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 Python

Студент гр. 3341 _____ Моисеева А.Е.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Изучить основные алгоритмы и структуры данных в языке Python, освоить основные методы для работы со списками в Python, затем создать программу и реализовать в ней односвязный список для хранения численных данных.

Задание

Вариант 4

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.

Пример того, как должно выглядеть взаимодействие с Вашим связным списком:

```
linked_list = LinkedList()
print(linked_list) # LinkedList[]
print(len(linked_list)) # 0
linked_list.append(10)
print(linked_list) # LinkedList[length = 1, [data: 10, next: None]]
print(len(linked_list)) # 1
linked_list.append(20)
print(linked_list) # LinkedList[length = 2, [data: 10, next:20; data: 20, next: None]]
print(len(linked_list)) # 2
linked_list.pop()
print(linked_list)
print(linked_list) # LinkedList[length = 1, [data: 10, next: None]]
```

print(len(linked_list)) # 1

Вам не требуется реализовывать создание экземпляров ваших классов и вызов методов, это сделает проверяющая система.

В отчете вам требуется:

Указать, что такое связный список. Основные отличия связного списка от массива.

Указать сложность каждого метода.

Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

Основные теоретические положения

Связный список — это базовая структура данных, в которой элементы хранятся в узлах. Каждый узел содержит два компонента: данные, хранимые в данном элементе, ссылка на следующий узел - указатель, который направляет к следующему элементу в последовательности.

Списки могут быть односвязными – каждый элемент ссылается на следующий за ним, двусвязными – элемент хранит ссылку на следующий и предыдущий элементы, кольцевыми, где каждый узел соответственно содержит ссылку только на следующий элемент, последний же узел снова указывает на начало списка.

Отличия связного списка от массива следующие: массив размещён в памяти последовательно, в списке же узлы размещены в памяти отдельно друг от друга, в массиве возможен доступ по индексам, в то время как в списке для доступа к элементам необходимо последовательно перемещаться по последовательности.

Сложность методов:

Класс Node: инициализация элементов класса *__init__()*, геттер *get_data()*, сеттер *change_data()* и метод для формирования строкового представления узла *__str__()* имеют сложность O(1) и выполняются за константное время, поскольку не требуют перемещения по спику.

Класс LinkedList: инициализация элементов класса *__init__()*, метод для получения длины *len()*, метод для очистки списка *clear()* имеют сложность O(1) и выполняются за константное время, не требуют прохода по списку (в методе *len()* длина списка *length* хранится как свойство объекта). Метод добавления элемента в конец списка *append()*, удаления последнего элемента в списке *pop()*, метод для строкового представления *__str__()*, метод изменения значения элемента под определённым номером *change_on_start()* имеют сложность O(n) поскольку требуют прохождения по списку.

Реализация бинарного поиска в связном списке:

Для реализации алгоритма бинарного поиске в отсортированном связном списке будем использовать указатели на начало и конец списка, а также на середину. В дальнейшем при равенстве искомого значения и значения в среднем узле списка, возвращаем данное значение, если же оно больше искомого, перемещаем конечный указатель на узел перед указателем на середину, если меньше искомого – перемещаем начальный указатель на узел, следующий за средним.

Отличие бинарного поиска для связного списка и для классического списка Python заключается в следующем: в связных списках доступ к элементу требует последовательного прохода по списку, в массивах же доступ осуществляется с помощью индексов. Бинарный поиск в связном списке не настолько рационален, как в массиве, поскольку нахождение начального, конечного и серединного элементов и дальнейшее их перемещение быстрее с использованием индексов и их арифметики.

Выполнение работы

Создается класс *Node*, который включает приватное поле *data* (хранит данные элемента *Node*) и публичное *next* (ссылка на следующий элемент). Метод *__init__()* инициализирует экземпляр класса *Node*, где *next* по умолчанию равен *None*. Метод *get_data()* возвращает значение приватного поля *data*, а *change_data()* заменяет данные объекта класса *Node* на новые. Метод *__str__()* создает строковое представление объекта класса *Node* с использованием форматной строки.

Далее реализуется класс *LinkedList*, содержащий два поля: *head* (данные первого элемента списка) и *length* (количество элементов в списке). Метод *__init__()* инициализирует объект класса *LinkedList* в зависимости от содержимого *head*. Метод *len()* возвращает значение поля *length*. Метод *append()* добавляет объект класса *Node*, инициализированный значением *element*, в конец списка. Метод *str()* возвращает представление списка в виде форматной строки. Метод *pop()* удаляет последний элемент списка, выбрасывая исключение *IndexError*, если список пуст. Метод *clean()* сбрасывает значения *head* на *Node* и *length* на 0. Метод *change_on_start()* удаляет *n*-ый элемент с начала списка, выбрасывая исключение *KeyError*, если элемента с индексом n нет в списке.

Разработанный программный код см. в приложении А.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	node = Node(1) print(node) node.next = Node(2, None) print(node) print(node.get_data()) l_l = LinkedList() l_l.append(10) l_l.append(20) l_l.append(30) print(l_l) print(len(l_l)) l_l.change_on_start(2, 2) print(l_l)	data: 1, next: None data: 1, next: 2 1 LinkedList[length = 3, [data: 10, next: 20; data: 20, next: 30; data: 30, next: None]] 3 LinkedList[length = 3, [data: 10, next: 2; data: 2, next: 30; data: 30, next: None]]	Создаются элементы класса Node, затем выводятся для проверки работоспособности класса, затем создаётся список, в него добавляются три элемента, затем второй с начала меняется на два, выводится полученный связный список.

Выводы

В ходе выполнения работы были изучены основные алгоритмы и структуры данных в языке Python, а также освоены основные методы работы со списками. С применением полученных знаний была разработана программа, в рамках которой был реализован односвязный список для хранения численных данных.

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 is not None:
                        return  f"data:  {self.get_data()},  next:
{self.next.get_data()}"
            else:
                return f"data: {self.get_data()}, next: None"

class LinkedList:
    def __init__(self, head = None):
        self.head = head
        if head is not None:
            self.length = 1
        else:
            self.length = 0

    def __len__(self):
        return self.length

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

    def __str__(self):
        if self.head is None:
            result = f"LinkedList[]"
        else:
            result = f"LinkedList[length = {self.length}, ["
            current = self.head
            while current is not None:
                if current.next is None:
                      result += f"data: {current.get_data()}, next:
None]]"
```

```
                    else:
                        result += f"data: {current.get_data()}, next:
{current.next.get_data()}; "
                    current = current.next
            return result

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

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

        def change_on_start(self, n, new_data):
            if n > self.length or n <= 0:
                raise KeyError("Element doesn't exist!")
            current = self.head
            for i in range(n-1):
                current = current.next
            current.change_data(new_data)
```