**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 |  | Анахин Е. Д. |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург

2024

## Цель работы

Изучение алгоритмов и структур данных на языке программирования Python, реализация программы при помощи односвязного списка.

## Задание

**Вариант 1**

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:

* data      # Данные элемента списка, приватное поле.
* next      # Ссылка на следующий элемент списка.

И следующие методы:

* \_\_init\_\_(self, data, next) - конструктор, у которого значения по умолчанию для аргумента next равно None.
* get\_data(self) - метод возвращает значение поля data (это необходимо, потому что в идеале пользователь класса не должен трогать поля класса Node).
* \_\_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.

**Linked List**

Класс, который описывает связный однонаправленный список.

Он должен иметь 2 поля:

* head        # Данные первого элемента списка.
* length     # Количество элементов в списке.

И следующие методы:

* \_\_init\_\_(self, head) - конструктор, у которого значения по умолчанию для аргумента head равно None.
  1. Если значение переменной head равна None, метод должен создавать пустой список.
  2. Если значение head не равно None, необходимо создать список из одного элемента.
* \_\_len\_\_(self) - перегрузка метода \_\_len\_\_, он должен возвращать длину списка (этот стандартный метод, например, используется в функции len).
* 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.
* \_\_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:

1. Если список пустой, то строковое представление: “LinkedList[]”
2. 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.

* pop(self) - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.
* clear(self) - очищение списка.
* delete\_on\_end(self, n) - удаление n-того элемента с конца списка. Метод должен выбрасывать исключение KeyError, с сообщением "Element doesn't exist!", если количество элементов меньше n.

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

## Выполнение работы

1) Связный список — это динамическая структура данных, состоящая из узлов. Каждый узел должен иметь ссылку на следующий элемент(возможно, и на предыдущий, если список двусвязный). Если элемент не имеет ссылки на следующий, то он является конечным. В отличие от массива, время вставки и удаления элемента является константым (если есть указатель на предыдущий элемент, иначе — тоже O(N) — время, необходимое, чтобы найти этот элемент), а время доступа к элементу является линейным. Также связный список может изменять свою длину, что не может делать не динамический массив. Элементы связного списка могут находиться в разных местах памяти, когда в массиве все элементы должны идти последовательно.

2) \_\_init\_\_() - сложность O(1)

\_\_len\_\_() - сложность O(1)

append() — сложность O(n)

\_\_str\_\_() - сложность O(n)

pop() - сложность O(n)

clear() - сложность O(1)

delete\_on\_end() - сложность O(n)

3) Возможная реализация бинарного поиска в односвязном списке может выглядеть так: находится длина списка — len. Далее, также как и в обычном списке, находится средний элемент. Если нужный элемент меньше, то верхней границей поиска является средний элемент, если меньше — средний элемент будет нижней границей. Добираться до нужного элемента нужно будет каждый раз, начиная с начала, если список будет односвязным. Реализация бинарного поиска будет отличаться сложностью: log(n) для обычного списка и n\*log(n) — для связного.

**Тестирование**

Результаты тестирования представлены в таблице 1.

Таблица 1 – Результаты тестирования

|  |  |  |
| --- | --- | --- |
| Тест | Выходные данные | Комментарии |
| node = Node(1)  print(node) # data: 1, next: None  node.next = Node(2, None)  print(node) # data: 1, next: 2  l\_l = LinkedList()  print(l\_l)  print(len(l\_l))  l\_l.append(10)  l\_l.append(20)  l\_l.append(30)  l\_l.append(40)  print(l\_l)  print(len(l\_l)) | data: 1, next: None  data: 1, next: 2  1  LinkedList[]  LinkedList[length = 4, [data: 10, next: 20; data: 20, next: 30; data: 30, next: 40; data: 40, next: None]]  4 |  |

**Выводы**

Были изучены алгоритмы и структуры данных на языке программирования Python, реализована программы при помощи односвязного списка.

# 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.\_\_data}, next: {self.next.get\_data() if self.next is not None else None}"

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 self.head is None:

self.head = new\_node

else:

current = self.head

while current.next is not None:

current = current.next

current.next = new\_node

self.length += 1

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

if self.head is None:

return "LinkedList[]"

else:

nodes = []

current = self.head

while current is not None:

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

current = current.next

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

def pop(self):

if self.head is None:

raise IndexError("LinkedList is empty!")

if 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 delete\_on\_end(self, n):

if n > len(self):

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

if n <= 0:

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

# Если n равно длине списка, удаляем первый элемент

if n == len(self):

self.head = self.head.next

else:

# Ищем элемент, предшествующий удаляемому

current = self.head

for \_ in range(len(self) - n - 1):

current = current.next

# Удаляем элемент

current.next = current.next.next

self.length -= 1