**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: Алгоритмы и структуры данных в Python

|  |  |  |
| --- | --- | --- |
| Студент гр. 3341 |  | Самокрутов А.Р. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Цель лабораторной работы состоит в ознакомлении с основными алгоритмами и структурами данных. Для достижения цели необходимо разработать программу на языке программирования Python, реализующую зависимые классы *Node* и *LinkedList*. Класс *Node* описывает узел списка с полями, в которых хранятся некоторые данные и ссылка на следующий элемент, а также методы для инициализации, получения данных и строкового представления объекта. Класс *LinkedList* описывает однонаправленный связный список с методами инициализации, возвращения длины списка, добавления нового элемента, строкового представления объекта, удаления последнего элемента, очищения списка, удаления *n*-ого элемента с начала списка.

## 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.

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 — структура данных, состоящая из элементов, называемых узлами, содержащих помимо собственных данных ещё и ссылку на следующий и/или предыдущий элемент списка. Если узел содержит ссылку только на следующий или предыдущий узел, то список называется односвязным, если на оба — двусвязным.

Главное отличие массива от списка заключается в том, что в массиве память хранится непрерывным блоком в памяти, в то время как элементы в связном списке благодаря ссылкам на другие узлы могут храниться отдельно друг от друга, что удобно при большом объёме данных в каждом элементе. Однако из-за такого расположения элементов в памяти значительно усложняется доступ к элементу по индексу.

Все методы класса *Node* имеют сложность *O(1)*.

Добавление элемента в конец списка, создание строкового представления, удаление последнего элемента, удаление *n*-ого элемента с начала списка имеют сложность *O(n)*, т.к. требуют итерации по списку.

Инициализация списка, нахождение длины списка и очистка списка имеют сложность *O(1)*, т.к. не требуют обхода списка.

В отсортированном однонаправленном связном списке для реализации бинарного поиска можно хранить в памяти указатели на «левый» и «правый» концы рассматриваемого участка списка, изначально равные первому и последнему элементам соответственно, а также «средний» между ними элемент. Находить средний элемент можно, например, итерируя от «левого» элемента на половину расстояния от «левого» до «правого». Далее эти указатели нужно менять в соответствии с обычным ходом бинарного поиска. Отличием от бинарного поиска в обычном списке в Python будет сложность алгоритма: найти *n*-ый элемент списка можно за константное время, в то время как доступ к *n*-ому элементу с связном списке будет осуществляться за *O(n)*.

## Выполнение работы

Реализуется класс *Node* с полями *\_\_data* (данные, приватное поле) и *next* (ссылка на следующий узел).

Определяется конструктор *\_\_init\_\_(self, data, next=None)*, присваивающий полям *\_\_data* и *next* значения аргументов *data* и *next* соответственно. Метод *get\_data(self)* возвращает значение приватного поля *\_\_data*. Метод *\_\_str\_\_(self)* возвращает строковое представление объекта класса *Node* в соответствии с заданным в задании форматом.

Далее описывается класс *LinkedList* с полями *head* (первый узел в списке) и *\_\_length* (длина списка, приватное поле).

Определяется конструктор класса *\_\_init(self, head=None)*, присваивающий значение переменной *head* полю *head*, а полю *\_\_length* — значение *1* если значение *head* отлично от *None*, в ином случае — *0.* Метод *\_\_len\_\_(self)* возвращает значение приватного поля *\_\_length*. Метод *append(self,* *element)* присваивает полю *head* значение объекта *Node(element)*, если длина списка равна нулю, иначе итерируется по всему списку до его конца и меняет значение поля *next* последнего элемента на *Node(element)*. При этом длина списка увеличивается на *1*. Метод *\_\_str\_\_(self)* проходит по всему списку и возвращает строковое представление объекта класса *LinkedList* в соответствии с условием. Метод *pop(self)* удаляет последний элемент списка: если список пустой, то вызывается ошибка *IndexError*; если в списке элемент всего один, то он меняет значение поля *head* на *None*; иначе итерируется по списку до предпоследнего элемента и меняет значение его ссылки на следующий элемент на *None*. При этом значение длины списка уменьшается на *1*. Метод *clear(self)* меняет значение поля *head* на *None*, тем самым очищая список. Метод *delete\_on\_start(self, n)* действует аналогично *pop()*, но итерируется не до предпоследнего элемента, а до *(n-1)*-го. При некорректном значении n вызывается ошибка *KeyError*.

Разработанный код см. в приложении А.

## Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | linked\_list = LinkedList()  try:  linked\_list.pop()  except Exception as exc:  print(exc)  linked\_list.append(100)  print(str(linked\_list))  linked\_list.pop()  print(str(linked\_list))  linked\_list.append(200)  linked\_list.append(300)  print(str(linked\_list))  linked\_list.pop()  print(str(linked\_list)) | LinkedList is empty!  LinkedList[length = 1, [data: 100, next: None]]  LinkedList[]  LinkedList[length = 2, [data: 200, next: 300; data: 300, next: None]]  LinkedList[length = 1, [data: 200, next: None]] | Проверка работы метода pop() на граничных случаях. |
|  | list = LinkedList()  list.append(50)  try:  list.delete\_on\_start(0)  except Exception as exc:  print(exc)  list.append(100)  print(str(list))  list.delete\_on\_start(1)  print(str(list))  list.append(200)  list.append(300)  print(str(list))  list.delete\_on\_start(3)  print(str(list)) | "Element doesn't exist!"  LinkedList[length = 2, [data: 50, next: 100; data: 100, next: None]]  LinkedList[length = 1, [data: 100, next: None]]  LinkedList[length = 3, [data: 100, next: 200; data: 200, next: 300; data: 300, next: None]]  LinkedList[length = 2, [data: 100, next: 200; data: 200, next: None]] | Проверка работы метода delete\_on\_start() на граничных случаях. |

## Выводы

Была изучены структура данных однонаправленный связный список и различные алгоритмы, связанные с ней.

Был реализован однонаправленный связный список на языке программирования Python с использованием двух зависимых классов Node и LinkedList, описанных в работе.

# 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 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):

node = Node(element)

if self.\_\_length == 0:

self.head = node

else:

current = self.head

while current.next is not None:

current = current.next

current.next = node

self.\_\_length += 1

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

if (self.\_\_length == 0):

return "LinkedList[]"

string = f"LinkedList[length = {self.\_\_length}, ["

current = self.head

while (current != None):

string += str(current)

if (current.next != None):

string += "; "

current = current.next

string += "]]"

return string

def pop(self):

if self.head is None:

raise IndexError("LinkedList is empty!")

if self.\_\_length == 1:

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\_start(self, n):

if (n > self.\_\_length) or (n < 1):

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

if n == 1:

self.head = self.head.next

else:

current = self.head

for \_ in range(n - 2):

current = current.next

current.next = current.next.next

self.\_\_length -= 1