**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 |  | Малиновский А.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  **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.

     ·  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.

## Выполнение работы

Связный список — это структура данных, которая состоит из узлов, содержащих данные и ссылки («связки») на следующий узел списка. Основное отличие связного списка от массива заключается в том, что массив — это структура данных, которая хранит элементы в памяти последовательно, а связный список — это структура данных, которая хранит элементы в памяти не последовательно, а связывая их между собой ссылками. В массиве доступ к элементам осуществляется по индексу, это позволяет быстро получать доступ к любому элементу массива. Но, при добавлении или удалении элементов в середине массива, необходимо перемещать все элементы после добавляемого или удаляемого, что занимает много времени и ресурсов. В связном списке доступ к элементам осуществляется последовательно, начиная с головы списка. Для доступа к элементу по индексу необходимо последовательно пройти все узлы до нужного. Однако, при добавлении или удалении элементов в середине списка, необходимо просто изменить ссылки на узлы, что занимает меньше времени и ресурсов, чем в массиве. Таким образом, связный список позволяет эффективно добавлять и удалять элементы в середине списка, а массив обеспечивает быстрый доступ к элементам по индексу.

Сложности методов:

Класс Node:

1. \_\_init\_\_ – O(1);

2. get\_data – O(1);

3. change\_data – O(1);

4. \_\_str\_\_ – O(1).

Класс LinkedList:

1. \_\_init\_\_ – O(n);

2. \_\_len\_\_ – O(1);

3. append – O(n);

4. \_\_str\_\_ – O(n);

5. pop – O(n);

6. clear – O(1);

7. change\_on\_start – O(n).

Алгоритм бинарного поиска в связном списке не рационален, т.к. для его

реализации требуется log(n) раз получать элементы каждый запрос займет O(n), тогда как в связном списке пройдя по элементам за O(n) будет найдет элемент.

## 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:

\_\_data=0

next=0

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():

head=None

length=0

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

if(head):

self.head=Node(head)

self.length=1

else:

self.head=None

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

return self.length

def append(self, element):

tmp = Node(element)

if self.length == 0:

self.head = tmp

self.length = 1

return

curr = self.head

while curr.next :

curr = curr.next

curr.next = tmp

self.length += 1

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

tmp=self.head

outp=[]

if self.head:

while(tmp):

outp.append(f"data: {tmp.get\_data()}, next: {tmp.next.get\_data() if tmp.next is not None else None}")

tmp=tmp.next

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

else:

return 'LinkedList[]'

def pop(self):

if len(self)==0:

raise IndexError("LinkedList is empty!")

elif len(self)==1:

self.length=0

self.head=None

return self

else:

tmp=self.head

while(tmp.next.next):

tmp=tmp.next

tmp.next=None

self.length-=1

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

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

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

temp=self.head

curr=Node(new\_data)

prev=temp

if n==1:

curr.next=self.head.next

self.head=curr

else:

i=1

while(i<n):

prev=temp

temp=temp.next

i+=1

prev.next = curr

curr.next=temp.next

def clear(self):

self.head=None

self.length=0