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

Студент гр. 3341 _____ Кудин А.А.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Изучение принципов работы и реализация базовых операций однонаправленного связного списка на языке программирования Python в рамках курса "Алгоритмы и структуры данных".

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.

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

o clear(self) - очищение списка.

o 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

Описание классов

Класс Node:

- Поля:
- __data: содержит данные узла списка.
- next: содержит ссылку на следующий узел списка.
- Методы:
- __init__(data, next=None): конструктор класса, инициализирующий узел с данными и ссылкой на следующий узел.
- get_data(): возвращает данные узла.
- change_data(new_data): изменяет данные узла на новые.
- __str__(): возвращает строковое представление узла в виде "data: <data>, next: <next_data>".

Класс LinkedList:

- Поля:
- head: ссылка на первый узел списка.
- length: количество элементов в списке.
- Методы:
- __init__(head=None): конструктор класса, создающий пустой список или список с одним элементом.
- __len__(): возвращает длину списка.
- append(element): добавляет новый узел в конец списка.
- __str__(): возвращает строковое представление всего списка.
- pop(): удаляет последний узел из списка.
- clear(): полностью очищает список.
- change_on_end(n, new_data): изменяет данные n-го узла с конца списка.

Алгоритм работы методов

Методы класса LinkedList реализованы следующим образом:

1. append(element): Создаёт новый узел с данными element и добавляет его в конец списка. Если список пуст, новый узел становится головным элементом. В противном случае происходит обход списка до последнего элемента и установка ссылки next последнего узла на новый узел.
2. pop(): Удаляет последний элемент списка. Если список пуст, выбрасывается исключение IndexError. Если в списке один элемент, head устанавливается в None. Если элементов больше, происходит обход до предпоследнего узла и его next устанавливается в None.
3. clear(): Устанавливает head в None и length в 0, тем самым полностью очищая список.
4. change_on_end(n, new_data): Изменяет данные узла, находящегося на n-ой позиции с конца списка. Производит обход списка до нужного узла и изменяет его данные.

Разработанный программный код см. в приложении А.

Тестирование

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии

1.	node = Node(1)
	print(node) # data: 1, next: None
	node.next = Node(2, None)
	print(node) # data: 1, next: 2
	print(node.get_data())
	l_l = LinkedList()
	print(l_l)
	print(len(l_l))
	l_l.append(111)
	l_l.append(222)
	l_l.append(333)
	print(l_l)
	print(len(l_l))
	l_l.pop()
	print(l_l)
	l_l.append(333)
	l_l.change_on_end(3, 1)
	print(l_l)
		data: 1, next: None
		data: 1, next: 2
		1
		LinkedList[]
		0
		LinkedList[length = 3, [data: 111, next: 222; data: 222, next: 333; data: 333, next: None]]
		3
		LinkedList[length = 2, [data: 111, next: 222; data: 222, next: None]]
		LinkedList[length = 3, [data: 111, next: 222; data: 222, next: 333; data: 333, next: None]]
			OK

Выводы

В ходе выполнения лабораторной работы были успешно реализованы и протестированы классы **Node** и **LinkedList**, представляющие структуру данных "однонаправленный связный список". Работа с этой структурой данных позволила глубже понять принципы организации и манипуляции узлами в памяти.

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


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

    def __len__(self):
        return self.length

    def append(self, element):
        new_node = Node(element)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
        self.length += 1
```

```
        def __str__(self):
            if not self.head:
                return "LinkedList[]"
            else:
                current = self.head
                nodes = []
                while current:
                    node_str = f"data: {current.get_data()}, next:
{current.next.get_data() if current.next else 'None'}"
                    nodes.append(node_str)
                    current = current.next
                return f"LinkedList[length = {self.length}, [{';
'.join(nodes)}]]"

        def pop(self):
            if not self.head:
                raise IndexError("LinkedList is empty!")
            if self.length == 1:
                self.head = None
            else:
                current = self.head
                while current.next and 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 <= 0 or n > self.length:
                raise KeyError("Element doesn't exist!")

            steps_to_move = self.length - n
            current = self.head
            for _ in range(steps_to_move):
                current = current.next
            current.change_data(new_data)
```