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(ка) гр. 3383 Логинова А. Ю.

Преподаватель Иванов Д. В.

Санкт-Петербург

2024

Цель работы

Цель данной работы заключается в создании структуры данных односвязного списка и реализации основных операций с этой структурой, таких как добавление, удаление и изменение элементов.

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_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.

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:

1. Указать, что такое связный список. Основные отличия связного списка от массива.
2. Указать сложность каждого метода.
3. Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

Выполнение работы

Класс Node представляет собой узел в односвязном списке. Он содержит следующие элементы:

- __init__(self, data, next=None): Конструктор класса, который инициализирует узел с данными (data) и ссылкой на следующий узел (next). Если next не указан, он по умолчанию будет инициализирован как None.
- get_data(self): Метод для получения данных из узла.
- change_data(self, new_data): Метод для изменения данных в узле на новые (new_data).
- __str__(self): Метод для представления узла в виде строки. Он возвращает строку, содержащую данные узла и данные следующего узла, если он существует, или None, если следующего узла нет.

Класс LinkedList представляет собой односвязный список, состоящий из узлов. Он содержит следующие элементы:

- __init__(self, head=None): Конструктор класса, который инициализирует список с начальным элементом (head). Если head не указан, список будет пустым.
- __len__(self): Метод для получения длины списка, вызывает метод len().
- append(self, element): Метод для добавления элемента в конец списка. Создает новый узел с данным элементом и добавляет его в список.
- __str__(self): Метод для представления списка в виде строки. Он возвращает строку, содержащую длину списка и данные всех узлов, проходясь итеративно по односвязному списку.
- pop(self): Метод для удаления последнего элемента из списка. Если список пуст, вызывает исключение IndexError.
- clear(self): Метод для очистки списка, вызывает метод clear().

- change_on_end(self, n, new_data): Метод для изменения данных на n-м элементе с конца списка на new_data. Если элемент не существует, вызывает исключение KeyError.

Задание 1

Связный список - это структура данных, состоящая из узлов, каждый из которых содержит какое-либо значение и ссылку на следующий узел в списке. Список отличается от массива тем, что массив представляет собой непрерывный блок памяти, в котором элементы хранятся последовательно.

Основные отличия связного списка от массива:

1. Память: В связном списке каждый элемент может быть распределен в памяти в произвольном порядке, в то время как элементы массива хранятся в непрерывной области памяти.

2. Динамичность: Размер связного списка может изменяться динамически, в отличие от массива, размер которого обычно фиксирован при создании.

3. Вставка и удаление: В связном списке вставка и удаление элементов происходят быстрее, так как не требуется сдвигать остальные элементы, в отличие от массива, где вставка и удаление могут потребовать перемещения всех последующих элементов.

4. Доступ к элементам: В связном списке доступ к элементам осуществляется последовательно, начиная с головы списка, в то время как в массиве доступ к элементам может быть произведен напрямую по индексу.

Таким образом, связный список и массив имеют различные особенности, и выбор между ними зависит от конкретных требований и характеристик приложения.

Задание 2

1. Метод `__init__` класса `Node` имеет сложность O(1), так как он выполняет только инициализацию узла, что не зависит от размера списка.

2. Метод `get_data` класса `Node` также имеет сложность O(1), так как он просто возвращает значение узла, не зависящее от размера списка.

3. Метод `change_data` класса `Node` также имеет сложность O(1), так как он просто изменяет значение узла, не зависящее от размера списка.

4. Метод `__str__` класса `Node` имеет сложность O(1), так как он просто возвращает строковое представление узла, не зависящее от размера списка.

5. Метод `__init__` класса `LinkedList` имеет сложность O(1), так как он выполняет только инициализацию списка, что не зависит от размера списка.

6. Метод `__len__` класса `LinkedList` имеет сложность O(1), так как он просто возвращает длину списка, не зависящую от размера списка.

7. Метод `append` класса `LinkedList` имеет сложность O(1), так как он добавляет элемент в конец списка, не зависящий от размера списка.

8. Метод `__str__` класса `LinkedList` имеет сложность O(n), где n - длина списка, так как он формирует строковое представление списка, проходя по всем его элементам.

9. Метод `pop` класса `LinkedList` имеет сложность O(1), так как он удаляет последний элемент списка, не зависящий от размера списка.

10. Метод `clear` класса `LinkedList` имеет сложность O(1), так как он очищает список, не зависящий от размера списка.

11. Метод `change_on_end` класса `LinkedList` имеет сложность O(n), где n - длина списка, так как он изменяет значение элемента в конце списка, что требует прохода по списку до нужного элемента.

Задание 3

Бинарный поиск в связном списке может быть реализован следующим образом:

1. Необходимо определить длину связного списка, чтобы затем определить границы для бинарного поиска.

2. Затем можно использовать два указателя, начальный и конечный, чтобы определить середину списка.

3. Сравниваем значение в середине списка с искомым значением. Если значение совпадает, возвращаем индекс. Если значение меньше, сужаем диапазон поиска до левой половины списка. Если значение больше, сужаем диапазон поиска до правой половины списка.

4. Повторяем шаги 2-3 до тех пор, пока не найдем искомый элемент или не исчерпаем весь диапазон поиска.

Отличие реализации алгоритма бинарного поиска для связного списка от классического списка Python заключается в том, что для связного списка нет прямого доступа к элементам по индексу, как в случае с массивом или списком Python. Поэтому для бинарного поиска в связном списке необходимо сначала определить длину списка и затем использовать указатели для перемещения по элементам списка и сужения диапазона поиска.

Тестирование

l_l = LinkedList() l_l.append(30) l_l.append(40) print(len(l_l)) print(l_l) l_l.pop() print(len(l_l)) print(l_l)	2 LinkedList[length = 2, [data: 30, next: 40; data: 40, next: None]] 1 LinkedList[length = 1, [data: 30, next: None]]
n = Node(250) b = Node(500) n.next = b print(n) l_l = LinkedList() l_l.append(100) l_l.append(40) l_l.append(20) print(l_l) l_l.change_on_end(2, 30) print(l_l)	data: 250, next: 500 LinkedList[length = 3, [data: 100, next: 30; data: 30, next: 20; data: 20, next: None]]
l_l = LinkedList() l_l.append(100) l_l.change_on_end(1, 30) print(l_l) l_l.clear() print(l_l)	LinkedList[length = 1, [data: 30, next: None]] LinkedList[]

Выводы

Была написана программа на языке Python, реализующая структуру данных "связный список" на основе массива с использованием классов Node и LinkedList. В программе были реализованы методы для добавления элементов в конец списка, удаления последнего элемента, изменения значения элемента по индексу, получения длины списка и формирования строкового представления 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_daya(self, new_data):
        self.data = new_data

    def __str__(self):
          return "data: {}, next: {}".format(self.data, self.next.data
if self.next else self.next)


class LinkedList:
    def __init__(self, head=None):
        if head is None:
            self.head = []
        else:
            self.head = [head]

    def __len__(self):
        return len(self.head)

    def append(self, element):
        new_element = Node(element)
        if len(self.head) != 0:
            self.head[-1].next = new_element
        self.head.append(new_element)

    def __str__(self):
        return "LinkedList[]" if len(self.head) == 0 \
            else ("LinkedList[length = {}, [".format(len(self.head)) +
                      "; ".join(["data: {}, next: {}".format(node.data,
node.next.data if node.next else node.next)
                                 for node in self.head]) + "]]")

    def pop(self):
        if len(self.head) == 0:
            raise IndexError("LinkedList is empty!")
        else:
            self.head.pop(-1)
        if len(self.head) > 0:
            self.head[-1].next = None

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

```
def change_on_end(self, n, new_data):
    if n <= 0 or (n > 0 and n > len(self.head)):
        raise KeyError("Element doesn't exist!")
    else:
        self.head[-n].data = new_data
```