

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

Студентка гр. 3341

Яковлева А.А.

Преподаватель

Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Целью данной работы является:

- изучение алгоритмов и структур данных
- реализовать связный однонаправленный список на языке Python

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`.

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` равно `None`, метод должен создавать пустой список.
- Если значение `head` не равно `None`, необходимо создать список из одного элемента.

ВПВПАВПАВПА

- `__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`.

Основные теоретические положения

Связный список - это динамическая структура данных, состоящая из узлов, содержащих данные и ссылки на следующий и/или предыдущий узел списка. Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

Преимущества связных списков:

- эффективное (за константное время) добавление и удаление элементов
- размер ограничен только объёмом памяти компьютера и разрядностью указателей
- динамическое добавление и удаление элементов

Выполнение работы

Создаём класс *Node*, содержащий приватное поле *name* и поле *next*. Создаём метод *get_data()*, который возвращает значение поля *data*, и метод *change_data()*, который меняет текущее значение поля *data*. Переопределяем метод *str*, который теперь возвращает строку в виде “*data: <node_data>, next: <node_next>*”.

Создаём класс *LinkedList*, содержащий поля *head*, *length*. В конструкторе значение по умолчанию для *head* равно *None*. При инициализации объекта, если значение *head* равно *None* создаём пустой список, иначе создаём список из одного элемента. Переопределяем метод *len*, который теперь возвращает длину списка, и метод *str*, который теперь возвращает строку в определённом виде. Создаём метод *append()*, который проходит по списку до последнего элемента и добавляет элемент в конце списка; метод *pop()*, который, если список пустой выбрасывает исключение *IndexError* с сообщением "LinkedList is empty!", иначе проходит по списку до предпоследнего элемента и удаляет последний элемент; метод *change_on_start()*, который меняет поле *data* *n*-ого элемента на *new_data*, если номер *n* больше количества элементов выбрасывает исключение *KeyError* с сообщением "Element doesn't exist! ".

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.	<pre>node = Node(7) print(node) node.next = Node(5, None) print(node) print(node.get_data()) node.change_data(10) print(node.get_data()) print(node)</pre>	<pre>data: 7, next: None data: 7, next: 5 7 10 data: 10, next: 5</pre>	Проверка методов класса Node
2.	<pre>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.change_on_start(2, 7) print(linked_list) linked_list.pop() print(linked_list) linked_list.clear() print(linked_list)</pre>	<pre>LinkedList[] 0 LinkedList[length = 1, [data: 10, next: None]] 1 LinkedList[length = 2, [data: 10, next: 20; data: 20, next: None]] 2 LinkedList[length = 2, [data: 10, next: 7; data: 7, next: None]] LinkedList[length = 1, [data: 10, next: None]] LinkedList[]</pre>	Проверка методов класса LinkedList
3.	<pre>linked_list = LinkedList() linked_list.append(10) linked_list.change_on_start(2, 10)</pre>	<pre>KeyError: "Element doesn't exist!"</pre>	Проверка исключения метода change_on_start

Выводы

В ходе выполнения лабораторной работы были изучены основные алгоритмы и структуры данных. Была написана программа на языке Python, реализующая связный однонаправленный список с помощью двух зависимых 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.get_data()}, next: {None if self.next == None else self.next.get_data()}"

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

    def __len__(self):
        return self.length

    def append(self, element):
        new_node = Node(element)

        if self.length == 0:
            self.head = new_node
        else:
            tmp = self.head
            while tmp.next != None:
                tmp = tmp.next
            tmp.next = new_node
        self.length += 1

    def __str__(self):
        if self.length == 0:
            result = "LinkedList[]"
        else:
            result = f"LinkedList[length = {self.length}, ["

            tmp = self.head
            while tmp != None:
                result += str(tmp)
                if tmp.next != None:
                    result += "; "
                tmp = tmp.next
            result += "]"

        return result
```

```

def pop(self):
    if self.length == 0:
        raise IndexError("LinkedList is empty!")
    else:
        if self.length == 1:
            head = None
        else:
            tmp = self.head
            while tmp.next.next != None:
                tmp = tmp.next
            tmp.next = None
            self.length -= 1

def change_on_start(self, n, new_data):
    if n < 1 or self.length < n:
        raise KeyError("Element doesn't exist!")
    else:
        tmp = self.head
        while n > 1:
            tmp = tmp.next
            n -= 1
        tmp.change_data(new_data)

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

```