**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 связанный список

|  |  |  |
| --- | --- | --- |
| Студент гр. 3388 |  | Глебова В.С. |
| Преподаватель |  | Шалагинов И.В. |

Санкт-Петербург

2024

## Цель работы

Провести исследование и реализацию развернутого связного списка. Сравнить время выполнения различных операций для развернутого связанного списка, простого связного списка и массива.

## Задание

Вам необходимо реализовать функцию *calculate\_optimal\_node\_size(num\_elements)* которая вычисляет оптимальный размер узла для хранения определенного количества элементов в памяти. Для этого вам необходимо узнать общий объем памяти(количество байт, где берем элемент int, занимающий 4 байт), необходимый для хранения всех элементов, количество строк кэша(общий объем памяти поделить на минимальный размер строки кэша с округлением вверх), необходимого для хранения всех элементов(минимальный размер строки кэша = 64). Необходимо вывести оптимальный размер узла, с добавлением одной дополнительной строки кэша.

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

## Выполнение работы

### Класс Node

Класс Node представляет отдельный узел в списке.

* Поля:
  + data: список, содержащий элементы узла.
  + next: ссылка на следующий узел в списке.
* Методы:
  + \_\_next\_\_: возвращает ссылку на следующий узел.
  + \_\_len\_\_: возвращает количество элементов в узле.
  + \_\_str\_\_: возвращает строковое представление данных узла через разделитель
  + \_\_getitem\_\_: доступ к элементу по индексу.
  + \_\_contains\_\_: проверка наличия элемента в узле.
  + bind: присвоение следущего узла и его возврат.
  + insert: вставка элемента в узел.
  + delete: удаление значения из узла по индексу.

### Класс UnfoldedLinkedList

Класс реализует основной функционал развёрнутого связанного списка.

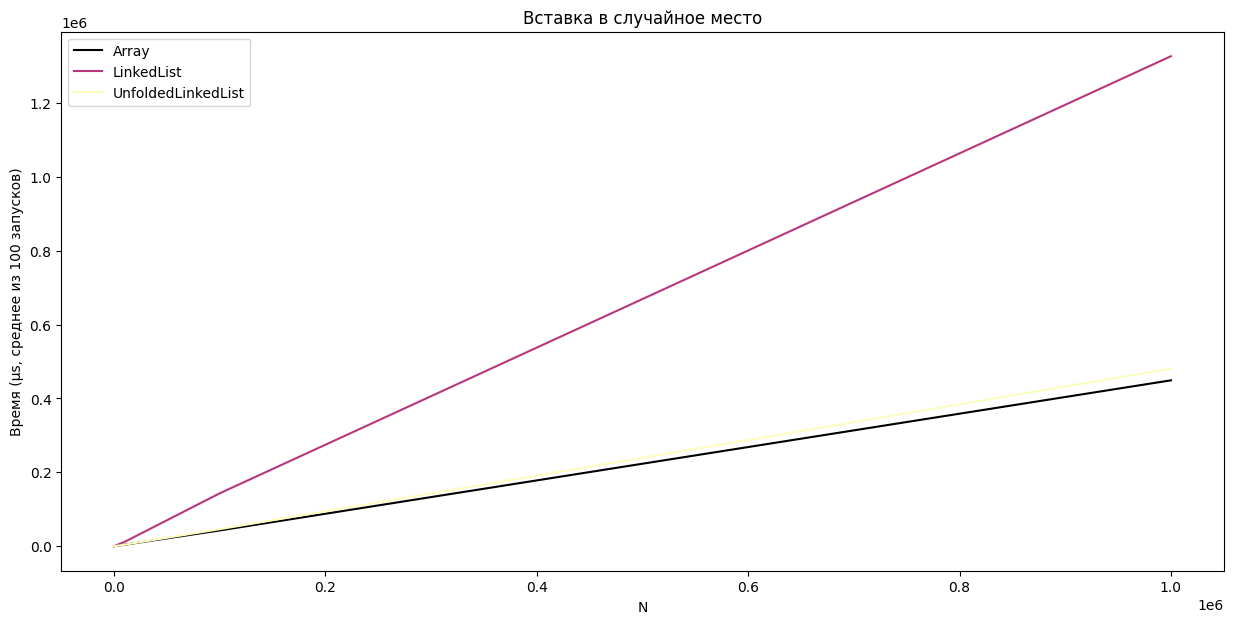
* Поля:
  + node\_size: оптимальный размер узла (сколько элементов вмещает один узел).
  + head: ссылка на первый узел.
  + tail: ссылка на последний узел.
* Основные методы:
  + \_\_len\_\_: возвращает количество элементов в списке.
  + \_\_str\_\_: возвращает строковое представление данных списка через разделитель.
  + print\_nodes: возвращает строковое представление данных узлов через разделитель.
  + rebalance: ребалансировка списка (перераспределение элементов при изменении оптимального размера узла).
  + insert: вставка элемента по указанному индексу.
  + delete: удаление элемента по индексу.
  + search: поиск элемента по значению.
* Вспомогательные методы:
  + calculate\_optimal\_node\_size: вычисляет оптимальный размер узла на основе количества элементов и параметров кэш-памяти.

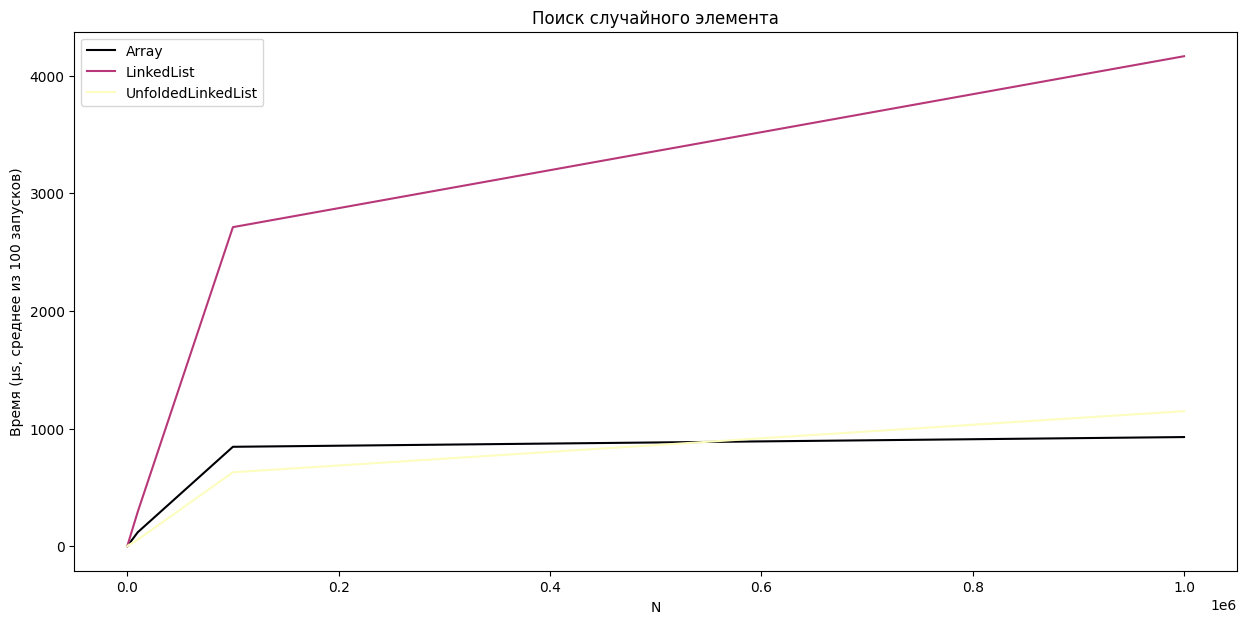
Исходный код программы см. в приложении А.

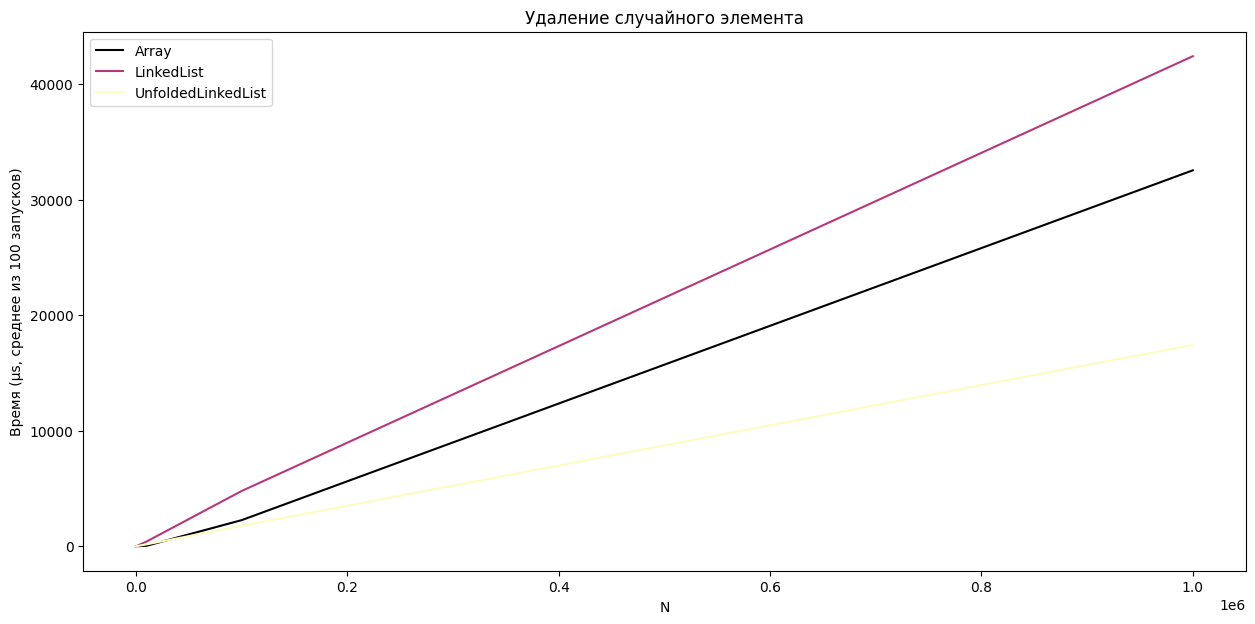
Результаты тестирования см. в приложении Б.

## Анализ полученных значений

Результаты скорости работы *UnfoldedLinkedList, LinkedList, Array* представлены в графическом виде ниже. Среднее время выполнения операции указано по оси ординат в микросекундах. По оси абцисс — N обозначает количество данных.

Рисунок 1 — Операция вставки

Рисунок 2 — Операция поиска

Рисунок 3 — Операция удаления

Обратите внимание, что при увеличении объема данных все три структуры демонстрируют схожее поведение, но развернутый список имеет свои особенности. При небольших объемах данных он немного медленнее обычного списка из-за дополнительных расходов на перебалансировку и выделение памяти для дополнительных массивов. Но при значительных объемах информации развернутый список становится быстрее, а в некоторых случаях даже превосходит массив, что свидетельствует о его большей эффективности по сравнению со стандартным списком.

## Выводы

Мы рассмотрели разные структуры данных, реализовали развернутый связный список на языке Python, измерили время выполнения операций вставки, поиска и удаления для развернутого связного списка, обычного связного списка и массива на различных объемах данных. Полученные результаты были проанализированы. Кроме того, мы написали тесты с помощью библиотеки pytest.

# Приложение A Исходный код программы

Файл: modules/unfolded\_linked\_list.py

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

memory\_cost = num\_elements \* 4

cache\_cost = memory\_cost // 64 + 2

return cache\_cost

class Node:

def \_\_init\_\_(self, data=None):

self.data = data if data is not None else []

self.next = None

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

return len(self.data)

def \_\_getitem\_\_(self, index):

return self.data[index]

def \_\_contains\_\_(self, value):

return value in self.data

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

return " ".join(map(str, self.data))

def bind(self, next):

self.next = next

return next

def insert(self, values):

self.data.extend(values)

def delete(self, index):

self.data.pop(index)

class UnfoldedLinkedList:

head = None

tail = None

def \_\_init\_\_(self, init\_data=None):

self.node\_size = calculate\_optimal\_node\_size(0)

if init\_data is not None:

self.insert(init\_data)

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

current\_node = self.head

length = 0

while current\_node is not None:

length += len(current\_node)

current\_node = current\_node.next

return length

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

node\_strings = []

current\_node = self.head

while current\_node is not None:

node\_strings.append(str(current\_node))

current\_node = current\_node.next

return " ".join(node\_strings)

def print\_nodes(self):

node\_strings = []

current\_node = self.head

while current\_node is not None:

node\_strings.append(f"Node {len(node\_strings)}: {current\_node}")

current\_node = current\_node.next

return "\n".join(node\_strings)

def rebalance(self, new\_size):

self.node\_size = new\_size

if self.head is None:

return

new\_head = Node()

current\_node, new\_current\_node = self.head, new\_head

while current\_node is not None:

for i in range(len(current\_node)):

if len(new\_current\_node) == new\_size:

new\_current\_node = new\_current\_node.bind(Node())

new\_current\_node.insert([current\_node[i]])

current\_node = current\_node.next

self.head = new\_head

self.tail = new\_current\_node

def insert(self, values):

new\_size = calculate\_optimal\_node\_size(len(self) + len(values))

if new\_size > self.node\_size:

self.rebalance(new\_size)

self.\_insert(values)

def \_insert(self, data):

if self.head is None:

self.head = Node(data[:self.node\_size])

self.tail = self.head

data = data[self.node\_size:].copy()

else:

remaining\_space = self.node\_size - len(self.tail)

if remaining\_space > 0:

self.tail.insert(data[:remaining\_space])

data = data[remaining\_space:].copy()

for i in range(0, len(data), self.node\_size):

new\_node = Node(data[i:i + self.node\_size])

self.tail = self.tail.bind(new\_node)

def delete(self, index):

previous\_node, current\_node = None, self.head

elements\_passed = len(self.head) if self.head is not None else 0

while (current\_node is not None) and (elements\_passed <= index):

elements\_passed += len(current\_node)

previous\_node = current\_node

current\_node = current\_node.next

if ((previous\_node is None) and (current\_node is None)) or (passed\_elements <= index):

raise IndexError(f"Element with index {index} does not exist!")

node\_index = index - (elements\_passed - len(current\_node))

current\_node.delete(node\_index)

if len(current\_node) == 0:

if previous\_node is None:

self.head = current\_node.next

else:

previous\_node.next = current\_node.next

if current\_node.next is None:

self.tail = previous\_node

new\_size = calculate\_optimal\_node\_size(len(self))

if new\_size < self.node\_size:

self.rebalance(new\_size)

def search(self, value):

current\_node = self.head

elements\_passed = 0

while current\_node is not None:

if value in current\_node:

return elements\_passed + current\_node.data.index(value)

elements\_passed += len(current\_node)

current\_node = current\_node.next

return -1

ull = UnfoldedLinkedList()

ull.insert(list(map(int, input().split())))

print(ull.print\_nodes())

Файл: tests.py

from modules.unfolded\_linked\_list import calculate\_optimal\_node\_size, Node

def test\_calculate():

assert calculate\_optimal\_node\_size(16) == 3

assert calculate\_optimal\_node\_size(0) == 2

def test\_node():

node = Node([1, 2, 3])

assert str(node) == "1 2 3"

# Приложение Б Результаты тестирования

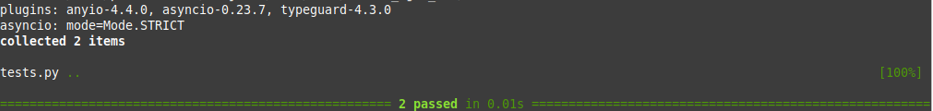


Рисунок 1 — Результаты тестирования