

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: Введение в алгоритмы и структуры данных

Студент гр. 3344

Щербак М.С.

Преподаватель

Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Целью работы является ознакомление с алгоритмами и структурами данных на языке Python.

Задание

Вариант 1. 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:

- 1) *data* # Данные элемента списка, приватное поле.
- 2) *next* # Ссылка на следующий элемент списка.

И следующие методы:

- 1) *__init__(self, data, next)* - конструктор, у которого значения по умолчанию для аргумента *next* равно *None*.
- 2) *get_data(self)* - метод возвращает значение поля *data* (это необходимо, потому что в идеале пользователь класса не должен трогать поля класса *Node*).
- 3) *__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:

- 1) *head* # Данные первого элемента списка.
- 2) *length* # Количество элементов в списке.

И следующие методы:

- 1) *__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.

- 2) `__len__(self)` - перегрузка метода `__len__`, он должен возвращать длину списка (этот стандартный метод, например, используется в функции `len`).
- 3) `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.
- 4) `__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:
 - a. Если список пустой, то строковое представление:
"*LinkedList[]*"
 - b. 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]*",
- 5) `pop(self)` - удаление последнего элемента. Метод должен выбрасывать исключение *IndexError* с сообщением "*LinkedList is empty!*", если список пустой.
- 6) `clear(self)` - очищение списка.
- 7) `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.

Выполнение работы

1.Связный список – это структура данных, состоящая из узлов, где каждый узел содержит данные и ссылку (указатель) на следующий узел в списке. Основные отличия связного списка от массива:

- В связном списке элементы хранятся не последовательно в памяти, а могут быть разбросаны по памяти, поэтому доступ к элементам осуществляется через ссылки.

- Добавление и удаление элементов в связном списке выполняется быстрее, чем в массиве, так как не требуется сдвигать другие элементы.(В общем случае, добавление и удаление элементов в связном списке может быть быстрее, чем в массиве, так как не требуется перемещать все элементы после вставляемого или удаляемого. Однако, в определенных ситуациях, таких как добавление элемента в конец массива или удаление элемента из середины массива, массив может быть более эффективным. В целом, выбор между массивом и связным списком должен зависеть от конкретных требований задачи.)

- Доступ к элементам связного списка имеет линейную сложность $O(n)$, в отличие от массива, где доступ к элементам имеет константную сложность $O(1)$.

2.Сложности методов. $O(1)$: `__init__`, `get_data`, `Node.__str__`, `__len__`, `__clear__`, `append`(если добавляем голову), `pop`(если список пуст), `delete_on_end`(если удаляем голову). $O(n)$: `LinkedList.__str__`, `append`, `pop`, `delete_on_end`.

3.Реализация бинарного поиска в связном списке отличается от классического списка Python из-за особенностей структуры связного списка. Для бинарного поиска в связном списке необходимо обходить список с помощью указателей, сравнивать значения узлов и перемещаться дальше по указателям в зависимости от результата сравнения. Алгоритм бинарного поиска для связного списка может быть реализован рекурсивно или итеративно, но

требует дополнительных проверок и операций с указателями. Реализация бинарного поиска для связного списка может выглядеть следующим образом:

1. Для нахождения середины связного списка понадобится два указателя: один будет двигаться по списку на одну позицию за каждую итерацию, а другой на две позиции. Когда быстрый указатель (продвигающийся на две позиции) достигнет конца списка, медленный указатель будет указывать на середину.
2. На каждом шаге бинарного поиска находится средний элемент списка. Сравниваем его с искомым значением: если средний элемент равен искомому значению, возвращаем его.
3. В случае, если ключ не совпадает со средним элементом, выбираем, какую половину списка использовать для следующего поиска.
4. Если ключ меньше среднего узла, то для следующего поиска используется левая часть списка.
5. Если ключ больше среднего узла, то для следующего поиска используется правая часть списка.
6. Продолжаем делить список пополам и сужать интервал поиска, пока не найдем искомый элемент или не исчерпаем весь список.

Основное отличие реализации алгоритма бинарного поиска для связного списка заключается в необходимости использования алгоритма для нахождения среднего элемента, так как прямого доступа по индексу нет. Это делает алгоритм менее эффективным.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Тест	Выходные данные	Комментарии
1.	<pre> node = Node(1) print(node) node.next = Node(2, None) print(node) print(node.get_data()) l_l = LinkedList() print(l_l) print(len(l_l)) l_l.append(10) l_l.append(20) l_l.append(30) l_l.append(40) print(l_l) print(len(l_l)) l_l.delete_on_end(3) print(l_l) </pre>	<pre> data: 1, next: None data: 1, next: 2 1 LinkedList[] 0 LinkedList[length = 4, [data: 10, next: 20; data: 20, next: 30; data: 30, next: 40; data: 40, next: None]] 4 LinkedList[length = 3, [data: 10, next: 30; data: 30, next: 40; data: 40, next: None]] </pre>	Данные обработаны корректно
2.	<pre> node = Node(1) print(node) node.next = Node(2, None) print(node) 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.delete_on_end(1) print(l_l) </pre>	<pre> 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 = 2, [data: 111, next: 222; data: 222, next: None]] </pre>	Данные обработаны корректно

Выводы

Были получены базовые навыки работы с алгоритмами и структурами данных. Была написана программа, с помощью которой были изучены сложность алгоритмов и методы работы со связными списками.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lab2.py

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

    def get_data(self):
        return self.__data

    def __str__(self):
        if self.next is not None:
            return f"data: {self.__data}, next: {self.next.get_data()}"
        else:
            return f"data: {self.__data}, next: None"

class LinkedList:
    def __init__(self, head=None):
        if head is None:
            self.head = None
            self.length = 0
        else:
            self.head = Node(head)
            self.length = 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[]"
        else:
            result = "LinkedList[length = {}, ".format(self.length)
            current = self.head
            while current is not None:
                result += "data: {}, next: {}";
                result = result[:-2] if current.next is not None else "None"
                current = current.next
            result += "]"
            # Remove the extra "; "
```

```

        return result

def pop(self):
    if self.head is None:
        raise IndexError("LinkedList is empty!")
    elif self.head.next is None:
        self.head = None
        self.length = 0
    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 self.length < n or n <= 0:
        raise KeyError("Element doesn't exist!")
    else:
        index = self.length - n
        tmp = self.head
        if index == 0:
            self.head = tmp.next
        else:
            for i in range(1, index):
                tmp = tmp.next
            tmp.next = tmp.next.next
        self.length -= 1

```