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

Студент гр. 3343 _____ Какира У.Н.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Изучить алгоритмы и структуры данных, а также способы их реализации в языке программирования Python. Разработать программу, выполняющую поставленную в задании задачу.

Задание

Вариант 2

В данной лабораторной работе Вам предстоит реализовать связный ***однонаправленный*** список.

Node

Класс, который описывает элемент списка.

Класс Node должен иметь 2 поля:

__data # данные, приватное поле

__next__ # ссылка на следующий элемент списка

Вам необходимо реализовать следующие методы в классе Node:

__init__(self, data, next)

конструктор, у которого значения по умолчанию для аргумента next равно None.

get_data(self)

метод возвращает значение поля __data.

__str__(self)

перегрузка метода __str__. Описание того, как должен выглядеть результат вызова метода смотрите ниже в примере взаимодействия с Node.

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.

Класс LinkedList должен иметь 2 поля:

__head__ # данные первого элемента списка

__length # количество элементов в списке

Вам необходимо реализовать конструктор:

__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.

и следующие методы в классе LinkedList:

__len__(self)

перегрузка метода __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 метода __str__. Описание того, как должен выглядеть результат вызова метода смотрите ниже в примере взаимодействия с LinkedList.

pop(self)

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

clear(self)

очищение списка.

delete_on_start(self, n)

удаление n-того элемента с НАЧАЛА 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

В ходе выполнения программы были определены 2 класса: Node и LinkedList. Класс Node представляет собой отдельный элемент(звено) связного списка, содержащий 2 поля: __data__, хранящее данные, и __next__, хранящее указатель на следующий элемент в списке.

Класс Node также имеет 2 метода: get_data(), который возвращает данные, хранящиеся в элементе, и str(), который возвращает строку, описывающую элемент.

Класс LinkedList представляет собой связный список из элементов класса Node. Класс LinkedList имеет 2 поля: __length, хранящее количество элементов связного списка, и __head__, хранящее ссылку на первый элемент в списке. Класс LinkedList также имеет несколько методов:

len(): возвращает длину списка.

append(element) добавляет новый элемент с данными в конец списка.

str(): возвращает строку, описывающую список, путем обхода всех элементов в списке и сбора информации о каждом элементе.

pop(): удаляет последний элемент списка.

delete_on_start(n): удаляет n-й элемент считая от начала списка.

clear(): очищает список путем удаления всех элементов.

Алгоритм бинарного поиска работает только для отсортированной структуры данных, поэтому сперва связный список нужно отсортировать, что не так просто нежели со стандартным массивом, но алгоритмическая сложность остается похожей. Сам алгоритм предполагает разбивку списка на более маленькие области путем сравнения серединного элемента каждой области с искомым. И если для ограничения области удобно использовать ссылки на первый и последний элемент области, после чего обновлять их, то поиск серединного элемента весьма трудоемкий процесс, ведь для каждой области нужно пройти n/2 элементов. Тогда сложность данного алгоритма выходит O(n), то есть для связных списков данный алгоритм практически бесполезен в отличие от классических списков, которые представляют собой последовательный

упорядоченный набор элементов, что позволяет обращаться к элементам по индексам и, следовательно, быстрее искать “середину”

Тестирование

Здесь результаты тестирования, которые помещаются на одну страницу.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
	A = LinkedList() a.append(1) a.clear() print(a)	LinkedList[]	Ответ верный
	a = LinkedList() a.append(1) a.append(2) a.append(3) a.delete_on_start(2) print(a)	LinkedList[length = 2, [data: 1, next: 3; data: 3, next: None]]	Ответ верный

Выводы

Были изучены алгоритмы и структуры данных, а также способы их реализации в языке программирования Python.

Была разработана программа, реализующая такую структуру данных как односвязный список. Он представляет собой класс с соответствующими полями и методами, благодаря которым можно пополнять список элементами, удалять их, а также выводить различную информацию.

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

class LinkedList:
    def __init__(self, head = None):
        if head is not None:
            self.__length = 1
            self.__head__ = Node(head)
        else:
            self.__length = 0
            self.__head__ = None

    def __len__(self):
        return self.__length

    def append(self, element):
        node = Node(element)
        if self.__head__ is None:
            self.__head__ = node
        else:
            tmp = self.__head__
            while tmp.next is not None:
                tmp = tmp.next
```

```
                tmp.next = node
            self.__length += 1

        def __str__(self):
            if self.__length > 0:
                node_list = []
                tmp = self.__head__
                while tmp is not None:
                    node_list.append(tmp.__str__())
                    tmp = tmp.next
                node_list = '; '.join(node_list)
                return     f"LinkedList[length     =     {self.__length},
[{node_list}]]"
            else:
                return "LinkedList[]"

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

        def delete_on_start(self, n):
            if n < 1 or n > self.__length:
                raise KeyError(f"{n} doesn't exist!")
            elif n == 1:
                self.__head__ = self.__head__.next
            else:
                tmp = self.__head__
                for i in range(n - 2):
                    tmp = tmp.next
                tmp.next = tmp.next.next
```

```
        self.__length -= 1

    def clear(self):
        self.__head__ = None
self.__length = 0
```