**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**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3341 |  | Кудин А.А. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Изучение принципов работы и реализация базовых операций однонаправленного связного списка на языке программирования Python в рамках курса "Алгоритмы и структуры данных".

## 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?

## Выполнение работы

### Описание классов

**Класс** Node**:**

* Поля:
  + \_\_data: содержит данные узла списка.
  + next: содержит ссылку на следующий узел списка.
* Методы:
  + \_\_init\_\_(data, next=None): конструктор класса, инициализирующий узел с данными и ссылкой на следующий узел.
  + get\_data(): возвращает данные узла.
  + change\_data(new\_data): изменяет данные узла на новые.
  + \_\_str\_\_(): возвращает строковое представление узла в виде "data: <data>, next: <next\_data>".

**Класс** LinkedList**:**

* Поля:
  + head: ссылка на первый узел списка.
  + length: количество элементов в списке.
* Методы:
  + \_\_init\_\_(head=None): конструктор класса, создающий пустой список или список с одним элементом.
  + \_\_len\_\_(): возвращает длину списка.
  + append(element): добавляет новый узел в конец списка.
  + \_\_str\_\_(): возвращает строковое представление всего списка.
  + pop(): удаляет последний узел из списка.
  + clear(): полностью очищает список.
  + change\_on\_end(n, new\_data): изменяет данные n-го узла с конца списка.

### Алгоритм работы методов

Методы класса LinkedList реализованы следующим образом:

1. append(element): Создаёт новый узел с данными element и добавляет его в конец списка. Если список пуст, новый узел становится головным элементом. В противном случае происходит обход списка до последнего элемента и установка ссылки next последнего узла на новый узел.
2. pop(): Удаляет последний элемент списка. Если список пуст, выбрасывается исключение IndexError. Если в списке один элемент, head устанавливается в None. Если элементов больше, происходит обход до предпоследнего узла и его next устанавливается в None.
3. clear(): Устанавливает head в None и length в 0, тем самым полностью очищая список.
4. change\_on\_end(n, new\_data): Изменяет данные узла, находящегося на n-ой позиции с конца списка. Производит обход списка до нужного узла и изменяет его данные.

Разработанный программный код см. в приложении А.

## Тестирование

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 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(111)  l\_l.append(222)  l\_l.append(333)  print(l\_l)  print(len(l\_l))  l\_l.pop()  print(l\_l)  l\_l.append(333)  l\_l.change\_on\_end(3, 1)  print(l\_l) | data: 1, next: None  data: 1, next: 2  1  LinkedList[]  0  LinkedList[length = 3, [data: 111, next: 222; data: 222, next: 333; data: 333, next: None]]  3  LinkedList[length = 2, [data: 111, next: 222; data: 222, next: None]]  LinkedList[length = 3, [data: 111, next: 222; data: 222, next: 333; data: 333, next: None]] | OK |

## Выводы

В ходе выполнения лабораторной работы были успешно реализованы и протестированы классы **Node** и **LinkedList**, представляющие структуру данных "однонаправленный связный список". Работа с этой структурой данных позволила глубже понять принципы организации и манипуляции узлами в памяти.

# 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):

next\_data = self.next.get\_data() if self.next else "None"

return f"data: {self.\_\_data}, next: {next\_data}"

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 not self.head:

self.head = new\_node

else:

current = self.head

while current.next:

current = current.next

current.next = new\_node

self.length += 1

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

if not self.head:

return "LinkedList[]"

else:

current = self.head

nodes = []

while current:

node\_str = f"data: {current.get\_data()}, next: {current.next.get\_data() if current.next else 'None'}"

nodes.append(node\_str)

current = current.next

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

def pop(self):

if not self.head:

raise IndexError("LinkedList is empty!")

if self.length == 1:

self.head = None

else:

current = self.head

while current.next and current.next.next:

current = current.next

current.next = None

self.length -= 1

def clear(self):

self.head = None

self.length = 0

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

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

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

steps\_to\_move = self.length - n

current = self.head

for \_ in range(steps\_to\_move):

current = current.next

current.change\_data(new\_data)