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. 3384 Рудаков А.Л.

Преподаватель Шестопалов Р.П.

Санкт-Петербург

2024

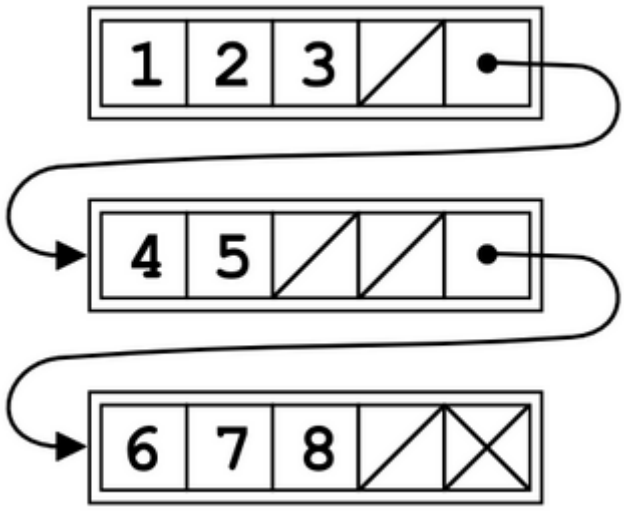
Цель работы.

Изучить, что из себя представляет развернутый связный список, и реализовать данную структуру данных. Также необходимо провести исследование его работы.

Задание.

В данном модуле курса Вы познакомились с различными структурами данных, такими как массив, связный список, абстрактными структурами данных (стек, очередь, днк и др.), но на этом структуры данных не заканчиваются... В рамках данной лабораторной работы необходимо реализовать “новую” структуру, которая называется развернутым связным списком.

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. Функция вычисления размера node находится в следующем блоке заданий.

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 (в начало, в середину, в конец). 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 предусмотрена в следующем блоке. Проверяться будет правильность выбора размера ноды, структуры списка и вставка элементов. Остальное будет проверяться на защите.

Задание 1.

Вам необходимо реализовать функцию calculate_optimal_node_size(num_elements) которая вычисляет оптимальный размер узла для хранения определенного количества элементов в памяти. Для этого вам необходимо узнать общий объем памяти(количество байт, где берем

элемент int, занимающий 4 байт), необходимый для хранения всех элементов, количество строк кэша(общий объем памяти поделить на минимальный размер строки кэша с округлением вверх), необходимого для хранения всех элементов(минимальный размер строки кэша = 64). Необходимо вывести оптимальный размер узла, с добавлением одной дополнительной строки кэша.

Задание 2.

Вам необходимо реализовать развернутый связный список который представляет собой связный список, в котором каждый узел содержит массив элементов и указатель на следующий узел. Для определения размера массива элеменов, вам необходимо использовать функцию из предыдущего задания. Выведите развернутый связный список по нодам.

Пример:

Input:

1 2 3 4 5 6 7 8 9 10

Output:

Node 0: 1 2

Node 1: 3 4

Node 2: 5 6

Node 3: 7 8

Node 4: 9 10

Выполнение работы.

В начале работы была написана функция *def calculate_optimal_node_size(num_elements)*, которая вычисляет оптимальный размер узла для хранения определенного количества элементов в памяти. Все формулы, используемые в ней, взяты из алгоритма, приведенного в задании.

Далее был написан класс *Node* - элемент развернутого связного списка, имеющий 2 поля: *data* — массив *int'ов* и *next_element* — указатель на следующий элемент.

После него был написан класс *Unrolled_Linked_List* — развернутый связный список, который имеет 3 поля: *__head* — указатель на начало списка, *__n_array* — размер массивов (элементов структуры) и *__length* — длина списка.

При инициализации класса в методе *__init__(self, data = None, n_array = 4)* изначально производятся проверки поступивших данных, они должны быть поданы в виде списка *list* элементами которого являются *int*, а так же длина поданного массива должна быть больше 0. В случае поступления неверных данных вызываются методы класса *__type_error_value()* и *__length_error()* соответственно. Также *n_array* должен иметь тип *int* и быть больше 0, иначе будет вызван метод *__type_error_index()*. Далее поля класса инициализируются поданными значениями, после чего начинается „заполнение“ списка. Поступивший массив данных *data* разбивается на части длины *n_array*, создаются структуры *Node* из этих частей при помощи метода *__create_node()*, после чего они объединяются в связный список массивов равной длины посредством добавления массива в конец списка методом *__add_node()*. Поле *__head* указывает на начало данного списка.

Далее идет метод *__getitem__(self, index),* который дает доступ к элементу по индексу. Вначале проверяется корректность входных данных, после чего вызывается метод *__find_current()* для поиска массива, в котором лежит значение, и возвращается значение по поданному индексу.

Далее идет метод *__add_node(self, head, element),* созданный для добавления массива в конец списка. На вход поступает начало списка *head* и элемент, который требуется вставить *element*. Далее, если *head* пустой, то ему присваивается значение *element*, иначе вставляется в конец.

Далее был создан метод *__str__(self)*, который нужен для вывода всего списка через пробел. Внутри инициализируется строка *output*, после чего идет обход всего списка подряд, при этом, проходя по каждому элементу (массиву) к строке добавляется строка, созданная из элементов текущего массива. В строку

они преобразуются при помощи метода *' '.join()* и функции *map()*. После прохождения по всему списку возвращается строка *output*.

Далее идет метод *__len__(self)*, который возвращает длину списка, беря значение из поля *__length*.

Далее реализован метод *find(self, value)*, созданный для того, чтобы искать индекс элемента в списке по значению *value*. Вначале идет проверка корректности поданного значение, если оно не корректно, вызывается метод *__type_error_value()*. После этого переменной *current* передается значение поля *head*, после чего в цикле *while* оно будет меняться на *current.next_element* до тех пор, пока не дойдет до последнего массива в списке либо пока в поле *current.data* не найдется значение *value*. в это же время в цикле к переменной *idx* прибавляется количество элементов внутри массивов, для того, чтобы если в каком либо массиве найдется нужное значение, к *idx* можно было прибавить индекс данного значения в этом массиве, тем самым получив индекс значения в списке, либо, если значение не найдется, присвоить *idx = -1*. Метод возвращает найденный индекс, либо *-1*, в случае, если значения нет в списке.

Далее был реализован метод *__find_current(self, n_idx)*, который похож на метод *find*, но выполняет немного другую функцию. Его задача по заданному индексу *n_idx* найти массив, в котором он находится, а также индекс, с которого начинаются элементы данного массива в списке, эта информация понадобится для последующих методов. Принцип работы метода отличается от *find()* тем, что цикл *while* идет не до момента, пока в *current.data* не найдется нужное значение, а до момента, пока сумма текущего индекса и длины текущего массива будет меньше чем заданный индекс. После цикла возвращается *current* - ссылка на нужный массив и *idx* — индекс начала массива в списке.

Далее идет метод *__create_node(self, data),* созданный для создания элементов списка. Возвращает созданный элемент класса *Node*.

После идет метод *__optimal_size(self),* который вызывается при добавлении или удалении элемент для перерасчета нужного *__n_array.* Если он меняется, то вызывается метод *__balance().*

Далее идет метод *__balance(self, n_array),* созданный для перебалансировки списка. Внутри метода создается новый список с *__n_array* равным поступившему, из элементов старого списка. После чего *head* нового списка передают в *__head* текущего.

После него идет метод *insert(self, value, in_idx)*, выполняющий функцию вставки значения *value* по индексу *in_idx*. Вначале проверяется правильность поданного значения, если оно неправильное, вызывается метод *__type_error_value()* и правильность индекса, если он неправилен, вызывается *__type_error_index()*. Далее при помощи метода *__find_current()* находится *current* - элемент (массив) списка, в который нужно выполнить вставку и *idx* — индекс начала данного элемента в списке. После чего проверяется корректность поданного индекса, если он выходит за границу списка, то вызывается метод *__index_error()*. Если же все в порядке, в первом условии проверяется, не является ли индекс, куда нужно вставить элемент, после последнего в массиве (например *n_array = 2; Node 0: 1, 2; Node 1: 3, 4*; требуется вставить элемент 10 на индекс 2, при этом найденный массив заканчивается индексом 1), если да, то создается *element* - новый элемент *Node*, с массивом, состоящим из 1 элемента *value*, после чего элемент вставляется между текущим и следующим (в примере станет *Node 0: 1, 2; Node 1: 10; Node 2: 3, 4*). Если же элемент не находится на границе массива, а лежит в нем, то проверяется наличие места в массиве, если в нем есть место (например *n_array = 4*, а в массиве *current.data* лежит *<= 3* значений), то значение *value* добавляется в массив *current.data* на нужное место. В противном случае создаются 2 новых элемента (массива), *first_element.data* включает в себя значения массива *current* до нужного индекса и *value*, а *second_element.data* включает в себя значения массива *current*, начиная с нужного индекса, после чего они расставляются на свои места (например *n_array = 4, Node 0: 1, 2, 3, 4; Node 1: 5, 6, 7, 8*; необходимо вставить элемент 10 на индекс *2*, получится *Node 0: 1, 2, 10 (first_element); Node 1: 3, 4 (second_element); Node 2: 5, 6 ,7, 8)*. Значение поля *length* увеличивается на 1. После выполнения вызывается метод *__optimal_size().*

Далее идет метод *__add_support(self, current, add_element)*, которая нужна для добавления промежуточного элемента между поступившим и его следующим.

Далее реализован метод *pop(self, idx)*, удаляющий элемент по индексу в списке. Вначале проверяется корректность поданного индекса, если же он некорректен, вызывается метод *__type_error_index()* либо *__index_error()*. Далее при помощи метода *__find_current()* определяется *current* и *idx*. и если длина массива *current.data* не равна *1*, то из него удаляется элемент стандартным методом *pop()* для списков (например *n_array = 3; Node 0: 1, 2, 3; Node 1: 4, 5, 6*; нужно удалить элемент с индексом *1*, тогда будет *Node 0: 1, 3; Node 1: 4, 5, 6*). Если же внутри элемент всего 1, то элемент *Node* удаляется из списка (например *n_array = 3; Node 0: 1, 2, 3; Node 1: 4; Node 2: 5, 6, 7*; нужно удалить элемент с индексом *3*, тогда будет *Node 0: 1, 2, 3; Node 1: 5, 6, 7*). Значение поля *length* уменьшается на 1. После вызывается метод *__optimal_size()*.

После идет метод *remove(self, value)*, который удаляет элемент по значению *value*. Внутри него вызывается метод *find*, который ищет индекс текущего значения, если индекс не найден, вызывается метод *__value_error()*, в противном случае вызывается метод *pop()* по найденному индексу.

Далее идет метод *print_Node(self)*, который нужен для того, чтобы вывести элементы списка так, как указано в задании 2.

Далее были написаны методы обработки ошибок

Первый из них метод *__index_error(self)* ловит ошибки индекса, выходящего за границы массива. Вызывается *raise IndexError('IndexError: Unrolled_Linked_List index out of range')*.

Далее идет метод *__value_error(self, value)*, который ловит ошибки некорректной подачи значений. Вызывается *raise ValueError(f'ValueError: {value} is not in Unrolled_Linked_List')*.

Далее метод *__type_error_value(self)*, который ловит ошибки неверного типа поданного значений. Вызывается *raise TypeError('TypeError: The Unrolled_Linked_List must consist of "int"')*.

Далее идет метод *__type_error_index(self)*, который ловит ошибки неверного типа поданного индекса. Вызывается *raise TypeError('TypeError: The index and n_array can only contain "int"')*.

Последний метод *__length_error(self)*, который ловит ошибки неправильно поданного массива значений при инициализации (длина равна 0). Вызывался *raise Exception('LengthError: The length of Unrolled_Linked_List cannot be equal to 0')*.

Кроме этого была реализована функция *check(arr_1, arr_2, n_array = 4)*, описанная в условии задания и созданная для проверки работоспособности новой структуры.

Описание пайплайна.

Объект класса *Unrolled_Linked_List* создается с входными данными: массивом и длиной элементов. В методе *__init__()* создается данный лист, используя в своей работе методы *__create_node()* для создания элемента списка (массива) и *__add_node()* для его добавления в конец. Для получения значения по индексу используется метод *__getitem__()*, использующий в работе метод *__find_current()* для поиска элемента списка (массива) в котором лежит нужный элемент и индекса начала этого массива. Для поиска элементов по значению используется *find()*. Для вставки элемента по индексу есть *insert()*, использующий методы *__find_current()*, *__create_node()*, *__add_support()* для вставка массива между другими массивами и *__optimal_size()* для вычисления нового размера элемента списка. Для удаления по индексу написан метод *pop()*, использующий *__find_current()*, *__add_support()* и *__optimal_size()*. Для удаления по значению есть *remove()*, использующий *find()* для поиска индекса значения и *pop()* для удаления по найденному индексу. Для расчета нового размера элемента массива есть метод *__optimal_size()*, который в случае изменения размера использует метод __balance, перестраивающий список под новый размер. Для обработки ошибок используются методы *__index_error()*, *__value_error()*, *__type_error_value()*, *__type_error_index()*, *__length_error()*. Для

возвращения длины есть метод *__len__()*. Для вывода списка на экран есть метод *__str__()* возвращающий все элементы через пробел и *print_Node()* для вывода элементов в формате, требуемом во втором задании.

Анализ полученных значений.

Также было проведено исследование. Замерено время работы функций поиска, удаления элемента и добавления элемента в начале, середине и конце развернутого связного списка, массива и односвязного списка.

Единицы измерения 10^{-6} с.

Таблица 1 – Результаты исследования

Операция	Набор данных	Место	Развернутый связный список	Массив (list)	Односвязный список
Поиск	маленький	начало	1.43	0.72	0.72
		середина	1.91	0.95	1.67
		конец	2.38	0.97	2.15
	средний	начало	1.43	1.19	4.29
		середина	25.51	19.79	491.39
		конец	41.01	36.72	2284.05
	большой	начало	4.77	2.38	4.53
		середина	291.11	222.21	4783.87
		конец	423.43	438.93	9811.64
Удаление (по значению)	маленький	начало	3.58	0.72	1.43
		середина	4.35	0.95	3.58
		конец	4.77	0.97	4.29
	средний	начало	3.81	1.43	4.77
		середина	28.85	29.33	557.18
		конец	77.25	57.46	2043.56
	большой	начало		20.98	4.77

		середина	267.27	295.64	4896.40
		конец	464.91	605.11	10256.77
Вставка	маленький	начало	3.57	0.48	0.72
		середина	4.05	0.72	2.92
		конец	4.77	0.72	4.29
	средний	начало	5.96	3.34	1.19
		середина	7.87	1.91	507.61
		конец	8.58	4.77	1820.31
	большой	начало	996.58	34.32	1.43
		середина	998.06	23.13	4802.12
		конец	1000.04	3.10	9856.23

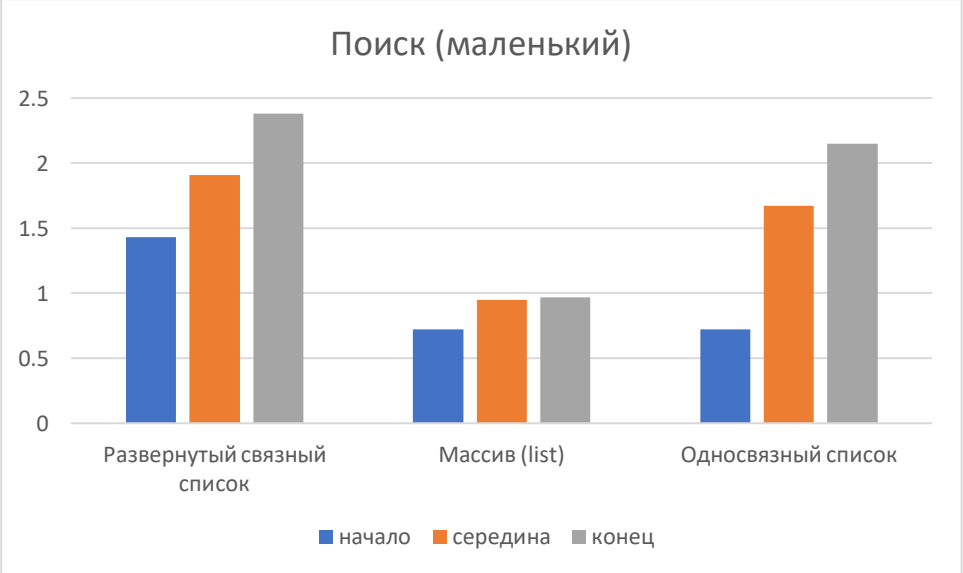


Рисунок 1 – Поиск элемента на маленьком объеме данных

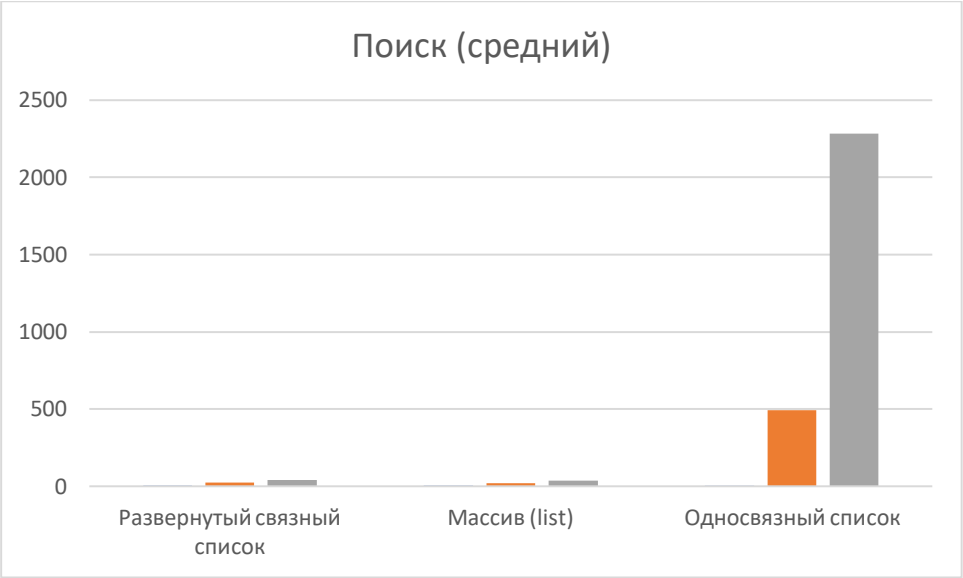


Рисунок 2 – Поиск элемента на среднем объеме данных

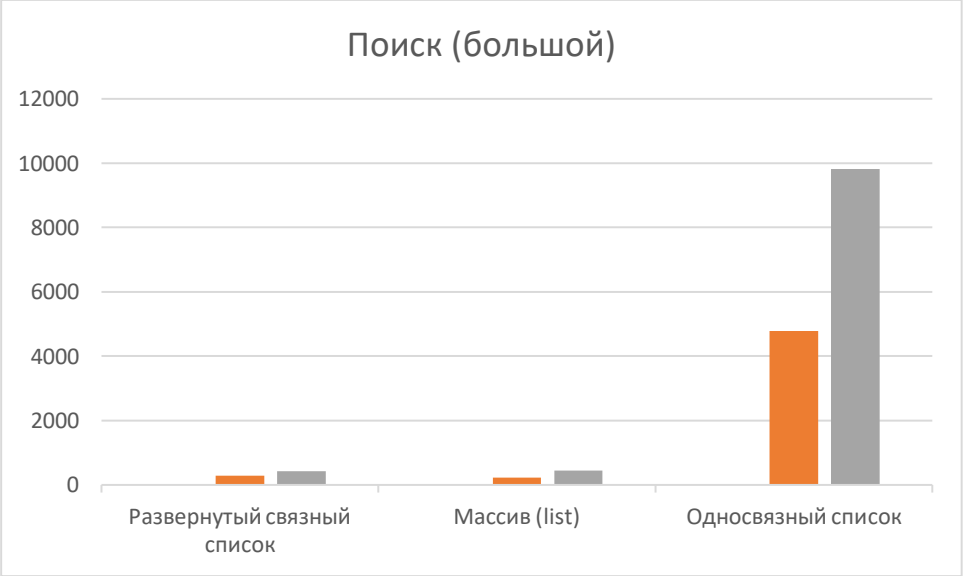


Рисунок 3 – Поиск элемента на большом объеме данных

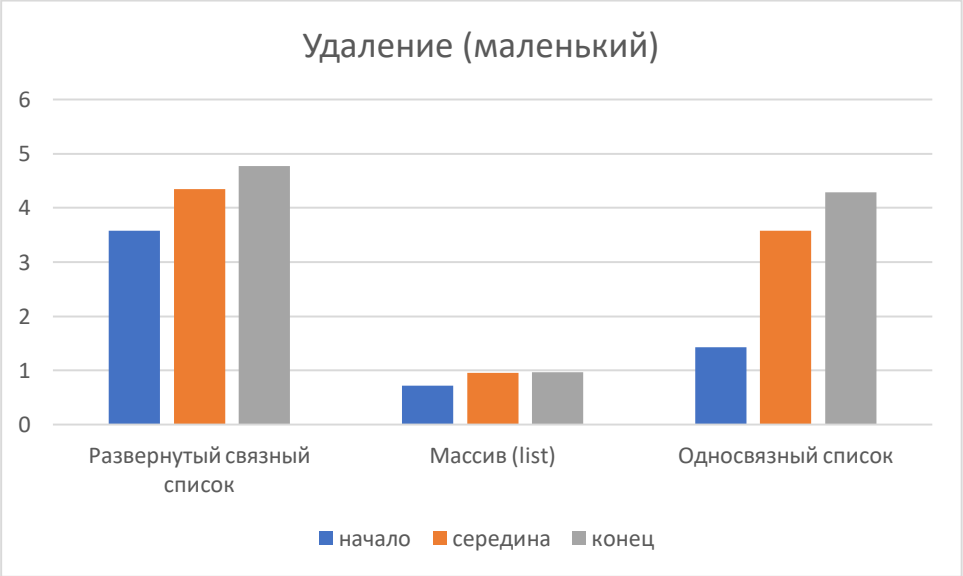


Рисунок 4 – Удаление элемента на маленьком объеме данных

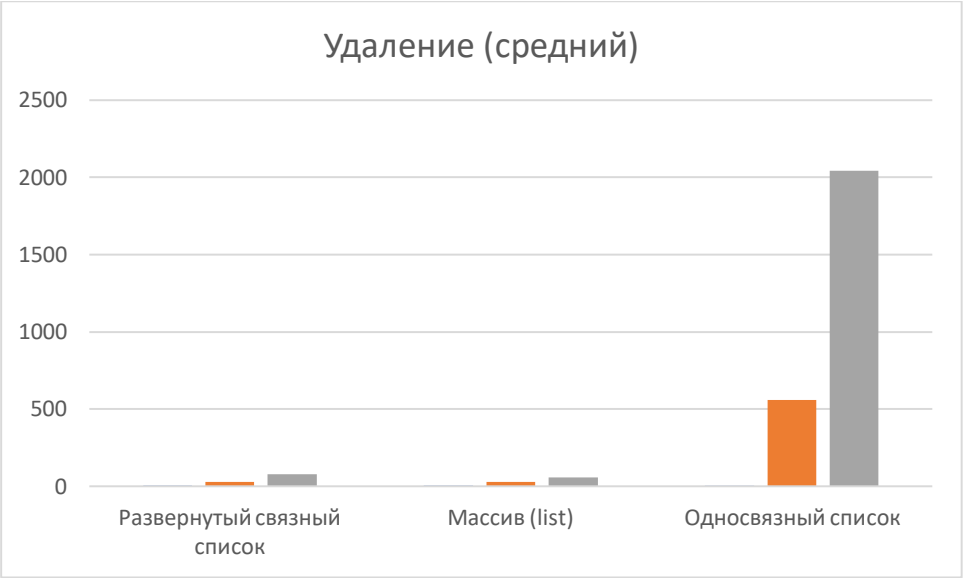


Рисунок 5 – Удаление элемента на среднем объеме данных

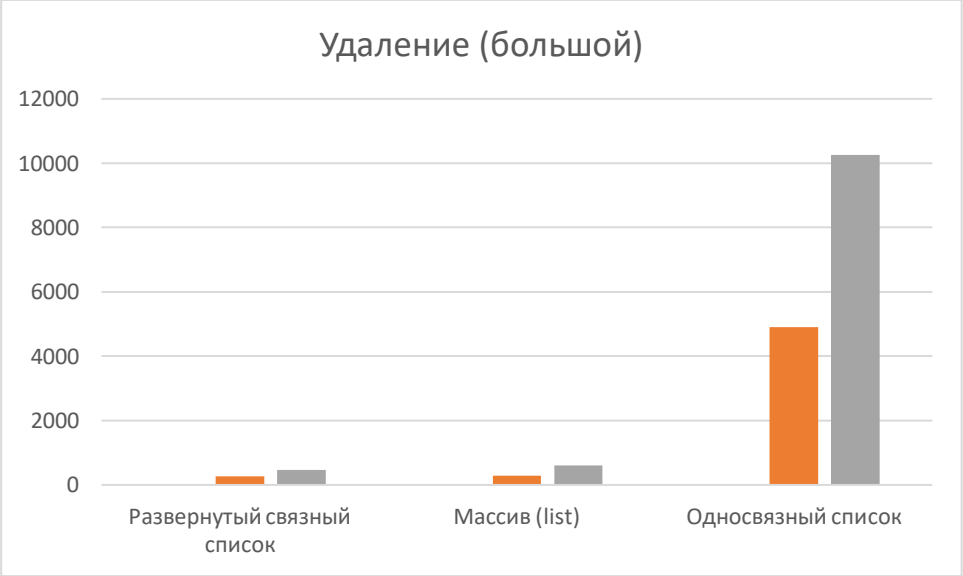


Рисунок 6 – Удаление элемента на большом объеме данных

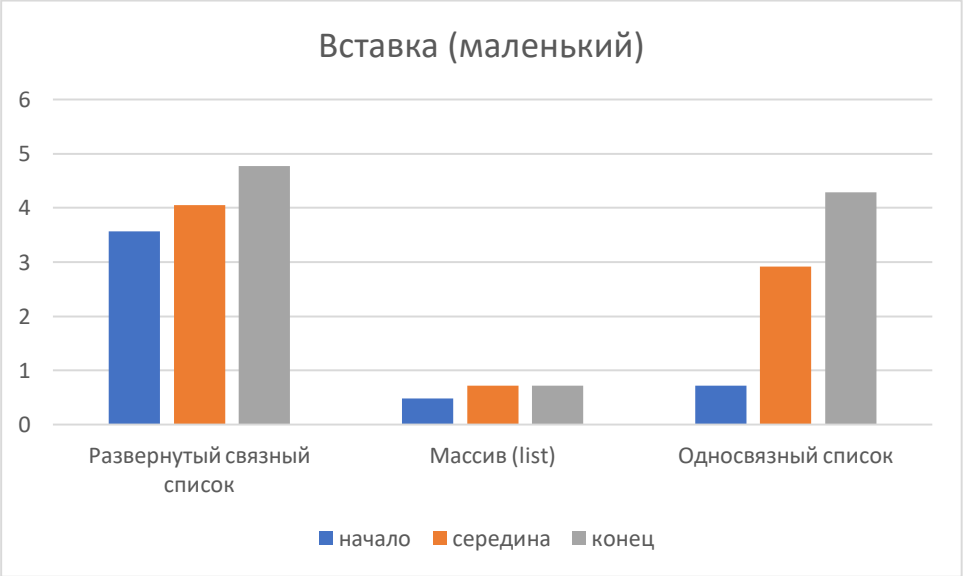


Рисунок 7 – Вставка элемента на маленьком объеме данных

Рисунок 8 – Вставка элемента на среднем объеме данных

Рисунок 9 – Вставка элемента на большом объеме данных

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Выводы.

Была изучена и реализована новая структура данных — развернутый связный список. Проведено исследование скорости его работы и сравнение с другими контейнерами. Оказалось, что при небольшом наборе данных он немного уступает в скорости односвязному списку и сильно уступает массиву, при средних наборах данных он примерно равен по скорости массиву и

значительно выигрывает у односвязного списка, при больших наборах данных он также примерно равен по скорости массиву и значительно выигрывает у односвязного списка.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
import math

def check(arr_1, arr_2, n_array = 4):
    my_list = Unrolled_Linked_List(arr_1, n_array)
    print(my_list)
    for i in arr_2:
        if my_list.find(i) != -1:
            my_list.remove(i)
            print(my_list)

def calculate_optimal_node_size(num_elements):
    size_of_int = 4
    total_memory = num_elements * size_of_int
    cache_min_size = 64
    num_lines = math.ceil(total_memory / cache_min_size) + 1
    return num_lines

class Node:
    def __init__(self, data = [], next_element = None):
        self.data = data
        self.next_element = next_element


class Unrolled_Linked_List:
    def __init__(self, data = None, n_array = 4):
        if not isinstance(n_array, int) or n_array == 0:
            self.__type_error_index()
        if not isinstance(data, list):
            self.__type_error_value()
        for value in data:
            if not isinstance(value, int):
                self.__type_error_value()
        if len(data) == 0:
            self.__length_error()
        i = 0
        self.__head = None
        self.__n_array = n_array
        self.__length = len(data)
        while len(data) > n_array * i:
            element = self.__create_node(data[n_array * i : n_array
* (i + 1)])
            self.__head = self.__add_node(self.__head, element)
            i += 1


    def __getitem__(self, index):
        if index >= self.__length or index < 0:
            self.__index_error()
        current, idx = self.__find_current(index)
        return current.data[index - idx]
```

```
        def __add_node(self, head, element):
            if head == None:
                head = element
            else:
                current = head
                while current.next_element != None:
                    current = current.next_element
                current.next_element = element
            return head


        def __str__(self):
            output = ''
            current = self.__head
            while current != None:
                output += ' '.join(map(str,current.data)) + ' '
                current = current.next_element
            return output


        def __len__(self):
            return self.__length


        def find(self, value):
            if not isinstance(value, int):
                self.__type_error_value()
            current = self.__head
            idx = 0
            while current.next_element != None and (not (value in
current.data)):
                idx += len(current.data)
                current = current.next_element
            if value in current.data:
                idx += current.data.index(value)
            else:
                idx = -1
            return idx


        def __find_current(self, n_idx):
            current = self.__head
            idx = 0
            while idx + len(current.data) <= n_idx and
current.next_element != None:
                idx += len(current.data)
                current = current.next_element
            return current, idx


        def __create_node(self, data):
            current = Node(data)
            return current


        def __optimal_size(self):
```

```
            current_n_array                                                 =
calculate_optimal_node_size(self.__length)
            if current_n_array != self.__n_array:
                self.__balance(current_n_array)
                self.__n_array = current_n_array


        def __balance(self, n_array):
            current = self.__head
            new_head = None
            data = []
            idx = 0
            curr_idx = 0
            while current != None:
                if 1 - curr_idx + idx + len(current.data) <= n_array:
                    data.extend(current.data[curr_idx:])
                    idx += len(current.data[curr_idx:])
                    curr_idx = 0
                    current = current.next_element
                else:
                    curr_idx = n_array - idx
                    data.extend(current.data[:curr_idx])
                if len(data) == n_array:
                    element = self.__create_node(data)
                    new_head = self.__add_node(new_head, element)
                    data = []
                    idx = 0
            if len(data) != 0:
                element = self.__create_node(data)
                new_head = self.__add_node(new_head, element)
            self.__head = new_head


        def insert(self, value, in_idx):
            if not isinstance(value, int):
                self.__type_error_value()
            if not isinstance(in_idx, int):
                self.__type_error_index()
            self.__length += 1
            current, idx = self.__find_current(in_idx)
            if idx + self.__n_array < in_idx or in_idx < 0:
                self.__length -= 1
                self.__index_error()
            elif idx + self.__n_array == in_idx:
                element = self.__create_node([value])
                element.next_element = current.next_element
                current.next_element = element
            else:
                if len(current.data) == self.__n_array:
                    first_data = current.data[:in_idx - idx]
                    first_data.append(value)
                    first_element = self.__create_node(first_data)
                    second_data = current.data[in_idx - idx:]
                    second_element = self.__create_node(second_data)
                    second_element.next_element = current.next_element
                    first_element.next_element = second_element
                    self.__add_support(current, first_element)
                else:
```

```
                    new_data = current.data[:in_idx - idx]
                    new_data.append(value)
                    new_data.extend(current.data[in_idx - idx:])
                    current.data = new_data
            self.__optimal_size()


        def __add_support(self, current, add_element):
            if current != self.__head:
                current_support = self.__head
                while current_support.next_element != current:
                    current_support = current_support.next_element
                current_support.next_element = add_element
            else:
                self.__head = add_element


        def pop(self, idx):
            if not isinstance(idx, int):
                self.__type_error_index()
            if idx < 0 or idx >= self.__length:
                self.__index_error()
            else:
                current, d_idx = self.__find_current(idx)
                if len(current.data) != 1:
                    current.data.pop(idx - d_idx)
                else:
                    self.__add_support(current, current.next_element)
                self.__length -= 1
                self.__optimal_size()


        def remove(self, value):
            idx = self.find(value)
            if idx == -1:
                self.__value_error(value)
            self.pop(idx)


        def print_Node(self):
            current = self.__head
            i = 0
            while current != None:
                print(f'Node {i}:', ' '.join(map(str, current.data)))
                i += 1
                current = current.next_element


        def __index_error(self):
            raise  IndexError('IndexError:  Unrolled_Linked_List  index
out of range')  # error


        def __value_error(self, value):
            raise   ValueError(f'ValueError:   {value}   is   not   in
Unrolled_Linked_List')  # error
```

```
        def __type_error_value(self):
            raise TypeError('TypeError: The Unrolled_Linked_List must
consist of "int"') #error


        def __type_error_index(self):
            raise TypeError('TypeError: The index and n_array can only
contain "int"') #error


        def __length_error(self):
            raise     Exception('LengthError:     The     length     of
Unrolled_Linked_List cannot be equal to 0')
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

Таблица Б.2 - Примеры тестовых случаев (функция check)

№ п/п	Входные данные	Выходные данные	Комментарии
1.	arr1 = [0,1,2,3,4,5,6] arr2 = [5,3]	0 1 2 3 4 5 6 0 1 2 3 4 6 0 1 2 4 6	Ok
2.	arr1 = [1,2,3] arr2 = [1,2,3]	1 2 3 2 3 3	Ok