**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. Вариант 2**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Поддубный В.А. |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург

2024

## Цель работы

Практическое применение объектно-ориентированного программирования (ООП) для создания собственной реализации односвязного списка на Python.

## Задание

**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

**1. Что такое связный список?**

Односвязный список — это линейная структура данных, состоящая из узлов, где каждый узел содержит данные (data) и ссылку (next) на следующий узел в последовательности.

**Отличия от массива:**

* **Организация в памяти:** Массив хранится в непрерывном блоке памяти, а узлы связного списка могут располагаться в произвольных местах памяти, связанных ссылками.
* **Доступ к элементам:** В массиве доступ к элементам осуществляется по индексу за O(1) время, в связном списке требуется пройти по ссылкам, что занимает O(n) времени в худшем случае.
* **Вставка/удаление:** Вставка/удаление в середине массива требует сдвига элементов, что занимает O(n) времени. В связном списке достаточно изменить ссылки, что занимает O(1) времени после нахождения нужного узла.

**2. Сложность методов:**

* \_\_len\_\_: O(n) - необходимо пройти по всем узлам, чтобы подсчитать количество элементов.
* append: O(n) - требуется найти последний узел, чтобы добавить новый.
* pop: O(n) - необходимо найти предпоследний узел, чтобы удалить последний.
* delete\_on\_start: O(n) - нужно пройти до n-1 узла, чтобы удалить n-ый.
* clear: O(1) - просто сбрасывает голову списка и длину.

**3. Бинарный поиск в связном списке:**

Классический бинарный поиск неэффективен для связного списка, так как он требует прямого доступа к середине списка, что занимает O(n) времени.

**Возможное решение:**

* **Использование сортированного списка:** Если отсортировать список, то каждый поиск будет занимать O(log n), однако сама сортировка займет O(n log n)

**Отличия от бинарного поиска в классическом списке Python:**

* В классическом списке Python бинарный поиск работает за O(log n) времени, так как доступ к элементам по индексу занимает O(1) время.
* В связном списке бинарный поиск в его классической форме не применим из-за необходимости последовательного доступа к элементам, но использование сортированного списка позволяет нам это сделать также за O(log n)

## Тестирование

Результаты тестирования содержатся в таблице 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 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 | 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 | Программа сработала корректно. |

## Выводы

Односвязные списки обладают своими преимуществами (эффективная вставка/удаление) и недостатками (медленный произвольный доступ) по сравнению с массивами. Выбор между ними зависит от конкретных требований задачи к эффективности операций и использованию 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 \_\_str\_\_(self):  
 return f"data: {self.\_\_data}, next: {self.next.\_\_data if self.next else None}"  
  
  
class LinkedList:  
 def \_\_init\_\_(self, head: Node = 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):  
 self.length += 1  
 current\_node = self.head  
 if current\_node is None:  
 self.head = Node(element)  
 return  
 while current\_node.next:  
 current\_node = current\_node.next  
 current\_node.next = Node(element)  
  
 def \_\_str\_\_(self):  
 if self.length == 0: return "LinkedList[]"  
 current\_node = self.head  
 node\_to\_str\_list = []  
 while current\_node:  
 node\_to\_str\_list.append(current\_node.\_\_str\_\_())  
 current\_node = current\_node.next  
 linkedlist\_elements = "; ".join(node\_to\_str\_list)  
 return f"LinkedList[length = {self.length}, [{linkedlist\_elements}]]"  
  
 def pop(self):  
 if self.length == 0: raise IndexError("LinkedList is empty!")  
 current\_node = self.head  
 if self.length == 1:  
 self.head = None  
 else:  
 while current\_node.next.next:  
 current\_node = current\_node.next  
 current\_node.next = None  
 self.length -= 1  
  
 def delete\_on\_start(self, n):  
 if self.length < n or n <= 0: raise KeyError("Element doesn't exist!")  
 current\_node = self.head  
 count = 1  
 if n == 1:  
 self.head = current\_node.next  
 else:  
 while count + 1 != n:  
 count += 1  
 current\_node = current\_node.next  
 current\_node.next = current\_node.next.next  
 self.length -= 1  
  
 def clear(self):  
 while self.head:  
 self.pop()