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 _____ Пчелкин Н.И.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Ознакомиться с базовыми алгоритмами и структурами данных, использующимися в разработке в целом и в программировании на языка Python в частности. Разработать программу, реализующую однонаправленный список.

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?

Основные теоретические положения

Связный список – структура данных, состоящая из элементов (узлов), каждый из которых помимо данных содержит в себе ссылки на следующий (предыдущий) элемент списка. Однонаправленный связный список состоит из элементов, содержащих ссылки на последующие элементы списка.

Основные отличия связного списка от массива заключаются в том, что массив хранится в памяти непрерывным блоком, в то время как узлы связного списка могут храниться отдельно друг от друга, что при больших объёмах данных позволяет оптимизировать хранение информации в памяти. Также в массиве доступ к элементам осуществляется по их индексам, в то время как в связном списке доступа по индексу элемента может и не быть.

Добавление элемента в конец списка (*append()*), создание строкового представления списка (*__str__()*), удаление последнего элемента (*pop()*) и замена данных в n-ном элементе с конца списка (*change_on_end()*) имеют сложность O(n), т.к. требуют итерации по всему списку.

Конструктор списка (*__init__()*), получение длины списка (*__len__()*) и очищение списка (*clean()*) имеют сложность O(1), т.к. не зависят от количества элементов в списке.

Если считать, что связный однонаправленный список отсортирован, то удобнее всего для бинарного поиска было бы хранить ссылки на первый, последний и центральный узлы рассматриваемого участка списка и менять их при поиске (что, конечно, потребует итерации по элементам списка, хранение длины участка между крайними элементами). Классический список в Python имеет индексацию и доступ по индексам, что означает, что бинарный поиск может обращаться к нужным для его работы крайним и центральным элементам по индексу, а находить эти индексы простой арифметикой.

Выполнение работы

Реализуется класс *Node*, содержащий поля *__data* (приватное поле, данные, хранящиеся в элементе *Node*) и *next* (ссылка на следующий элемент). Метод *__init__(self, data, next=None)* инициализирует объект класса *Node* (поле next по умолчанию *None*), *get_data(self)* возвращает значение приватного поля *__data*, *change_data(self, new_data)* заменяет данные объекта класса *Node* на новые, а *__str__(self)* с помощью форматной строки создаёт строковое представление объекта класса *Node*.

Далее реализуется класс *LinkedList*, имеющий два поля: *head* – данные первого элемента списка, и *length* – количество элементов в списке.

Метод *__init__(self, head)* – конструктор класса, который в зависимости от содержимого *head* инициализирует объект класса *LinkedList*.

Метод *__len__(self)* возвращает значение поля *length*.

Метод *append(self, element)* итерируется по всему списку до его конца (если длина списка равна 0, создаётся голова списка), добавляет в конец списка объект класса *Node*, инициализированный из значения *element*.

Метод *__str__(self)* итерируется по всему списку и возвращает представление списка форматной строкой.

Метод *pop(self)* итерируется по всему списку до его предпоследнего элемента (если список пустой, выбрасывается исключение *IndexError*), затем удаляет ссылку на последний элемент списка.

Метод *clean(self)* присваивает *head* значение *Node*, а *length* – значение 0.

Метод *change_on_end(self, n, new_data)* в случае правильного индекса *n* удаляет *length-n*-ый элемент с начала (т.е. *n*-ый с конца), итерируясь до искомого элемента. Если элемента с индексом n нет в списке, выбрасывается исключение *KeyError*.

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

Выводы

В ходе выполнения лабораторной работы были изучены основные алгоритмы и структуры данных, применяющиеся в разработке на языке Python. Была написана программа, реализующая однонаправленный связный список с помощью двух классов Node и LinkedList.

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):
          return f"data: {self.__data}, next: {self.next.__data if
self.next != None else None}"


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

    def __len__(self):
        return self.length

    def append(self, element):
        cur_element = self.head
        if cur_element == None:
            self.head = Node(element)
            self.length = 1
        else:
            while(cur_element.next != None):
                cur_element = cur_element.next
            cur_element.next = Node(element)
            self.length += 1

    def __str__(self):
        if self.length == 0:
            return "LinkedList[]"
        else:
            result = f"LinkedList[length = {self.length}, ["
            cur = self.head
            while(cur != None):
                result += str(cur)
                cur = cur.next
                if(cur != None):
                    result += "; "
            result += "]]"
        return result

    def pop(self):
```

```
    if self.length == 0:
        raise IndexError ("LinkedList is empty!")
    elif self.length == 1:
        self.clear()
    else:
        cur = self.head
        while (cur.next.next != None):
            cur = cur.next
        cur.next = None
        self.length -= 1

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

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