**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 |  | Мурдасов М.К. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

Изучение алгоритмов и структур данных в языке Python.

## Задание

Вариант 4

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  **change\_data(self, new\_data)**- метод меняет значение поля data объекта 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**change\_on\_start(self, n, new\_data)**- изменение поля data n-того элемента с НАЧАЛА списка на new\_data. 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:*

1. Указать, что такое связный список. Основные отличия связного списка от массива.
2. Указать сложность каждого метода.
3. Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

## Выполнение работы

Связный список - это линейная структура данных, в которой элементы (узлы) хранятся в отдельных областях памяти и связаны друг с другом указателями. Каждый узел содержит данные и ссылку на следующий (иногда и на предыдущий) узел в списке.

**Связный список**

* **Структура данных:** Линейная структура данных, состоящая из узлов, каждый из которых содержит данные и ссылку на следующий узел.
* **Доступ к элементам:** Доступ к элементам осуществляется путем последовательного перехода от одного узла к другому, следуя ссылкам.
* **Вставка и удаление:** Вставка и удаление элементов могут выполняться в любом месте списка, перенаправляя ссылки.
* **Память:** Связные списки не требуют непрерывного блока памяти, как массивы.
* **Производительность:** Доступ к элементам может быть медленным, особенно для больших списков, из-за необходимости последовательного перехода.
* **Использование:** Подходит для ситуаций, когда требуется частая вставка и удаление элементов в произвольных местах.

**Массив**

* **Структура данных:** Непрерывная область памяти, состоящая из элементов фиксированного типа данных.
* **Доступ к элементам:** Доступ к элементам осуществляется с помощью индексов, что обеспечивает быстрый прямой доступ.
* **Вставка и удаление:** Вставка и удаление элементов могут быть дорогостоящими операциями, поскольку они требуют сдвига других элементов.
* **Память:** Элементы массива хранятся в непрерывной области памяти, что обеспечивает эффективное использование памяти.
* **Производительность:** Доступ к элементам быстрый, особенно для больших массивов, благодаря прямому доступу.
* **Использование:** Подходит для ситуаций, когда требуется быстрый доступ к элементам по их индексам и когда частота вставок и удалений невелика.

**Сложности методов:**

* О(1):
* \_\_init\_\_
* get\_data()
* change\_data()
* Node.\_\_str\_\_
* \_\_len\_\_
* append() – при self.head == None
* pop() - при self.head == None
* change\_on\_start() – при self.length < n or n <= 0
* \_\_clear\_\_

* O(n):
* LinkedList.\_\_str\_\_
* append()
* pop()
* change\_on\_start

**Реализация бинарного поиска в связном списке:**

1. Найти средний узел списка.
2. Если значение среднего узла равно искомому значению, вернуть узел.
3. Если значение среднего узла меньше искомого значения, перейти к правой половине списка.
4. Если значение среднего узла больше искомого значения, перейти к левой половине списка.
5. Повторять шаги 1-4, пока не будет найден узел с искомым значением или не будет достигнуто начало или конец списка.

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

* В связном списке нет прямого доступа к элементам по индексу, поэтому поиск среднего узла требует перебора списка.

**Тестирование**

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | 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.change\_on\_start(2, 2)  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 = 4, [data: 10, next: 2; data: 2, next: 30; data: 30, next: 40; data: 40, next: None]] | Корректно |

## Выводы

Была написана программа на Python с применением алгоритмов и структур данных.

# Приложение А Исходный код программы

Название файла: Murdasov\_Mikhail\_lb2.py

class Node:

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

self.data = data

self.next = next

def get\_data(self):

return self.data

def change\_data(self, new\_data):

self.data = new\_data

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

if self.next != None:

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

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

class LinkedList:

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

if head == None:

self.head = None

self.length = 0

else:

self.head = Node(head)

self.length = 1

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

return self.length

def append(self, element):

if self.head == None:

self.head = Node(element)

else:

node = self.head

while node.next != None:

node = node.next

node.next = Node(element)

self.length += 1

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

if self.head == None:

return "LinkedList[]"

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

node = self.head

while node != None:

if node.next == None:

res += f"{str(node)}"

break

res += f"{str(node)}; "

node = node.next

res += "]]"

return res

def pop(self):

if self.head == None:

raise IndexError("LinkedList is empty!")

elif self.head.next == None:

self.head = None

self.length = 0

else:

node = self.head

while node.next.next != None:

node = node.next

node.next = None

self.length -= 1

def change\_on\_start(self, n, new\_data):

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

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

else:

node = self.head

for i in range(n-1):

node = node.next

node.data = new\_data

def clear(self):

self.head = None