|  |
| --- |
| 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 № 59**

Выполнил: студент 2 курса

Группы **\_\_\_\_\_\_\_\_\_\_\_**

Шифр \_\_\_\_\_\_\_\_

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

Проверил:

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

Москва 2021 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 №59.**

**Реализация связи элементов линейного списка: Указатели**

**Способ организации линейного связанный список: Очередь с 1 головой**

**Алгоритм сортировки: Простая вставка**

**Теория о сортировках.**

Сортировка вставками (*Insertion Sort*) — это простой алгоритм сортировки. Суть его заключается в том, что, на каждом шаге алгоритма мы берем один из элементов массива, находим позицию для вставки и вставляем. Стоит отметить, что массив из 1-го элемента считается отсортированным.  
  
Словесное описание алгоритма звучит довольно сложно, но на деле это самая простая в реализации сортировка. Каждый из нас, не зависимо от рода деятельности, применял алгоритм сортировки, просто не осознавал это. Например когда сортировали купюры в кошельке — берем 100 рублей и смотрим — идут 10, 50 и 500 рублёвые купюры. Вот как раз между 50 и 500 и вставляем нашу сотню:) Или приведу пример из всех книжек — игра в карточного «Дурака». Когда мы тянем карту из колоды, смотрим на наши разложенные по возрастанию карты и в зависимости от достоинства вытянутой карты помещаем карту в соответствующее место. Для большей наглядности приведу анимацию из википедии.

**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):

        self.\_head = None

        self.\_size = 0

        self.\_n\_op = 0

    def push(self, value: VT):  # 2n + 6

        node = Node(value)

        if self.is\_empty():  # 1

            self.\_head = node  # 1

        else:

            tail = self.\_head  # 2

            while tail.next:  # 2n

                tail = tail.next

            tail.next = node  # 2

        self.\_size += 1  # 1

        self.\_n\_op += (self.\_size - 1) \* 2 + 6

    def pop(self) -> VT:  # 4

        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 += 4

        return node.value

    def is\_empty(self) -> bool:

        return self.\_size == 0

    @property

    def head(self) -> VT:

        return self.\_head.value

    @property

    def size(self) -> int:

        return self.\_size

    @property

    def n\_op(self) -> int:

        return self.\_n\_op

def print\_queue(queue: Queue):

    elems = []

    for \_ in range(queue.size):

        el = queue.pop()

        elems.append(el)

        queue.push(el)

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

def print\_queue(queue: Queue):

    elems = []

    for \_ in range(queue.size):

        elems.append(queue.head)

        rotate(queue)

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

def rotate(queue: Queue):  # 2n + 10

    queue.push(queue.pop())

def seek(queue: Queue[VT], pos: int) -> VT:  # 2n^2 + 10n + 2

    for \_ in range(pos):  # (2n + 10)pos = 2n \* pos + 10pos

        rotate(queue)

    el = queue.head  # 2

    for \_ in range(

        queue.size - pos

    ):  # (2n + 10) \* (n - pos) =  2n^2 + 10n - 2n \* pos - 10pos

        rotate(queue)

    return el

def pop\_by\_pos(queue: Queue[VT], pos: int) -> VT:  # 2n^2 + 10n + 6

    for \_ in range(pos):  # (2n + 10)pos = 2n \* pos + 10pos

        rotate(queue)

    el = queue.pop()  # 6

    for \_ in range(

        queue.size - pos

    ):  # (2n + 10) \* (n - pos) =  2n^2 + 10n - 2n \* pos - 10pos

        rotate(queue)

    return el

def push\_by\_pos(queue: Queue[VT], el: VT, pos: int):  # 2n^2 + 8n - 4

    if pos >= queue.size:  # 2

        queue.push(el)  # 2n + 6

        return

    for \_ in range(pos):  # 2n \* pos + 10pos

        rotate(queue)

    queue.push(el)  # 6

    for \_ in range(

        queue.size - pos - 1

    ):  # (2n + 10) \* (n - pos - 1) =  2n^2 + 10n - 2n \* pos - 10pos - 2n - 10

        rotate(queue)

def swap(queue: Queue, pos1: int, pos2: int):  # 8n^2 + 36n + 4

    temp = pop\_by\_pos(queue, pos1)  # 2n^2 + 10n + 6

    push\_by\_pos(

        queue, pop\_by\_pos(queue, pos2 - 1), pos1

    )  # 2n^2 + 8n - 4 + 2n^2 + 10n + 6

    push\_by\_pos(queue, temp, pos2)  # 2n^2 + 8n - 4

def simple\_insertion\_sort(queue: Queue[VT]) -> Queue[VT]:

    for i in range(1, queue.size): # n \* (

        for j in range(i, 0, -1): # n \* (

            if seek(queue, j - 1) > seek(queue, j): # 4n^2 + 20n + 6

                swap(queue, j - 1, j) # 8n^2 + 36n + 5

            else:

                break

        # ) = 12n^3 + 56n^2 + 11n

    # ) = 12n^4 + 56n^3 + 11n^2

    return queue

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

    import time

    from random import randint

    tests = 10

    step = 40

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

        queue = Queue[int]()

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

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

        start\_time = time.time()

        simple\_insertion\_sort(queue)

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

        print(f"Test: {test\_num}")

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

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

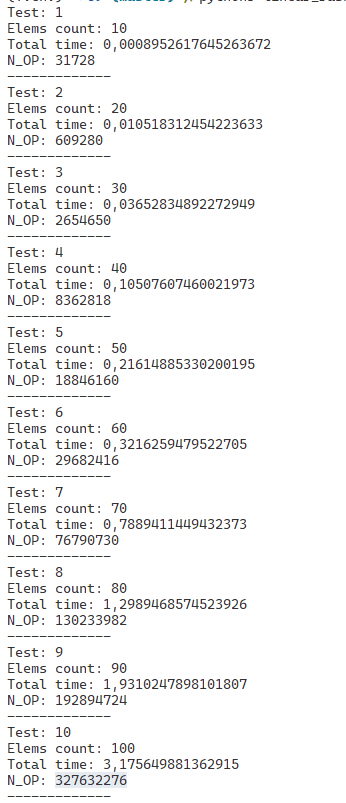
        print(f"N\_OP: {queue.n\_op}")

        print("-------------")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 10 | 177200 | 10000 | 0,000895262 | 31728 |
| 20 | 2372800 | 160000 | 0,010518312 | 609280 |
| 30 | 11242800 | 810000 | 0,036528349 | 2654650 |
| 40 | 34323200 | 2560000 | 0,105076075 | 8362818 |
| 50 | 82030000 | 6250000 | 0,216148853 | 18846160 |
| 60 | 167659200 | 12960000 | 0,321625948 | 29682416 |
| 70 | 307386800 | 24010000 | 0,788941145 | 76790730 |
| 80 | 520268800 | 40960000 | 1,298946857 | 130233982 |
| 90 | 828241200 | 65610000 | 1,93102479 | 192894724 |
| 100 | 1256120000 | 100000000 | 3,175649881 | 327632276 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 197930937,1 | 11169917,44 | 5,584972264 | 0,315179022 |
| 225587518 | 15211565,61 | 3,894432773 | 0,262605042 |
| 307782868,1 | 22174558,22 | 4,235134575 | 0,305124969 |
| 326650953,9 | 24363300,68 | 4,104262463 | 0,306116909 |
| 379506986,7 | 28915258,65 | 4,352610824 | 0,331632545 |
| 521286298,8 | 40295256,28 | 5,648435087 | 0,43662214 |
| 389619431 | 30433195,37 | 4,002915456 | 0,312667948 |
| 400531243,5 | 31533237,69 | 3,994877466 | 0,314510847 |
| 428912774,4 | 33976777,69 | 4,293747298 | 0,34013372 |
| 395547383 | 31489617,48 | 3,833932405 | 0,305220234 |

**Таблица результата экспериментов и графики зависимостей**

**Скриншот работы программы:**

**Выводы.**

По результатам экспериментов было установлено, что алгоритм сортировки простой вставкой для очереди с одной головой на указателях имеет зависимость от числа элементов. Чем больше количество чисел и меньше диапазон, тем ближе сложность будет приближаться к асимптотической оценке.

**Литература:**

1. Левитин А. В. Глава 7. Пространственно-временной компромисс: Пирамидальная сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006.