**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. 3344 |  | Жаворонок Д.Н. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Введение в алгоритмы и структуры данных. Освоение алгоритмов и структур данных на языке Python.

## Задание.

Вариант 2. 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   \_\_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   delete\_on\_start(self, n) - удаление n-того элемента с НАЧАЛА списка. Метод должен выбрасывать исключение KeyError,  с сообщением "Element doesn't exist!", если количество элементов меньше n.

## Выполнение работы

Связный список - это структура данных, состоящая из узлов, где каждый узел содержит данные и ссылку (указатель) на следующий узел в списке. Основные отличия связного списка от массива:

Память: В массиве элементы хранятся в непрерывной области памяти, в то время как узлы связного списка могут быть разбросаны по памяти, связываясь друг с другом через указатели. Также связный список требует дополнительной памяти для хранения указателей на следующие узлы, в то время как массив требует память только для хранения элементов.

Размер: Размер массива фиксирован и определяется при создании, в то время как связный список может динамически изменять свой размер путем добавления или удаления узлов.

Вставка и удаление: Вставка и удаление элементов в массиве может быть затратной операцией, так как требуется сдвигать другие элементы. В связном списке вставка и удаление узлов более эффективны, так как требуется только изменить указатели.

Доступ к элементам: В массиве доступ к элементам осуществляется по индексу, что делает операцию доступа быстрой. В связном списке доступ к элементам может быть медленнее, так как требуется последовательно переходить от одного узла к другому.

|  |  |
| --- | --- |
| Сложность методов: | |
| О(n):  *LinkedList.\_\_str\_\_*  *append*  *pop*  *delete\_on\_start* | О(1):  *\_\_init\_\_*  *get\_data*  *Node.\_\_str\_\_*  *\_\_len\_\_*  \_\_clear\_\_ |
|
|
|
|
|

Для связного списка, бинарный поиск может быть реализован следующим образом: начиная с головы списка, двигаемся по элементам, чтобы найти средний элемент. Для этого используем две переменные, одна из которых движется на 2 ссылки за итерацию, а другая на одну. Когда первая переменная достигнет конца списка, вторая будет указывать на середину. Затем сравниваем средний элемент с ключом. Если ключ найден, поиск завершается. Если нет, определяем, какая половина списка будет использоваться для следующего поиска: левая, если ключ меньше среднего элемента, и правая, если больше. Процесс продолжается до тех пор, пока ключ не будет найден или список не будет исчерпан. Основное отличие бинарного поиска для связного списка от классического заключается в том, что в связном списке используются указатели для перемещения, что делает поиск более медленным из-за необходимости просмотра всех элементов.

## Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № пп | Входные данные | Выходные данные | Комментарии |
|  | node = Node(1)  print(node)  node.next = Node(2, None)  print(node) | data: 1, next: None  data: 1, next: 2 | - |
|  | 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 | - |

## Выводы

Были получены базовые знания об алгоритмах и структурах данных и их применении в языке Python.

## 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 \_\_str\_\_(self):

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

class LinkedList():

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

self.head = head

self.length = 1 if head else 0

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

return self.length

def append(self, element):

newElement = Node(element)

temp = self.head

if temp:

while temp.next:

temp = temp.next

temp.next = newElement

else:

self.head = newElement

self.length += 1

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

temp = self.head

if not temp:

return "LinkedList[]"

nodeArray = []

while temp:

nodeArray.append(temp)

temp = temp.next

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

def pop(self):

if not self.head:

raise IndexError("LinkedList is empty!")

elif not self.head.next:

self.head = None

self.length -= 1

else:

temp = self.head

while temp.next.next:

temp = temp.next

temp.next = None

self.length -= 1

def clear(self):

self.head = None

self.length = 0

def delete\_on\_start(self, n):

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

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

elif n == 1:

self.head = self.head.next

else:

temp = self.head

for \_ in range(n-2):

temp = temp.next

temp.next = temp.next.next

self.length -= 1