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: Введение в алгоритмы и структуры данных

Студентка гр. 3344 Коняева М.В.

Преподаватель Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Целью работы является ознакомление с алгоритмами и структурами данных на языку Python.

Задание

Вариант 1. 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:

1) *data* # Данные элемента списка, приватное поле.
2) *next* # Ссылка на следующий элемент списка.

И следующие методы:

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

1) *head* # Данные первого элемента списка.
2) *length* # Количество элементов в списке.

И следующие методы:

1) *__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.

2) *__len__(self)* - перегрузка метода *__len__*, он должен возвращать длину списка (этот стандартный метод, например, используется в функции *len*).
3) *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.
4) *__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:
    a. Если список пустой, то строковое представление: *“LinkedList[]”*
    b. 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]”,*
5) *pop(self)* - удаление последнего элемента. Метод должен выбрасывать исключение *IndexError* с сообщением *"LinkedList is empty!",* если список пустой.
6) *clear(self)* - очищение списка.
7) *delete_on_end(self, n)* - удаление n-того элемента с конца списка. Метод должен выбрасывать исключение *KeyError*, с сообщением *"Element doesn't exist!"*, если количество элементов меньше n.

Выполнение работы

Связный список – это структура данных, которая состоит из узлов, каждый из которых содержит как собственно данные, так и ссылку (или указатель) на следующий узел в списке. Основные отличия связного списка от массива:

1. Устройство памяти. В массивах элементы хранятся в последовательных блоках памяти. В связном списке каждый узел может располагаться в произвольном месте в памяти, а его связь с другими узлами обеспечивается указателями.
2. Доступ к элементам. В массивах доступ к элементам осуществляется по индексу за константное время O(1). В связных списках доступ к элементам осуществляется путем последовательного прохода от начала до нужного элемента.
3. Размер. Размер массива обычно фиксирован, то есть он определяется при создании и не изменяется без специальных операций. Связный список может быть динамическим - его размер может динамически расти или уменьшаться во время выполнения программы.
4. Вставка и удаление. В массивах вставка и удаление элементов в середину списка могут быть дорогостоящими, так как требуется сдвиг всех элементов после изменяемого. В связных списках вставка и удаление элементов в середину списка обычно эффективны, так как требуется только изменение указателей на узлы.

Сложности методов. O(1): *__init__*, *get_data*, *Node. __str__*, *__len__*, *__clear__*, *append*(если добавляем голову), *pop*(если список пуст), *delete_on_end*(если удаляем голову). O(n*): LinkedList.__str__*, *append*, *pop*, *delete_on_end*.

Реализация бинарного поиска для связного списка может выглядеть следующим образом:

1. Для нахождения середины связного списка понадобится два указателя: один будет двигаться по списку на одну позицию за каждую итерацию, а другой на две позиции. Когда быстрый

указатель (продвигающийся на две позиции) достигнет конца списка, медленный указатель будет указывать на середину.

2. На каждом шаге бинарного поиска находится средний элемент списка. Сравниваем его с искомым значением: если средний элемент равен искомому значению, возвращаем его.
3. В случае, если ключ не совпадает со средним элементом, выбираем, какую половину списка использовать для следующего поиска.
4. Если ключ меньше среднего узла, то для следующего поиска используется левая часть списка.
5. Если ключ больше среднего узла, то для следующего поиска используется правая часть списка.
6. Продолжаем делить список пополам и сужать интервал поиска, пока не найдем искомый элемент или не исчерпаем весь список.

Основное отличие реализации алгоритма бинарного поиска для связного списка заключается в необходимости использования алгоритма для нахождения среднего элемента, так как прямого доступа по индексу нет. Это делает алгоритм менее эффективным.

Разработанный программный код см. в приложении А. Результаты тестирования см. в приложении Б.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Тест	Выходные данные	Комментарии
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(10) l_l.append(20) l_l.append(30) l_l.append(40) print(l_l) print(len(l_l)) l_l.delete_on_end(3) 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 = 3, [data: 10, next: 30; data: 30, next: 40; data: 40, next: None]]	Данные обработаны корректно
2.	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.delete_on_end(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 = 2, [data: 111, next: 222; data: 222, next: None]]	Данные обработаны корректно

Выводы

Были получены базовые навыки работы с алгоритмами и структурами данных. Была написана программа, с помощью которой были изучены сложность алгоритмов и методы работы со связными списками.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lb2.py

```
class Node:
    def __init__(self, data, next=None):
        self.__data = data
        self.next = next

    def get_data(self):
        return self.__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.length = 0
        self.head = head
        while head:
            head = head.next
            self.length += 1


    def __len__(self):
        return self.length

    def append(self, element):
        self.length += 1
        if self.head is not None:
            tmp = self.head
            while tmp.next:
                tmp = tmp.next
            tmp.next = Node(element)
        else:
            self.head = Node(element)


    def __str__(self):
        if self.head is not None:
            res = f'LinkedList[length = {self.length}, ['
            tmp = self.head
            while tmp:
                res += str(tmp) + '; '
                tmp = tmp.next
            res = res[:-2] + ']]'
            return res
        else:
            return 'LinkedList[]'

    def pop(self):
        if self.head is None:
            raise IndexError("LinkedList is empty!")
```

```
    if self.length == 1:
        self.__init__()
    if self.head is not None:
        tmp = self.head
        while tmp.next.next:
            tmp = tmp.next
        tmp.next = None
        self.length -= 1


def delete_on_end(self, n):
    if self.length < n or n <= 0:
        raise KeyError("Element doesn't exist!")
    else:
        index = self.length - n
        tmp = self.head
        if index == 0:
            self.head = tmp.next
        else:
            for i in range(1, index):
                tmp = tmp.next
            tmp.next = tmp.next.next
        self.length -= 1

def clear(self):
    self.__init__()
```