**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

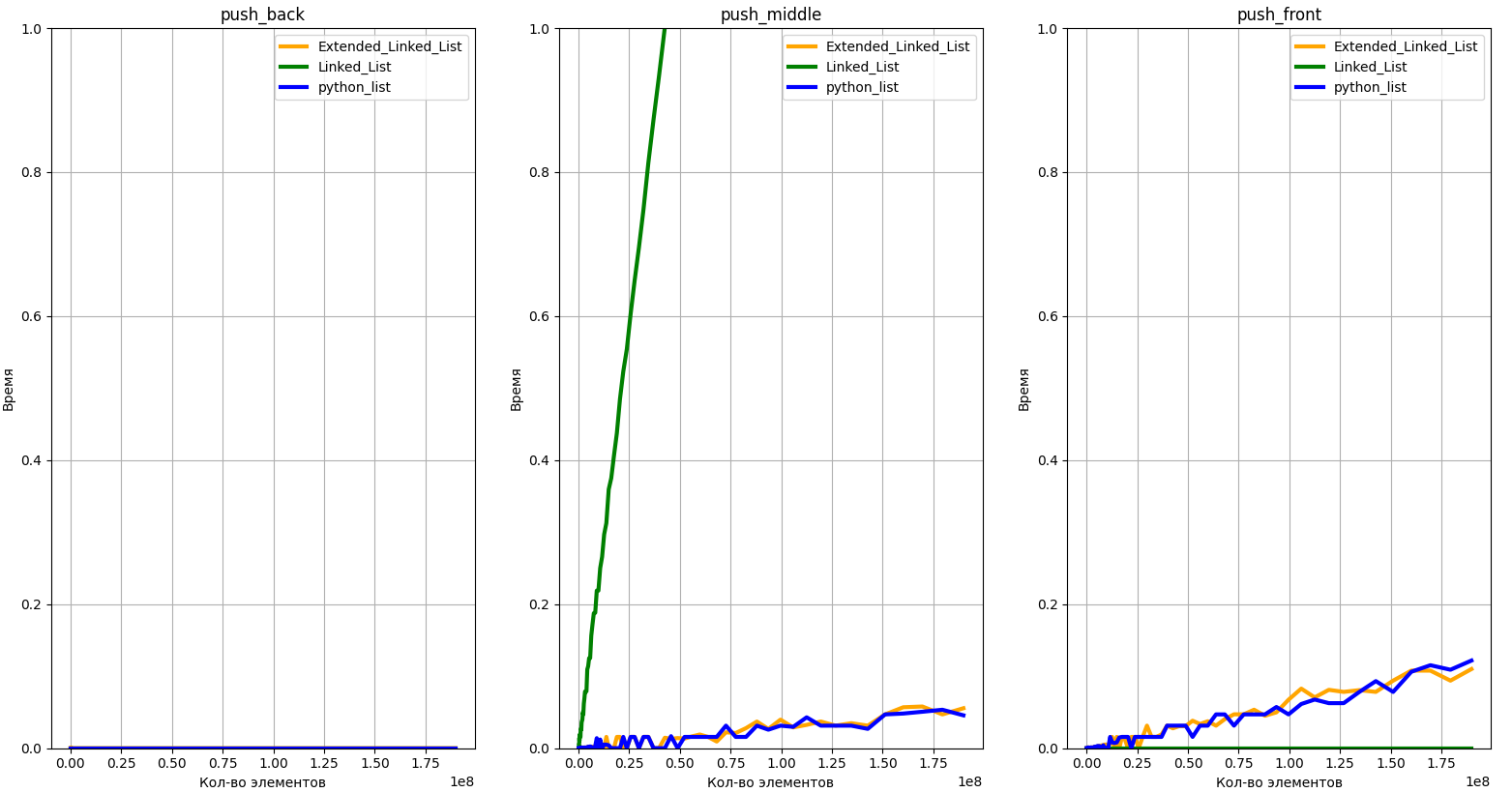
|  |  |  |  |
| --- | --- | --- | --- |
| Поиск, связный список | | | |
|  | search\_back | search\_in\_middle | search\_front |
| 10 | 0 | 0 | 0 |
| 10000 | 0 | 0 | 0 |
| 100000 | 0 | 0.002 | 0 |

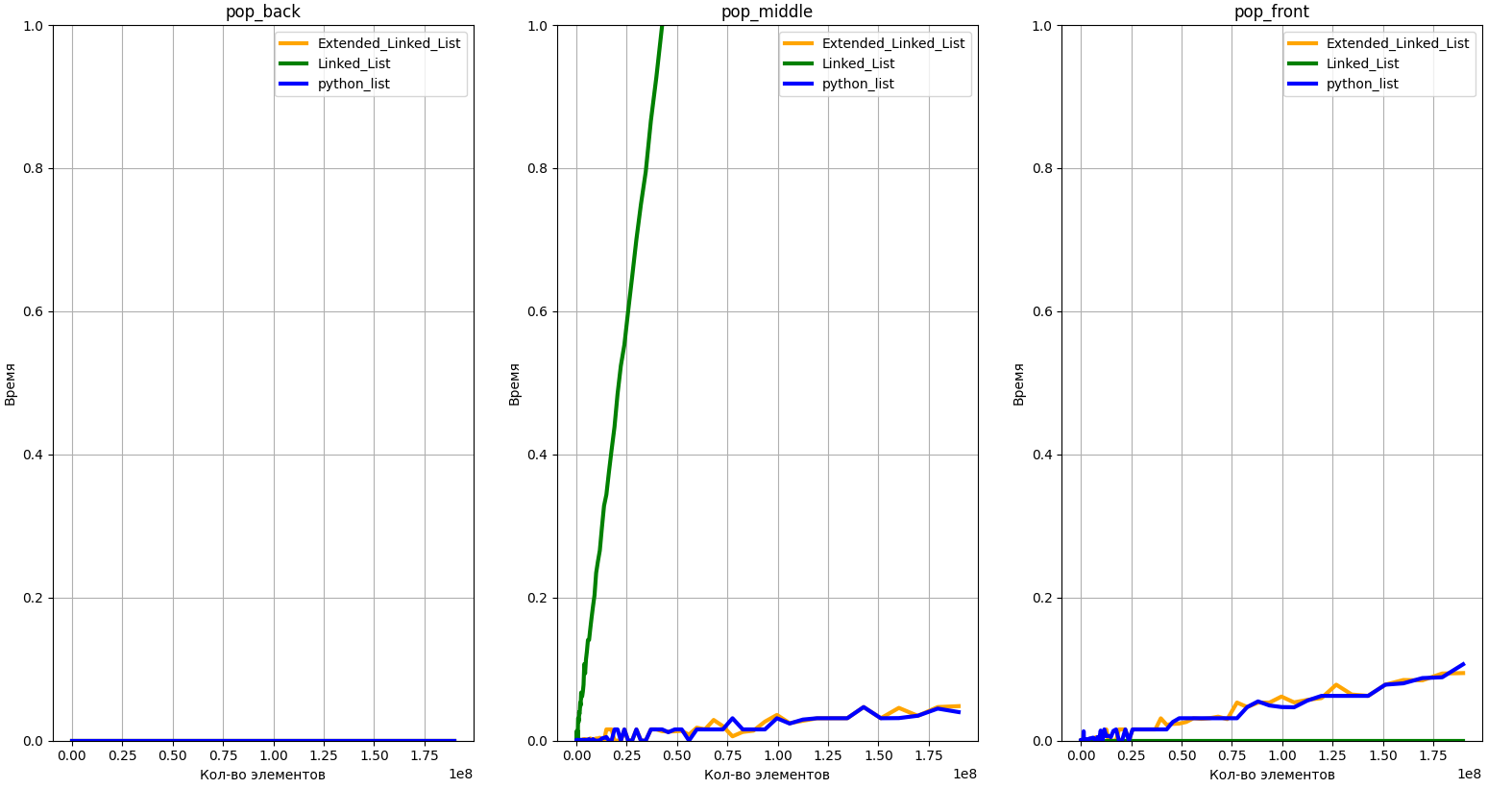
Таблица 8 — Поиск в связном списке

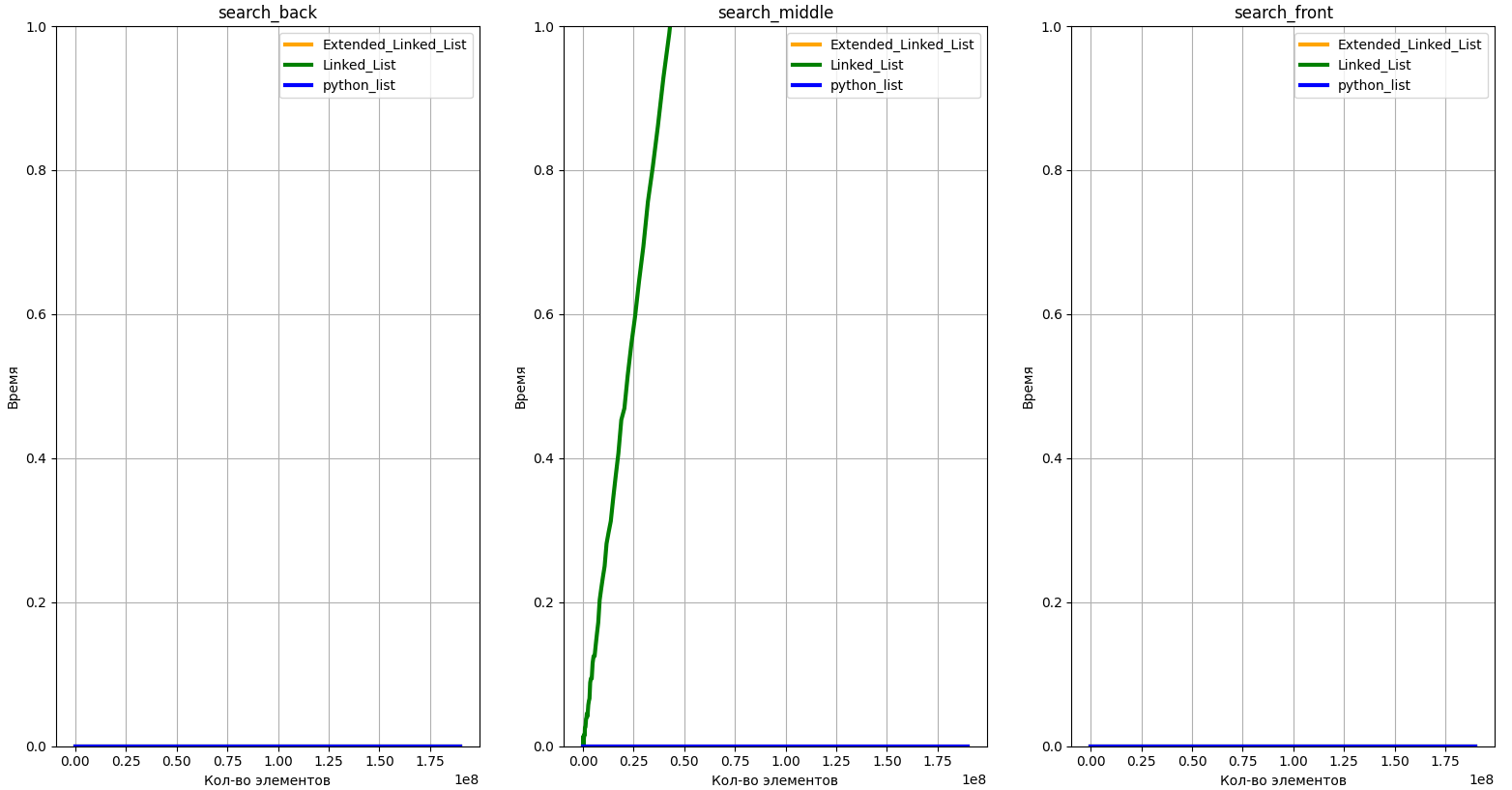
|  |  |  |  |
| --- | --- | --- | --- |
| Поиск, python list | | | |
|  | search\_back | search\_in\_middle | search\_front |
| 10 | 0 | 0 | 0 |
| 10000 | 0 | 0 | 0 |
| 100000 | 0 | 0 | 0 |

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

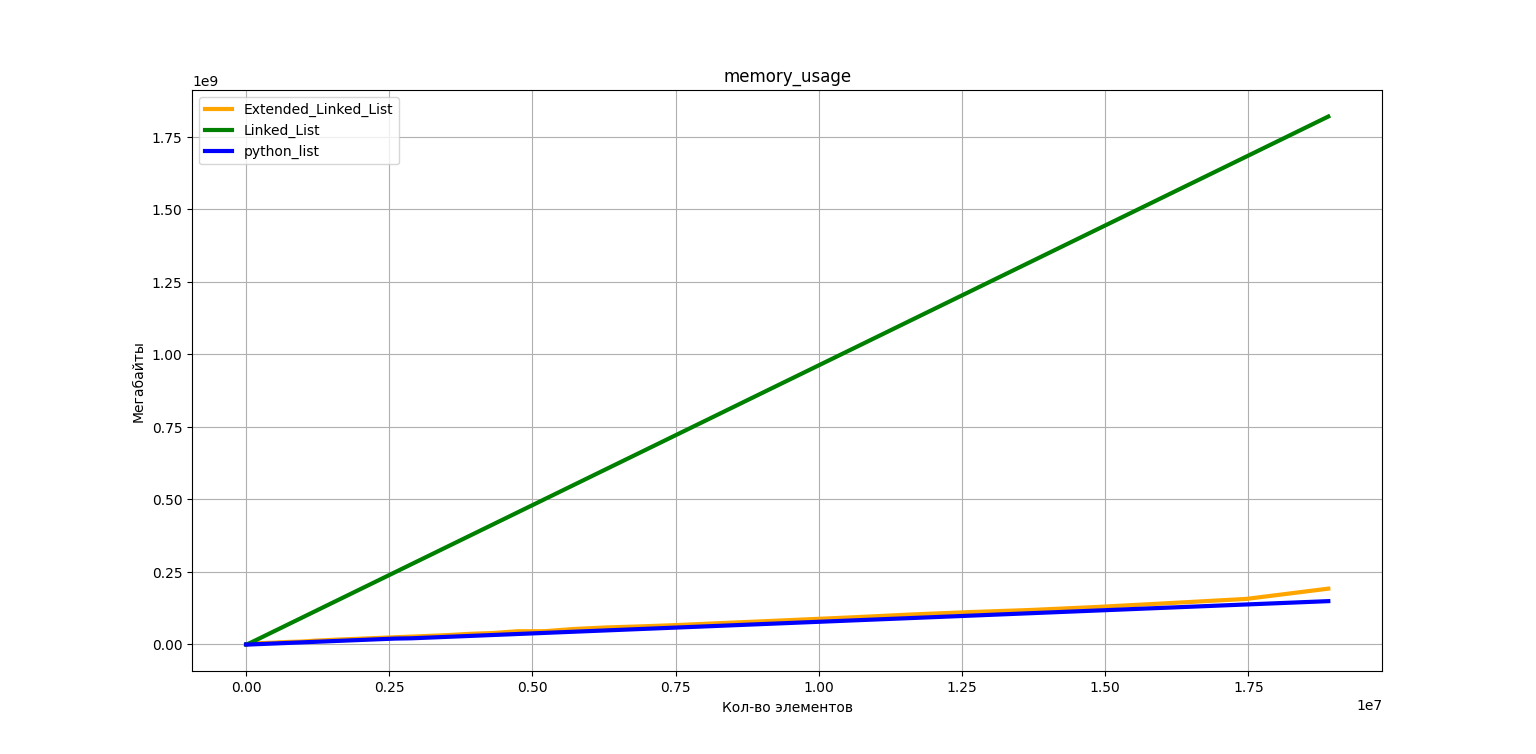


Рисунок 4 — Использование памяти

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)