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 _____ Иванов С. С.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург
2024

Цель работы

Изучение работы с линейным списком и его реализации на языке Python.

Задание

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:

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_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 созданы классы Node и LinkedList. Реализованы методы согласно условию.

Связный список — это структура данных, представляющая собой последовательность элементов, в которой каждый элемент хранит ссылку на следующий элемент в последовательности. Каждый элемент списка называется узлом. Узлы могут содержать как данные (например, числа или объекты), так и ссылки на следующие узлы.

Основные отличия связного списка от массива:

1. **Хранение данных:** В массиве элементы хранятся в последовательной области памяти, в то время как в связном списке каждый элемент хранится отдельно, и ссылки соединяют их в список.
2. **Доступ к элементам:** В массиве доступ к элементам осуществляется по индексу за константное время (O(1)), тогда как в связном списке время доступа к элементам зависит от их позиции в списке и может быть линейным (O(n)) в худшем случае.
3. **Вставка и удаление элементов:** В связном списке вставка и удаление элементов могут быть выполнены за константное время (O(1)), даже в начале и середине списка, тогда как в массиве эти операции могут потребовать сдвиг всех последующих элементов, что может быть дорого с точки зрения времени (O(n)).

Сложность каждого метода в реализованном коде:

1. **__init__**: O(1) - создание объекта LinkedList или Node.
2. **__len__**: O(n) - вычисление длины списка.
3. **append**: O(n) - добавление элемента в конец списка.
4. **__str__**: O(n) - создание строкового представления списка.
5. **pop**: O(n) - удаление последнего элемента списка.
6. **change_on_end**: O(n) - изменение элемента по позиции с конца списка.
7. **clear**: O(1) - очистка списка.

Отличие реализации бинарного поиска для связного списка от классического списка Python заключается в том, что для связного списка нет прямого доступа к элементам по индексу, что делает невозможным применение такой же стратегии деления списка на две части, как в случае массива. Вместо этого бинарный поиск в связном списке будет осуществляться с использованием указателей на начало и конец текущего диапазона, итеративно сокращая этот диапазон вдвое, пока не будет найден искомый элемент или не определится его отсутствие.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	node = Node(1) print(node) # data: 1, next: None node.next = Node(2, None) print(node) # data: 1, next: 2 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_end(2, “new”) 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: 20; data: 20, next: 30; data: “new”, next: 40; data: 40, next: None]]	Верный вывод

Выводы

Был реализован связный список с помощью классов на языке Python. Исследована скорость работы методов созданного класса и возможность использования бинарного поиска в связном списке.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from typing import Any


class Node:
    def __init__(self, data: Any, next=None) -> None:
        self.data = data
        self.next = next

    def get_data(self) -> Any:
        return self.data

    def __str__(self) -> str:
        _next_data = self.next.data if self.next is not None else
None
        return f"data: {self.get_data()}, next: {_next_data}"

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



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

        if head is None:
            return

        self.append(head)

    def __len__(self) -> int:
        return self.length

    def append(self, element: Any) -> None:
        if not len(self):
            self.head = Node(element)
            self.length += 1
            return

        _node = self.head

        while _node.next != None:
            _node = _node.next

        _node.next = Node(element)
        self.length += 1


    def __str__(self) -> str: # LinkedList[length = 2, [data: 10,
next:20; data: 20, next: None]]
        _res = 'LinkedList['
```

```
        if not len(self):
            return _res + ']'

        _res += f"length = {len(self)}, ["

        _node = self.head
        while _node != None:
            _res += str(_node) + '; '
            _node = _node.next

        return _res[:-2] + ']]'

    def pop(self) -> None:
        if not len(self):
            raise IndexError("LinkedList is empty!")

        if self.head.next is None:
            self.head = None
            self.length = 0
            return

        _node = self.head
        while _node.next.next != None:
            _node = _node.next

        _node.next = None
        self.length -= 1

    def change_on_end(self, n: int, new_data) -> None:
        if len(self) < n or n < 1:
            raise KeyError(f"{n} 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

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