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

Студентка гр. 3343 _____ Синицкая Д.В.

Преподаватель _____ Иванов Д. И.

Санкт-Петербург

2024

Цель работы

Изучение создания зависимых классов для реализации односвязного списка на языке программирования Python, приобретение умения описывать элементы списка и методы изменения элементов в односвязном списке.

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).

__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 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.

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

clear(self) - очищение списка.

delete_on_end(self, n) - удаление n-того элемента с конца списка. 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

1. Связный список - это структура данных, состоящая из узлов, каждый из которых содержит как сами данные, так и ссылку на следующий узел в списке.

Отличие связного списка от массива заключается в следующем:

Хранение данных: В массиве элементы хранятся непосредственно в ячейках памяти по порядковым номерам, в то время как в связном списке каждый узел может находиться в произвольном месте памяти, а указатели связывают их между собой.

Сложность доступа к элементам: В массиве доступ к элементу по индексу осуществляется за константное время O(1), тогда как в связном списке для доступа к произвольному элементу нужно перебирать узлы по порядку, что требует O(n) времени, где n - это длина списка.

Сложность вставки/удаления элементов: Вставка или удаление элемента в массиве в произвольное место может потребовать сдвига всех элементов справа от данного индекса, что делает эти операции неэффективными с асимптотикой O(n). В связном списке вставка или удаление элемента из середины списка может быть выполнена быстро с асимптотикой O(1) при наличии указателей на соседние узлы.

Занимаемая память: Связный список требует дополнительной памяти для хранения указателей на следующие узлы, в то время как массив требует память для хранения всех элементов подряд.

2. Методы класса Node:

__init__ : O(1) - константная сложность, так как независимо от размера списка создается только один узел.

get_data : O(1) - также константная сложность, поскольку просто возвращает значение атрибута data.

__str__ : O(1) - константная сложность, так как обращение к атрибутам data и next выполняется за постоянное время.

Методы класса LinkedList:

__init__ : O(1) - константная сложность, так как инициализируются атрибуты объекта.

__len__ : O(1) - константная сложность, поскольку длина списка уже хранится как атрибут length

append : O(n) - линейная сложность, так как при добавлении элемента нужно пройти по всему списку до конца.

__str__ : O(n) - линейная сложность, так как нужно пройти по всем узлам списка для формирования строки.

pop : O(n) - линейная сложность, так как при удалении последнего элемента нужно пройти по всему списку до предпоследнего элемента.

clear : O(1) - константная сложность, так как элементы списка обнуляются.

delete_on_end : O(n) - линейная сложность, так как позиция удаляемого элемента может потребовать проход по всем узлам до этого элемента.

Таким образом, у методов append, __str__, pop, и delete_on_end сложность зависит от длины списка и может быть линейной O(n), а у остальных методов - константной O(1).

3. Бинарный поиск – это эффективный алгоритм поиска элемента в упорядоченном списке. Он работает в логарифмическом времени относительно размера списка. Однако применение бинарного поиска в связном списке может быть не таким прямолинейным, как в классическом списке Python, который поддерживает прямой доступ к элементам по индексу.

Для реализации бинарного поиска в связном списке можно использовать итеративный или рекурсивный подход. Важно учитывать, что в обычном связном списке нет прямого доступа к элементам по индексу, поэтому для проведения бинарного поиска в связном списке требуется осуществлять пошаговое перемещение от начала списка к нужному элементу.

Алгоритм реализации бинарного поиска в связном списке:

1. Указание на первый и последний элемент списка. Установка указателя на начало списка (голову) в начало и на указатель на конец списка (хвост) в конец.

2. Нахождение середины списка, используя указатели на начало и конец. Для этого нужно последовательно перемещаться по указателям в направлении с середины списка.

3. Сравнение значений элемента в середине с искомым значением:

- Если значение в середине равно искомому значению, то элемент найден, и поиск завершается.

- Если значение в середине больше искомого значения, то продолжить поиск в левой половине списка. Для этого переместить указатель на конец списка на элемент перед серединой.

- Если значение в середине меньше искомого значения, то продолжить поиск в правой половине списка. Для этого переместить указатель на начало списка на элемент после середины.

4. Повторение шагов 2 и 3 до тех пор, пока искомый элемент не будет найден или пока указатель на начало списка не станет больше указателя на конец списка. Если указатель на начало списка станет больше указателя на конец списка, то элемент не найден.

Отличия между реализацией бинарного поиска для связного списка и классического списка Python:

Доступ к элементам: В классическом списке Python доступ к элементам осуществляется по индексу за константное время, что позволяет быстро реализовать бинарный поиск. В связном списке каждый узел имеет указатель на следующий узел, и для доступа к элементу по индексу требуется последовательно пройти от начала списка до нужного узла.

Сложность операций: В классическом списке операции получения элемента по индексу выполняются за O(1), в то время как в связном списке для такой операции требуется O(n), где n - номер нужного элемента.

Ресурсы и память: Связные списки требуют больше памяти на хранение указателей на следующие элементы, чем классические списки. Поэтому использование бинарного поиска в связном списке может потребовать дополнительных ресурсов из-за необходимости пошагового перемещения по узлам.

Таким образом, реализация бинарного поиска в связном списке может быть более сложной и требует дополнительных операций по перемещению между узлами, в отличие от классического списка Python, где достаточно просто обращаться к элементам по индексу.

Разработанный программный код см. в приложении А.

Выводы

Была написана программа, содержащая в себе реализацию связанного однонаправленного списка. Изучены создание списка, добавление и удаление элементов в различные позиции, а также вывод информации о каждом элементе списка.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
    class Node:
        def __init__(self, data, next=None):
            self.__data = data
            self.next = next

        def get_data(self):
            return self.__data

        def __str__(self):
             next_data = self.next.__data if self.next is not None 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 self.head is None:
                self.head = new_node
            else:
                current = self.head
                while current.next is not None:
                    current = current.next
                current.next = new_node
            self.length += 1

        def __str__(self):
            if self.head is None:
                return "LinkedList[]"

            def node_to_string(node):
                return str(node)

            current = self.head
            elements = []
            while current is not None:
                elements.append(node_to_string(current))
                current = current.next

                    return  f"LinkedList[length  =  {self.length},  [{';
'.join(elements)}]]"

        def pop(self):
            if self.head is None:
                raise IndexError("LinkedList is empty!")

            if self.head.next is None:
```

```
            self.head = None
        else:
            current = self.head
            while current.next.next is not None:
                current = current.next
            current.next = None

        self.length -= 1

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

    def delete_on_end(self, n):
        if n <= 0 or self.length < n:
            raise KeyError("Element doesn't exist!")
        current = self.head
        end = self.length - n
        position = 0
        if position == end:
            self.head = self.head.next
            self.length -= 1
            return
        while current is not None and position + 1 != end:
            position = position + 1
            current = current.next
        if current is not None:
            current.next = current.next.next
        self.length -= 1
```