if self.length == 0:

self.length = 1

self.head = pt

return

current = self.head

while current.next:

current = current.next

self.length += 1

current.next = pt

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

pt = self.head

if self.head is None:

return 'LinkedList[]'

else:

value = []

while pt != None:

value.append(f'data: {pt.get\_data()}, next: {None if pt.next is None else pt.next.get\_data()}')

pt = pt.next

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

def pop(self):

if self.length == 0:

raise IndexError('LinkedList is empty!')

elif self.length == 1:

self.clear()

else:

pt = self.head

while pt.next.next:

pt = pt.next

self.length -= 1

pt.next = None

def clear(self):

self.length = 0

self.head = None

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

pt = self.head

previous = pt

current = Node(new\_data)

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

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

if n == 1:

current.next = self.head.next

self.head = current

else:

for i in range(n - 1):

previous = pt

pt = pt.next

current.next = pt.next

previous.next = current