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

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Алгоритмы и структуры данных»
Тема: Реализация и исследование развернутого связного списка

Студент гр. 2384 ___ Кузьминых Е.М

Преподаватель ___ Иванов Д.В.

Санкт-Петербург
2023

Цель работы.

Написать реализацию развернутого односвязного списка на языке программирования Python.

Задачи.

Развёрнутый связный список — список, каждый физический элемент которого содержит несколько логических элементов (обычно в виде массива, что позволяет ускорить доступ к отдельным элементам).

Данная структура позволяет значительно уменьшить расход памяти и увеличить производительность по сравнению с обычным списком. Особенно большая экономия памяти достигается при малом размере логических элементов и большом их количестве.

У данной структуры необходимо реализовать основные операции: поиск, удаление, вставка, а также функцию вывода всего списка в консоль через пробел. В качестве элементов для заполнения используются целые числа. Так же при инициализации списка существует необязательный параметр n_array (значение по умолчанию = 4), отвечающий за размер физического элемента (т.е. за размер массива).

Для проверки работоспособности структуры необходимо реализовать функцию (не метод класса) check, принимающую на вход два массива: массив arr_1 для заполнения структуры, массив arr_2 для поиска и удаления, а также необязательный параметр n_array (описан выше). Функция должна сначала заполнять развернутый связный список данным arr_1, затем искать элементы arr_2 и удалять их. После каждой операции по обновлению списка необходимо осуществлять полный его вывод в консоль.

Помимо реализации описанного класса Вам необходимо провести исследование его работы: сравнить время (дополнительные исследуемые параметры, такие как память и на то, что Вам хватит

фантазии - будут плюсом) у реализованной структуры, массива (для Python используйте list, для Cpp - стандартный массив) и односвязного списка (код реализации массива и односвязного списка загружать не нужно!).

Чтобы провести исследование необходимо проверить основные операции на маленьком (около 10), среднем (10000) и большом (100000) наборах данных для всех трёх случаев операции (лучший, средний, худший). По итогам исследования в отчёте необходимо предоставить таблицу с результатами замеров, а так же их графическое представление (на одном графике необходимо изобразить одну операцию в одном случае для трёх структур, т.е. суммарно должно получиться 9 графиков).

Автоматическая проверка вашего кода пока не предусмотрена, поэтому он будет проверяться на защите - загружайте в репозитории исходный код программы (весь), а также тесты для тестирования реализованного вами класса и функций.

Выполнение работы.

Код состоит из двух классов: *Node* и *Unrolled_linked_list*.

Класс *Node* представляет узел, используемый для хранения элементов внутри *Unrolled Linked List*.

Узел содержит следующие поля:

length: Поле, которое отслеживает количество элементов в массиве *array* данного узла. При создании узла оно инициализируется нулем.

array: Массив, который хранит элементы данного узла. При создании узла массив пуст.

next: Ссылка на следующий узел в *Unrolled Linked List*. По умолчанию равно *None*.

Класс *unrolled_linked_list* представляет саму *Unrolled Linked List*

и содержит методы для вставки, удаления и поиска элементов, а также методы для управления структурой данных.

Класс имеет следующие поля:

n_array: Максимальная длина массива в каждом узле. По умолчанию равно 4.

head: Ссылка на первый узел в *Unrolled Linked List*. При создании списка равно *None*.

tail: Ссылка на последний узел в *Unrolled Linked List*. При создании списка равно *None*.

all_length: Общая длина списка, то есть количество всех элементов в *Unrolled Linked List*. При создании списка равно 0.

Методы, реализованные для работы со *unrolled linked list*:

append(self, value): Данный метод добавляет элемент *value* в конец списка. Если текущий последний узел полон (длина массива достигла *n_array*), создается новый узел и элемент добавляется в него.

insert(self, value, index): Этот метод вставляет элемент *value* в

указанный индекс внутри списка. Если индекс выходит за пределы списка, ничего не происходит. Если индекс больше или равен общей длине, элемент добавляется в конец списка. Если индекс в пределах списка, элемент добавляется в соответствующий узел.

delete(self, value): Этот метод удаляет все вхождения элемента *value* из *Unrolled Linked List*. Если после удаления узел становится недозаполненным (длина массива меньше *n_array // 2*), узлы объединяются.

delete_by_index(self, index): Метод удаляет элемент по глобальному индексу. Если после удаления узел становится недозаполненным, узлы объединяются.

search(self, value): Этот метод выполняет поиск элемента *value* в *Unrolled Linked List* и выводит индексы всех найденных элементов.

search_by_index(self, index): Метод выполняет поиск элемента по

глобальному индексу и выводит найденный элемент.

print_info(self): Метод выводит все элементы списка в порядке их следования.

Исследование работы Unrolled Linked List:

Ниже предоставлены графики с сравнением времени работы реализованного развернутого списка с *list* и обычным односвязным списком.

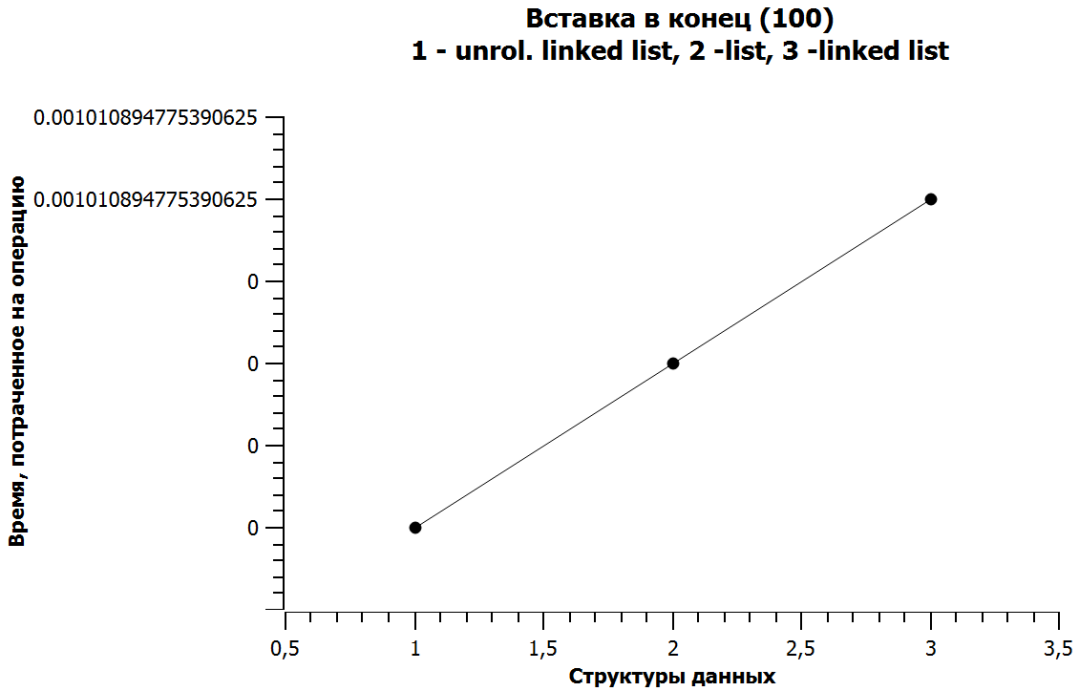

Рисунок 1 – Вставка в конец

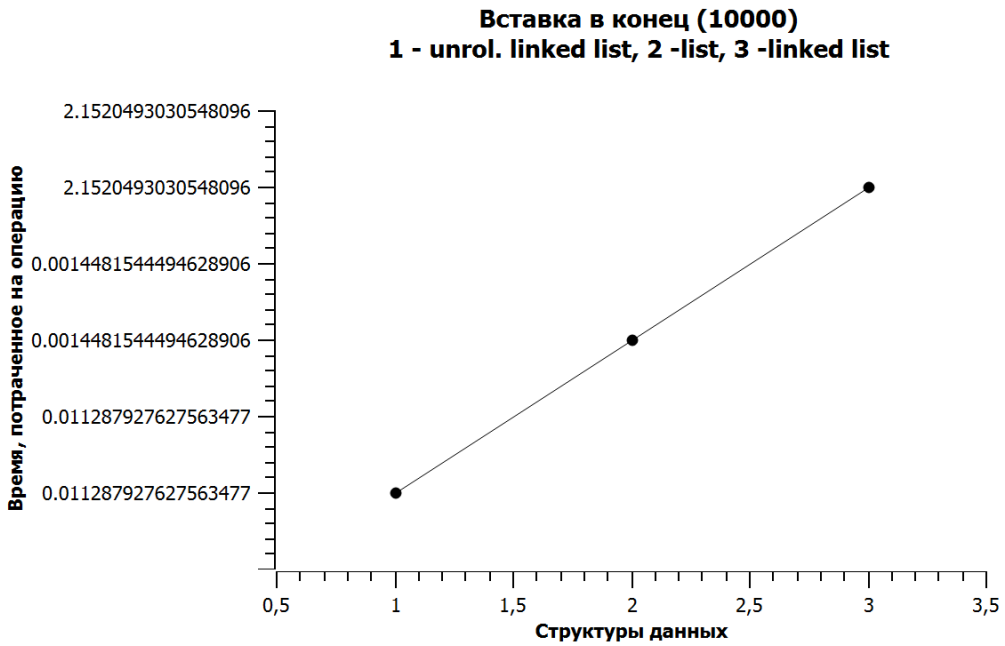

Рисунок 2 – Вставка в конец

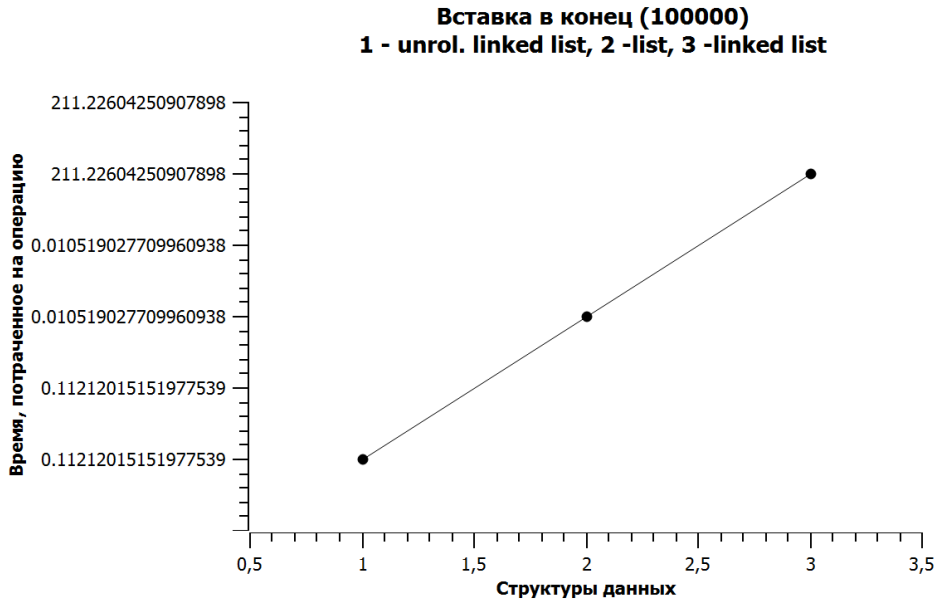

Рисунок 3 – Вставка в конец

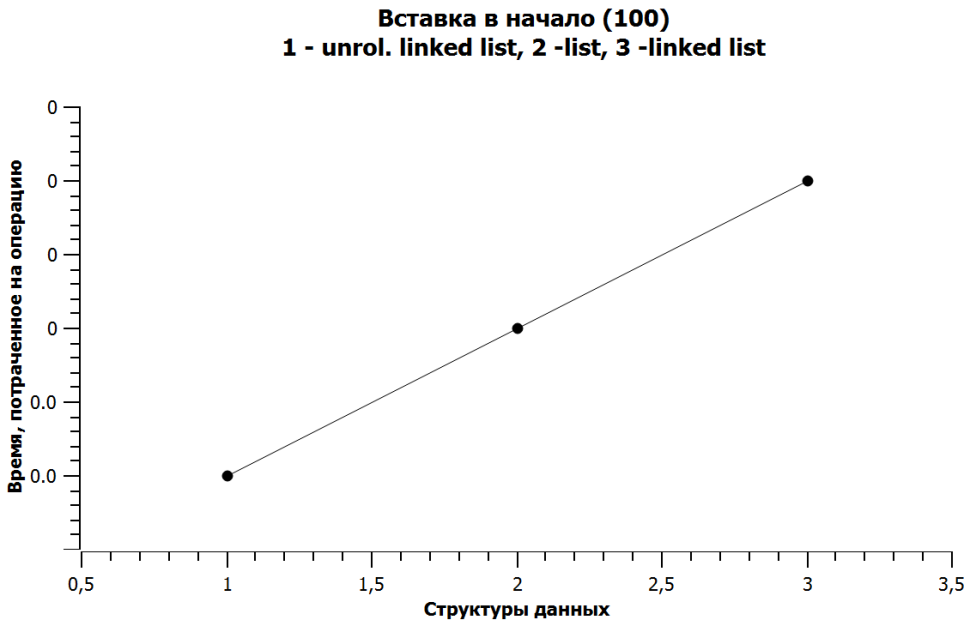

Рисунок 4 – Вставка в начало

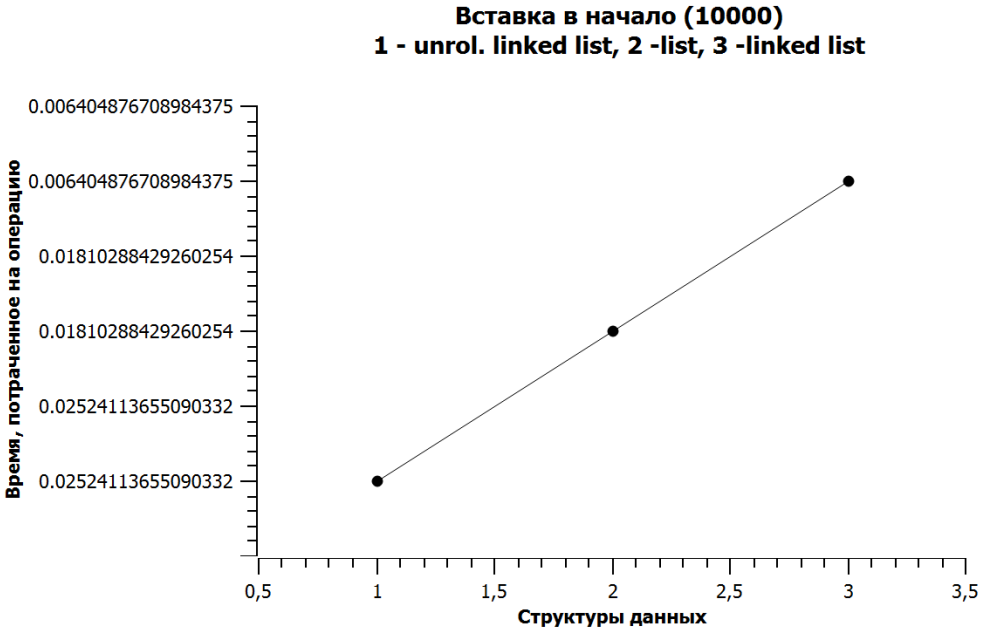

Рисунок 5 – Вставка в начало

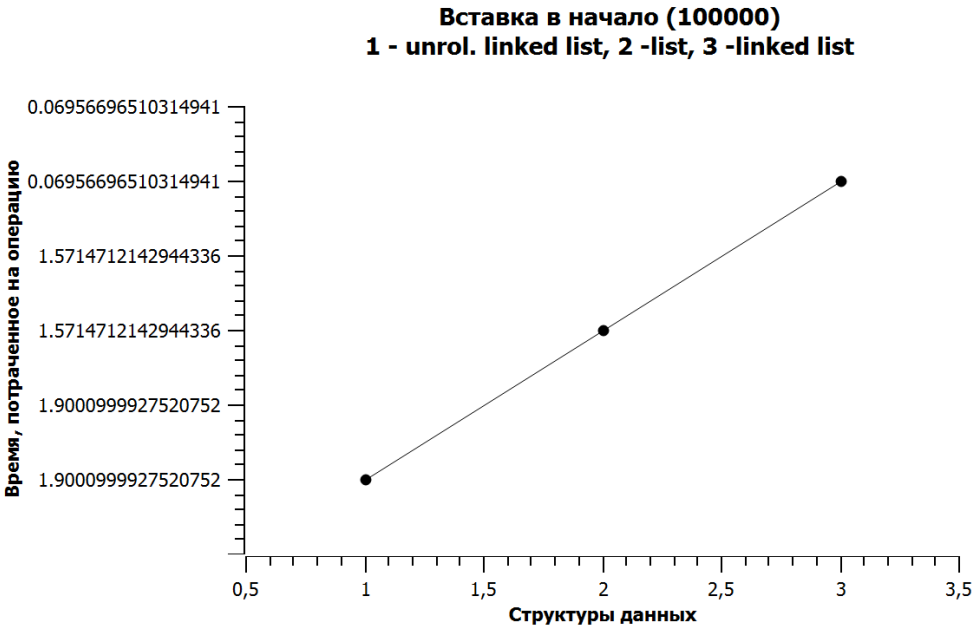

Рисунок 6 – Вставка в начало

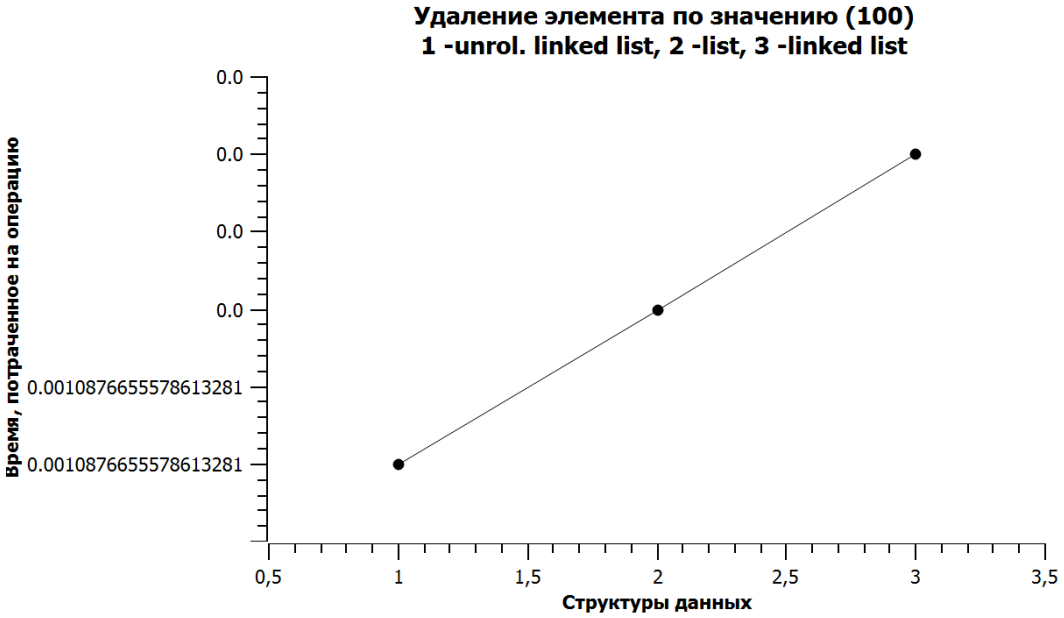

Рисунок 7 – Удаление элемента

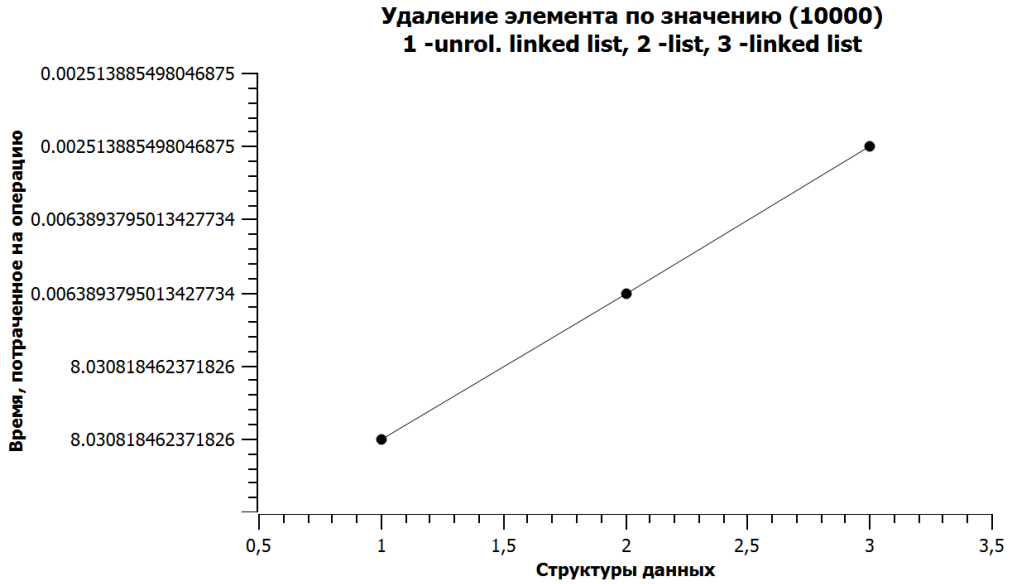

Рисунок 8 – Удаление элемента

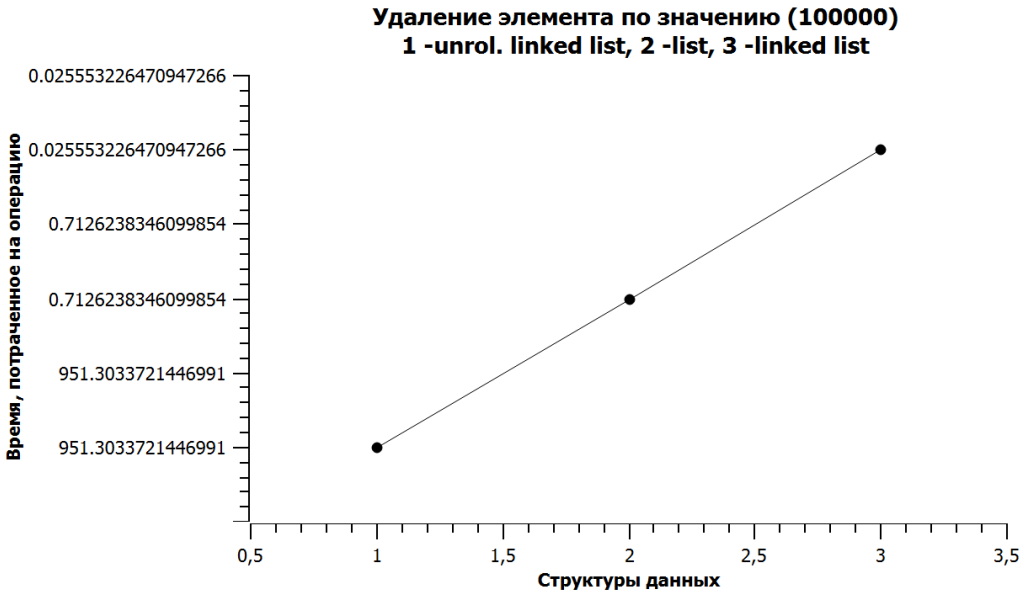

Рисунок 9 – Удаление элемента

Тестирование.

В ходе написания лабораторной работы для тестирования был создан файл *tests.py* с функцией *check*, принимающую на вход два массива: массив *arr_1* для заполнения структуры, массив *arr_2* для поиска и удаления. Тесты проводились с помощью *pytest*.

№	Входные данные	Выходные данные	Комментарий

1	def check(arr1, arr2):
	  my_unrolled_list = unrolled_linked_list()
	  for i in range(len(arr1)):
	
	my_unrolled_list.append(arr1[i])
	
	my_unrolled_list.print_info()
	  for i in range(len(arr2)):
	
	my_unrolled_list.delete(arr2[i])
	
	my_unrolled_list.print_info()
	
	  assert (my_unrolled_list.all_length==0)
	arr1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
	arr2 = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
	check(arr1, arr2)
		0==0	Ответ верный

2	def test_assert(value):
	  my_unrolled_list = unrolled_linked_list()
	  for i in range(10):
	
	my_unrolled_list.insert(value, i)
	  assert (my_unrolled_list.all_length==10)
	check_assert(100)
		10==10	Ответ верный

3	def test_print():
	  my_unrolled_list = unrolled_linked_list()
	  for i in range(10):
	
	my_unrolled_list.append(i)
	  assert my_unrolled_list.print_info()=='0 1 2 3 4 5 6 7 8 9 '
		'0 1 2 3 4 5 6 7 8 9 '== '0 1 2 3 4 5 6 7 8 9 '	Ответ верный

Вывод.

В ходе выполнения лабораторной работы была реализована структура данных - развернутый связный список, операции для взаимодействия с ним. Кроме этого, были написаны тесты для проверки корректности работы реализованного списка, использован *pytest* для тестирования программы. Произведен анализ скорости работы развернутого связного списка по сравнению с списком и связным списком

Приложение А.

Исходный код программы.

Main.py

```
class Node:
    def __init__(self):
        self.length = 0
        self.array = [0] * self.length
        self.next = None

class unrolled_linked_list:
    def __init__(self, n_array=4):
        self.n_array = n_array
        self.head = None
        self.tail = None
        self.all_length = 0

    def insert(self, value, index):
        if index < 0 or index > self.all_length:
            return

        if self.all_length == 0 or self.all_length <= index:
            self.append(value)
        else:
            global_index = 0
            my_temp = self.head
            prev_temp = None
            while my_temp is not None:
                for i in range(len(my_temp.array)):
                    if global_index == index:
                        if my_temp.length + 1 <= self.n_array:
                            my_temp.array.insert(i, value)
                        else:
                            new_node = Node()
                            split_index = self.n_array // 2
                            new_node.array =
my_temp.array[split_index:]
                            my_temp.array =
my_temp.array[:split_index]

                            if prev_temp is None:
                                self.head = new_node
                            else:
                                prev_temp.next = new_node
                            new_node.next = my_temp.next

                            if i < split_index:
                                my_temp.array.insert(i, value)
                            else:
                                new_node.array.insert(i - split_index,
value)
                        self.all_length += 1
                        return
                    global_index += 1
```

```
                prev_temp = my_temp
                my_temp = my_temp.next

    def append(self, value):
        if self.head == None:
            self.head = Node()
            self.head.array.append(value)
            self.head.length += 1
            self.all_length += 1
            self.tail = self.head

        elif self.tail.length + 1 <= self.n_array:
            self.tail.array.append(value)
            self.tail.length += 1
            self.all_length += 1
        else:
            new_node = Node()
            half_length = self.tail.length // 2
            mini_array = self.tail.array[-half_length:]
            new_node.array.extend(mini_array)
            self.tail.array = self.tail.array[:-half_length]

            new_node.array.append(value)
            new_node.length = len(new_node.array)
            self.all_length += 1
            self.tail.length = half_length
            self.tail.next = new_node
            self.tail = new_node

    def print_info(self):
        temp = self.head
        result_string = ''
        while temp != None:
            for i in range(len(temp.array)):
                result_string += f'{temp.array[i]} '
            temp = temp.next
        return result_string

    def delete(self, value):
        temp = self.head
        prev_temp = None
        while temp is not None:
            i = 0
            while i < len(temp.array):
                if temp.array[i] == value:
                    temp.array.pop(i)
                    temp.length -= 1
                else:
                    i += 1

            if len(temp.array) < self.n_array // 2:
                if prev_temp is not None and len(prev_temp.array) +
len(temp.array) <= self.n_array:
                    prev_temp.array.extend(temp.array)
                    prev_temp.next = temp.next
                    temp = prev_temp
                elif temp.next is not None and len(temp.array) +
len(temp.next.array) <= self.n_array:
```

```
                    temp.array.extend(temp.next.array)
                    temp.next = temp.next.next

            prev_temp = temp
            temp = temp.next

        if self.head is not None and len(self.head.array) == 0:
            self.head = self.head.next

        self.all_length -= 1

    def delete_by_index(self, index):
        if index < 0 or index >= self.all_length:
            return
        temp = self.head
        prev_temp = None
        global_index = 0

        while temp is not None:
            i = 0
            while i < len(temp.array):
                if global_index == index:

                    temp.array.pop(i)
                    temp.length -= 1

                    if len(temp.array) < self.n_array // 2:
                        if prev_temp is not None and
len(prev_temp.array) + len(temp.array) <= self.n_array:
                            prev_temp.array.extend(temp.array)
                            prev_temp.next = temp.next
                        elif temp.next is not None and len(temp.array)
+ len(temp.next.array) <= self.n_array:
                            temp.array.extend(temp.next.array)
                            temp.next = temp.next.next

                    self.all_length -= 1
                    return

                i += 1
                global_index += 1

            prev_temp = temp
            temp = temp.next

        if self.head is not None and len(self.head.array) == 0:
            self.head = self.head.next

    def search(self, value):
        temp = self.head
        while temp.next != None:
            for i in range(len(temp.array)):
                if value == temp.array[i]:
                    return temp.array[i]
            temp = temp.next

    def search_by_index(self, index):
        global_index = 0
```

```
        temp = self.head

        while temp != None:
            for i in range(len(temp.array)):
                if global_index == index:
                    return temp.array[i]
                global_index += 1
            temp = temp.next
```

Tests.py

```
from main import unrolled_linked_list
from main import Node

def check(arr1, arr2):
    my_unrolled_list = unrolled_linked_list()
    for i in range(len(arr1)):
        my_unrolled_list.append(arr1[i])
        print(my_unrolled_list.print_info())
    for i in range(len(arr2)):
        my_unrolled_list.delete(arr2[i])
        print(my_unrolled_list.print_info())
assert (my_unrolled_list.all_length == 0)
def test_assert(value=100):
    my_unrolled_list = unrolled_linked_list()
    for i in range(10):
        my_unrolled_list.insert(value, i)
    assert (my_unrolled_list.all_length == 10)
def test_search(len=10):
    my_unrolled_list = unrolled_linked_list()
    for i in range(len):
        my_unrolled_list.append(i)
    array = []
    for i in range(my_unrolled_list.all_length):
        array.append(my_unrolled_list.search_by_index(i))
    assert array == [x for x in range(len)]
def test_print():
    my_unrolled_list = unrolled_linked_list()
    for i in range(10):
        my_unrolled_list.append(i)
assert my_unrolled_list.print_info() == '0 1 2 3 4 5 6 7 8 9 '
```