МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе № 2

по дисциплине «Информационные технологии»

Тема: Алгоритмы и структуры данных в

Студент гр. 3344 _____ Волков А.А.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Изучить структуру линейного односвязного списка, реализовать его при помощи ООП в языке программирования Python. Создать методы для работы с ним. Произвести оценку сложности каждого метода. Рассмотреть алгоритм бинарного поиска в контексте линейного списка.

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. 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.

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

- 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.

Основные теоретические положения

1. Односвязный линейный список

Односвязный (однонаправленный связный) список – структура данных, каждый элемент которой содержит 2 поля: собственные данные и ссылку на следующий элемент. Порядок элементов связного списка не совпадает с тем, как расположены элементы в памяти компьютера (элементы могут располагаться не в непрерывном участке памяти), а порядок обхода списка всегда явно задается ссылками между элементами. Таким образом, индексация в связном списке, как в массиве, недоступна. Направление обхода в односвязном списке всегда в одну сторону. В односвязном списке нельзя, находясь на каком-то элементе, обратиться к предыдущему элементу. Для этого понадобится еще одна ссылка – на предыдущий элемент, и тогда список станет уже двусвязным. Для того, чтобы работать со списком надо иметь лишь ссылку на голову, с которой можно начинать обход остальных элементов.

2. Оценка сложности

Оценка сложности будет производиться при помощи нотации «О большое» (Big O). Она показывает, как сильно растет время выполнения программы относительно объема входных данных.

Для примера:

O(1) – «константное время», то есть программа затратит одинаковое вне зависимости от объема входных данных.

O(n) – линейное время, то есть время выполнения программы растет «линейно» в зависимости от объема входных данных.

Выполнение работы

Рассмотрим методы обоих классов и оценим их сложность при помощи Big O:

Class Node:

1. __init__ (инициализирует узел списка)

Имеет константную сложность O(1)

2. get_data (возвращает данные, которые хранятся в узле списка)

Имеет константную сложность O(1)

3. __str__ (возвращает строковое представление узла в заданном формате)

Имеет константную сложность O(1)

Class LinkedList:

1. __init__ (инициализирует линейный односвязный список)

Имеет константную сложность O(1)

2. __len__ (возвращает кол-во элементов списка)

Имеет константную сложность O(1)

3. append (добавляет узел с данным значением в конец списка)

Имеет линейную сложность O(n), то есть в зависимости от размера списка линейно меняется время работы, так как для вставки в конец надо дойти до последнего элемента, пройдя по всем остальным

4. __str__ (возвращает строковое представление линейного списка)

Имеет линейную сложность O(n), то есть в зависимости от размера списка линейно меняется время работы

5. pop (удаляет последний элемент списка, если сам список не пуст)

Имеет линейную сложность O(n), то есть в зависимости от размера списка линейно меняется время работы

6. clear (очищает список)

Имеет константную сложность O(1

7. change_on_end(меняет значение поля data с конца списка)

Имеет линейную сложность O(n), то есть в зависимости от размера списка линейно меняется время работы

Стоит рассмотреть различия между массивом (классическим списком в Python) и линейным списком. Ключевое различие заключается в том, что по своей сути обычный массив реализован при помощи последовательно расположенных в памяти ячеек с указателями на его элементы, именно благодаря этому мы получаем возможность обращаться к элементам по индексу за константное время, в то время как в линейном списке это занимает линейное время. Однако преимущество линейного списка как раз заключается в его «децентрализации», которая выгодно выделяет его в случае, когда память сильно сегментирована и объем информации, способный хранится, важнее, чем обращение по индексу. Кроме того, неотъемлемое преимущество линейного списка над обычным массивом – более простой и удобный механизм вставки и удаления элементов, потому что надо найти нужное место и переставить указатели, массив же требует расширения памяти, сдвига всех остальных элементов, что является очень ресурсоемким процессом.

Рассмотрим возможность реализации алгоритма бинарного поиска в линейном списке (предполагается, что он отсортирован по возрастанию). Тогда определяем средний элемент и проходим до него, сравниваем с искомым, если больше, то ищем средний элемент той части списка, которая находится справа от текущего, иначе в левой части. Однако в отличии от массива (классического списка Python), где мы имеем константную сложность для получения элемента по индексу, в линейном списке эта же операция имеет линейную сложность, что делает бинарный поиск в этом случае бессмысленным.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии

1.	`l_list = LinkedList(1)`
	`print(l_list)`
	`l_list.append(2)`
	`l_list.append(3)`
	`print(l_list)`
	`l_list.pop()`
	`print(l_list)`
		LinkedList[length = 1, [data: 1, next: None]]
		LinkedList[length = 3, [data: 1, next: 2; data: 2, next: 3; data: 3, next: None]]
		LinkedList[length = 2, [data: 1, next: 2; data: 2, next: None]]
			Корректно создается объект линейного списка, добавление и удаление элементов также работает верно

2.	`l_list = LinkedList()`
	`print(l_list)`
	`l_list.append(2)`
	`l_list.append(3)`
	`l_list.append(4)`
	`print(l_list)`
	`print(len(l_list))`
	`l_list.pop()`
	`l_list.pop()`
	`l_list.pop()`
	`print(l_list)`
	`try:`
	`l_list.change_on_end(5 4, 5)`
	`except KeyError as msg:`
	`  print(msg)`
	`try:`
	`  l_list.pop()`
	`except IndexError as msg:`
	`  print(msg)`
	`l_list.append(1)`
	`print(l_list)`
		LinkedList[]
		LinkedList[length = 3, [data: 2, next: 3; data: 3, next: 4; data: 4, next: None]]
		3
		LinkedList[]
		"Element doesn't exist!"
		LinkedList is empty
		LinkedList[length = 1, [data: 1, next: None]]
		LinkedList[]
			Методы корректно работают, в случае критических ситуаций выбрасываются соответствующие ошибки. Очищение списка работает верно.

```
l_list.clear()
```

Выводы

Изучена структура линейного списка, получены знания в оценке сложности алгоритмов.

Был реализован односвязный список при помощи ООП, а также методы для взаимодействия с узлом и самим списком.

Были выделены основные отличия между массивами и линейными списками.

Был рассмотрен и оценен алгоритм бинарного поиска в случае связного списка.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lb_2.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.get_data()}"
        return f"data: {self.__data}, next: None"


class LinkedList:
    def __init__(self, head=None):
        if head:
            self.head = Node(head)
            self.length = 1
        else:
            self.head = None
            self.length = 0

    def __len__(self):
        return self.length

    def append(self, element):
        new_node = Node(element)
        if self.head:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
        else:
            self.head = new_node
        self.length += 1

    def __str__(self):
        print_list = []
        current = self.head
        while current:
            print_list.append(str(current))
            current = current.next
        if print_list:
                        return  f'LinkedList[length  =  {self.length},  [{";
".join(print_list)}]]'
        return 'LinkedList[]'
```

```
def pop(self):
    if not self.head:
        raise IndexError('LinkedList is empty')
    if self.head.next is None:
        self.head = None
        self.length = 0
    else:
        current = self.head
        while current.next.next:
            current = current.next
        current.next = None
        self.length -= 1

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

def change_on_end(self, n, new_data):
    if n > self.length or n < 0:
        raise KeyError("Element doesn't exist!")
    current = self.head
    for _ in range(self.length - n):
        current = current.next
    if current is None:
        raise KeyError("Element doesn't exist!")
    current.change_data(new_data)
```