**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 |  | Моисеева А.Е. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Изучить основные алгоритмы и структуры данных в языке Python, освоить основные методы для работы со списками в Python, затем создать программу и реализовать в ней односвязный список для хранения численных данных.

## Задание

Вариант 4

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\_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.

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:

Указать, что такое связный список. Основные отличия связного списка от массива.

Указать сложность каждого метода.

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 — это базовая структура данных, в которой элементы хранятся в узлах. Каждый узел содержит два компонента: данные, хранимые в данном элементе, ссылка на следующий узел - указатель, который направляет к следующему элементу в последовательности.

Списки могут быть односвязными – каждый элемент ссылается на следующий за ним, двусвязными – элемент хранит ссылку на следующий и предыдущий элементы, кольцевыми, где каждый узел соответственно содержит ссылку только на следующий элемент, последний же узел снова указывает на начало списка.

Отличия связного списка от массива следующие: массив размещён в памяти последовательно, в списке же узлы размещены в памяти отдельно друг от друга, в массиве возможен доступ по индексам, в то время как в списке для доступа к элементам необходимо последовательно перемещаться по последовательности.

Сложность методов:

Класс Node: инициализация элементов класса *\_\_init\_\_()*, геттер *get\_data()*, сеттер *change\_data()* и метод для формирования строкового представления узла *\_\_str\_\_()* имеют сложность O(1) и выполняются за константное время, поскольку не требуют перемещения по спику.

Класс LinkedList: инициализация элементов класса *\_\_init\_\_()*, метод для получения длины *len()*, метод для очистки списка *clear()* имеют сложность O(1) и выполняются за константное время, не требуют прохода по списку (в методе *len()* длина списка *length* хранится как свойство объекта). Метод добавления элемента в конец списка *append()*, удаления последнего элемента в списке *pop()*, метод для строкового представления *\_\_str\_\_()*, метод изменения значения элемента под определённым номером *change\_on\_start()* имеют сложность O(n) поскольку требуют прохождения по списку.

Реализация бинарного поиска в связном списке:

Для реализации алгоритма бинарного поиске в отсортированном связном списке будем использовать указатели на начало и конец списка, а также на середину. В дальнейшем при равенстве искомого значения и значения в среднем узле списка, возвращаем данное значение, если же оно больше искомого, перемещаем конечный указатель на узел перед указателем на середину, если меньше искомого – перемещаем начальный указатель на узел, следующий за средним.

Отличие бинарного поиска для связного списка и для классического списка Python заключается в следующем: в связных списках доступ к элементу требует последовательного прохода по списку, в массивах же доступ осуществляется с помощью индексов. Бинарный поиск в связном списке не настолько рационален, как в массиве, поскольку нахождение начального, конечного и серединного элементов и дальнейшее их перемещение быстрее с использованием индексов и их арифметики.

## Выполнение работы

Создается класс *Node*, который включает приватное поле *data* (хранит данные элемента *Node*) и публичное *next* (ссылка на следующий элемент). Метод *\_\_init\_\_()* инициализирует экземпляр класса *Node*, где *next* по умолчанию равен *None*. Метод *get\_data()* возвращает значение приватного поля *data*, а *change\_data()* заменяет данные объекта класса *Node* на новые. Метод *\_\_str\_\_()* создает строковое представление объекта класса *Node* с использованием форматной строки.

Далее реализуется класс *LinkedList*, содержащий два поля: *head* (данные первого элемента списка) и *length* (количество элементов в списке). Метод *\_\_init\_\_()* инициализирует объект класса *LinkedList* в зависимости от содержимого *head*. Метод *len()* возвращает значение поля *length*. Метод *append()* добавляет объект класса *Node*, инициализированный значением *element*, в конец списка. Метод *str()* возвращает представление списка в виде форматной строки. Метод *pop()* удаляет последний элемент списка, выбрасывая исключение *IndexError*, если список пуст. Метод *clean()* сбрасывает значения *head* на *Node* и *length* на 0. Метод *change\_on\_start()* удаляет *n*-ый элемент с начала списка, выбрасывая исключение *KeyError*, если элемента с индексом n нет в списке.

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 |
|  | node = Node(1)  print(node)  node.next = Node(2, None)  print(node)  print(node.get\_data())  l\_l = LinkedList()  l\_l.append(10)  l\_l.append(20)  l\_l.append(30)  print(l\_l)  print(len(l\_l))  l\_l.change\_on\_start(2, 2)  print(l\_l) | data: 1, next: None  data: 1, next: 2  1  LinkedList[length = 3, [data: 10, next: 20; data: 20, next: 30; data: 30, next: None]]  3  LinkedList[length = 3, [data: 10, next: 2; data: 2, next: 30; data: 30, next: None]] | Создаются элементы класса Node, затем выводятся для проверки работоспособности класса, затем создаётся список, в него добавляются три элемента, затем второй с начала меняется на два, выводится полученный связный список. |

## Выводы

В ходе выполнения работы были изучены основные алгоритмы и структуры данных в языке Python, а также освоены основные методы работы со списками. С применением полученных знаний была разработана программа, в рамках которой был реализован односвязный список для хранения численных данных.

# 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):

if self.next is not None:

return f"data: {self.get\_data()}, next: {self.next.get\_data()}"

else:

return f"data: {self.get\_data()}, next: None"

class LinkedList:

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

self.head = head

if head is not None:

self.length = 1

else:

self.length = 0

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

return self.length

def append(self, element):

new\_node = Node(element)

if self.head is not None:

current = self.head

while current.next is not None:

current = current.next

current.next = new\_node

else:

self.head = new\_node

self.length += 1

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

if self.head is None:

result = f"LinkedList[]"

else:

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

current = self.head

while current is not None:

if current.next is None:

result += f"data: {current.get\_data()}, next: None]]"

else:

result += f"data: {current.get\_data()}, next: {current.next.get\_data()}; "

current = current.next

return result

def pop(self):

if self.head is None:

raise IndexError("LinkedList is empty!")

elif self.head.next is None:

self.head = None

else:

current = self.head

while current.next.next is not None:

current = current.next

current.next = None

self.length -= 1

def clear(self):

self.head = None

self.length = 0

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

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

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

current = self.head

for i in range(n-1):

current = current.next

current.change\_data(new\_data)