Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский химико-технологический университет имени Д.И. Менделеева»

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 3

Выполнил студент группы КС-30 Положенцев Аксель Алексеевич

Ссылка на репозиторий: [MUCTR-IKT-CPP/PolozhencevAA\_30](https://github.com/MUCTR-IKT-CPP/PolozhencevAA_30)

Приняли: Крашенинников Роман Сергеевич

Дата сдачи: 17.02.2025

Москва 2025

Оглавление

[Описание задачи 3](#_Toc191294879)

[Описание метода 3](#_Toc191294880)

[Выполнение задачи 4](#_Toc191294881)

[Заключение 9](#_Toc191294882)

# Описание задачи

В данной лабораторной работе требуется реализовать два варианта структуры данных "очередь":

1. Очередь, основанная на односвязном списке (LinkedQueue).
2. Очередь, реализованная с использованием двух стеков (TwoStackQueue).

Также необходимо провести тестирование разработанных структур, включая:

* Операции добавления и удаления элементов.
* заполнение контейнера 1000 целыми числами в диапазоне от -1000 до 1000 и подсчет их суммы, среднего, минимального и максимального.
* Провести проверку работы операций вставки и изъятия элементов на коллекции из 10 строковых элементов.
* заполнение контейнера 100 структур, содержащих фамилию, имя, отчество и дату рождения (от 01.01.1980 до 01.01.2020) значения каждого поля генерируются случайно из набора заранее заданных. После заполнение необходимо найти всех людей младше 20 лет и старше 30 и создать новые структуры, содержащие результат фильтрации, проверить выполнение на правильность подсчётом кол-ва элементов, не подходящих под условие в новых структурах. Тесты по вариантам:
* Измерение времени выполнения операций.

# Описание метода

1. Разработаны классы LinkedQueue и TwoStackQueue, содержащие базовые операции: push(), dequeue(), peek(), empty(), count().
2. В классе LinkedQueue используются узлы односвязного списка, содержащие значение и ссылку на следующий элемент.
3. В классе TwoStackQueue для реализации очереди применяются два стека (входной и выходной), что обеспечивает корректный порядок извлечения элементов.
4. Проведены тесты, включающие:
   * Заполнение очереди случайными числами и вычисление статистики (сумма, среднее значение, минимум, максимум).
   * Вставку и удаление строковых данных.
   * Фильтрацию списка объектов Person по году рождения.
   * Измерение скорости выполнения операций push() и dequeue() для больших объемов данных.

# Выполнение задачи

1. Реализованы классы LinkedQueue и TwoStackQueue.
2. import random
3. import time
4. from collections import deque
5. from typing import List
6. # python lab3.py
7. # Этот код представляет собой реализацию двух типов очередей:
8. # через односвязный список (LinkedQueue) и через два стека (TwoStackQueue).
9. # Далее, в нем выполняются различные тесты для этих очередей.
10. # Очередь через односвязный список
11. class LinkedQueue:
12. # Вложенный класс для представления узла очереди
13. class Node:
14. def \_\_init\_\_(self, value):
15. self.data = value
16. self.next = None # Указатель
17. # Инициализация пустой очереди
18. def \_\_init\_\_(self):
19. self.head = None
20. self.rear = None
21. self.size = 0 # Размер очереди
22. # Метод добавления элемента в очередь
23. def push(self, value):
24. new\_node = self.Node(value) # Создаем новый узел с переданным значением
25. if self.rear:
26. self.rear.next = new\_node
27. else:
28. # Если очередь пуста, новый узел будет и началом, и концом
29. self.head = new\_node
30. self.rear = new\_node # Новый узел становится концом очереди
31. self.size += 1
32. # Метод извлечения элемента из очереди
33. def dequeue(self):
34. # Если очередь пуста, выбрасываем исключение
35. if self.empty():
36. raise IndexError("Queue is empty")
37. # Временная переменная для удаления первого элемента
38. temp = self.head
39. # Смещаем начало очереди на следующий элемент
40. self.head = self.head.next
41. if not self.head:
42. self.rear = None
43. # Удаляем временный элемент
44. del temp
45. self.size -= 1
46. # Метод для получения первого элемента без удаления
47. def peek(self):
48. # Если очередь пуста, выбрасываем исключение
49. if self.empty():
50. raise IndexError("Queue is empty")
51. return self.head.data
52. # Метод проверки пустоты очереди
53. def empty(self):
54. return self.size == 0
55. # Метод для получения размера очереди
56. def count(self):
57. return self.size
58. # Превращает ваш односвязный список в итератор
59. def \_\_iter\_\_(self):
60. current = self.head
61. while current:
62. yield current.data
63. current = current.next
64. # Класс для представления человека
65. class Person:
66. def \_\_init\_\_(self, last\_name, first\_name, middle\_name, birth\_year):
67. self.last\_name = last\_name
68. self.first\_name = first\_name
69. self.middle\_name = middle\_name
70. self.birth\_year = birth\_year
71. # Тест 1: статистика для очереди из 1000 чисел
72. def test1():
73. queue = LinkedQueue()  # Создаем пустую очередь
74. random.seed(time.time())  # Инициализируем генератор случайных чисел
75. sum\_val = 0
76. min\_val = float('inf')
77. max\_val = float('-inf')
79. for \_ in range(1000):
80. value = random.randint(-1000, 1000) # Генерируем случайное число
81. queue.push(value) # Добавляем его в очередь
82. sum\_val += value # Считаем сумму всех чисел
83. min\_val = min(min\_val, value) # Определяем минимум
84. max\_val = max(max\_val, value)  # Определяем максимум
86. average = sum\_val / queue.count() # Вычисляем среднее значение
87. print("Queue statistics:")
88. print(f"Sum: {sum\_val}")
89. print(f"Average: {average}")
90. print(f"Min: {min\_val}")
91. print(f"Max: {max\_val}")
92. # Тест 2: проверка операций вставки и изъятия на 10 строковых элементах
93. def test2():
94. queue = LinkedQueue() # Создаем очередь
95. words = ["orange", "blueberry", "strawberry", "peach", "plum", "apricot", "pear", "mango", "watermelon", "pineapple"]
97. print("Enqueuing elements:")
98. for word in words:
99. queue.push(word) # Добавляем слова в очередь
100. print(word, end=" ")
101. print("\nDequeuing elements:")
102. while not queue.empty(): # Пока очередь не пуста
103. print(queue.peek(), end=" ") # Выводим первый элемент
104. queue.dequeue() # Удаляем его
105. print("\nTest2 completed.")
106. # Тест 3: работа с очередью из 100 структур Person
107. def test3():
108. queue = LinkedQueue() # Создаем очередь
109. names = ["Daniel", "Sophia", "Liam", "Olivia", "Ethan"]
110. last\_names = ["Miller", "Davis", "Wilson", "Moore", "Thomas"]
111. random.seed(time.time())
113. for \_ in range(100):
114. person = Person(random.choice(last\_names), random.choice(names), "", random.randint(1980, 2020))
115. queue.push(person) # Добавляем случайных людей
116. filtered\_queue = LinkedQueue()
117. count\_excluded = 0
118. while not queue.empty():
119. person = queue.peek()
120. queue.dequeue()
121. if person.birth\_year < 1994 or person.birth\_year > 2004:
122. filtered\_queue.push(person)  # Оставляем только тех, кто НЕ родился с 1994 по 2004
123. else:
124. count\_excluded += 1 # Считаем исключенных
125. print(f"Test3: Excluded count: {count\_excluded}")
126. reversed\_queue = LinkedQueue()
127. while not filtered\_queue.empty():
128. person = filtered\_queue.peek()
129. filtered\_queue.dequeue()
130. reversed\_queue.push(person)  # Разворачиваем очередь
131. print("Test3 completed.")
132. # Шаблонная очередь через два стека
133. class TwoStackQueue:
134. def \_\_init\_\_(self):
135. self.input\_stack = []
136. self.output\_stack = []
137. self.size = 0
138. # Добавляет элемент в очередь
139. def push(self, value):
140. self.input\_stack.append(value)
141. self.size += 1
142. # Удаляет первый элемент из очереди
143. def dequeue(self):
144. if self.empty():
145. raise IndexError("Queue is empty")
146. if not self.output\_stack:
147. self.transfer\_to\_output()
148. self.output\_stack.pop()
149. self.size -= 1
150. # Возвращает первый элемент в очереди без удаления
151. def peek(self):
152. if self.empty():
153. raise IndexError("Queue is empty")
154. if not self.output\_stack:
155. self.transfer\_to\_output()
156. return self.output\_stack[-1]
157. # Проверяет, пуста ли очередь
158. def empty(self):
159. return self.size == 0
160. # Возвращает количество элементов в очереди
161. def count(self):
162. return self.size
163. # Переносит элементы из input\_stack в output\_stack, меняя их порядок
164. def transfer\_to\_output(self):
165. while self.input\_stack:
166. self.output\_stack.append(self.input\_stack.pop())
167. # Тест 1 для TwoStackQueue
168. def test1\_two\_stack():
169. queue = TwoStackQueue()
170. random.seed(time.time())
171. sum\_val = 0
172. min\_val = float('inf')
173. max\_val = float('-inf')
175. for \_ in range(1000):
176. value = random.randint(-1000, 1000)
177. queue.push(value)
178. sum\_val += value
179. min\_val = min(min\_val, value)
180. max\_val = max(max\_val, value)
182. average = sum\_val / queue.count()
183. print("TwoStackQueue statistics:")
184. print(f"Sum: {sum\_val}")
185. print(f"Average: {average}")
186. print(f"Min: {min\_val}")
187. print(f"Max: {max\_val}")
188. # Тест 2 для TwoStackQueue
189. def test2\_two\_stack():
190. queue = TwoStackQueue()
191. words = ["orange", "pear", "mango", "plum", "peach", "apricot", "blueberry", "strawberry", "watermelon", "pineapple"]
193. print("Enqueuing elements (TwoStackQueue):")
194. for word in words:
195. queue.push(word)
196. print(word, end=" ")
197. print("\nDequeuing elements (TwoStackQueue):")
198. while not queue.empty():
199. print(queue.peek(), end=" ")
200. queue.dequeue()
201. print("\nTest2\_TwoStack completed.")
202. # Тест 3 для TwoStackQueue
203. def test3\_two\_stack():
204. queue = TwoStackQueue()
205. names = ["Daniel", "Sophia", "Liam", "Olivia", "Ethan"]
206. last\_names = ["Miller", "Davis", "Wilson", "Moore", "Thomas"]
207. random.seed(time.time())
209. for \_ in range(100):
210. person = Person(random.choice(last\_names), random.choice(names), "", random.randint(1980, 2020))
211. queue.push(person)
212. filtered\_queue = TwoStackQueue()
213. count\_excluded = 0
214. while not queue.empty():
215. person = queue.peek()
216. queue.dequeue()
217. if person.birth\_year < 1994 or person.birth\_year > 2004:
218. filtered\_queue.push(person)
219. else:
220. count\_excluded += 1
221. print(f"Test3\_TwoStack: Excluded count: {count\_excluded}")
222. reversed\_queue = TwoStackQueue()
223. while not filtered\_queue.empty():
224. person = filtered\_queue.peek()
225. filtered\_queue.dequeue()
226. reversed\_queue.push(person)
227. print("Test3\_TwoStack completed.")
228. # Тест скорости для LinkedQueue
229. def test\_linked\_queue\_speed(elements):
230. queue = LinkedQueue() # Создаём экземпляр очереди на связном списке
232. # Засекаем время выполнения операции добавления элементов
233. start = time.time()
234. for i in range(elements):
235. queue.push(i)   # Добавляем элементы в очередь
236. push\_time = (time.time() - start) \* 1\_000\_000   # Конвертируем в микросекунды
237. # Засекаем время выполнения операции удаления элементов
238. start = time.time()
239. while not queue.empty():
240. queue.dequeue()  # Удаляем элементы из очереди
241. dequeue\_time = (time.time() - start) \* 1\_000\_000 # Конвертируем в микросекунды
242. print(f"push time: {push\_time} microseconds")
243. print(f"Dequeue time: {dequeue\_time} microseconds")
244. def test\_two\_stack\_queue\_speed(elements):
245. queue = TwoStackQueue() # Создаём экземпляр очереди, реализованной через два стека
247. # Засекаем время выполнения операции добавления элементов
248. start = time.time()
249. for i in range(elements):
250. queue.push(i)   # Добавляем элементы в очередь
251. push\_time = (time.time() - start) \* 1\_000\_000   # Конвертируем в микросекунды
252. # Засекаем время выполнения операции удаления элементов
253. start = time.time()
254. while not queue.empty():
255. queue.dequeue() # Удаляем элементы из очереди
256. dequeue\_time = (time.time() - start) \* 1\_000\_000    # Конвертируем в микросекунды
257. print(f"push time for TwoStackQueue: {push\_time} microseconds")
258. print(f"Dequeue time for TwoStackQueue: {dequeue\_time} microseconds")
259. def main():
260. print("Testing LinkedQueue:")
261. test1()
262. test2()
263. test3()
264. print("\nTesting TwoStackQueue:")
265. test1\_two\_stack()
266. test2\_two\_stack()
267. test3\_two\_stack()
268. ELEMENTS = 10000
269. print(f"Performance comparison for {ELEMENTS} elements:")
270. test\_linked\_queue\_speed(ELEMENTS)
271. test\_two\_stack\_queue\_speed(ELEMENTS)
272. if \_\_name\_\_ == "\_\_main\_\_":
273. main()

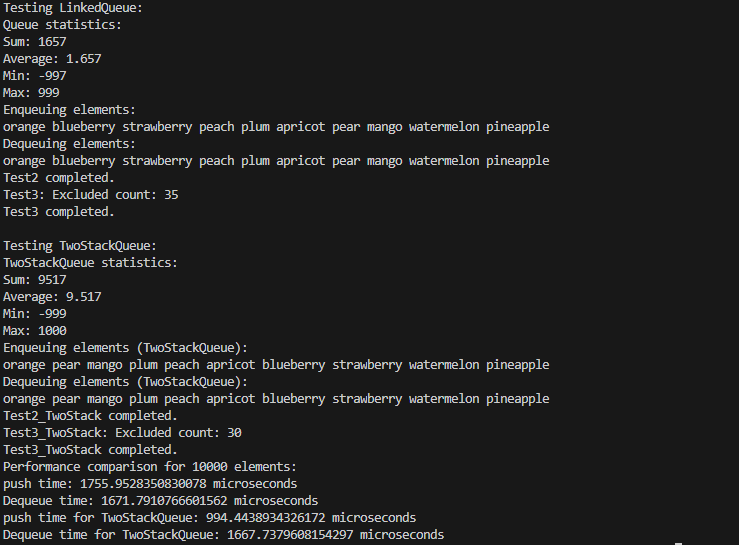
Проведены тесты:

* + Тест 1: добавление 1000 случайных чисел в очередь, вычисление их суммы, среднего, минимума и максимума.
  + Тест 2: работа с 10 строковыми элементами (добавление и удаление).
  + Тест 3: работа с 100 объектами класса Person, фильтрация по году рождения.
  + Тест скорости работы очередей на 1000000 элементов.

По результатам тестов подтверждена корректность работы обеих реализаций очереди.

Анализ показал, что LinkedQueue более эффективна по скорости удаления элементов, тогда как TwoStackQueue может демонстрировать ускоренную работу при множественных операциях push().

**Результат выполнения программы:**



# Заключение

В ходе тестирования были проверены основные операции с очередями: добавление (push), извлечение (dequeue) и получение первого элемента (peek). Оба типа очередей продемонстрировали корректную работу с элементами, что подтверждается успешным завершением всех тестов с различными наборами данных (строки, числа, структуры).

1. **LinkedQueue** показала более высокие времена на операции вставки и удаления элементов при большом числе (10000 элементов), что связано с особенностями работы с односвязным списком. Каждая операция добавления или удаления требует времени на перемещение указателей.
2. **TwoStackQueue** работает быстрее по времени вставки элементов, так как использование двух стеков позволяет эффективнее управлять порядком данных, однако время извлечения элементов оказалось немного выше, так как требуется дополнительное время для переноса элементов между стеками.

На основе результатов тестов можно заключить, что для операций с большим количеством вставок более эффективен **TwoStackQueue**, так как он быстрее справляется с операцией push. Однако для операций с удалением **LinkedQueue** может быть предпочтительнее в некоторых случаях, так как затраты времени на dequeue в TwoStackQueue могут быть больше из-за необходимости переноса данных между стеков.

В общем, выбор между этими двумя структурами зависит от специфики задачи: если важна скорость вставки — стоит использовать **TwoStackQueue**, если приоритет на скорость извлечения элементов — **LinkedQueue**.

Анализ сложности операций для обеих очередей:

**LinkedQueue (Очередь на основе связного списка):**

* **Добавление (enqueue)**:  
  Время выполнения операции добавления (enqueue) для LinkedQueue — **O(1)**. Это связано с тем, что при добавлении элемента в конец очереди мы просто изменяем указатель последнего элемента на новый элемент. В связном списке не требуется сдвигать другие элементы.
* **Удаление (dequeue)**:  
  Время выполнения операции удаления (dequeue) для LinkedQueue — **O(1)**. Мы просто перемещаем указатель на первый элемент, после чего старый элемент можно удалить. Этот процесс не требует перебора всех элементов, так как мы всегда работаем только с первым элементом списка.
* **Поиск (find)**:  
  Время поиска элемента в LinkedQueue — **O(n)**, где n — это количество элементов в очереди. Для поиска элемента нужно пройти по всему списку, начиная с первого, до тех пор, пока не будет найден нужный элемент. Это обусловлено тем, что в LinkedQueue элементы не упорядочены, и для поиска необходимо последовательное перебирание всех элементов.

**2. TwoStackQueue (Очередь на основе двух стеков):**

* **Добавление (enqueue)**:  
  Время выполнения операции добавления (enqueue) для TwoStackQueue — **O(1)** в случае обычного добавления элемента в первый стек. Время вставки не зависит от количества элементов в очереди, так как операции push в стеке происходят за константное время.
* **Удаление (dequeue)**:  
  Время выполнения операции удаления (dequeue) для TwoStackQueue в худшем случае — **O(n)**, где n — количество элементов в очереди. Когда второй стек пуст, все элементы из первого стека должны быть перенесены во второй стек, что занимает время, пропорциональное числу элементов в очереди. После этого операция удаления (pop) происходит за **O(1)**, если второй стек не пуст.
* **Поиск (find)**:  
  Время поиска элемента в TwoStackQueue — **O(n)**, где n — количество элементов в очереди. Для поиска нужно пройти через оба стека, что также требует линейного времени, так как элементы не отсортированы, и нам нужно перебрать все элементы.

**Сравнение:**

* **Добавление**: Операции добавления в обеих очередях имеют сложность **O(1)**.
* **Удаление**: Удаление в LinkedQueue имеет сложность **O(1)**, в то время как в TwoStackQueue может требовать **O(n)** в худшем случае из-за необходимости переноса элементов между стеками.
* **Поиск**: Операции поиска для обеих очередей имеют сложность **O(n)**, так как для поиска элемента нужно пройти через все элементы очереди.

Таким образом, **LinkedQueue** более эффективна в плане операций удаления и поиска, в то время как **TwoStackQueue** предоставляет более быстрые вставки, но может иметь ухудшение производительности при удалении элементов.