**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. 3341 |  | Бойцов В.А. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Ознакомиться с базовыми алгоритмами и структурами данных, использующимися в разработке в целом и в программировании на языка Python в частности. Разработать программу, реализующую однонаправленный список.

## 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  **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.

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 **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. *Пример того, как должно выглядеть взаимодействие с Вашим связным списком:*

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

***Вам не требуется реализовывать создание экземпляров ваших классов и вызов методов, это сделает проверяющая система.***

*В отчете вам требуется:*

1. Указать, что такое связный список. Основные отличия связного списка от массива.
2. Указать сложность каждого метода.
3. Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

## Основные теоретические положения

Связный список – структура данных, состоящая из элементов (узлов), каждый из которых помимо данных содержит в себе ссылки на следующий (предыдущий) элемент списка. Однонаправленный связный список состоит из элементов, содержащих ссылки на последующие элементы списка.

Основные отличия связного списка от массива заключаются в том, что массив хранится в памяти непрерывным блоком, в то время как узлы связного списка могут храниться отдельно друг от друга, что при больших объёмах данных позволяет оптимизировать хранение информации в памяти. Также в массиве доступ к элементам осуществляется по их индексам, в то время как в связном списке доступа по индексу элемента может и не быть.

Добавление элемента в конец списка (*append()*), создание строкового представления списка (*\_\_str\_\_())*, удаление последнего элемента (*pop()*) и замена данных в n-ном элементе с конца списка (*change\_on\_end()*) имеют сложность O(n), т.к. требуют итерации по всему списку.

Конструктор списка (*\_\_init\_\_()*), получение длины списка (*\_\_len\_\_()*) и очищение списка (*clean()*) имеют сложность O(1), т.к. не зависят от количества элементов в списке.

Если считать, что связный однонаправленный список отсортирован, то удобнее всего для бинарного поиска было бы хранить ссылки на первый, последний и центральный узлы рассматриваемого участка списка и менять их при поиске (что, конечно, потребует итерации по элементам списка, хранение длины участка между крайними элементами). Классический список в Python имеет индексацию и доступ по индексам, что означает, что бинарный поиск может обращаться к нужным для его работы крайним и центральным элементам по индексу, а находить эти индексы простой арифметикой.

## Выполнение работы

Реализуется класс *Node*, содержащий поля *\_\_data* (приватное поле, данные, хранящиеся в элементе *Node*) и *next* (ссылка на следующий элемент). Метод *\_\_init\_\_(self, data, next=None)* инициализирует объект класса *Node* (поле next по умолчанию *None*), *get\_data(self)* возвращает значение приватного поля *\_\_data*, *change\_data(self, new\_data)* заменяет данные объекта класса *Node* на новые, а *\_\_str\_\_(self)* с помощью форматной строки создаёт строковое представление объекта класса *Node*.

Далее реализуется класс *LinkedList*, имеющий два поля: *head* – данные первого элемента списка, и *length* – количество элементов в списке.

Метод *\_\_init\_\_(self, head)* – конструктор класса, который в зависимости от содержимого *head* инициализирует объект класса *LinkedList*.

Метод *\_\_len\_\_(self)* возвращает значение поля *length*.

Метод *append(self, element)* итерируется по всему списку до его конца (если длина списка равна 0, создаётся голова списка), добавляет в конец списка объект класса *Node*, инициализированный из значения *element*.

Метод *\_\_str\_\_(self)* итерируется по всему списку и возвращает представление списка форматной строкой.

Метод *pop(self)* итерируется по всему списку до его предпоследнего элемента (если список пустой, выбрасывается исключение *IndexError*), затем удаляет ссылку на последний элемент списка.

Метод *clean(self)* присваивает *head* значение *Node*, а *length* – значение 0.

Метод *change\_on\_end(self, n, new\_data)* в случае правильного индекса *n* удаляет *length-n*-ый элемент с начала (т.е. *n*-ый с конца), итерируясь до искомого элемента. Если элемента с индексом n нет в списке, выбрасывается исключение *KeyError*.

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()  print(linked\_list)  print(len(linked\_list))  linked\_list.append(10)  print(linked\_list)  print(len(linked\_list))  linked\_list.append(20)  print(linked\_list)  print(len(linked\_list))  linked\_list.pop()  print(linked\_list)  print(linked\_list)  print(len(linked\_list)) | LinkedList[]  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 | Тестирование основных методов списка |

## Выводы

В ходе выполнения лабораторной работы были изучены основные алгоритмы и структуры данных, применяющиеся в разработке на языке Python. Была написана программа, реализующая однонаправленный связный список с помощью двух классов Node и LinkedList.

# 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==None else self.next.\_\_data}"

class LinkedList:

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

if head == None:

self.head=None

self.length=0

else:

self.head=head

self.length=1

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

return self.length

def append(self, element):

new\_node = Node(element)

if self.length==0:

self.head=Node(element)

self.length=1

else:

tail=self.head

while(tail.next != None):

tail=tail.next

tail.next=new\_node

self.length+=1

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

if self.length==0:

return "LinkedList[]"

else:

output = f"LinkedList[length = {self.length}, ["

current=self.head

while(current!=None):

output=output+str(current)+"; "

current=current.next

output=output[:-2]+"]]"

return output

def pop(self):

if self.length==0:

raise IndexError("LinkedList is empty!")

else:

current=self.head

if self.length == 1:

self.head=None

else:

while(current.next.next!=None):

current=current.next

current.next=None

self.length-=1

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

if not(1<=n<=self.length):

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

else:

element = self.head

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

element=element.next

element.change\_data(new\_data)

def clear(self):

self.head=None

self.length=0

# Приложение Б Тестирование

Если результаты тестирования велики (больше 1 страницы), то их выносят в приложение.

Процесс тестирования можно представить в виде таблицы, например:

Таблица Б.2 - Примеры тестовых случаев

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| ... |  |  |  |

Обратите внимание, что в нумерации таблицы в приложении обязательно должен быть в качестве префикса номер самого приложения: Б.