МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Информатика»

Тема: Алгоритмы и структуры данных в Python

Студент гр. 9381 _____ Колованов Р.А.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2019

Цель работы.

Изучение алгоритмов и структур данных в Python, а также использование парадигмы ООП для написания программ.

Задание.

В данной лабораторной работе Вам предстоит реализовать связный двунаправленный список.

Node

Класс, который описывает элемент списка. Класс Node должен иметь 3 поля:

- __data # данные, приватное поле
- __prev__ # ссылка на предыдущий элемент списка
- __next__ # ссылка на следующий элемент списка

Вам необходимо реализовать следующие методы в классе Node:

- __init__(self, data, prev, next)

конструктор, у которого значения по умолчанию для аргументов prev и next равны 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.

Linked List

Класс, который описывает связный двунаправленный список. Класс LinkedList должен иметь 3 поля:

- __length # длина списка
- __first__ # данные первого элемента списка
- __last__ # данные последнего элемента списка

Вам необходимо реализовать конструктор:

- __init__(self, first, last)

конструктор, у которого значения по умолчанию для аргументов first и last равны None.

Если значение переменной first равно None, а переменной last не равно None, метод должен вызывать исключение ValueError с сообщением: "invalid value for last".

Если значение переменной first не равно None, а переменной last равна None, метод должен создавать список из одного элемента. В данном случае, first равен last, ссылки prev и next равны None, значение поля __data для элемента списка равно first.

Если значения переменных не равны None, необходимо создать список из двух элементов. В таком случае, значение поля __data для первого элемента списка равно first, значение поля __data для второго элемента списка равно last.

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.

- popitem(self, element)

удаление элемента, у которого значение поля __data равно element. Метод должен выбрасывать исключение KeyError, с сообщением "<element> doesn't exist!", если элемента в списке нет.

- clear(self)

очищение списка.

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 данной лабораторной работе был разработан класс связного списка. *Связный список* - базовая динамическая структура данных в информатике, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка. Принципиальным отличием перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

Преимущества:

- Эффективное добавление и удаление элементов.
- Размер ограничен только объёмом памяти компьютера и разрядностью указателей.
- Динамическое добавление и удаление элементов.

Недостатки:

- Сложность прямого доступа к элементу, а именно определения физического адреса по его индексу в списке.
- На поля-указатели расходуется дополнительная память.
- Некоторые операции со списками медленнее, чем с массивами, так как к произвольному элементу списка можно обратиться, только пройдя все предшествующие ему элементы.

- Соседние элементы списка могут быть распределены в памяти нелокально, что снизит эффективность кэширования данных в процессоре.
- Над связными списками, по сравнению с массивами, гораздо труднее производить параллельные векторные операции, такие, как вычисление суммы: накладные расходы на перебор элементов снижают эффективность распараллеливания.

Описание и сложность методов приведена в Таблице 1.

Класс	Метод	Описание	Сложность
Node	*__init__*	Конструктор. Задает начальные значения полям объекта.	O(1)
	get_data	Получает данные текущего узла.	O(1)
	__str__	Преобразует объект к типу *str*.	O(1)
LinkedList	*__init__*	Конструктор. Задает начальные значения полям объекта.	O(1)
	__len__	Возвращает количество элементов в списке.	O(1)
	__iter__	Возвращает итератор типа *LinkedList*.	O(1)
	__str__	Преобразует объект к типу *str*.	O(n)
	append	Добавляет элемент в конец списка.	O(1)
	clear	Очищает список.	O(1)
	pop	Удаляет последний элемент в списке.	O(1)

	popitem	Удаляет первый встретившийся элемент с определенным значением.	O(n)
LinkedListIterator	*__init__*	Конструктор. Задает начальные значения полям объекта.	O(1)
	__next__	Возвращает текущий элемент и перемещается на следующий.	O(1)

Таблица 1.

Для класса можно переопределить метод *__getitem__*, который бы позволял обращаться к элементам списка по индексу. В таком случае реализация бинарного поиска для данного класса и для списков в Python ничем бы не отличалась.

Листинг 1. Метод *__getitem__* и бинарный поиск.

```
def __getitem__(self, index):
    begin = self.__first__

    for i in range(index):
        begin = begin.__next__

    return begin.get_data()

def binary_search(L, x):
    i = 0
    j = len(L) - 1
    m = int(j / 2)

    while L[m] != x and i < j:
        if x > L[m]:
            i = m + 1
        else:
            j = m - 1
        m = int((i + j) / 2)

    if i > j:
        return -1
    else:
        return m

Функция binary_search будет работать как со списками list, так и со
списками LinkedList.
```

Разработанный программный код см. в приложении А.

Тестирование.

Тестирование производилось на образовательной платформе Stepik.

Выводы.

Были изучены основные алгоритмы и структуры данных в Python. На языке Python, руководствуясь парадигмой ООП, была разработана программа, предоставляющая класс связного списка. Для написания программы использовались условные операторы, операторы цикла, классы, а также обработчик исключений *try-except*. Помимо этого, для класса связного списка был написан класс-итератор.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

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

    def get_data(self):
        return self.__data

    def __str__(self):
        s1, s2 = "None", "None"

        if self.__prev__ != None:
            s1 = self.__prev__.get_data()
        if self.__next__ != None:
            s2 = self.__next__.get_data()

        return "data: {}, prev: {}, next: {}".format(self.__data, s1,
s2)


class LinkedListIterator:
    def __init__(self, linked_list):
        self.current = linked_list.__first__

    def __next__(self):
        if self.current != None:
            node = self.current
            self.current = self.current.__next__
            return node
        else:
            raise StopIteration


class LinkedList:
    def __init__(self, first=None, last=None):
        if first == None:
            self.__length = 0
            self.__first__ = None
            self.__last__ = None

            if last != None:
                raise ValueError("invalid value for last")

        elif first != None and last == None:
            self.__length = 1
            self.__first__ = Node(first)
            self.__last__ = self.__first__

        else:
```

```
            self.__length = 2
            self.__first__ = Node(first)
            self.__last__ = Node(last, prev=self.__first__)
            self.__first__.__next__ = self.__last__

    def __len__(self):
        return self.__length

    def __iter__(self):
        return LinkedListIterator(self)

    def __str__(self):
        if self.__length > 0:
            return "LinkedList[length = {},
[{}]]".format(self.__length, "; ".join(list(map(str, self))))
        else:
            return "LinkedList[]"

    def append(self, element):
        if self.__length == 0:
            self.__first__ = Node(element)
            self.__last__ = self.__first__
        else:
            self.__last__.__next__ = Node(element, prev=self.__last__)
            self.__last__ = self.__last__.__next__

        self.__length += 1

    def clear(self):
        self.__length = 0
        self.__first__ = None
        self.__last__ = None

    def pop(self):
        if self.__length == 0:
            raise IndexError("LinkedList is empty!")
        else:
            if self.__last__.__prev__ == None:
                self.__last__ = None
                self.__first__ = None
            else:
                self.__last__ = self.__last__.__prev__
                self.__last__.__next__ = None

            self.__length -= 1

    def popitem(self, element):
        current = None
        is_exists = False

        for node in self:
            if node.get_data() == element:
                is_exists = True
                current = node

        if is_exists:
```

```
        if current is self.__last__:
            if self.__last__.__prev__ == None:
                self.__last__ = None
                self.__first__ = None
            else:
                self.__last__ = self.__last__.__prev__
                self.__last__.__next__ = None
        elif current is self.__first__:
            if self.__last__.__prev__ == None:
                self.__last__ = None
                self.__first__ = None
            else:
                self.__first__ = self.__first__.__next__
                self.__first__.__prev__ = None
        else:
            current.__next__.__prev__ = current.__prev__
            current.__prev__.__next__ = current.__next__
        self.__length -= 1
    else:
        raise KeyError("{} doesn't exist!".format(element))
```