**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: Развёрнутый связанный список

|  |  |  |
| --- | --- | --- |
| Студент гр. 3341 |  | Лодыгин И. А. |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург

2024

## Цель работы

Целью данной лабораторной работы является изучение и реализация развернутого связного списка как новой структуры данных. В процессе работы необходимо будет проанализировать его характеристики и производительность по сравнению с традиционными структурами данных, такими как массив и односвязный список.

Необходимо продемонстрировать основные операции над развернутым связным списком, такие как поиск, удаление и вставка элементов, а также провести эксперимент с измерением времени выполнения этих операций. Работа включает в себя заполнение структуры, выполнение операций поиска и удаления, а далее сравнение полученных результатов с аналогичными операциями в других структурах данных.

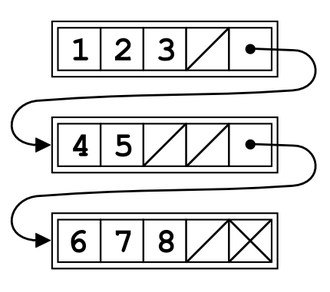
В результате исследования будут представлены таблицы с результатами замеров и графическое представление данных. Это позволит оценить эффективность развернутого связного списка в разных сценариях и сделать выводы о его применимости в различных задачах программирования.

## 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. Реализацию поиска и удаления делать на свое усмотрение. Данные операции будут проверяться на защите.

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).

## Выполнение работы

Программа начинается с импорта модуля math, который предоставляет необходимые математические функции, такие как ceil.

Далее, объявляется функция calculate\_optimal\_node\_size, которая принимает количество элементов num\_elements и вычисляет оптимальный размер узла. Она использует фиксированные значения размера кэша (cache\_size = 64) и размера одного целого числа (int\_size = 4), после чего вычисляет общий объем памяти, требуемый для хранения всех элементов. Для этого перемножается количество элементов на размер одного элемента. Затем деление этого объема на размер кэша с округлением в большую сторону (math.ceil) позволяет получить приблизительный оптимальный размер узла. Добавление единицы гарантирует наличие места для размещения заголовка узла.

Класс Node представляет собой узел развернутого связного списка. Конструктор этого класса принимает два аргумента: node\_size и node\_num.

1. Поле elements инициализируется как пустой список, где будут храниться элементы.
2. Поле node\_size хранит размер узла, то есть максимальное количество элементов, которые могут быть в узле.
3. Поле node\_num хранит номер узла в списке.
4. Поле next по умолчанию имеет значение None и используется для связи с следующим узлом в списке.

Класс UnrolledLinkedList реализует развернутый связный список. Конструктор этого класса принимает список элементов и создает структуру, разбивая их на узлы.

1. Поле element\_nums хранит количество элементов в списке.
2. Поле node\_size получает значение с помощью вызова функции calculate\_optimal\_node\_size, которая определяет оптимальный размер узла.
3. Поле head представляет собой первый узел списка. Этот узел инициализируется с размером узла и номером 0.
4. Поле node\_nums содержит общее количество узлов в списке.

Элементы списка равномерно распределяются по узлам. Если текущий узел заполнен, создается новый узел с увеличенным номером, и добавляется к предыдущему узлу через указатель next.

Методы класса UnrolledLinkedList:

1. Метод recreate\_list: Воссоздает структуру списка с заданным набором элементов. Этот метод используется в случае, если изменяется оптимальный размер узла после добавления или удаления элементов.
2. Метод push\_back: Добавляет элемент в конец списка. Если добавление элемента приводит к изменению оптимального размера узла, список пересоздается с новым размером.
3. Метод insert\_element: Вставляет элемент на заданный индекс в списке. При необходимости перераспределяет элементы между узлами, чтобы сохранить правильную структуру.
4. Метод push\_forward: Вставляет элемент в начало списка. Если после добавления элемента изменяется оптимальный размер узла, пересоздает список с новым размером узла.
5. Метод delete\_element: Удаляет элемент из списка. Если удаление элемента изменяет оптимальный размер узла, список пересоздается.
6. Метод find\_element: Ищет элемент в списке и возвращает информацию о его позиции в узле и индексе в списке. Если элемент не найден, возвращается None.
7. Метод get\_elements\_list: Возвращает список всех элементов в связном списке.
8. Метод \_\_len\_\_: Возвращает количество элементов в списке, суммируя количество элементов во всех узлах.
9. Метод \_\_str\_\_: Возвращает строковое представление списка, в котором каждый узел представлен с его номером и элементами.

Функция check используется для проверки работы программы. Она создает объект класса UnrolledLinkedList, наполняет его элементами и затем удаляет некоторые элементы, выводя текущее состояние списка на каждом шаге

## Тестирование

Результаты тестирования представлены в табл. 1.

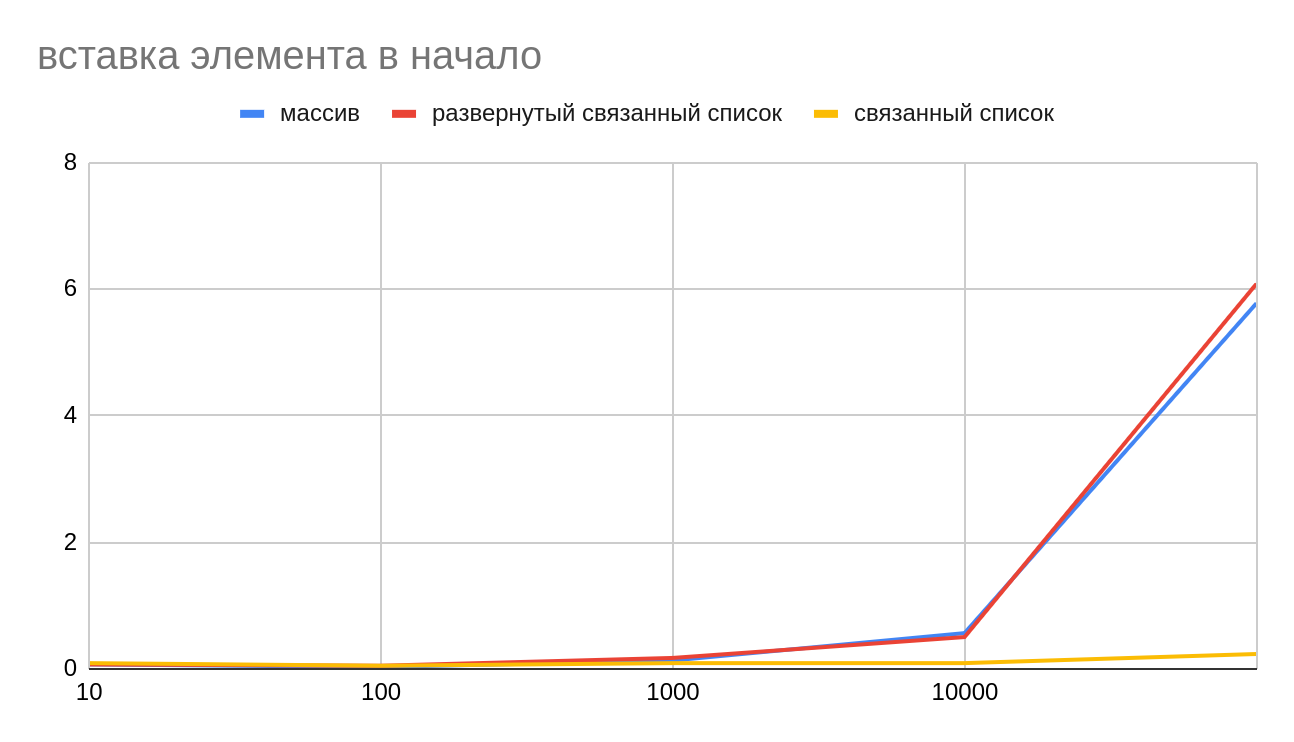
Таблица 1 сложность основных операций в структурах данных (время)

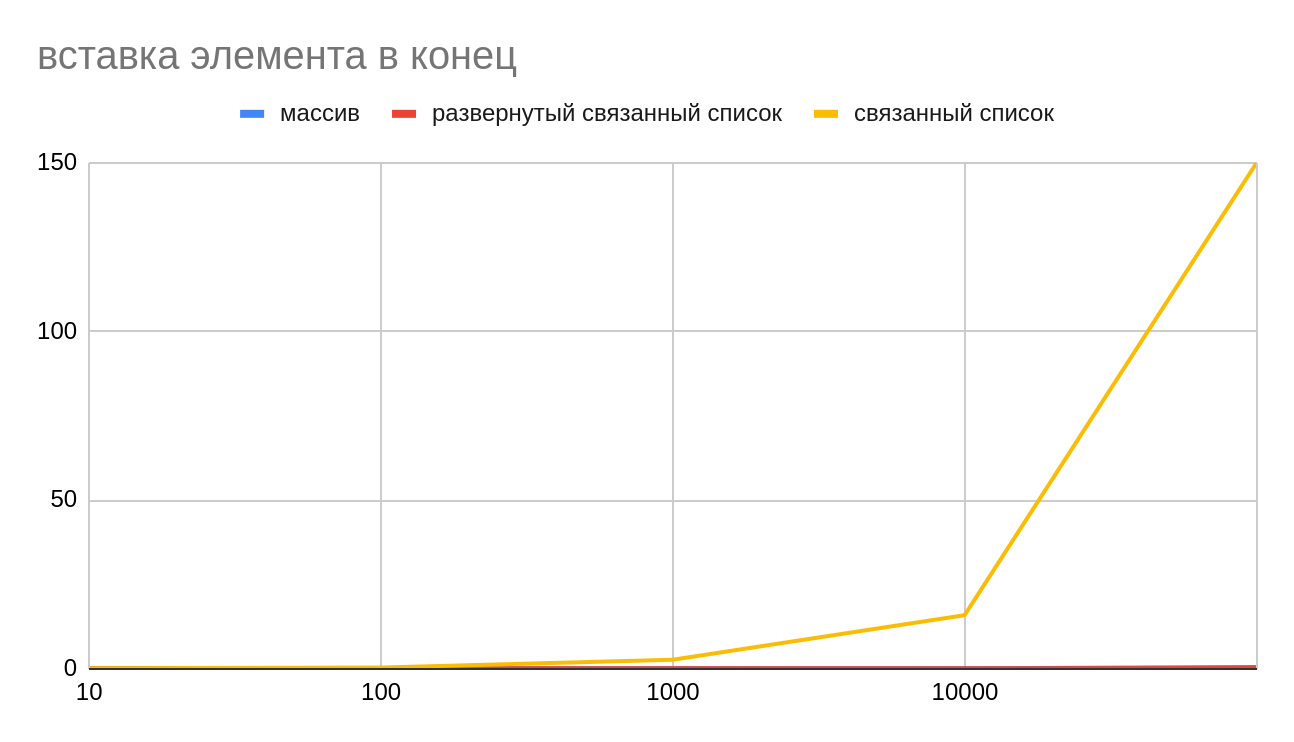
|  |  |  |  |
| --- | --- | --- | --- |
| Операция | Массив | Связанный список | Развёрнутый связанный список |
| Добавление элемента в начало | O(n), линейная | O(1), константная | O(n), линейная |
| Добавление элемента в конец | O(1), константная | O(n), линейная | O(n), линейная |
| Добавление/удаление элемента по индексу | O(1), константная | O(n), линейная | O(n), линейная |
| Удаление элемента в начале | O(n), линейная | O(1), константная | O(n), линейная |
| Удаление элемента в конце | O(1), константная | O(n), линейная | O(n), линейная |
| Линейный поиск | O(n), линейная | | |

Таблица 2 сложность основных операций в структурах данных (память)

|  |  |  |  |
| --- | --- | --- | --- |
| Операция | Массив | Связанный список | Развёрнутый связанный список |
| Добавление элемента в начало | O(n), линейная | O(1), константная | O(n), линейная |
| Добавление элемента в конец | O(1), константная | O(1), константная | O(n), линейная |
| Добавление/удаление элемента по индексу | O(1), константная | O(1), константная | O(n), линейная |
| Удаление элемента в начале | O(n), линейная | O(1), константная | O(n), линейная |
| Удаление элемента в конце | O(1), константная | O(1), константная | O(n), линейная |
| Линейный поиск | O(1), константная | | |

Ниже представлены графики, отображающие зависимость времени выполнения операций с различными структурами данных от числа элементов. Можно заметить, что развернутый связный список демонстрирует значительно лучшую производительность по сравнению с обычным связным списком при операциях доступа по индексу, включая удаление и добавление элементов в середину или конец. При этом его быстродействие практически на уровне массива.Вставка элемента в начало

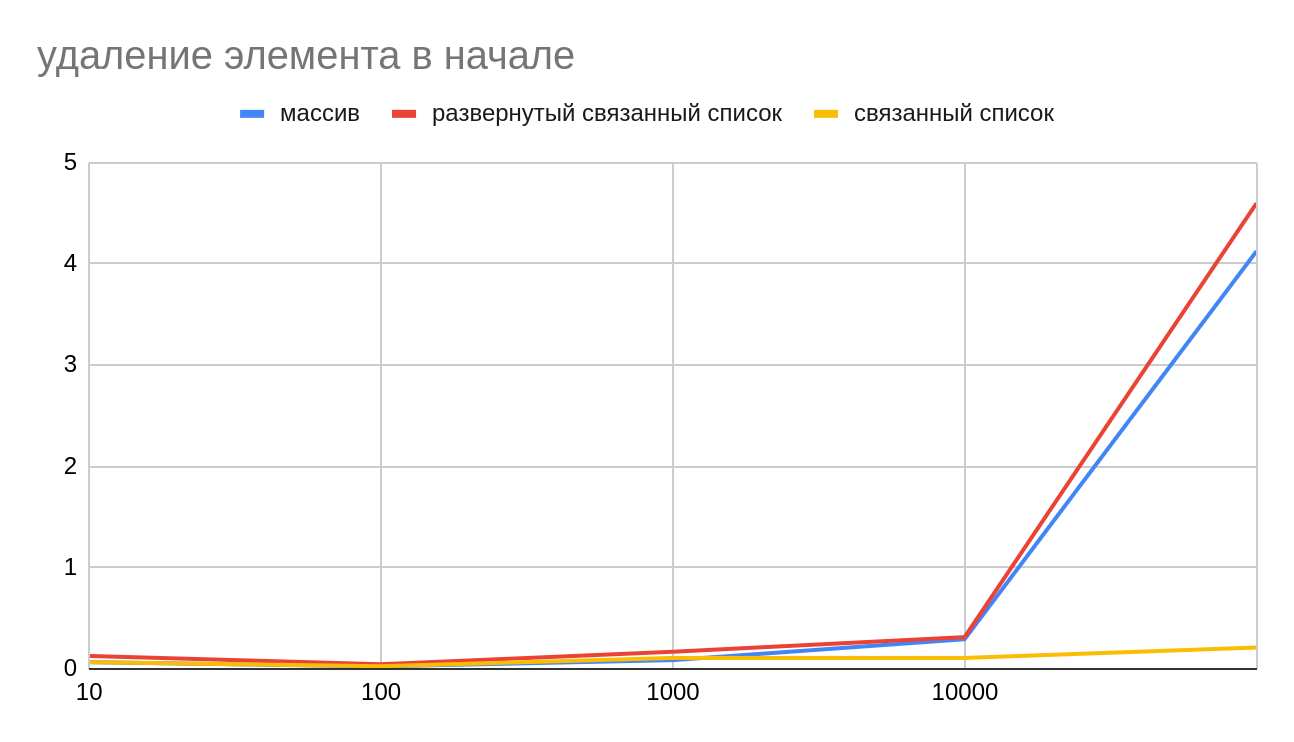
  
Вставка элемента в конец

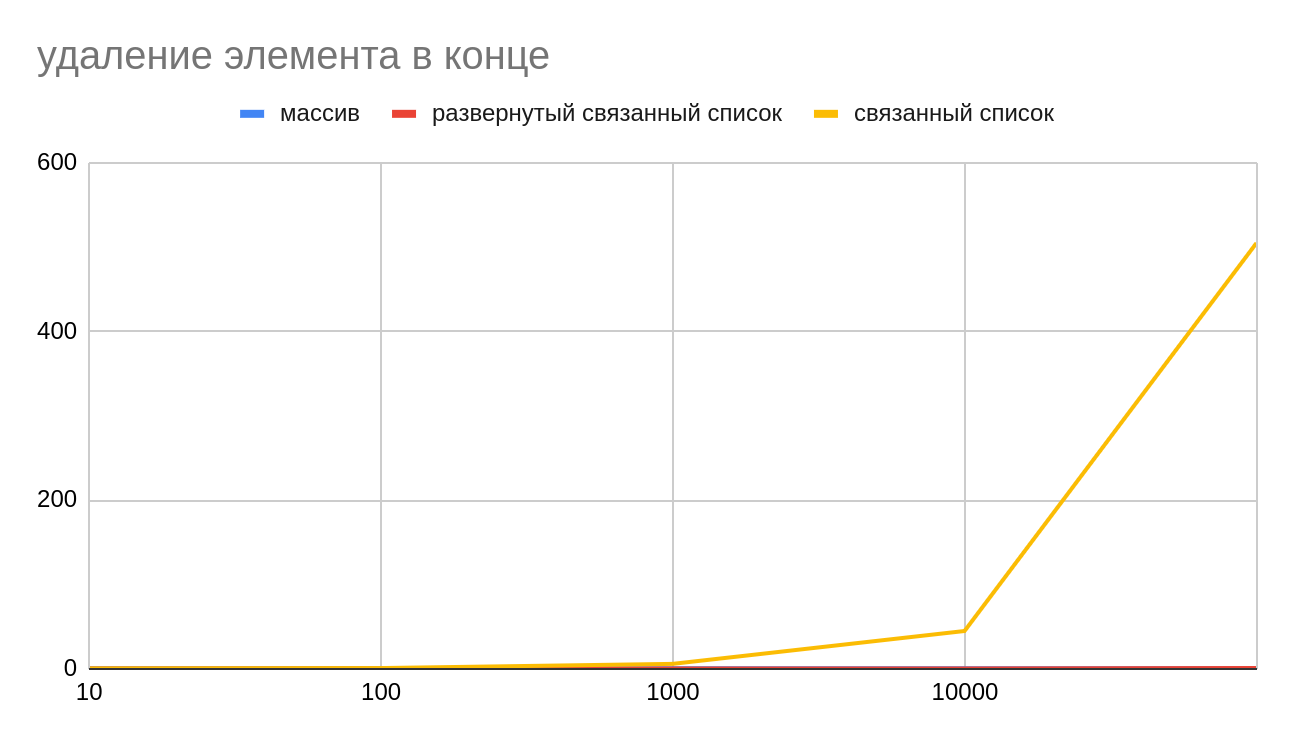


Вставка элемента в середину



Удаление элемента в начале

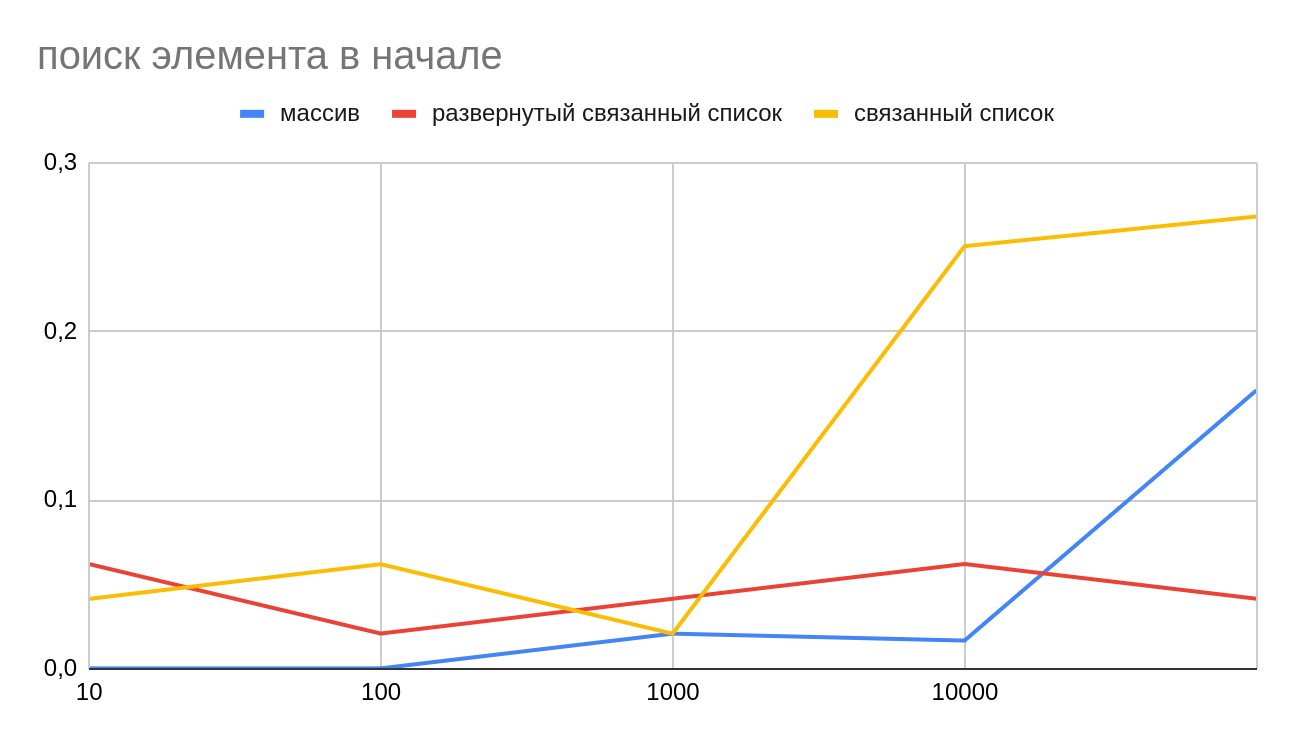
  
Удаление элемента в конце:



Удаление элемента в середине:



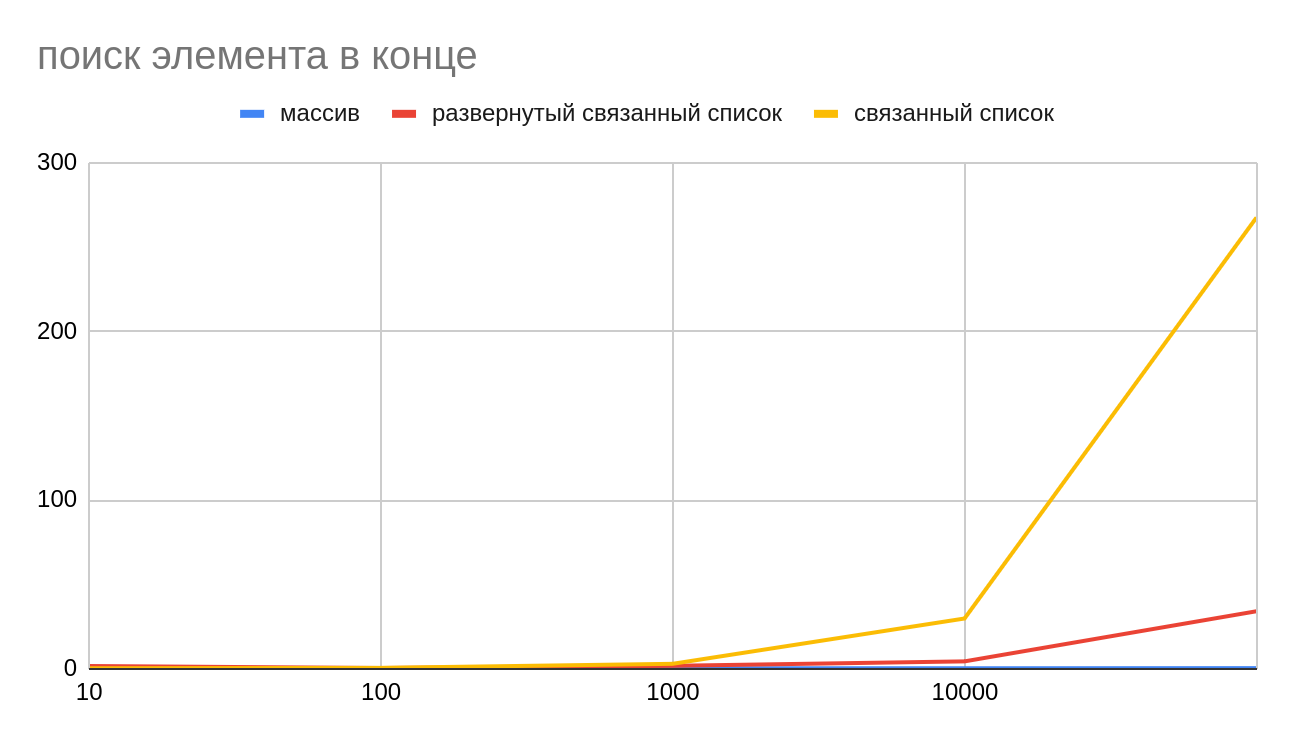
Поиск элемента в начале:



Поиск элемента в середине:



Поиск элемента в конце:



## Выводы

В результате выполнения данной лабораторной работы мы изучили и проанализировали развернутый связный список как альтернативную структуру данных. Основное внимание было уделено его операциям, таким как вставка, удаление и поиск элементов. Развернутый связный список представляет собой динамическую структуру, что позволяет ему увеличиваться или уменьшаться в размере без необходимости перераспределения памяти. Это делает его более гибким по сравнению с массивами.

Ключевым преимуществом развернутого связного списка является эффективность операций. Вставка и удаление элементов осуществляются со временем O(1), когда известен непосредственно узел, что значительно быстрее, чем в массиве. Кроме того, такая структура данных позволяет избежать значительной фрагментации памяти, что положительно сказывается на производительности при работе с большими объемами данных.

Однако, у развернутого связного списка есть и недостатки. Каждый элемент списка требует дополнительного пространства для хранения указателей, что может быть ощутимым недостатком при работе с большим количеством данных. Также сложность доступа к элементам является одним из существенных минусов: для доступа требуется пройти через несколько узлов, что делает операции доступа менее эффективными в сравнении с массивами, где они осуществляются за время O(1).

В заключение, развернутый связный список является эффективным инструментом для реализации динамических и изменяемых наборов данных, особенно когда важна скорость вставки и удаления. Тем не менее, при использовании данной структуры следует учитывать ограничения, такие как дополнительные затраты на память и медленный доступ к элементам. В дальнейшем могут быть интересны исследования по комбинированию развернутого связного списка с другими структурами данных для достижения оптимальной производительности в различных сценариях. Работа показала, что понимание свойств каждой структуры данных и их уместность в различных задачах является ключом к успешной разработке программных решений.

# Приложение А Исходный код программы

Название файла: main.py

import math

def calculate\_optimal\_node\_size(num\_elements):

cache\_size = 64

int\_size = 4

memory\_size = num\_elements\*int\_size

optimal\_node\_size = math.ceil(memory\_size/cache\_size) + 1

return optimal\_node\_size

class Node:

def \_\_init\_\_(self, node\_size, node\_num):

self.elements = list()

self.node\_size = node\_size

self.node\_num = node\_num

self.next = None

class UnrolledLinkedList:

def \_\_init\_\_(self, elements):

self.element\_nums = len(elements)

self.node\_size = calculate\_optimal\_node\_size(self.element\_nums)

self.head = Node(self.node\_size, 0)

self.node\_nums = 1

counter = 0

for element in elements:

counter += 1

current\_node = self.head

while(True):

if(current\_node.next is None):

break

current\_node = current\_node.next

current\_node.elements.append(element)

if(len(current\_node.elements) == self.node\_size):

if(counter == self.element\_nums):

current\_node.next = None

else:

current\_node.next = Node(self.node\_size, current\_node.node\_num + 1)

self.node\_nums += 1

def recreate\_list(self, elements):

self.element\_nums = len(elements)

self.node\_size = calculate\_optimal\_node\_size(self.element\_nums)

self.head = Node(self.node\_size, 0)

self.node\_nums = 1

counter = 0

for element in elements:

counter += 1

current\_node = self.head

while(True):

if(current\_node.next is None):

break

current\_node = current\_node.next

current\_node.elements.append(element)

if(len(current\_node.elements) == self.node\_size):

if(counter == self.element\_nums):

current\_node.next = None

else:

current\_node.next = Node(self.node\_size, current\_node.node\_num + 1)

self.node\_nums += 1

def push\_back(self, node\_element):

if(calculate\_optimal\_node\_size(self.element\_nums + 1) != self.node\_size):

elements = self.get\_elements\_list()

elements.append(node\_element)

self.recreate\_list(elements)

return

current\_node = self.head

while(True):

if(current\_node.next) == None:

break

current\_node = current\_node.next

if(len(current\_node.elements) == self.node\_size):

current\_node.next = Node(self.node\_size, current\_node.node\_num + 1)

current\_node = current\_node.next

self.node\_nums += 1

current\_node.elements.append(node\_element)

self.element\_nums += 1

def insert\_element(self, index, node\_element):

if(index >= self.\_\_len\_\_() or index < 0):

raise IndexError('list index out of range')

if(calculate\_optimal\_node\_size(self.element\_nums + 1) != self.node\_size):

elements = self.get\_elements\_list()

elements.insert(index, node\_element)

self.recreate\_list(elements)

return

self.element\_nums += 1

node\_num = int(index/self.node\_size)

target\_node = self.head

for \_ in range(node\_num):

target\_node = target\_node.next

target\_node.elements.insert(index % self.node\_size, node\_element)

if(len(target\_node.elements) > self.node\_size):

deleted\_element = target\_node.elements.pop()

else: return

if(target\_node.next is None):

self.push\_back(deleted\_element)

return

while(True):

target\_node = target\_node.next

if(target\_node.next is None):

if(len(target\_node.elements) >= self.node\_size):

target\_node.elements.insert(0, deleted\_element)

deleted\_element = target\_node.elements.pop()

self.push\_back(deleted\_element)

return

else:

target\_node.elements.insert(0, deleted\_element)

deleted\_element = target\_node.elements.pop()

def push\_forward(self, node\_element):

if(calculate\_optimal\_node\_size(self.element\_nums + 1) != self.node\_size):

elements = self.get\_elements\_list()

elements.insert(0, node\_element)

self.recreate\_list(elements)

return

self.insert\_element(0, node\_element)

def delete\_element(self, node\_element):

element\_index = self.find\_element(node\_element)

if(element\_index is None):

return

else: element\_index = element\_index['index']

if(calculate\_optimal\_node\_size(self.element\_nums - 1) != self.node\_size):

elements = self.get\_elements\_list()

elements.pop(element\_index)

self.recreate\_list(elements)

return node\_element

node\_num = int(element\_index/self.node\_size)

target\_node = self.head

for \_ in range(node\_num):

target\_node = target\_node.next

target\_node.elements.remove(node\_element)

self.element\_nums -= 1

while(True):

if(target\_node.next is None):

if(not target\_node.elements):

self.node\_nums -= 1

target\_node = None

return node\_element

target\_node.elements.append(target\_node.next.elements[0])

if(len(target\_node.next.elements) == 1):

self.node\_nums -= 1

target\_node.next = None

return node\_element

else:

target\_node.next.elements.pop(0)

target\_node = target\_node.next

def find\_element(self, element):

current\_node = self.head

while(True):

if element in current\_node.elements:

return {'position': f'Node: {current\_node.node\_num}, node index: {current\_node.elements.index(element)}',

'index': current\_node.node\_num\*self.node\_size + current\_node.elements.index(element)}

if(current\_node.next is None): return None

current\_node = current\_node.next

def get\_elements\_list(self):

elements = list()

current\_node = self.head

while(True):

elements += current\_node.elements

if(current\_node.next is None): break

current\_node = current\_node.next

return elements

def \_\_len\_\_(self):

lenght = 0

current\_node = self.head

while(True):

lenght += len(current\_node.elements)

if(current\_node.next is None): break

current\_node = current\_node.next

return lenght

def \_\_str\_\_(self):

current\_node = self.head

res = ''

while(True):

if(current\_node.elements):

res+=f'Node {current\_node.node\_num}: ' + ' '.join([str(element) for element in current\_node.elements]) + '\n'

if(current\_node.next is None): break

current\_node = current\_node.next

return res

def check(arr\_1, arr\_2):

my\_list = UnrolledLinkedList([])

for el in arr\_1:

my\_list.push\_back(el)

print(my\_list)

for el in arr\_2:

my\_list.delete\_element(el)

print(my\_list)

elements = list(map(int, input().split()))

my\_list = UnrolledLinkedList(elements)

print(my\_list, end='')