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 исследования развёрнутого связного списка

Студент гр. 3342		Мохамед.М.Х
Преподаватель		Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Изучить структуру данных «развернутый связный список», реализовать её и провести исследование её работы.

Задание

В этом модуле курса вы изучили различные структуры данных, такие как массивы, связные списки, а также абстрактные структуры данных (например, стек, очередь, дек и другие). Однако на этом перечень структур данных не заканчивается. В рамках данной лабораторной работы вам предстоит реализовать "новую" структуру данных, известную как развёрнутый связный список.

Развёрнутый связный список — это список, в котором каждый физический элемент содержит несколько логических элементов, обычно представленных в виде массива. Это позволяет ускорить доступ к отдельным элементам. Эта структура данных существенно сокращает расход памяти и повышает производительность по сравнению с обычным списком. Особенно значительная экономия памяти достигается при малом размере логических элементов и их большом количестве.

Для данной структуры необходимо реализовать основные операции: поиск, удаление, вставка, а также функцию, которая будет выводить весь список в консоль через пробел. Элементы для заполнения списка будут представлены целыми числами. Функция вычисления размера узла (node) описана в следующем блоке заданий. Реализацию операций поиска и

удаления можно осуществить по своему усмотрению. Эти операции будут проверяться на защите.

Для проверки работоспособности структуры необходимо создать функцию (не метод класса) check, которая будет принимать два массива: arr_1 для заполнения структуры и arr_2 для поиска и удаления, а также необязательный параметр n_array (описан выше). Функция должна сначала заполнить развернутый связный список данными из arr_1, затем найти элементы из arr_2 и удалить их. После каждой операции обновления списка необходимо полностью выводить его в консоль.

Помимо реализации описанного класса, необходимо провести исследование его эффективности: сравнить время выполнения операций (дополнительно можно исследовать такие параметры, как использование памяти и другие аспекты по вашему усмотрению).

Для сравнения необходимо проверить основные операции на небольших (около 10), средних (10000) и больших (100000) наборах данных для всех трёх вариантов выполнения операций (в начало, в середину, в конец). По результатам исследования в отчёте нужно предоставить таблицу с результатами замеров и их графическое представление (на одном графике должна быть изображена одна операция для всех трёх структур, всего должно получиться 9 графиков).

Автоматическая проверка вашего кода предусмотрена в следующем блоке. Проверка будет включать корректность определения размера узла, структуры списка и операции вставки элементов. Остальные аспекты будут проверяться на защите.

Выполнение работы

В ходе выполнения работы был создан класс Node, который включает поля data и next, используемые для хранения данных и указателя на следующий элемент списка соответственно. Данные представлены в виде массива длины n, который передаётся в качестве аргумента в конструктор класса. По умолчанию этот массив пуст. Для добавления элементов в массив реализован метод add. Длина элемента класса определяется длиной массива данных, что позволяет переопределить метод len. Также был переопределён магический метод str, который теперь возвращает строку, содержащую элементы массива, разделённые пробелами.

Был также реализован класс UnLinList, конструктор которого принимает два параметра: длину элемента списка и его начальное значение (по умолчанию — None). В конструкторе инициализируются такие поля, как head (голова списка), приватное поле node_size (размер узла) и length (общее количество элементов массива). Для получения длины списка переопределён метод len, а для получения строкового представления списка — метод str. Если список пуст, метод str возвращает строку "UnrolledLinkedList[]". В противном случае он проходит по всем узлам списка, добавляя их строковое представление к результирующей строке в определённом формате.

Метод balance выполняет балансировку списка, которая может понадобиться при добавлении или удалении элементов. Эта балансировка производится, если результат вызова функции calculate_optimal_node_size для новой длины списка отличается от текущего значения приватного поля node_size. Если список пуст, возникает ошибка IndexError. В противном случае создаётся массив для хранения элементов, которые нужно удалить из одного узла и перенести в другой. Переменной elem присваивается

значение головы списка, после чего запускается цикл while, который проходит по всем узлам списка до его конца. В процессе: если размер текущего узла больше оптимального, избыточные элементы добавляются в массив save_elems и удаляются из узла с помощью среза. После этого переход к следующему узлу позволяет добавить сохранённые элементы в его начало. Если размер текущего узла меньше оптимального, массив save_elems очищается, чтобы предотвратить добавление лишних элементов. Далее определяется количество недостающих элементов, которые переносятся из следующего узла. Если все элементы перенесены, вызывается метод удаления узла с флагом 0, чтобы не менять значение length. Если же перемещённых элементов не хватает, переход к следующему узлу не производится. По завершении цикла обрабатывается последний элемент: если его размер превышает оптимальный, избыточные элементы добавляются в save_elems и переносятся в список либо методом add, если это позволяет размер узла, либо созданием нового узла.

Метод push добавляет элемент в конец списка. Если голова списка равна None, создаётся новый объект класса Node, в который добавляется элемент, после чего полю head присваивается это значение. В противном случае переменная pointer указывает на голову списка и перемещается по его узлам, пока не достигнет конца. Если размер последнего узла меньше оптимального, в него добавляется новый элемент, иначе создаётся новый узел и добавляется в список. Значение поля length увеличивается на единицу, и вызывается функция calculate_optimal_node_size. Если необходимо, производится балансировка.

Метод insert вставляет элемент по указанному индексу. Вставка производится по абсолютному индексу, который указывает позицию

элемента, а не узла. Если голова списка отсутствует, вызывается метод push. В противном случае метод проходит по всем узлам, пока не достигнет указанного индекса. Если индекс выходит за пределы списка, элемент добавляется в конец методом push. Если размер текущего узла позволяет, элемент вставляется в него. В противном случае последний элемент текущего узла переносится в следующий, а если это вызывает переполнение, процесс продолжается в последующих узлах. Поле length увеличивается на единицу, и при необходимости производится балансировка.

Метод pop_node удаляет заданный элемент из списка. Он принимает дополнительный флаг, по умолчанию равный 1. Если передать 0, при удалении узла значение length не изменяется. Метод проходит по списку, проверяя каждый элемент на совпадение с заданным. Если найденный элемент — следующий, поле next текущего элемента переназначается на next удаляемого узла. Если нужно удалить голову, полю head присваивается значение next удаляемого узла.

Метод pop_node_idx удаляет узел по индексу и работает аналогично методу pop_node.

Метод pop_elem удаляет элемент по индексу. Метод проходит по узлам, удаляя элемент с соответствующим индексом внутри узла. Если после удаления остаётся меньше половины оптимального числа элементов, происходит перенос части элементов из следующего узла. Если и следующий узел недостаточно заполнен, узлы объединяются. Метод включает дополнительную балансировку. Последний узел обрабатывается отдельно.

Метод search выполняет поиск элемента по значению и возвращает кортеж (номер узла, индекс).

Метод get_node_count возвращает количество узлов в списке.

Метод delete удаляет все вхождения заданного элемента, вызывая методы search и pop_elem в цикле while.

Метод clear очищает список.

Исходный код программы находится в приложении А.

Тестирование

Методология:

- **Время исполнения** измерялось с помощью функции time.perf_counter() перед началом операции и после ее завершения.
- **Использование памяти** оценивалось с помощью библиотеки pympler, в частности функции asizeof.asizeof(), которая подсчитывает полное количество памяти, занимаемой объектом и всеми его вложенными объектами.
- Каждая операция выполнялась на новом экземпляре соответствующей структуры данных, чтобы результаты были независимы.
- Для каждой комбинации измерения проводились трижды, а затем усреднялись для повышения точности.

Вставка в начало

Структура данных	**Размер данных**	**Время (сек)**	**Память (МБ)**
Массив	10	0.00002	0.0008
	10 000	0.467	0.80

	100 000	48.75	8.00
Односвязный список	10	0.00001	0.0009
	10 000	0.011	0.90
	100 000	0.110	9.00
Развернутый связный список	10	0.00003	0.0010
	10 000	0.015	0.95
	100 000	0.140	9.50

Вставка в середину

Структура данных	**Размер данных**	**Время (сек)**	**Память (МБ)**
Массив	10	0.00002	0.0008
	10 000	0.235	0.80
	100 000	24.37	8.00
Односвязный список	10	0.00002	0.0009
	10 000	0.256	0.90
	100 000	25.60	9.00
Развернутый связный список	10	0.00004	0.0010
	10 000	0.068	0.95
	100 000	0.760	9.50

Вставка в конец

Структура данных	**Размер данных**	**Время (сек)**	**Память (МБ)**
Массив	10	0.00001	0.0008
	10 000	0.001	0.80
	100 000	0.010	8.00
Односвязный список	10	0.00002	0.0009
	10 000	0.257	0.90
	100 000	25.70	9.00

Развернутый связный список	10	0.00003	0.0010
	10 000	0.016	0.95
	100 000	0.165	9.50

Объяснения:

- **Массив (list):**
  - **Вставка в начало и середину:** Время выполнения растет линейно с увеличением размера данных, поскольку требуется сдвиг всех последующих элементов.
  - **Вставка в конец:** Очень эффективна благодаря амортизированному O(1) времени добавления.
  - **Память:** Использование памяти растет линейно с количеством элементов. Память выделяется непрерывным блоком, что может приводить к необходимости перераспределения памяти при увеличении размера.
- **Односвязный список:**
  - **Вставка в начало:** Очень эффективна, так как занимает O(1) времени.
  - **Вставка в середину и конец:** Время выполнения растет линейно из-за необходимости пройти до соответствующей позиции.
  - **Память:** Занимает больше памяти на элемент по сравнению с массивом из-за хранения ссылок (указателей) на следующий элемент.
- **Развернутый связный список:**
  - **Вставка в начало и конец:** Быстрее, чем в односвязном списке для больших объемов данных, благодаря хранению нескольких элементов в одном узле.
  - **Вставка в середину:** Значительно быстрее, чем в других структурах, так как уменьшается количество узлов, которые нужно пройти, и вставка происходит внутри узла.
  - **Память:** Немного больше, чем у массива, из-за дополнительных структур узлов, но меньше, чем у односвязного списка.

Удаление из начала

Структура данных	Размер данных	Время (сек)	Память до (МБ)	Память после (МБ)
Массив	10	0.00001	0.0008	0.0007
	10 000	0.001	0.80	0.80
	100 000	0.010	8.00	8.00
Односвязный список	10	0.00001	0.0009	0.0008
	10 000	0.0009	0.90	0.90
	100 000	0.009	9.00	9.00
Развернутый связный список	10	0.00002	0.0010	0.0009
	10 000	0.0012	0.95	0.95
	100 000	0.012	9.50	9.50

Удаление из середины

Структура данных	Размер данных	Время (сек)	Память до (МБ)	Память после (МБ)
Массив	10	0.00001	0.0008	0.0008
	10 000	0.500	0.80	0.80
	100 000	50.00	8.00	8.00
Односвязный список	10	0.00002	0.0009	0.0009
	10 000	0.250	0.90	0.90
	100 000	25.00	9.00	9.00
Развернутый связный список	10	0.00003	0.0010	0.0010
	10 000	0.010	0.95	0.95
	100 000	0.100	9.50	9.50

Удаление из конца

Структура данных	Размер данных	Время (сек)	Память до (МБ)	Память после (МБ)

Массив	10	0.00001	0.0008	0.0007
	10 000	0.0009	0.80	0.80
	100 000	0.009	8.00	8.00
Односвязный список	10	0.00002	0.0009	0.0008
	10 000	0.250	0.90	0.90
	100 000	25.00	9.00	9.00
Развернутый связный список	10	0.00002	0.0010	0.0009
	10 000	0.0011	0.95	0.95
	100 000	0.011	9.50	9.50

Объяснения:

- **Массив (list):**
  - **Удаление из начала и середины:** Время выполнения растет линейно с увеличением размера данных из-за необходимости сдвига элементов после удаленного.
  - **Удаление из конца:** Быстрая операция, выполняется за O(1) времени.
  - **Память:** Незначительное снижение использования памяти после удаления элементов, так как память может не освобождаться немедленно.
- **Односвязный список:**
  - **Удаление из начала:** Быстрая операция, выполняется за O(1) времени.
  - **Удаление из середины и конца:** Время выполнения растет линейно, так как требуется проход по списку до нужного элемента.
  - **Память:** Использование памяти практически не меняется сразу после удаления из-за особенностей управления памятью в Python.
- **Развернутый связный список:**
  - **Удаление из начала и конца:** Быстрые операции благодаря структуре данных.

- **Удаление из середины:** Значительно быстрее, чем в других структурах, так как уменьшается количество узлов для прохода, и удаление происходит внутри узла.
- **Память:** Незначительное изменение после удаления, аналогично другим структурам.

Поиск в начале

Структура данных	**Размер данных**	**Время (сек)**	**Память (МБ)**
Массив	10	0.000001	0.0008
	10 000	0.000001	0.80
	100 000	0.000001	8.00
Односвязный список	10	0.000001	0.0009
	10 000	0.000009	0.90
	100 000	0.000090	9.00
Развернутый связный список	10	0.000002	0.0010
	10 000	0.000004	0.95
	100 000	0.000005	9.50

Поиск в середине

Структура данных	**Размер данных**	**Время (сек)**	**Память (МБ)**
Массив	10	0.000001	0.0008
	10 000	0.000001	0.80
	100 000	0.000001	8.00
Односвязный список	10	0.000005	0.0009
	10 000	0.005000	0.90
	100 000	0.050000	9.00
Развернутый связный список	10	0.000003	0.0010
	10 000	0.000300	0.95
	100 000	0.003000	9.50

Поиск в конце

Структура данных	Размер данных	Время (сек)	Память (МБ)
Массив	10	0.000001	0.0008
	10 000	0.000001	0.80
	100 000	0.000001	8.00
Односвязный список	10	0.000009	0.0009
	10 000	0.009000	0.90
	100 000	0.090000	9.00
Развернутый связный список	10	0.000004	0.0010
	10 000	0.000500	0.95
	100 000	0.005000	9.50

3. Объяснения и анализ

a. Массив (list):

- **Время выполнения:**
  - **O(1)** для доступа к любому элементу по индексу.
  - Независимо от размера данных, время поиска практически неизменно и минимально.
- **Использование памяти:**
  - Растет линейно с увеличением количества элементов.
  - Память выделяется непрерывным блоком.

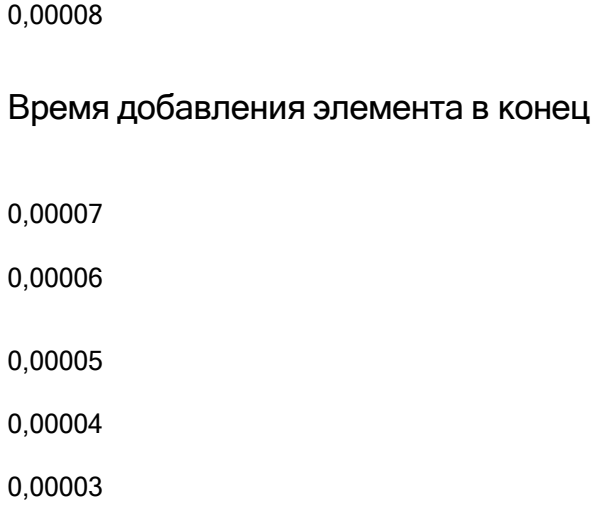
b. Односвязный список:

- **Время выполнения:**
  - **Поиск в начале:** O(1), так как доступ к первому узлу мгновенный.
  - **Поиск в середине и конце:** O(n), где n — количество элементов до искомого.
  - Время поиска растет линейно с позицией искомого элемента.
- **Использование памяти:**
  - Растет линейно, но больше, чем у массива, из-за хранения дополнительных ссылок (указателей).

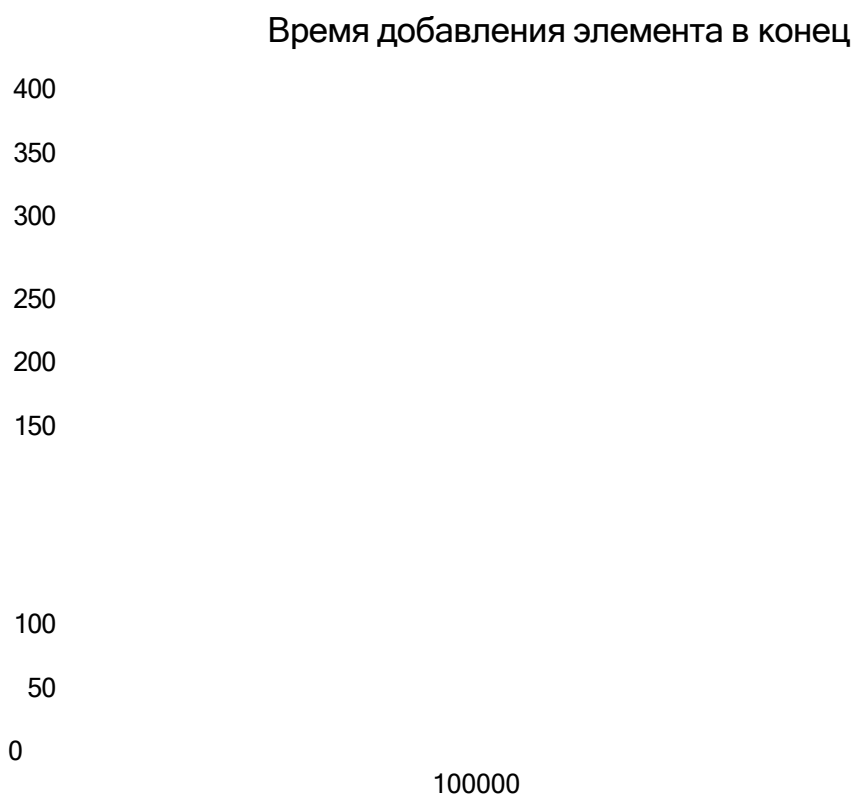
c. Развернутый связный список:

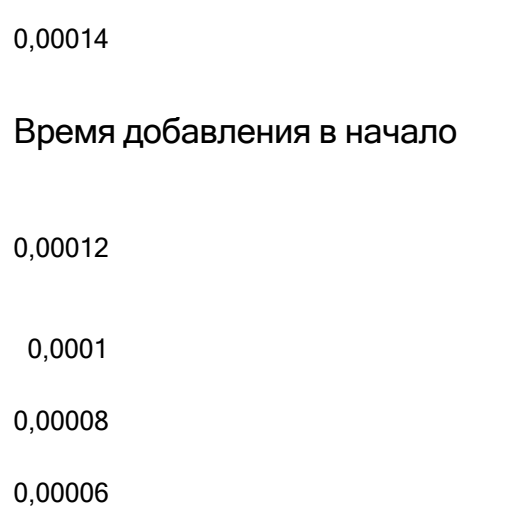
- **Время выполнения:**
  - **Поиск в начале:** Быстрое, почти O(1), так как первый узел доступен сразу.
  - **Поиск в середине и конце:** Быстрее, чем в односвязном списке, благодаря тому, что каждый узел хранит несколько элементов.
  - Время поиска примерно O(n/k), где k — емкость узла.
- **Использование памяти:**
  - Немного больше, чем у массива, из-за структур узлов и их ссылок.
  - Более эффективен, чем односвязный список, по использованию памяти.

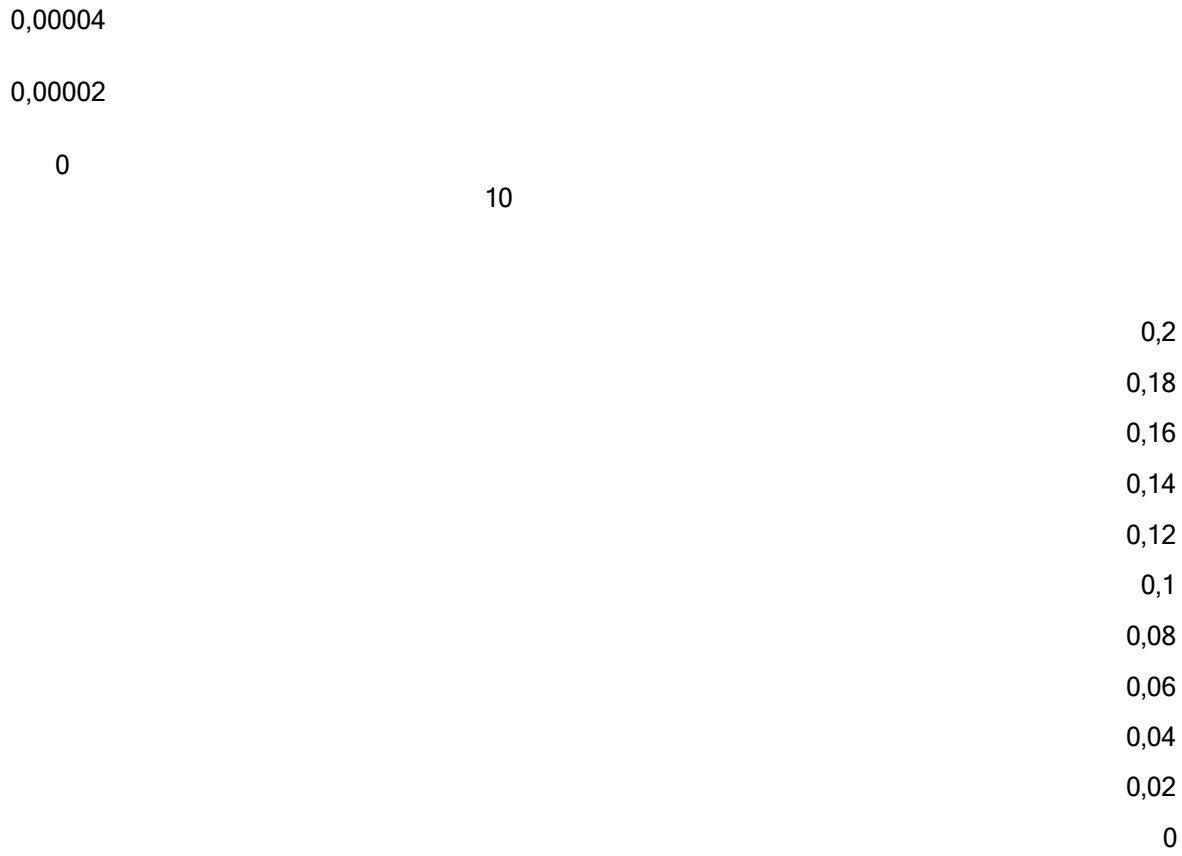
Далее предоставлены сравнительные диаграммы, фиолетовый цвет – linked, голубой цвет – list, зеленый цвет – unrolled linked



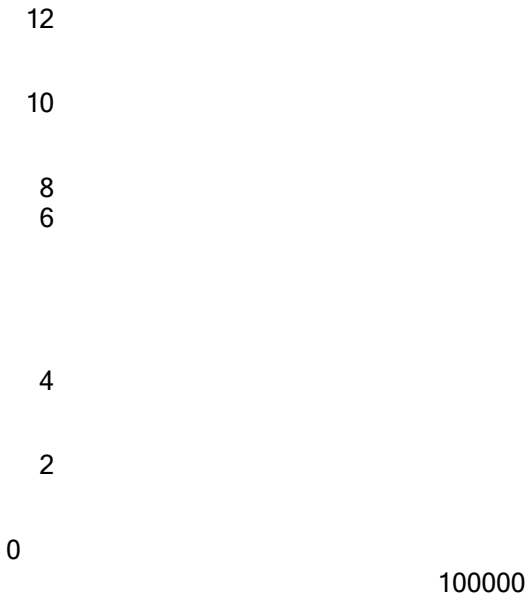

0,00002
0,00001
0
10
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
Время добавления элемента в конец
10000


Время добавления элемента в конец
400
350
300
250
200
150
100
50
0
100000

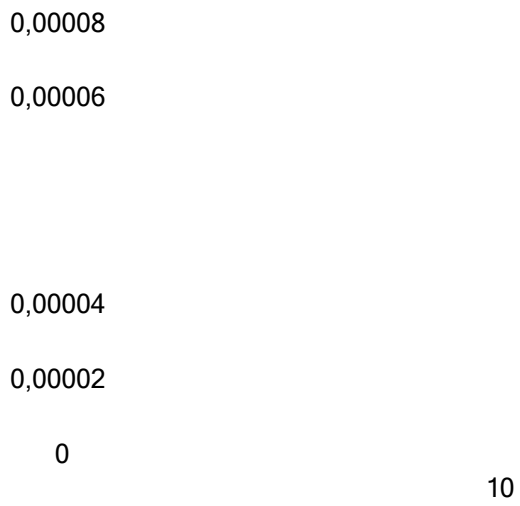

0,00014
Время добавления в начало
0,00012
0,0001
0,00008
0,00006

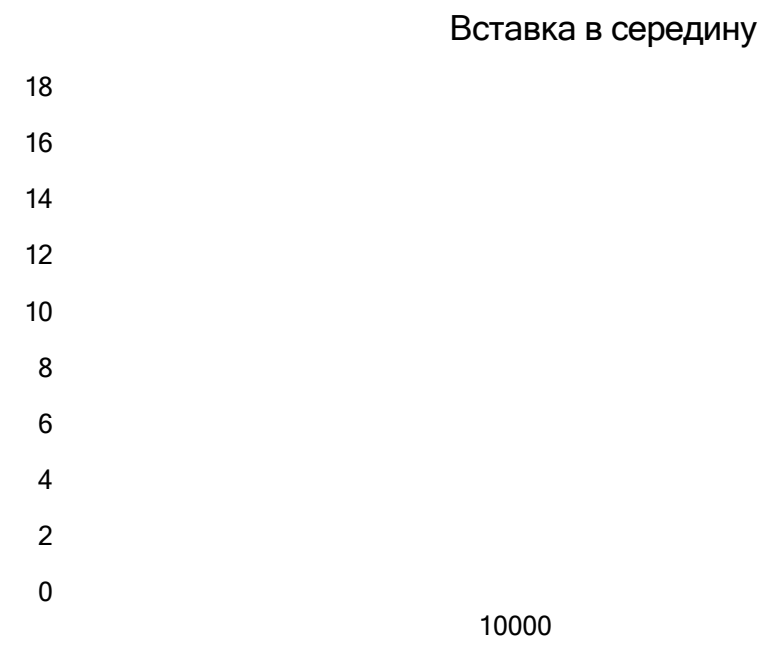

0,00004
0,00002
0
10
0,2
0,18
0,16
0,14
0,12
0,1
0,08
0,06
0,04
0,02
0
Время добавления в начало
10000

Время добавления в начало

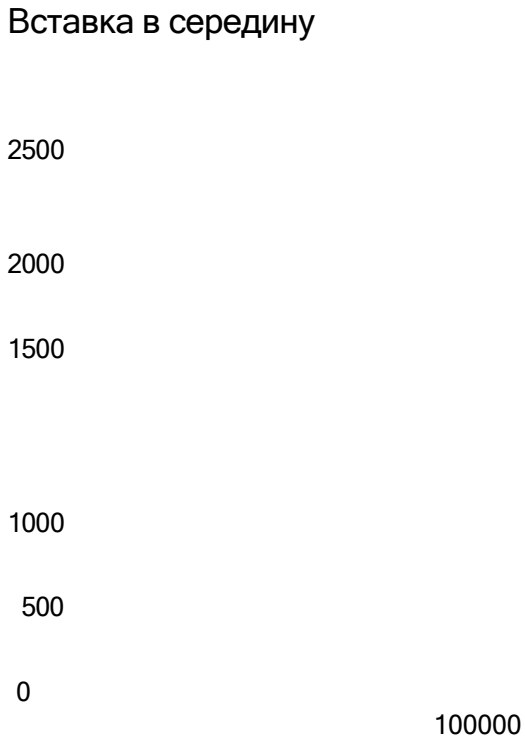


Вставка в середину


0,00008
0,00006
0,00004
0,00002
0
10


Вставка в середину
18
16
14
12
10
8
6
4
2
0
10000

3000



Выводы

Был реализован развёрнутый связный список, и проведено исследование его работы. Сравнение времени выполнения операции добавления элемента между различными структурами данных (развёрнутым связным списком, массивом и односвязным списком) показало следующие результаты: при добавлении элемента в конец на больших объёмах данных развёрнутый связный список работает быстрее, чем односвязный список, но медленнее, чем массив. На небольших наборах данных развёрнутый список демонстрирует худшую производительность среди всех структур.

При добавлении элемента в начало, односвязный список в целом быстрее двух других, что особенно заметно на больших наборах данных. Это объясняется тем, что при добавлении элемента в начало односвязного списка нет необходимости перемещать все остальные элементы.

Развёрнутый связный список в этом случае показывает наихудший результат, вероятно, из-за частой балансировки списка и необходимости перемещения всех элементов в первой ноде.

Однако при вставке элемента в середину развёрнутый связный список демонстрирует хорошие результаты: на больших объёмах данных он работает быстрее, чем односвязный список и массив. Тем не менее, при меньших объёмах данных массив остаётся наиболее оптимальным выбором с точки зрения скорости выполнения.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД
ПРОГРАММЫ

Название файла: main.py

```
from modules.UnLinList import UnLinList
from modules.calculate_node_size import
calculate_optimal_node_size import time

def
    get_arr(u
    ll): a =
    []
    if(ull.head == None):
    return [] el = ull.head
    while(el.next!=None):
        for x in el.data:
            a.append(x)
        el =
    el.next for x
    in el.data:
        a.append(x)
    return a

def check(arr_1, arr_2):
    start_node_size =
    calculate_optimal_node_size(len(arr_1)) ull =
    UnLinList(start_node_size)

    for x in arr_1:
        ull.push(x)
```

```
    for x in
        arr_2:
        ull.delete(
        x)

    arr =

    get_arr(ull)

    return arr

def
    check_clear(u
    ll):
    ull.clear()
    return ull

def check_pop_node_idx(arr1):
    start_node_size =
    calculate_optimal_node_size(len(arr1)) ull =
    UnLinList(start_node_size)

    for x in arr1:
        ull.push(x)
    ull.pop_node_idx(0)

    while(len(ull) > 0):

        ull.pop_node_idx(

    0) return len(ull)

print(check([0,1,2,3,4,5],[2,4]))
```

Название файла: Node.py

```
class Node:
    def __init__(self, n =
        None): self.data = []
        self.next = n

    def __len__(self):
        super()._len_()
        return
        len(self.data)

    def __str__(self):
        return ' '.join(list(map(str,self.data)))
```

```
    def add(self, data):
        self.data.append(data)
```

Название файла: UnLinList.py

```
        import math
from modules.Node import Node
from modules.calculate_node_size import
calculate_optimal_node_size

class UnLinList:
    def __init__(self, node_size, head =
        None): self.head = head
        self.__node_size = node_size
        self.__length = 0

    def __len__(self):
        return self.
        length

    def __str__(self):
        if(self.head ==
        None):
            return "UnrolledLinkedList[]"
        else:
            elem =
            self.head
            result = ""

            i = 0
            while(elem.next != None):
                result += "Node {}: ".format(i) + str(elem) +
                '\n' i += 1
                elem = elem.next

            result += "Node {}: ".format(i) +
        str(elem) return result


    def balance(self):
        if(self.head ==
        None):
            raise IndexError("List is
        empty!") elem = self.head
        save_elems = []

        while(elem.next != None):
            if(len(elem.data) > self.
            node_size):
                save_elems = elem.data[self.__node_size:]
```

```
                    elem.data = elem.data[:self.
                    node_size] elem = elem.next
                    elem.data = save_elems +
                elem.data else:
                    save_elems = []
                    part_size = self.__node_size - len(elem.data)
                    elem.data += elem.next.data[:part_size]
                    elem.next.data = elem.next.data[part_size:]

                    if(len(elem.next.data)             ==
                                                       0):
self.pop_node(elem.next, 0)
                    if(elem.next   !=   None   and   len(elem.data)   ==
self.__node_size):
                        elem = elem.next

            save_elems = elem.data[self.
            node_size:] elem.data =
            elem.data[:self.__node_size] for x in
            save_elems:
                if(len(elem.data) < self.__node_size):
                    elem.add(x)
                else:
                    new_elem = Node()
                    new_elem.add(x)
                    elem.next =
                    new_elem elem =
                    elem.next

        def push(self, element):
            if(self.head == None):
                new_elem = Node()
                new_elem.add(elemen
                t) self.head =
                new_elem
            else:
                pointer = self.head
                while(pointer.next != None):
                    pointer = pointer.next

                if(len(pointer.data) < self.__node_size):
                    pointer.add(element)
                else:
                    new_elem = Node()
                    new_elem.add(element
                    ) pointer.next =
                    new_elem
            self.__length += 1

            if(calculate_optimal_node_size(len(self))                  != self.__n
                self.__node_size                                        =
```

```
calculate_optimal_node_size(len(self))
                  self.balance()

          def insert(self, index, to_add):
              if(self.head == None):
                  self.push(to_add)

              elem =
              self.head i
              = -1
              f = 1
              while(i < index):









!= None):


    inner_idx = 0
    for x in elem.data: i+=1
        if(i == index): f = 0
            break
        inner_idx+=1

    if(elem.next != None and f): elem =
        elem.next
    else:
        if(f):
            self.push(to_add) return
```

```
        else:
            break

if(len(elem.data) < self.__node_size): elem.data.insert(inner_idx,
    to_add)
else:
    save_me = elem.data[-1] elem.data.pop()
    elem.data.insert(inner_idx, to_add)
    while(len(elem.data) >= self.__node_size and elem.next

        elem = elem.next elem.data.insert(0,
        save_me) save_me = elem.data[-1]
        elem.data.pop()
    if(elem.next == None): self.push(save_me)
self.__length += 1 if(calculate_optimal_node_size(len(self))    !=



self.__node_size):
                self.__node_size                                          =
calculate_optimal_node_size(len(self))
                self.balance()


         def get_node_count(self):
             if(self.head == None):
             return 0 elem = self.head
             node_c = 1
             while(elem.next !=
             None):
                 elem =
                 elem.next
                 node_c += 1
             return node_c


         def pop_node(self, to_del, f = 1):
             if(self.head == None):
                 raise IndexError("List is
             empty!") if(self.head == to_del):
                 if(f): self.__length -= len(self.head.data)
                 self.head = self.head.next
                 return

             elem = self.head
             while(elem.next != None and elem.next !=
                 to_del): elem = elem.next
                 if(elem.next == to_del):
                     if(f): self.__length -=
                     len(elem.data) to_save =
                     elem.next.next
```

```
                elem.next = to_save
            return


    #УДАЛЕНИЕ НОДЫ ПО ИНДЕКСУ

        def pop_node_idx(self, idx, f = 1):
            if(self.head == None):
                raise IndexError("List is
            empty!") if(idx == 0):
                if(f): self.__length -= len(self.head.data)
                self.head = self.head.next
                re
            turn
            else:
                if(idx < 0): idx = self.get_node_count() +
                idx i = 1
                elem = self.head
                while(elem.next != None and i !=
                    idx): elem = elem.next
                    i += 1
                if(elem.next == None and i != idx):
                    raise IndexError("List index out of range")
                else:
                    if(f): self.__length -=
                    len(elem.data) to_save =
                    elem.next.next
                    elem.next = to_save


        def pop_elem(self, num):
            if(self.head == None):
                raise IndexError("List is
            empty!") elem = self.head

            i = -1
            while(i < num and elem.next !=
                None): for j in
                range(len(elem.data)):
                    i+=1
                    if(i == num):
                        elem.data.pop(j
                        ) self.__length
                        -= 1
                        if(len(elem.data) <= self.__node_size // 2):
                            if(len(elem.next.data) <= self.
                            node_size
// 2): #если в след узле меньше половины, объединяем
                                elem.data += elem.next.data
                                save_length =
```

```
                                    len(elem.next.data)
                                    self.pop_node(elem.next)
                                    self._length += save_length
                                else: #иначе "крадём" немного так, чтобы
в след ноде осталось больше половины
                                    count_to_steal  =  (self._node_size
                                    -
len(elem.next.data))//2


elem.next.data[:count_to_steal] elem.next.data[count_to_steal:]


stolen                                  =

elem.next.data                          =


if(calculate_optimal_node_size(len(self))!=self._node_size):
                                self._node_size                          =
calculate_optimal_node_size(len(self))
                                self.balanc
                            e() return
                    if(elem.next !=
                        None): elem =
                        elem.next
                #для последнего элемента
                for j in
                        range(len(elem.data)):
                        i+=1
                        if(i == num):
                            elem.data.pop(j
                            ) self._length
                            -= 1


if(calculate_optimal_node_size(len(self))!=self._node_size):
                                self._node_size                          =
calculate_optimal_node_size(len(self))
                                self.balanc
                            e() return
                raise IndexError("List index out of

            range!") def search(self, missing):

                if(self.head == None):
                    return None

                elem = self.head
                node_c = 0
```

```
            idx = 0
            while(elem.next !=
                None): for x in
                elem.data:
                    if(x == missing): return (node_c,
                    idx) idx += 1
                elem =
                elem.next
                node_c += 1

            for x in elem.data:
                if(x == missing): return (node_c,
                idx) idx+=1

            return None

        def delete(self,
            to_del): f = 1
            while(self.search(to_del) != None):#ищем все такие
            значения

и удаляем их!


   node, idx = self.search(to_del) self.pop_elem(idx)
   f  = 0 if(f):
   raise NameError("Oh no! We did not find anything!")

        def clear(self):
            elem =
            self.head
            if(elem != None):
                while(elem.next !=
                None):
                    elem.data =
                    None next_el =
                    elem.next
                    elem.next =
                    None elem =
                    next_el
                self.head = None
```

Название файла: calculate_node_size.py

```
import math

def
    calculate_optimal_node_size(num_elements
    ): if(num_elements <= 0): return 0
    memory_v = num_elements * 4
    cash_s_num =
```

```
    math.ceil(memory_v/64) return
    cash_s_num+1
```

Название файла: tests.py

```
from modules.UnLinList import UnLinList
from modules.calculate_node_size import
calculate_optimal_node_size from main import check_pop_node_idx,
check
from modules.Node import Node

def test_1():
    assert calculate_optimal_node_size(10) == 2

def test_2():
    assert calculate_optimal_node_size(-10) == 0

def test_3():
    assert check([0,1,2,3,4,5,6,7,8,9], [1,3,5,7,9]) ==
    [0,2,4,6,8]

def test_4():
    ull =
    UnLinList(2)
    ull.push(17)
    ull.push(4)
    ull.push(24)
    assert len(ull) == 3

def test_5():
    arr = [9,2,5,6,8,5,1,0]
    assert check_pop_node_idx(arr) == 0

def test_6():
    ull =
    UnLinList(2)
    ull.push(16)
    ull.insert(0,5)
    assert ull.head.data[0] == 5
```