

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. 3342

Лапшов К.Н.

Преподаватель

Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Изучение основ работы с линейными списками и их практическая реализация на языке программирования Python.

Задание

Вариант 3.

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::

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

И следующие методы:

- **__init__(self, data, next)** - конструктор, у которого значения по умолчанию для аргумента next равно None.
- **get_data(self)** - метод возвращает значение поля data (это необходимо, потому что в идеале пользователь класса не должен трогать поля класса Node).
- **change_data(self, new_data)** - метод меняет значение поля data объекта Node.
- **__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::

- **head** # Данные первого элемента списка.

- **length** # Количество элементов в списке.

И следующие методы:

- **__init__(self, head)** - конструктор, у которого значения по умолчанию для аргумента head равно None.

- **__len__(self)** - перегрузка метода **__len__**, он должен возвращать длину списка (этот стандартный метод, например, используется в функции **len**).

- **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.

- **__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.

- **pop(self)** - удаление последнего элемента. Метод должен выбрасывать исключение **IndexError** с сообщением "LinkedList is empty!", если список пустой.максимальная скорость (в км/ч, положительное целое число)

- **clear(self)** - очищение списка.

- **change_on_end(self, n, new_data)** - меняет значение поля **data** n-того элемента с конца списка на **new_data**. Метод должен выбрасывать исключение **KeyError**, с сообщением "Element doesn't exist!", если количество элементов меньше **n**.

Выполнение работы

Были разработаны классы Node и LinkedList, в которых реализованы соответствующие методы в соответствии с поставленной задачей.

Разработанный программный код см. в приложении А.

Связанный список — это структура данных, состоящая из последовательности элементов, называемых узлами, где каждый узел содержит данные и ссылку (или указатель) на следующий элемент в списке. Каждый узел в связанном списке соединен с последующим узлом, образуя цепочку элементов.

Основные отличия связанного списка от массива:

- **Хранение данных:** В массиве элементы хранятся в последовательной области памяти, в то время как в связанном списке каждый элемент хранится отдельно, и ссылки соединяют их в список.
- **Вставка и удаление элементов:** В случае связанного списка вставка и удаление элементов могут быть выполнены за постоянное время ($O(1)$), включая начало и середину списка, в то время как в массиве эти операции могут потребовать сдвиг всех последующих элементов, что может быть затратным по времени ($O(n)$).
- **Доступ к элементам:** В массиве элементы доступны по индексу за постоянное время ($O(1)$), в то время как в связанном списке время доступа к элементам зависит от их позиции и может быть линейным ($O(n)$) в худшем случае.

Сложность каждого метода в реализованном коде:

- **__init__:** $O(1)$ - создание объекта LinkedList или Node.
- **__len__:** $O(n)$ - вычисление длины списка.
- **append:** $O(n)$ - добавление элемента в конец списка.
- **__str__:** $O(n)$ – создание строкового представления списка.
- **pop:** $O(n)$ - удаление последнего элемента списка.

- **change_on_end: $O(n)$** - изменение элемента по позиции с конца списка.
- **clear: $O(1)$** - очистка списка.

Реализация бинарного поиска в связном списке отличается от классического списка Python тем, что в связном списке отсутствует прямой доступ к элементам по индексу, что мешает применению той же стратегии разделения списка на две части, как в массиве. Вместо этого в бинарном поиске для связного списка используются указатели на начало и конец текущего диапазона, которые итеративно сокращаются вдвое, пока не будет найден искомый элемент или не будет определено его отсутствие.

Выводы

Был создан связный список с использованием классов на языке Python, после чего была проанализирована производительность методов этого класса и рассмотрена возможность применения бинарного поиска в связном списке.

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 change_data(self, new_data):
        self.data = new_data

    def __str__(self):
        next_data = self.next.data if self.next is not None else None
        return f"data: {self.data}, next: {next_data}"

class LinkedList:
    def __init__(self, head=None):
        self.head = None
        self.length = 0

        if head is not None:
            self.append(head)

    def __len__(self):
        return self.length

    def append(self, element):
        new_element = Node(element)

        if self.length == 0:
            self.head = new_element
            self.length += 1
            return

        tmp_node = self.head
        while tmp_node.next is not None:
            tmp_node = tmp_node.next

        tmp_node.next = new_element
        self.length += 1

    def __str__(self):
        if self.length == 0:
            return "LinkedList[]"

        result = "LinkedList["
        result += f"length = {self.length}, ["
        tmp_node = self.head
```



```

        while tmp_node.next is not None:
            result += f"data: {tmp_node.data}, next:
{tmp_node.next.data};"
            tmp_node = tmp_node.next

        result += f"data: {tmp_node.data}, next: None]"
        return result

def pop(self):
    if self.length == 0:
        raise IndexError("LinkedList is empty!")
    if self.length == 1:
        self.clear()
        return

    tmp_node = self.head
    while tmp_node.next.next is not None:
        tmp_node = tmp_node.next

    tmp_node.next = None
    self.length -= 1

def change_on_end(self, n, new_data):
    if self.length < n or n <= 0:
        raise KeyError("Element doesn't exist!")

    count = 0
    tmp_node = self.head
    while count != self.length-n:
        tmp_node = tmp_node.next
        count += 1

    tmp_node.data = new_data

def clear(self):
    self.head = None
    self.length = 0

```