**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: Введение в алгоритмы и структуры данных**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3344 |  | Гусева Е.А. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Целью работы является ознакомление с алгоритмами и структурами данных в языке Python. Написание кода для лабораторной работы 2.

## 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:

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

И следующие методы:

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

*Linked List:* класс, который описывает связный однонаправленный список.Он должен иметь 2 поля:

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

И следующие методы:

1. *\_\_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.
2. *\_\_len\_\_(self)* - перегрузка метода *\_\_len\_\_*, он должен возвращать длину списка (этот стандартный метод, например, используется в функции *len*).
3. *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.
4. *\_\_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]”,*
5. *pop(self)*- удаление последнего элемента. Метод должен выбрасывать исключение *IndexError* с сообщением *"LinkedList is empty!",* если список пустой.
6. *clear(self)*- очищение списка.
7. *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.

## 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). Связные списки бывают двух видов: односвязные и двусвязные(во втором варианте есть ссылка еще и на предыдущий узел). Отличие такой структуры данных от массива состоит в том, что в связном списке операции вставки и удаления элементов выполняются быстрее, чем в массиве, легко менять размер связного списка, т. к. он не фиксирован в отличии от размера массива. 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) Сложности методов. О(1):

\_\_init\_\_

get\_data

Node.\_\_str\_\_

\_\_len\_\_

\_\_clear\_\_

append(если добавляем head)

pop(если список пуст)

delete\_on\_start(если удаляем первый элемент)

О(n):

LinkedList.\_\_str\_\_

append

pop

delete\_on\_start

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. к. Основное отличие реализации бинарного поиска для связного списка от реализации для классического списка заключается в том, что в связном списке поиск осуществляется путем перемещения указателей, а не прямого доступа к элементам по индексу. Это делает поиск более медленным, так как требуется пройти по всем элементам до нужного.

## Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Тест | Выходные данные | Комментарии |
|  | node = Node(1)  print(node)  node.next = Node(2, None)  print(node)  print(node.get\_data())  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))  l\_l.delete\_on\_end(3)  print(l\_l) | data: 1, next: None  data: 1, next: 2  1  LinkedList[]  0  LinkedList[length = 4, [data: 10, next: 20; data: 20, next: 30; data: 30, next: 40; data: 40, next: None]]  4  LinkedList[length = 3, [data: 10, next: 30; data: 30, next: 40; data: 40, next: None]] | Данные обработаны корректно |
|  | node = Node(1)  print(node)  node.next = Node(2, None)  print(node)  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.delete\_on\_end(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 = 2, [data: 111, next: 222; data: 222, next: None]] | Данные обработаны корректно |

## Выводы

Были получены базовые навыки работы с алгоритмами и структурами данных. Была написана программа для лабораторной работы, изучены сложность алгоритмов и методы работы со связными списками.

# Приложение А Исходный код программы

Название файла: main\_for\_lb2(cs).py

class Node:

def \_\_init\_\_(self, data, next=None):

self.data, self.next = data, next

def get\_data(self):

return self.data

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

if self.next:

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

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

class LinkedList:

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

self.head, self.length = head, 0

while head:

head=head.next

self.length+=1

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

return self.length

def append(self, element):

self.length += 1

if self.head:

tmp = self.head

while tmp.next:

tmp = tmp.next

tmp.next = Node(element)

else:

self.head = Node(element)

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

if self.head is not None:

res = f'LinkedList[length = {self.length}, ['

tmp = self.head

while tmp:

res += str(tmp) + '; '

tmp = tmp.next

res = res[:-2] + ']]'

return res

else:

return 'LinkedList[]'

def pop(self):

if not self.head:

raise IndexError("LinkedList is empty!")

if self.length == 1:

self.\_\_init\_\_()

if self.head:

tmp=self.head

while tmp.next.next:

tmp=tmp.next

tmp.next=None

self.length-=1

def clear(self):

self.\_\_init\_\_()

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

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

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

else:

index = self.length - n

tmp = self.head

if index == 0:

self.head = tmp.next

else:

for i in range(1, index):

tmp = tmp.next

tmp.next = tmp.next.next

self.length -= 1