|  |
| --- |
| 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 № 27**

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 № 27.**

**Реализация связи элементов линейного списка: Указатели**

**Способ организации линейного связанный список: Очередь с одной головой**

**Алгоритм сортировки: Быстрая по Хоару (без медианного (pivot) элемента)**

**Теория о сортировках.**

**Быстрая сортировка Хоара (QuickSort).**

Один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем O ( n log ⁡ n ) {\displaystyle O(n\log n)} обменов при упорядочении n {\displaystyle n} элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка» и «Шейкерная сортировка»), известного в том числе своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы (таким образом улучшение самого неэффективного прямого метода сортировки дало в результате один из наиболее эффективных улучшенных методов).

Рисунок иллюстрирует сортировку Хоара.

Шаги алгоритма:

1. Выбрать элемент из массива. Назовём его опорным.

2. Разбиение: перераспределение элементов в массиве таким образом, что элементы, меньшие опорного, помещаются перед ним, а большие или равные - после.

3. Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

**Листинг программы с расчетами.**

from typing import Generic, Optional, TypeVar

VT = TypeVar("VT")

class Node(Generic[VT]):

    \_value: VT

    next: Optional["Node[VT]"]

    def \_\_init\_\_(self, value: VT, next: Optional["Node[VT]"] = None):

        self.\_value = value

        self.next = next

    @property

    def value(self) -> VT:

        return self.\_value

    def \_\_str\_\_(self) -> str:

        return str(self.\_value)

class Queue(Generic[VT]):

    \_head: Optional[Node[VT]]

    \_size: int

    \_n\_op: int

    def \_\_init\_\_(self) -> None:

        self.\_head = None

        self.\_size = 0

        self.\_n\_op = 0

    def push(self, value: VT) -> None:  # 2n + 9

        node = Node(value)  # 2

        if self.is\_empty():  # 1

            self.\_head = node  # 1

        else:

            if self.\_head.next != None:

                tail = self.\_head.next  # 3

                while tail.next is not None:  # n \* (

                    tail = tail.next  # 2

                # ) = 2n

            else:

                tail = self.\_head  # 2

            tail.next = node  # 2

        self.\_size += 1  # 1

        self.\_n\_op += self.\_size \* 2 + 8

    def pop(self) -> VT:  # 7

        if self.is\_empty():  # 1

            raise Exception("Can't pop from empty queue!")

        node = self.\_head  # 1

        self.\_head = node.next  # 2

        self.\_size -= 1  # 1

        self.\_n\_op += 7

        return node.value

    def is\_empty(self) -> bool:  # 2

        return self.\_size == 0

    @property

    def head(self) -> VT:  # 2

        return self.\_head.value

    @property

    def size(self) -> int:  # 1

        return self.\_size

    @property

    def n\_op(self) -> int:

        return self.\_n\_op

def print\_queue(queue: Queue[VT]):

    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:  # 2n + 16

    queue.push(queue.pop())  # 2n + 16

def seek(queue: Queue[VT], pos: int) -> VT:  # 4n^2 + 32n + 2

    for \_ in range(pos):  # n \* (

        rotate(queue)  # 2n + 16

    # ) = 2n^2 + 16n

    el = queue.head  # 2

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

        rotate(queue)  # 2n + 16

    # ) = 2n^2 + 16n

    return el

def pop\_by\_pos(queue: Queue[VT], pos: int) -> VT:  # 4n^2 + 32n + 7

    for \_ in range(pos):  # n \* (

        rotate(queue)  # 2n + 16

    # ) = 2n^2 + 16n

    el = queue.pop()  # 7

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

        rotate(queue)  # 2n + 16

    # ) = 2n^2 + 16n

    return el

def push\_by\_pos(queue: Queue[VT], el: VT, pos: int) -> None:  # 4n^2 + 34n + 11

    if pos >= queue.size:  # 2

        queue.push(el)  # 2n + 9

        return

    for \_ in range(pos):  # n \* (

        rotate(queue)  # 2n + 16

    # ) = 2n^2 + 16n

    queue.push(el)  # 2n + 9

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

        rotate(queue)  # 2n + 16

    # ) = 2n^2 + 16n

def swap(queue: Queue, pos1: int, pos2: int):  # 16n^2 + 132n + 36

    temp = pop\_by\_pos(queue, pos1)  # 4n^2 + 32n + 7

    push\_by\_pos(queue, pop\_by\_pos(queue, pos2 - 1), pos1)  # 8n^2 + 66n + 18

    push\_by\_pos(queue, temp, pos2)  # 4n^2 + 34n + 11

def slice\_(queue: Queue[VT], l: int = 0, r: int = -1) -> Queue[VT]:  # 8n^2 + 57n + 6

    buffer = Queue[VT]()  # 2

    if r == -1:  # 1

        r = queue.size - 1  # 3

    for \_ in range(l):  # n \* (

        rotate(queue)  # 2n + 16

    # ) = 2n^2 + 16n

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

        el = queue.pop()  #  7

        buffer.push(el)  # 2n + 9

        queue.push(el)  # 2n + 9

    # ) = 4n^2 + 25n

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

        rotate(queue)  # 2n + 16

    # ) = 2n^2 + 16n

    return buffer

def quick\_sort(queue: Queue[VT]) -> Queue[VT]:  # 24n^3 + 200n^2 + 2nlog(n) + 74n + 2

    def \_quick\_sort(

        left: int, right: int

    ) -> None:  # 24n^3 + 200n^2 + 2nlog(n) + 74n + 2

        i = left  # 1

        j = right  # 1

        partition\_index = seek(queue, (i + j) // 2)  # 4n^2 + 32n + 2

        while i < j:  # n \* (

            while seek(queue, i) < partition\_index:  # 4n^2 + 32n + 3

                i += 1

            while seek(queue, j) > partition\_index:  # 4n^2 + 32n + 3

                j -= 1

            if i <= j:  # 1

                if i < j:  # 1

                    swap(queue, i, j)  # 16n^2 + 132n + 36

                i += 1  # 1

                j -= 1  # 1

        # ) = 24n^3 + 196n^2 + 42n

        if left < j:  # 1

            \_quick\_sort(left, j)  # ~= nlog(n)

        if i < right:  # 1

            \_quick\_sort(i, right)  # ~= nlog(n)

    \_quick\_sort(0, queue.size - 1)

    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))

        quick\_sort(struct)

        print\_queue(struct)

    elif sys.argv[1] == "tests":

        tests = 10

        step = 60

        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()

            quick\_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}")

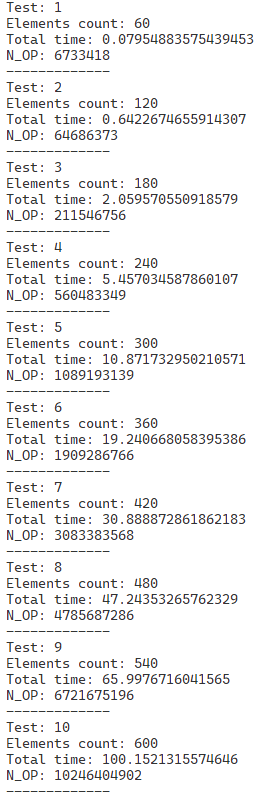
            print(f"N\_OP: {struct.n\_op}")

            print("-------------")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 60 | 5915151 | 216000 | 0,079548836 | 6733418 |
| 120 | 44374540 | 1728000 | 0,642267466 | 64686373 |
| 180 | 146482019 | 5832000 | 2,059570551 | 211546756 |
| 240 | 343341557 | 13824000 | 5,457034588 | 560483349 |
| 300 | 666057139 | 27000000 | 10,87173295 | 1089193139 |
| 360 | 1145732756 | 46656000 | 19,24066806 | 1909286766 |
| 420 | 1813472402 | 74088000 | 30,88887286 | 3083383568 |
| 480 | 2700380073 | 110592000 | 47,24353266 | 4785687286 |
| 540 | 3837559765 | 157464000 | 65,9976716 | 6721675196 |
| 600 | 5256115477 | 216000000 | 100,1521316 | 10246404902 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 74358735,37 | 2715313,153 | 0,8784767 | 0,032078805 |
| 69090436,67 | 2690467,901 | 0,685995173 | 0,026713509 |
| 71122603,21 | 2831658,278 | 0,692433303 | 0,027568374 |
| 62917240,45 | 2533243,977 | 0,612581191 | 0,024664426 |
| 61265038,64 | 2483504,711 | 0,611514263 | 0,024788992 |
| 59547451,92 | 2424863,828 | 0,600084166 | 0,02443635 |
| 58709568,66 | 2398533,619 | 0,588143629 | 0,024028149 |
| 57158724,61 | 2340891,838 | 0,564261706 | 0,023108906 |
| 58146896,27 | 2385902,353 | 0,570923119 | 0,023426303 |
| 52481314,13 | 2156718,95 | 0,512971674 | 0,021080565 |

**Таблица результата экспериментов и графики зависимостей**

**Скриншот работы программы:**

**Выводы.**

В данной работе был реализован класс Queue для работы с очередью на указателях с одной головой. Также были реализованы базовые операции над этой структурой.

По результатам экспериментов было установлено, что алгоритм сортировки Хоара для очереди на указателях с одной головой имеет зависимость от числа элементов

**Литература:**

1. Левитин А. В. Глава 4. Метод декомпозиции: Быстрая сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006.