**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**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3342 |  | Галеев А.Д. |
| Преподаватель |  | Шалагинов И.В. |

Санкт-Петербург

2024

## Цель работы

Изучение основных алгоритмов, методов и структур данных для эффективной обработки информации в программах.

## Задание

#### Реализовать связный *однонаправленный* список. Для этого необходимо реализовать 2 зависимых класса:

#### Node - Класс, который описывает элемент списка.

#### Linked List - Класс, который описывает связный однонаправленный список.

## Основные теоретические положения

Для решения задач в программе использовались функции стандартной библиотеки языка Python

## Выполнение работы

Связный список - это структура данных, которая состоит из узлов, каждый из которых содержит какое-то значение и ссылку (или указатель) на следующий узел в списке. Главным отличием от массива является то, что массив это статическая структура данных, где элементы располагаются последовательно в памяти.

Основные отличия между связным списком и массивом:

Хранение данных в памяти: В связном списке каждый элемент может быть расположен в разных местах в памяти, и доступ к элементам осуществляется через ссылки между ними. В массиве все элементы хранятся последовательно в памяти, и доступ к ним осуществляется через индексы.

Размер и изменяемость: Размер массива фиксирован и обычно определяется при его создании, а размер связного списка может изменяться динамически при добавлении или удалении элементов.

Вставка и удаление элементов: В связном списке вставка и удаление элементов происходят быстрее, чем в массиве, особенно если это касается элементов в середине списка. При этом не требуется перемещать все последующие элементы, как в массиве.

Память: Связный список может потреблять больше памяти на хранение указателей или ссылок на следующие элементы, чем массив на хранение данных.

Сложность доступа к элементам: В массиве доступ к элементу по индексу происходит за константное время (O(1)), а в связном списке доступ к элементу может быть медленнее, так как требуется пройти от начала списка до нужного элемента, что занимает линейное время в среднем (O(n)).

Сложность каждого метода:

**\_\_init\_\_** Сложность: O(1).

**\_\_len\_\_** Сложность: O(1).

**append** Сложность: O(n), где n - количество элементов в списке.

**\_\_str\_\_** Сложность: O(n), где n - количество элементов в списке.

**pop** Сложность: O(n), где n - количество элементов в списке.

**clear** Сложность: O(1).

**change\_on\_end** Сложность: O(n), где n - количество элементов в списке.

Для реализации бинарного поиска в связном списке нужно сначала определить длину списка (что может потребовать пройти по всему списку), а затем использовать двоичный поиск, учитывая, что доступ к элементу осуществляется только последовательно.

В реализации бинарного поиска для связного списка особенность заключается в том, что нет прямого доступа к элементам по индексу, как в случае с классическим списком Python. Поэтому для доступа к элементам по индексу и выполнения бинарного поиска необходимо пройти по списку с помощью указателей. Это означает, что худшая сложность бинарного поиска в связном списке остается O(n), так как для доступа к середине списка требуется пройти половину его длины. Однако, если список предварительно отсортирован и поддерживается в упорядоченном виде, бинарный поиск все равно будет эффективным.

:

**Тестирование**

Результаты тестирования представлены в табл. 1  
Таблица 1 – Результаты тестирования

|  |  |  |
| --- | --- | --- |
| № проверки | Входные данные | Выходные данные |
| 1. | 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.change\_on\_end(3, 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 = 3, [data: 1, next: 222; data: 222, next: 333; data: 333, next: None]] |

## Выводы

Было изучено понятие алгоритмов и структур данных.

Разработана программа выполняющая обработку данных с помощью двух зависимых классов.

# Приложение А Исходный код программы

Название файла: main\_lb2

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):

next\_data = self.next.data if self.next else None

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

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 not self.head:

self.head = new\_node

else:

current\_node = self.head

while current\_node.next:

current\_node = current\_node.next

current\_node.next = new\_node

self.length += 1

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

if not self.head:

return "LinkedList[]"

else:

result = "LinkedList[length = " + str(self.length) + ", ["

current\_node = self.head

while current\_node:

result += str(current\_node) + "; "

current\_node = current\_node.next

result = result.rstrip("; ") + "]]"

return result

def pop(self):

if not self.head:

raise IndexError("LinkedList is empty!")

elif not self.head.next:

self.head = None

else:

previous\_node = None

current\_node = self.head

while current\_node.next:

previous\_node = current\_node

current\_node = current\_node.next

previous\_node.next = None

self.length -= 1

def clear(self):

self.head = None

self.length = 0

def change\_on\_end(self, n: int, new\_data) -> None:

if len(self) < n or n < 1:

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

n = len(self) - n

if not n:

self.head.data = new\_data

return

\_node = self.head

for \_ in range(n):

\_node = \_node.next

\_node.data = new\_data