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. 3344 _____ Сербиновский Ю.М.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Получить опыт реализации структур данных и алгоритмов работы с ними.

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.

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:

- 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.
  - Если значение переменной head равна None, метод должен создавать пустой список.
  - Если значение head не равно None, необходимо создать список из одного элемента.
- __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.

Пример того, как должен выглядеть результат реализации см. ниже.

- pop(self) - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.
- clear(self) - очищение списка.
- 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.

Выполнение работы

1) Был реализован связной список, то есть список, в котором элементы связаны с помощью указателей друг на друга. В отличие от массива, в связном списке могут храниться элементы разного размера и типа, однако эти элементы хранятся в памяти неупорядоченно.
2) Сложность методов класса Node:
    - get_data() – O(1)
    - change_data – O(1)

Сложность методов класса LinkedList:

- append() – O(n)
- pop() – O(n)
- change_on_start() – O(n)
- clear() – O(1)

3) Бинарный поиск в связном списке:

При выполнении бинарного поиска в связном списке функция ищет средний элемент, сравнивает его с входным значением. Если значение совпадает, функция завершается; в противном случае, она вызывает саму себя для левой или правой части связного списка в зависимости от результата сравнения среднего элемента и входного значения.

В классическом списке для поиска элементов можно использовать индексы, что значительно ускоряет процесс выполнения поиска. В случае связного списка, такая возможность отсутствует, что делает выполнение бинарного поиска более время затратным по сравнению с классическим списком, где доступ к элементам осуществляется напрямую по индексам.

Выводы

На основе двух классов был реализован связной список на языке Python. Был реализован API для связного списка.

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):
        if self.next:
            return f"data: {self.__data}, next: {self.next.__data}"
        return f"data: {self.__data}, next: None"
class LinkedList:
    def __init__(self, _head=None):
        self._head = _head
        self._length = 0
        tmp = self._head
        while tmp:
            self._length += 1
            tmp = tmp.next
    def __len__(self):
        return self._length
    def append(self, element):
        if self._head:
            tmp = self._head
            while tmp.next:
                tmp = tmp.next
            tmp.next = Node(element, None)
        else:
            self._head = Node(element, None)
        self._length += 1
    def __str__(self):
        if self._length == 0:
            return "LinkedList[]"
        else:
            rtr_str = f"LinkedList[length = {self._length}, ["
            tmp = self._head
            while tmp.next:
                rtr_str += tmp.__str__() + "; "
                tmp = tmp.next
            rtr_str += tmp.__str__() + "]]"
            return rtr_str
    def pop(self):
        if self._length == 0:
            raise IndexError("LinkedList is empty!")
        elif self._length == 1:
            self._head = None
            self._length -= 1
        else:
            tmp = self._head
            while tmp.next.next:
                tmp = tmp.next
```

```
        tmp.next = None
        self._length -= 1
def clear(self):
    self._head = None
    self._length = 0
def change_on_start(self, n, new_data):
    if self._length < n or n <= 0:
        raise KeyError("Index out of the range")
    else:
        tmp = self._head
        for i in range(n-1):
            tmp = tmp.next
        tmp.change_data(new_data)
```