|  |
| --- |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА – Российский технологический университет"**  **РТУ МИРЭА** |

Институт комплексной безопасности и специального приборостроения

Кафедра КБ-3 «Управление и моделирование систем»

**ОТЧЕТ   
о выполнении лабораторной работы №1**

**«Реализация сортировки линейных структур данных»**

**по дисциплине   
«Программная реализация нелинейных структур»**

**Вариант № 47**

Выполнил: студент 2 курса

Группы **\_\_\_\_\_\_\_\_\_\_\_**

шифр \_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
*(фио студента)*

Проверил:

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Москва 2022 г.

**Задание на лабораторную работу № 1.**

В рамках лабораторной работы №1 требуется программно реализовать (с помощью указателей (однонаправленных/двунаправленный динамический линейный связанный список, массива или используя стандартный контейнер библиотеки STL “stack” или «queue» - по варианту) абстрактный тип данных (АТД) в соответствии с заданием (стек, дек, очередь с одной головой, очередь с головой и хвостом).

Абстрактный тип данных должен позволять осуществлять только операции, присущие типу линейного связанного списка:

* получить значение первого элемента (на выходе),
* добавить элемент (на вход),
* удалить элемент из списка (на выходе),
* проверить – список пуст,
* обнулить (проинициализировать) список (конструктор, при необходимости).
* деструктор (при необходимости)

Используя разработанный АТД и указанный набор операций, необходимо реализовать заданный алгоритм сортировки последовательности элементов заданного типа, при этом следует учитывать, что разрешен доступ (чтение/извлечение) только к элементу на выходе.

На основе исходного текста программы получить аналитическую оценку трудоемкости работы алгоритма сортировки, используя О-символику для каждого реализованного метода АТД и сортировки в целом.

**Вариант № 47.**

**Реализация связи элементов линейного списка: Очередь**

**Способ организации линейного связанный список: Библиотека классов**

**Алгоритм сортировки: Естественное двухпутевое слияние**

**Теория о сортировках.**

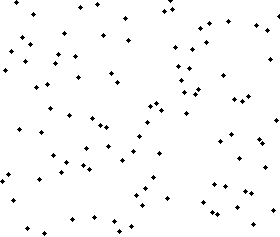
**Сортировка слиянием** (англ. *merge sort*) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Рисунок иллюстрирует сортировку слиянием.

**Алгоритм**

1. Сортируемый массив разбивается на две части примерно одинакового размера;
2. Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
3. Два упорядоченных массива половинного размера соединяются в один.

1.1. — 2.1. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).

3.1. Соединение двух упорядоченных массивов в один.  
Основную идею слияния двух отсортированных массивов можно объяснить на следующем примере. Пусть мы имеем два уже отсортированных по возрастанию подмассива. Тогда:  
3.2. Слияние двух подмассивов в третий результирующий массив.  
На каждом шаге мы берём меньший из двух первых элементов подмассивов и записываем его в результирующий массив. Счётчики номеров элементов результирующего массива и подмассива, из которого был взят элемент, увеличиваем на 1.  
3.3. «Прицепление» остатка.  
Когда один из подмассивов закончился, мы добавляем все оставшиеся элементы второго подмассива в результирующий массив.

**Листинг программы с расчетами.**

from queue import SimpleQueue

from typing import TypeVar

VT = TypeVar("VT")

class Queue(SimpleQueue):

    \_n\_op: int

    def \_\_init\_\_(self) -> None:

        super(SimpleQueue, self).\_\_init\_\_()

        self.\_n\_op = 0

    def push(self, el: VT) -> None:  # 1

        self.put(el)  # 1

        self.\_n\_op += 1

    def pop(self) -> VT:  # 1

        v = self.get()  # 1

        self.\_n\_op += 1

        return v

    def is\_empty(self) -> bool:

        return self.empty()

    @property

    def n\_op(self) -> int:

        return self.\_n\_op

    @property

    def size(self) -> int:  # 1

        return self.qsize()

def print\_queue(queue: Queue[VT]) -> None:

    elements = []

    for \_ in range(queue.size):

        el = queue.pop()

        elements.append(el)

        queue.push(el)

    print("Queue[" + ", ".join(map(str, elements)) + "]")

def rotate(queue: Queue[VT]) -> None:  # 2

    queue.push(queue.pop())  # 2

def seek(queue: Queue[VT], pos: int) -> VT:  # 4n + 1

    for \_ in range(pos):  # n \* (

        rotate(queue)  # 2

    # ) = 2n

    el = queue.pop()  # 2

    queue.push(el)  # 1

    for \_ in range(queue.size - pos - 1):  # (n - 1) \* (

        rotate(queue)  # 2

    # ) = 2n - 2

    return el

def pop\_by\_pos(queue: Queue[VT], pos: int) -> VT:  # 4n + 2

    for \_ in range(pos):  # n \* (

        rotate(queue)  # 2

    # ) = 2n

    el = queue.pop()  # 2

    for \_ in range(queue.size - pos):  # n \* (

        rotate(queue)  # 2

    # ) = 2n

    return el

def push\_by\_pos(queue: Queue[VT], el: VT, pos: int) -> None:  # 4n - 1

    if pos >= queue.size:  # 2

        queue.push(el)  # 1

        return

    for \_ in range(pos):  # n \* (

        rotate(queue)  # 2

    # ) = 2n

    queue.push(el)  # 1

    for \_ in range(queue.size - pos - 1):  # (n - 1) \* (

        rotate(queue)  # 2

    # ) = 2n - 2

def swap(queue: SimpleQueue, pos1: int, pos2: int):  # 16n + 2

    temp = pop\_by\_pos(queue, pos1)  # 4n + 2

    push\_by\_pos(queue, pop\_by\_pos(queue, pos2 - 1), pos1)  # 8n + 1

    push\_by\_pos(queue, temp, pos2)  # 4n - 1

def slice\_(queue: Queue[VT], l: int = 0, r: int = -1) -> Queue[VT]:  # 7n + 4

    buffer = Queue[VT]()  # 2

    if r == -1:  # 1

        r = queue.size - 1  # 3

    for \_ in range(l):  # n \* (

        rotate(queue)  # 2

    # ) = 2n

    for \_ in range(r - l + 1):  # n \* (

        el = queue.pop()  # 1

        buffer.push(el)  # 1

        queue.push(el)

    # ) = 3n

    for \_ in range(queue.size - r - 1):  # (n - 1) \* (

        rotate(queue)  # 2

    # ) = 2n - 2

    return buffer

def merge\_sort(queue: Queue[VT]) -> Queue[VT]: # 2n^2 + 2nlog(n) + 14n + 11

    if queue.size <= 1: # 2

        return queue

    left = slice\_(queue, 0, queue.size // 2 - 1) # 3n + 2

    right = slice\_(queue, queue.size // 2) # 4n + 2

    left = merge\_sort(left) # ~= nlog(n)

    right = merge\_sort(right) # ~= nlog(n)

    result = Queue[VT]() # 1

    while not left.is\_empty() and not right.is\_empty(): # n \* (

        if seek(left, 0) <= seek(right, 0): # 2n + 2

            result.push(left.pop()) # 2

        else:

            result.push(right.pop()) # 2

    # ) = 2n^2 + 4n

    while not left.is\_empty(): # m (where m << n) \* (

        result.push(left.pop()) # 2

    # ) ~= 2

    while not right.is\_empty(): # k (where k << n) \* (

        result.push(right.pop()) # 2

    # ) ~= 2

    for \_ in range(queue.size): # n \* (

        queue.pop() # 1

        queue.push(result.pop()) # 2

    # ) = 3n

    queue.\_n\_op += left.n\_op + right.n\_op

    return queue

if \_\_name\_\_ == "\_\_main\_\_":

    import sys

    import time

    from random import randint

    if len(sys.argv) < 2 or sys.argv[1] not in ["example", "tests"]:

        print(f"Usage: python3 {sys.argv[0]} [example/tests]")

        exit(1)

    if sys.argv[1] == "example":

        struct = Queue[int]()

        for \_ in range(20):

            struct.push(randint(-10000, 10000))

        merge\_sort(struct)

        print\_queue(struct)

    elif sys.argv[1] == "tests":

        tests = 10

        step = 1000

        for test\_num in range(1, tests + 1):

            struct = Queue[int]()

            for \_ in range(test\_num \* step):

                struct.push(randint(-10000, 10000))

            start\_time = time.time()

            merge\_sort(struct)

            total\_time = time.time() - start\_time

            print(f"Test: {test\_num}")

            print(f"Elements count: {test\_num \* step}")

            print(f"Total time: {total\_time}".replace(".", ","))

            print(f"N\_OP: {struct.n\_op}")

            print("-------------")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 1000 | 2033943 | 1000000 | 0,147462845 | 2085594 |
| 2000 | 8071874 | 4000000 | 0,575965166 | 8189228 |
| 3000 | 18111315 | 9000000 | 1,247863531 | 18301850 |
| 4000 | 32151737 | 16000000 | 2,192739487 | 32414492 |
| 5000 | 50192888 | 25000000 | 3,402697563 | 50536170 |
| 6000 | 72234620 | 36000000 | 4,822574377 | 72657086 |
| 7000 | 98276835 | 49000000 | 6,604558945 | 98779966 |
| 8000 | 128319464 | 64000000 | 8,604237556 | 128901272 |
| 9000 | 162362454 | 81000000 | 11,1317029 | 163031026 |
| 10000 | 200405765 | 100000000 | 13,57478762 | 201162346 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 13792915,57 | 6781369,239 | 0,975234187 | 0,479479707 |
| 14014517,91 | 6944864,439 | 0,985669728 | 0,48844653 |
| 14513859,11 | 7212327,13 | 0,98958933 | 0,491753566 |
| 14662816,75 | 7296808,443 | 0,991893912 | 0,493606378 |
| 14750910,77 | 7347111,971 | 0,993207204 | 0,494695186 |
| 14978435,65 | 7464892,646 | 0,994185481 | 0,49547817 |
| 14880151,09 | 7419117,675 | 0,994906548 | 0,496052003 |
| 14913519,38 | 7438195,375 | 0,995486403 | 0,496504022 |
| 14585589,93 | 7276514,719 | 0,99589911 | 0,496837945 |
| 14763086,61 | 7366597,756 | 0,996238954 | 0,497110926 |

**Таблица результата экспериментов и графики зависимостей**

**Скриншоты работы программы:**

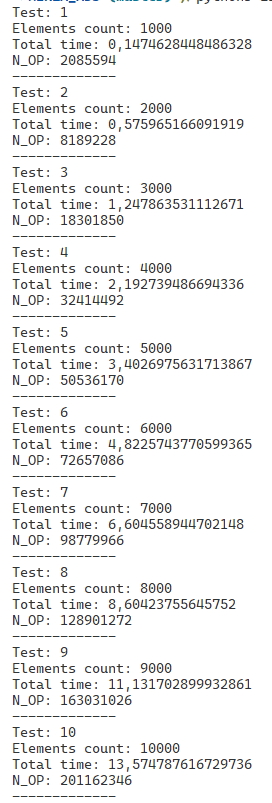


Рисунок 1. Скриншот работы программы в режиме tests.



Рисунок 2. Скриншот работы программы в режиме example.

**Выводы.**

В данной работе был реализован класс Queue для работы с очередью из библиотеки классов. Были реализованы вспомогательные операции и счетчик количества операций.

По результатам экспериментов было установлено, что алгоритм сортировки слиянием для очереди из библиотеки классов имеет зависимость от числа элементов

**Литература:**

1. Левитин А. В. Глава 6. Метод декомпозиции: Сортировка слиянием // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006.