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 Мильхерт А.С.

Преподаватель Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Изучить алгоритмы и структуры данных. На практике реализовать связный однонаправленный список на языке Python путем реализации двух зависимых классов Node и LinkedList.

Задание

Вариант 3

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 data 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 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>.next; data:<second_node>.data, next:<second_node>.next; …; data:<last_node>.data, next: <last_node>.next]”, где <len> - длина связного списка, <first_node>, <second_node>, …, <last_node> - элементы однонаправленного списка.
- 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.

Основные теоретические положения

1) Связный список - это структура данных, состоящая из узлов, каждый из которых содержит какое-то значение и ссылку на следующий узел в списке. Основное отличие связного списка от массива заключается в том, что элементы связного списка хранятся в произвольных областях памяти и связаны между собой ссылками, в то время как элементы массива располагаются в последовательных ячейках памяти.
2) Добавление элемента в конец, замена элемента с конца, создание строкового представления и удаление последнего элемента имеют сложность O(n), так как нет индексации элементов и, соответственно, необходимо пройти по цепочке узлов до нужного. Конструктор класса, метод очистки и метод нахождения длины списка имеют сложность O(1), так как не требуют итерации по списку.
3) Бинарный поиск в связном списке отличается от классического бинарного поиска для массивов из-за того, что доступ к элементам происходит последовательно, а не по индексу. Отличие от классического бинарного поиска заключается в том, что мы не можем сразу перейти к середине списка из-за отсутствия прямого доступа по индексу. Вместо этого мы используем два указателя - медленный и быстрый, чтобы найти середину списка.

Выполнение работы

Реализуется класс Node, с полями data и next и методами __init__, get_data, change_data и __str__ согласно условию задания.

Реализуется класс LinkedList, с полями head, по умолчанию равным None, и length.

Метод __init__ создает пустой список, если параметр head равен None, иначе создается список из одного элемента.

Метод __len__ возвращает значение поля length.

Метод append создает объект класса Node. Если список пустой, то head присваивается ссылка на этот объект. Иначе метод итерируется по списку до его последнего элемента и присваивает его полю next ссылку на этот объект.

Метод pop возвращает ошибку IndexError, если список пустой. Иначе метод итерируется по списку до его предпоследнего элемента и присваивает его полю next значение None.

Метод clear присваивает head значение None, а length значение 0.

Метод __str__ возвращает строковое представление списка, согласно условию задания.

Метод change_on_end проверяет индекс n на правильность. Если он больше длины списка или меньше 1, возвращается KeyError. Иначе меняется значение n-ого элемента с конца.

Для реализации бинарного поиска в связном списке можно использовать следующий алгоритм:

1. Организовать два указателя start и end на границах списка. start будет присвоено значение head, а для поиска end будет необходимо пройти по списку до его последнего элемента.

2. Вычислить середину списка при помощи итерации нужное количество раз, определяемое длиной списка.

3. На каждом шаге сравнивать значение середины с искомым значением.

4. Если значение середины равно искомому значению, вернуть индекс.

5. Если значение середины меньше искомого значения, двигать указатель start на один шаг вперед от середины.

6. Если значение середины больше искомого значения, двигать указатель end на один шаг назад от середины.

7. Повторять шаги 2-6 до тех пор, пока значения не совпадут или пока start не станет больше end.

Отличие от классического списка Python в том, что в случае связного списка доступ к элементам происходит последовательно через ссылки, что делает сложным проведение бинарного поиска, так как нет прямого доступа к элементам по индексу. Таким образом, для реализации бинарного поиска в связном списке потребуется дополнительные операции перемещения по списку, что делает его менее эффективным по сравнению с классическим списком в Python.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Выводы

Были изучены различные алгоритмы и структуры данных. На практике реализован однонаправленный связный список на языке Python при помощи 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):
        if self.next is None:
            next_data = "None"
        else:
            next_data = str(self.next.get_data())
        return f"data: {self.__data}, next: {next_data}"


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

    def __len__(self):
        return self.length

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

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

```
                result += '; '
            current_node = current_node.next
        result += ']]'
        return result

def pop(self):
    if self.length == 0:
        raise IndexError("LinkedList is empty!")
    elif self.length == 1:
        self.head = None
        self.length = 0
    else:
        current_node = self.head
        while current_node.next.next is not None:
            current_node = current_node.next
        current_node.next = None
        self.length -= 1

def change_on_end(self, n, new_data):
    if self.length < n or n < 1:
        raise KeyError("Element doesn't exist!")
    current_node = self.head
    n = self.length - n
    while n > 0:
        current_node = current_node.next
        n -= 1
    current_node.change_data(new_data)

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

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица Б.1 - Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	node = Node(1) print(node.get_data()) print(node) node.next = Node(2, None) print(node) node.change_data(3) print(node)	1 data: 1, next: None data: 1, next: 2 data: 3, next: 2	Тестирование класса Node
2.	linked_list = LinkedList() print(linked_list, len(linked_list)) linked_list.append(10) linked_list.append(20) print(linked_list) linked_list.change_on_end(2, 30) print(linked_list) linked_list.pop() print(linked_list) linked_list.clear() print(linked_list)	LinkedList[] 0 LinkedList[length = 2, [data: 10, next: 20; data: 20, next: None]] LinkedList[length = 2, [data: 30, next: 20; data: 20, next: None]] LinkedList[length = 1, [data: 30, next: None]] LinkedList[]	Тестирование класса LinkedList

3.	try:
	  linked_list = LinkedList()
	  linked_list.append(10)
	  linked_list.pop()
	  linked_list.pop()
	except IndexError as err:
	  print(err.args[0])
		LinkedList is empty!	Проверка исключения метода pop

4.	try: linked_list = LinkedList()	Element doesn't exist! Element doesn't exist!	Проверка исключения метода change_on_end

	linked_list.append(10) linked_list.append(20) linked_list.change_on_end(0, 30) except KeyError as err: print(err.args[0]) try: linked_list = LinkedList() linked_list.append(10) linked_list.append(20) linked_list.change_on_end(- 1, 30) except KeyError as err: print(err.args[0]) try: linked_list = LinkedList() linked_list.append(10) linked_list.append(20) linked_list.change_on_end(3, 30) except KeyError as err: print(err.args[0])	Element doesn't exist!	