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: Развернутый связный список

Студент гр. 3384 _____ Пьянков М.Ф.

Преподаватель _____ Шестопалов Р.П.

Санкт-Петербург
2024

Цель работы

Изучение и реализация структуры данных: расширенный связный список. Тестирование и исследование возможностей структуры. Сравнение времени работы различных методов с методами в других структурах данных.

Задание

Вам необходимо реализовать развернутый связный список который представляет собой связный список, в котором каждый узел содержит массив элементов и указатель на следующий узел.

Выполнение работы

Каждый узел структуры данных представлен классом Node, со следующими методами:

Метод __init__(node_size = 1, auto_balance_flag = True) - создание узла с заданной длиной массива если необходимо, также есть возможность создавать список с жёсткой балансировкой, если указать auto_balance_flag = False.

Метод is_fool() - возвращает true или false в зависимости от того, заполнен ли массив узла до размера node_size.

Метод __str__() - возвращает вывод всех элементов в виде строки или посредством функции print().

Класс Extended_Linked_List:

Метод __init__(node_size = 1) – создаёт экземпляр нашей структуры данных с заданным вручную или по умолчанию размером массива в каждом узле.

Метод insert(index, value) – осуществляет вставку в структуру данных элемента value по индексу index, при этом все элементы, находящиеся после элемента с необходимым индексом включительно циклически сдвигаются в право на один элемент. Если индекс имеет недопустимое значение, то метод вернёт значение None.

Метод push_back(value) – добавляет элемент в конец структуры, используя указатель __tail__.

Метод push_front(value) – добавляет элемент в начало структуры, используя указатель __head__.

Метод pop(index) – удаляет элемент с индексом index и циклически сдвигает все элементы до элемента с индексом index влево на один элемент. В случае недопустимого значения индекса метод вернёт значение None.

Метод pop_back() - удаляет последний элемент структуры, используя указатель на конец - __tail__.

Метод pop_front() - удаляет первый элемент структуры, используя указатель __head__.

Метод pop_by_value_first(value) - удаляет первый элемент структуры с заданным значением, посредством вызова метода search_by_value_first(value).

Метод pop_by_value_first(value) - удаляет последний элемент структуры с заданным значением, посредством вызова метода search_by_value_last(value).

Метод pop_by_value_all(value) - удаляет все элементы структуры с заданным значением, посредством вызова метода search_by_value_first(value) (пока он не выдаст None).

Метод calculate_optimal_node_size() – высчитывает оптимальное значение размера узла (внутренний метод).

Метод balance(new_node_size) — метод осуществляет балансировку структуры данных, с помощью динамического алгоритма (сдвиг).

Метод search(index) – возвращает значение элемента с необходимым индексом.

Методы get_first() и get_last() - возвращают значение первого и последнего элемента соответственно, используя указатели на начало __head__ и конец __tail__ структуры данных.

Методы search(index) и search_by_value_first(value), search_by_value_last(value), search_by_value_all(value) – циклически перебирают все узлы и элементы и ищут элементы с соответствующими индексами или значениями.

Метод __str__ - вызывая метод __str__ для каждого узла Node, формирует вывод всех элементов, содержащихся в нашей структуре.

Метод replication_by_func(func, buffer) и replication_by_index(index, buffer) — заполнение буфера buffer копией ущла с нужными характеристиками.

Метод filter(func) – фильтрация СД, удаление неподходящих под условие элементов.

Методы __copy__() и __deepcopy__() - создание копии и глубокой копии объекта соответственно.

Метод __iadd__() - осуществляет слияние двух СД.

Метод __del__() - удаление объекта, освобождение памяти.

Метод __make_beauty__() - приведение СД в стандартизированный вид.

Пайплайн работы методов:

Вызов push_back(value) → рассмотрение указателя __tail__ и проверка его заполненности → успех - создание нового узла, вставка в его конец элемента и переназначение __tail__ / неудача — вставка в конец __tail__ элемента → балансировка, если это необходимо.

Вызов insert(index, value) → переход к элементу с необходимым индексом с помощью цикла, вставка → циклический сдвиг всех элементов после вставленного влево на один → балансировка, если это необходимо.

Вызов push_front(value) → рассмотрение узказателя __head__ и проверка его заполненности → успех — создание нового узла и переназначение __head__ / неудача → вставка в конец __head__.

Вызов pop_back() → удаление последнего элемента в узле __tail__ → проверка __tail__ на пустоту → успех — переназначение __tail__ и его соседей / неудача → балансировка, если это необходимо → завершение работы метода.

Вызов pop(index) → переход к элементу с индексом index, его удаление → циклический сдвиг всех элементов после удалённого вправо на один → балансировка, если это необходимо.

Вызов pop_front() → вызов pop(0).

Вызов search(index) / search_by_value(value) и другие методы поиска элементов СД → циклических поиск элемента по индексу/значению и возврат его значения/индекса в случае успеха или None в случае неудачи.

Вызов get_first() → обращение к первому узлу __head__ и получение первого элемента его массива.

Вызов get_last() → обращение к последнему узлу __tail__ и получение последнего элемента его массива.

Вызов __str__() или print() → Циклический перебор всех узлов структуры → вызов для каждого узла метода __str__ и добавление полученной строки к итоговому результату → возврат итоговой строки вывода всех элементов структуры данных.

Разработанный программный код см. в приложении А.

Тестирование

Тестирование для созданной структуры реализовано при помощи библиотеки pytest.

Ход и принцип тестирования: сначала проверяется метод структуры данных search (сверяются значения массива и расширенного списка по определённому индексу). После проверяются такие методы как вставка: в конец, начало, середину; удаление: в конце, начале, середине. В конце проверяется вывод структуры данных.

Каждый метод, кроме __str__ и search тестируется следующем образом: создаётся массив случайных чисел, иногда двумерный (индекс, значение), применяется соответствующий метод, и методом search проверяется наличие или отсутствие элемента в СД (в случае удаление создаётся массив уникальных чисел). Для тестирования каждого метода также прилагаются граничные случаи и случаи (к набору данных добавляется отрицательный индекс или индекс превышающий кол-во элементов в СД).

Исследование

В качестве исследуемых объектов были взяты следующие структуры данных: расширенных связный список, связный список, стандартный список в python – list.

Вставка, расширенный связный список			
	push_back	push_in_middle	push_front
10	0	0	0
10000	0	0	0
100000	0	0	0

Таблица 1 — Вставка в расширенный связный список

Вставка, связный список			
	push_back	push_in_middle	push_front
10	0	0	0
10000	0	0.001	0
100000	0	0.002	0

Таблица 2 — Вставка в связный список

Вставка, python list			
	push_back	push_in_middle	push_front
10	0	0	0
10000	0	0	0
100000	0	0	0

Таблица 3 — Вставка в список python

Удаление, расширенный связный список			
	pop_back	pop_in_middle	pop_front
10	0	0	0
10000	0	0	0
100000	0	0	0

Таблица 4 — Удаление из расширенного связного списка

Удаление, связный список

	pop_back	pop_in_middle	pop_front
10	0	0	0
10000	0	0	0
100000	0	0.003	0

Таблица 5 — Удаление из связного списка

Удаление, python list			
	pop_back	pop_in_middle	pop_front
10	0	0	0
10000	0	0	0
100000	0	0	0

Таблица 6 — Удаление из списка python

Поиск, расширенный связный список			
	search_back	search_in_middle	search_front
10	0	0	0
10000	0	0	0
100000	0	0	0

Таблица 7 — Поиск в расширенном связном списке

Поиск, связный список			
	search_back	search_in_middle	search_front
10	0	0	0
10000	0	0	0
100000	0	0.002	0

Таблица 8 — Поиск в связном списке

Поиск, python list			
	search_back	search_in_middle	search_front
10	0	0	0
10000	0	0	0
100000	0	0	0

Таблица 9 — Поиск в списке python

К сожалению на количестве элементов в СД 10, 10000, 100000 скорость выполнения методов настолько велика, что время практически равно 0, вследствие этого рассмотрим время выполнения этих операций на большем количестве элементов в СД и построим соответсвующие графики.

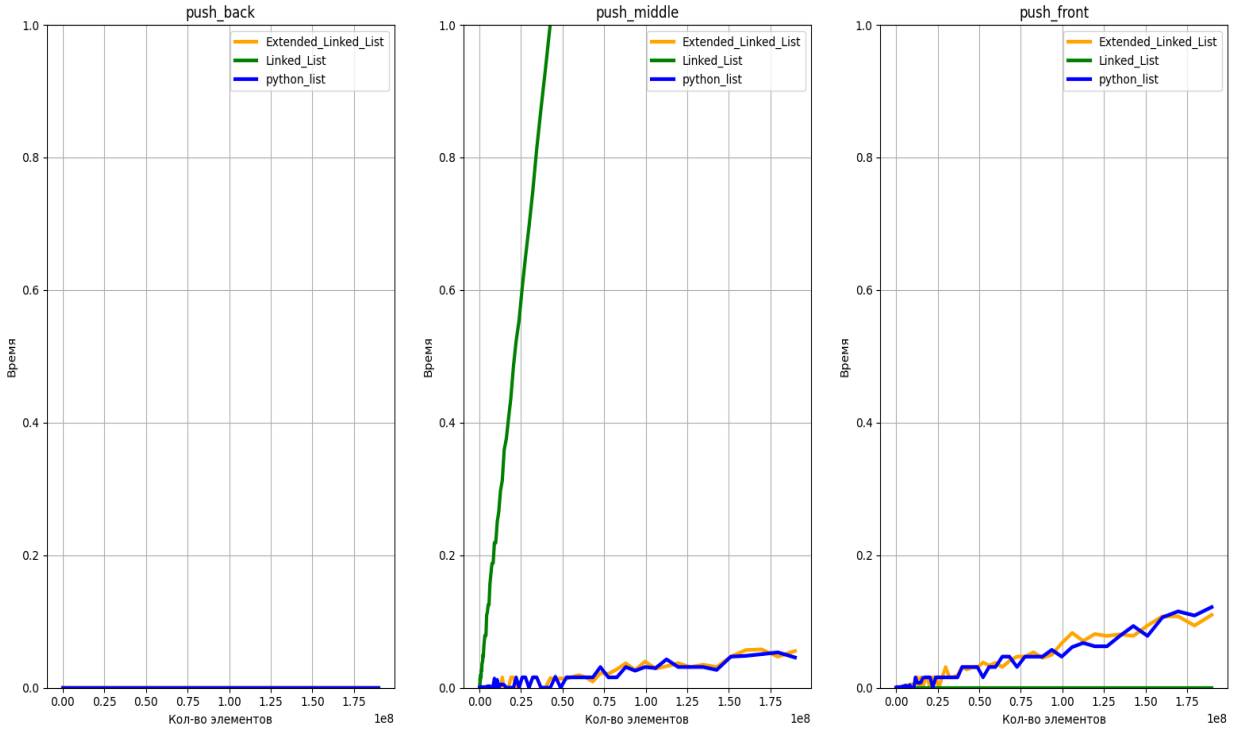

Рисунок 1 — Вставка

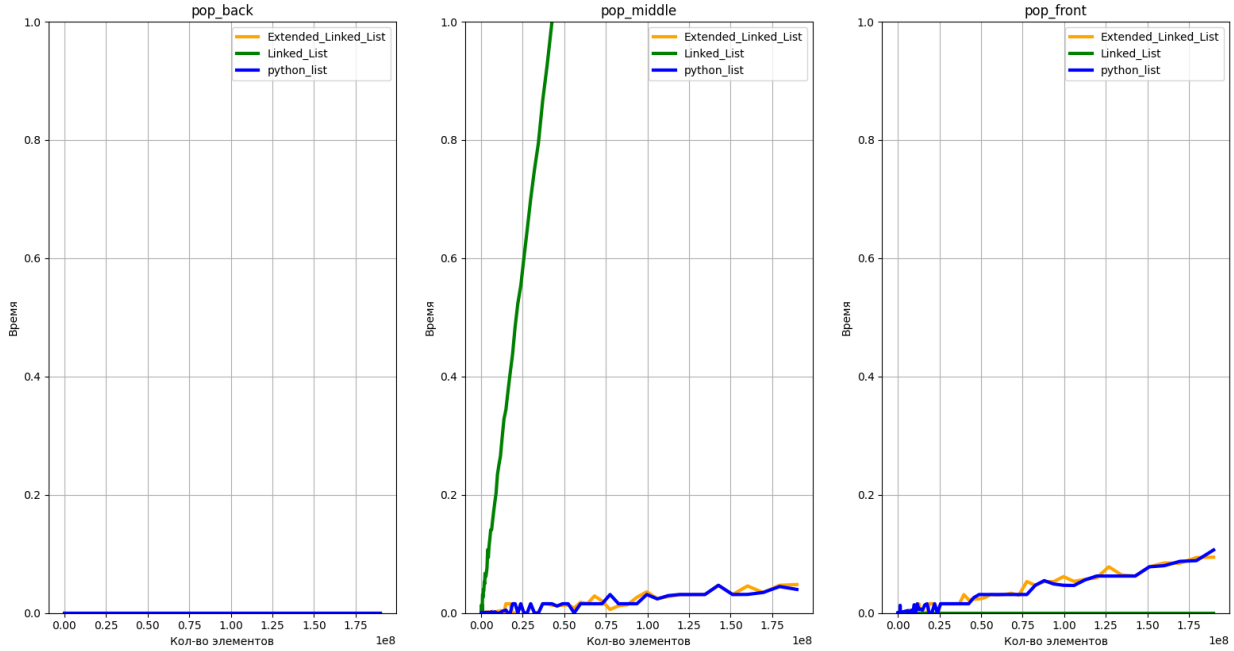

Рисунок 2 — Удаление

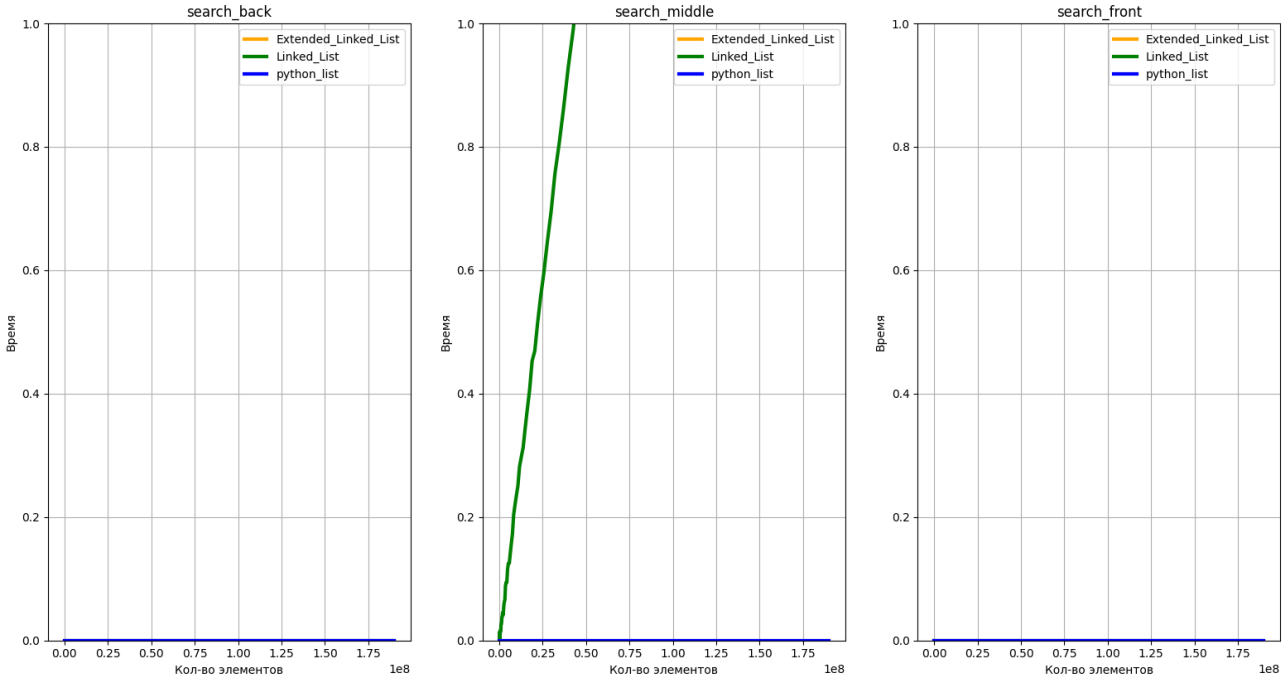

Рисунок 3 — Поиск

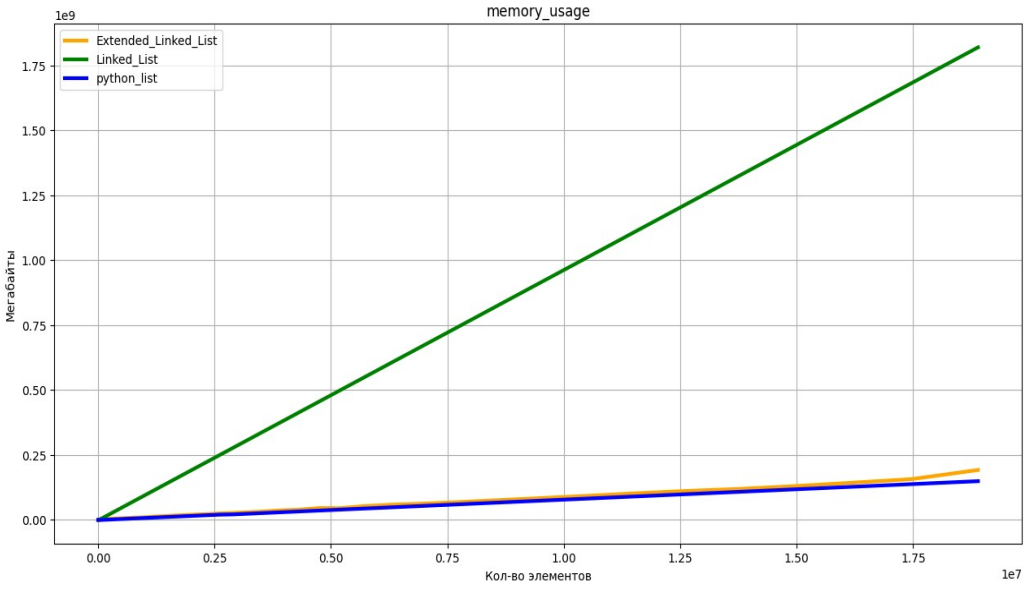

Рисунок 4 — Использование памяти

Графики позволяют нам оценить время работы методов исследуемых объектов.

	Расширенный связный список	Связный список	Python list

push_back	O(1)	O(1)	O(1)
push_in_middle	O(n)	O(n)	O(n)
push_front	O(n)	O(1)	O(n)
pop_back	O(1)	O(1)	O(1)
pop_in_middle	O(n)	O(n)	O(n)
pop_front	O(n)	O(1)	O(n)
search_back	O(1)	O(1)	O(1)
search_in_middle	O(n)	O(n)	O(n)
search_front	O(1)	O(1)	O(1)

Таблица 10 — Сложность методов СД

Примечание сложность методов при действиях в центре СД, различается: связный список справляется хуже всего, так как идёт перебор всех элементов непосредственно, python list справляется хорошо за счёт оптимизации, а расширенный список справляется почти также, как python list за счёт того, что идёт перебор не каждого элемента, а количества узлов, которое позволяет сократить время в десятки раз, к тому же элементами узлов также являются python list-ы. Больше всего памяти потребовалось для работы связного списка, что соответствует реальности, так как помимо численного значения int в каждом узле содержится указатель на следующий и предыдущий узел.

Выводы

Была успешно реализована новая структура данных — расширенный связный список. Также были получены навыки организации тестирования методов и исследования сложности работы алгоритмов. Соответствующие навыки были применены на практике в следствии чего полученные результаты в целом отражают действительность. Проведённое исследование показало, что расширенный список схож с СД python list, и выигрывает по всем методам, кроме удаления и вставки в начало СД простой связный список.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main_lb1.py

```
class Node:
    def __init__(self, node_size):
        self.node_size = node_size
        self.data = list()
        self.next = None
        self.prev = None

    def is_fool(self):
        return len(self.data) == self.node_size

    def __str__(self):
        return ' '.join([str(i) for i in self.data])

    def __del__(self):
        self.data = None
        self.node_size = None
        self.next = None
        self.prev = None
        del self.data
        del self.node_size


class Extended_Linked_List:
    def __init__(self, node_size = 1, auto_balance_flag = True):
        self.__head__ = Node(node_size)
        self.__tail__ = self.__head__
        self.length = 0
        self.node_size = node_size
        self.node_count = 1
        self.auto_balance_flag = auto_balance_flag

    def __sizeof__(self):
        return self.length * int().__sizeof__()

    def fullness(self):
          return f'{self.length / (self.node_count * self.node_size) *
100 :.3f}%'

    def push_back(self, value):
        if not(self.__tail__.is_fool()):
            self.__tail__.data.append(value)
        else:
            self.node_count += 1
            new_node = Node(self.node_size)
                new_node.data += self.__tail__.data[(self.node_size +
1) // 2:]
             self.__tail__.data = self.__tail__.data[:(self.node_size +
1) // 2]
            new_node.data.append(value)
            new_node.prev = self.__tail__
```

```
            self.__tail__.next = new_node
            self.__tail__ = new_node
        self.length += 1
                                    if    self.auto_balance_flag    and
self.calculate_optimal_node_size() != self.node_size:
            self.balance()

    def push_front(self, value):
        if not(self.__head__.is_fool()):
            self.__head__.data.insert(0, value)
        else:
            self.node_count += 1
            new_node = Node(self.node_size)
            new_node.data.append(value)
                new_node.data += self.__head__.data[:(self.node_size +
1) // 2]
              self.__head__.data = self.__head__.data[(self.node_size +
1) // 2:]
            new_node.next = self.__head__
            self.__head__.prev = new_node
            self.__head__ = new_node
        self.length += 1
                                    if    self.auto_balance_flag    and
self.calculate_optimal_node_size() != self.node_size:
            self.balance()

    def insert(self, index, value):
        if index < 0 or index > self.length:
            return None
        elif index == 0:
            self.push_front(value)
        elif index == self.length:
            self.push_back(value)
        else:
            current = self.__head__
            while(current != None and index >= 0):
                if index < len(current.data):
                    if not(current.is_fool()):
                        current.data.insert(index, value)
                    else:
                        current.data.insert(index, value)
                        self.node_count += 1
                        new_node = Node(self.node_size)
                         new_node.data = current.data[(self.node_size +
1) // 2:]
                         current.data = current.data[:(self.node_size +
1) // 2]
                        new_node.prev = current
                        new_node.next = current.next
                        current.next = new_node
                        if current == self.__tail__:
                            self.__tail__ = current
                    self.length += 1
                    break
                index -= len(current.data)
                current = current.next
```

```
                                    if    self.auto_balance_flag    and
self.calculate_optimal_node_size() != self.node_size:
            self.balance()

    def pop_back(self):
        if self.length != 0:
            self.__tail__.data = self.__tail__.data[:-1]
            self.length -= 1
              if  len(self.__tail__.data)  ==  0  and  self.__tail__  !=
self.__head__:
            self.__tail__.prev.next = None
            tail_node  = self.__tail__
            del self.__tail__
            self.__tail__ = tail_node.prev
            self.node_count -= 1
        else:
            return None

                                    if    self.auto_balance_flag    and
self.calculate_optimal_node_size() != self.node_size:
            self.balance()

    def pop_front(self):
        if self.length != 0:
            self.__head__.data = self.__head__.data[1:]
            self.length -= 1
              if  len(self.__head__.data)  ==  0  and  self.__head__  !=
self.__tail__:
            self.__head__.next.prev = None
            head_node = self.__head__
            del self.__head__
            self.__head__ = head_node.next
            self.node_count -= 1
        else:
            return None

                                    if    self.auto_balance_flag    and
self.calculate_optimal_node_size() != self.node_size:
            self.balance()
    def pop(self, index):
        if index == None or index < 0 or index >= self.length:
            return None
        elif index == 0:
            self.pop_front()
        elif index == self.length - 1:
            self.pop_back()
        else:
            current = self.__head__
            while (current != None and index >= 0):
                if index < len(current.data):
                    current.data.pop(index)
                      if current != self.__tail__ and len(current.data)
< (self.node_size + 1) // 2:
                        length = len(current.data)
                                    current.data  +=  current.next.data[:
(self.node_size + 1) // 2 - length]
```

```
                                                    current.next.data  =
current.next.data[(self.node_size + 1) // 2 - length:]
                            if len(current.next.data) < (self.node_size +
1) // 2:
                                current.data += current.next.data
                                next_node = current.next
                                if current.next != self.__tail__:
                                    current.next.next.prev = current
                                    del current.next
                                    current.next = next_node.next
                                    self.node_count -= 1
                                else:
                                    current.next = None
                                    del self.__tail__
                                    self.__tail__ = current
                                    self.node_count -= 1
                            elif current == self.__tail__:
                                if len(current.data) == 0:
                                    if self.__head__ != self.__tail__:
                                        self.__tail__.prev.next = None
                                        tail_node = self.__tail__
                                        del self.__tail__
                                        self.__tail__ = tail_node.prev
                                        self.node_count -= 1
                                    else:
                                        return None
                                self.length -= 1
                                break
                        index -= len(current.data)
                        current = current.next
                                        if    self.auto_balance_flag    and
self.calculate_optimal_node_size() != self.node_size:
            self.balance()

    def pop_by_value_first(self, value):
        self.pop(self.search_by_value_first(value))

    def pop_by_value_last(self, value):
        self.pop(self.search_by_value_last(value))

    def pop_by_value_all(self, value):
        index = self.search_by_value_first(value)
        while(index != None):
            self.pop(index)
            index = self.search_by_value_first(value)

    def search(self, index):
        if index < 0 or index >= self.length:
            return None
        elif index == 0:
            return self.get_first()
        elif index == self.length - 1:
            return self.get_last()
        current = self.__head__
        while (current != None and index >= 0):
            if index < len(current.data):
```

```
                return current.data[index]
            index -= len(current.data)
            current = current.next

    def get_last(self):
        if self.__tail__ != None and len(self.__tail__.data) != 0:
            return self.__tail__.data[-1]

    def get_first(self):
        if self.__head__ != None and len(self.__head__.data) != 0:
            return self.__head__.data[0]
    def search_by_value_first(self, value):
        current = self.__head__
        index = 0
        while (current != None):
            for i in range(len(current.data)):
                if current.data[i] == value:
                    return i + index
            index += len(current.data)
            current = current.next
        return None

    def search_by_value_last(self, value):
        current = self.__head__
        index = 0
        res = -1
        while (current != None):
            for i in range(len(current.data)):
                if current.data[i] == value:
                    res = i + index
            index += len(current.data)
            current = current.next
        if res != -1:
            return res
        return None

    def search_by_value_all(self, value):
        current = self.__head__
        indexes = list()
        index = 0
        while (current != None):
            for i in range(len(current.data)):
                if current.data[i] == value:
                    indexes.append(i + index)
            index += len(current.data)
            current = current.next
        if len(indexes) != 0:
            return indexes
        return None

    def replication_by_func(self, func, buffer):
        current = self.__head__
        while(current != None):
            flag = True
            for i in range(len(current.data)):
                if not(func(current.data[i])):
```

```
                flag = False
                break
        if flag:
            buffer.append((current,current.__str__()))
        current = current.next

def replication_by_index(self, index, buffer):
    current = self.__head__
    while(current != None):
        if index < len(current.data):
            buffer.append((current,current.__str__()))
            break
        index -= len(current.data)
        current = current.next

def filter(self, func):
    current = self.__head__
    index = 0
    values = list()
    while(current != None):
        for i in range(len(current.data)):
            if not(func(current.data[i])):
                values.append(current.data[i])
        index += len(current.data)
        current = current.next
    if len(values) != 0:
        for i in values:
            self.pop_by_value_first(i)

def calculate_optimal_node_size(self):
    return int(self.length ** 0.5)

def balance(self, node_size = None):
    if node_size == None:
        node_size = self.calculate_optimal_node_size()
    if node_size >= self.length:
        self.node_size = node_size
        self.node_count = 1
        self.__head__.node_size = node_size
        if self.__head__ != self.__tail__:
            current = self.__head__.next
            while(current != None):
                self.__head__.data += current.data
                current_copy = current.next
                if current != self.__tail__:
                    current.prev.next = current.next
                    current.next.prev = current.prev

                else:
                    current.prev.next = None
                del current
                current = current_copy
            self.__tail__ = self.__head__
    elif node_size > self.node_size:
        current = self.__head__
        self.node_size = node_size
```

```
            while(current != self.__tail__ and current != None):
                current.node_size = node_size
                temp = current.next
                while(temp != None):
                    cnt = (node_size + 1) // 2 - len(current.data)
                    if len(temp.data) > cnt:
                        current.data += temp.data[:cnt]
                        temp.data = temp.data[cnt:]
                        break
                    else:
                        self.node_count -= 1
                        current.data += temp.data
                        if temp != self.__tail__:
                            temp.prev.next = temp.next
                            temp.next.prev = temp.prev
                        else:
                            self.__tail__.prev.next = None
                            tail_copy = self.__tail__
                            del self.__tail__
                            self.__tail__ = tail_copy.prev
                            del tail_copy
                        temp = temp.next
                current = current.next
        elif node_size < self.node_size:
            current = self.__head__
            self.node_size = node_size
            while(current != self.__tail__):
                current.node_size = node_size
                if len(current.data) > (node_size + 1) // 2:
                    cnt = len(current.data) - (node_size + 1) // 2
                            current.next.data = current.data[-cnt:] +
current.next.data
                    current.data = current.data[:-cnt]
                current = current.next
            while(len(self.__tail__.data) > (node_size + 1) // 2):
                new_node = Node(node_size)
                cnt = len(self.__tail__.data) - (node_size + 1) // 2
                new_node.data = self.__tail__.data[-cnt:]
                self.__tail__.data = self.__tail__.data[:-cnt]
                new_node.prev = self.__tail__
                self.__tail__.next = new_node
                self.__tail__ = new_node
                self.node_count += 1

    def __iadd__(self, other):
        other.__head__.prev = self.__tail__
        self.__tail__.next = other.__head__
        self.__tail__ = other.__tail__
        self.length += other.length
        self.node_count += other.node_count
                                    if    self.auto_balance_flag    and
self.calculate_optimal_node_size() != self.length:
            self.balance()
        return self

    def __copy__(self):
```

```
                                        extended_linked_list_copy     =
Extended_Linked_List(self.node_size)
        extended_linked_list_copy.length = self.length
        extended_linked_list_copy.__tail__ = self.__tail__
        extended_linked_list_copy.__head__ = self.__head__
        extended_linked_list_copy.node_count = self.node_count
        return extended_linked_list_copy

    def __deepcopy__(self):
                                    extended_linked_list_deep_copy    =
Extended_Linked_List(self.node_size)
        extended_linked_list_deep_copy.length = self.length
        extended_linked_list_deep_copy.node_count = self.node_count
        current = self.__head__
        while(current != None):
            extended_linked_list_deep_copy.__tail__.data = [i for i in
current.data]
            if current.next != None:
                new_node = Node(self.node_size)
                                                    new_node.prev   =
extended_linked_list_deep_copy.__tail__
                        extended_linked_list_deep_copy.__tail__.next =
new_node
                extended_linked_list_deep_copy.__tail__ = new_node
            current = current.next
        return extended_linked_list_deep_copy

    def __del__(self):
        current = self.__head__
        while(current != None):
            temp = current.next
            current.__del__()
            current = temp
        self.length = 0

    def __make_beaty__(self):
        if len(self.__tail__.data) > ((self.node_size + 1) // 2) :
             cnt = len(self.__tail__.data) - ((self.node_size + 1) //
2)
            new_node = Node(self.node_size)
            new_node.data = self.__tail__.data[-cnt:]
            self.__tail__.data = self.__tail__.data[:-cnt]
            new_node.prev = self.__tail__
            self.__tail__.next = new_node
            self.__tail__ = new_node

    def __str__(self):
        if self.length == 0:
            return 'Empty'
        res = ''
        self.__make_beaty__()
        index = 0
        current = self.__head__
        while(current != None):
            res += f'Node {index}: ' + current.__str__() + '\n'
            current = current.next
```

```
            index += 1
        return res[:-1]


def calculate_optimal_node_size(num_elements):
    return((num_elements * 4 + 63) // 64 + 1)

data = list(map(int, input().split()))
A = Extended_Linked_List(auto_balance_flag=False)

A.balance(calculate_optimal_node_size(len(data)) * 2)

for i in data:
    A.push_back(i)

print(A)
```