|  |
| --- |
| 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 № 13**

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 № 13.**

**Реализация связи элементов линейного списка: Массив**

**Способ организации линейного связанный список: Очередь**

**Алгоритм сортировки: Алгоритм Шелла**

**Теория о сортировках.**

**Сортировка Шелла** (англ. *Shell sort*) — алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами — это сортировка вставками с предварительными «грубыми» проходами. Аналогичный метод усовершенствования пузырьковой сортировки называется сортировка расчёской.

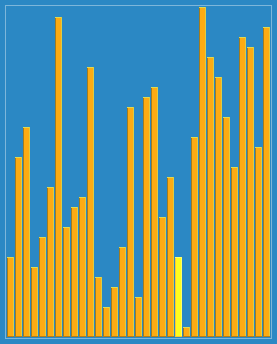


Рисунок 1. Иллюстрация сортировки алгоритмом Шелла

**Алгоритм**

При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии d {\displaystyle d} (о выборе значения d {\displaystyle d} см. ниже). После этого процедура повторяется для некоторых меньших значений d {\displaystyle d} , а завершается сортировка Шелла упорядочиванием элементов при d = 1 {\displaystyle d=1} (то есть обычной сортировкой вставками). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, пузырьковой, каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, а при сортировке Шелла это число может быть больше).

Невзирая на то, что сортировка Шелла во многих случаях медленнее, чем быстрая сортировка, она имеет ряд преимуществ:

* отсутствие потребности в памяти под стек;
* отсутствие деградации при неудачных наборах данных — быстрая сортировка легко деградирует до O(n²), что хуже, чем худшее гарантированное время для сортировки Шелла.

**Листинг программы с расчетами.**

from typing import Generic, List, TypeVar

VT = TypeVar("VT")

class Queue(Generic[VT]):

    \_queue: List[VT]

    \_bias: int

    \_size: int

    \_max\_size: int

    \_n\_op: int

    def \_\_init\_\_(self, max\_size: int = 10000) -> None:

        self.\_queue = [None for \_ in range(max\_size)]

        self.\_max\_size = max\_size

        self.\_size = 0

        self.\_n\_op = 0

        self.\_bias = 0

    def push(self, value: VT) -> None:  # 8

        if self.\_size == self.\_max\_size:  # 3

            raise RuntimeError("Queue overflow")

        self.\_queue[(self.\_size + self.\_bias) % self.\_max\_size] = value  # 3

        self.\_size += 1  # 2

        self.\_n\_op += 8

    def pop(self) -> VT:  # 12

        if self.is\_empty():

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

        el = self.\_queue[self.\_bias]  # 4

        self.\_size -= 1  # 2

        self.\_bias = (self.\_bias + 1) % self.\_max\_size  # 6

        self.\_n\_op += 12

        return el

    def is\_empty(self) -> bool:  # 2

        return self.\_size == 0

    @property

    def size(self) -> int:  # 1

        return self.\_size

    @property

    def tail(self) -> VT:  # 8

        return self.\_queue[(self.\_size - 1 + self.\_bias) % self.\_max\_size]

    @property

    def head(self) -> VT:  # 3

        return self.\_queue[self.\_bias]

    @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:  # 20

    queue.push(queue.pop())  # 20

def seek(queue: Queue[VT], pos: int) -> VT:  # 40n + 20

    for \_ in range(pos):  # n \* (

        rotate(queue)  # 20

    # ) = 20n

    el = queue.pop()  # 12

    queue.push(el)  # 8

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

        rotate(queue)  # 20

    # ) = 20n

    return el

def pop\_by\_pos(queue: Queue[VT], pos: int) -> VT:  # 40n + 12

    for \_ in range(pos):  # n \* (

        rotate(queue)  # 20

    # ) = 20n

    el = queue.pop()  # 12

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

        rotate(queue)  # 20

    # ) = 20n

    return el

def push\_by\_pos(queue: Queue[VT], el: VT, pos: int) -> None:  # 40n + 10

    if pos >= queue.size:  # 2

        queue.push(el)  # 8

        return

    for \_ in range(pos):  # n \* (

        rotate(queue)  # 20

    # ) = 20n

    queue.push(el)  # 8

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

        rotate(queue)  # 20

    # ) = 20n

def swap(queue: Queue[VT], pos1: int, pos2: int) -> None:  #  160n + 46

    temp = pop\_by\_pos(queue, pos1)  # 40n + 13

    push\_by\_pos(queue, pop\_by\_pos(queue, pos2 - 1), pos1)  # 80 + 23

    push\_by\_pos(queue, temp, pos2)  # 40n + 10

def slice\_(queue: Queue[VT], l: int = 0, r: int = -1) -> Queue[VT]:  # 68n + 6

    buff = Queue[VT](max\_size=queue.size)  # 2

    if r == -1:  # 1

        r = queue.size - 1  # 3

    for \_ in range(l):  # n \* (

        rotate(queue)  # 20

    # ) = 20n

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

        buff.push(queue.head)  # 8

        rotate(queue)  # 20

    # ) = 28n

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

        rotate(queue)  # 20

    # ) = 20n

    return buff

def shell\_sort(queue: Queue): # 240n^2 \* log(n) + 89n \* log(n) + 6

    last\_index = queue.size - 1  # 3

    step = queue.size // 2  # 3

    while step > 0:  # log(n) \* (

        for i in range(step, last\_index + 1, 1):  # n \* (

            j = i  # 1

            delta = j - step  # 2

            while delta >= 0 and seek(queue, delta) > seek(

                queue, j

            ):  # k (where k << n) \* ( 80n + 40 +

                swap(queue, delta, j)  # 160n + 46

                j = delta  # 1

                delta = j - step  # 2

            # ) = 240n + 89

        # ) = 240n^2 + 89n

        step //= 2

    # ) = 240n^2 \* log(n) + 89n \* log(n)

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))

        shell\_sort(struct)

        print\_queue(struct)

    elif sys.argv[1] == "tests":

        tests = 10

        step = 100

        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()

            shell\_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}")

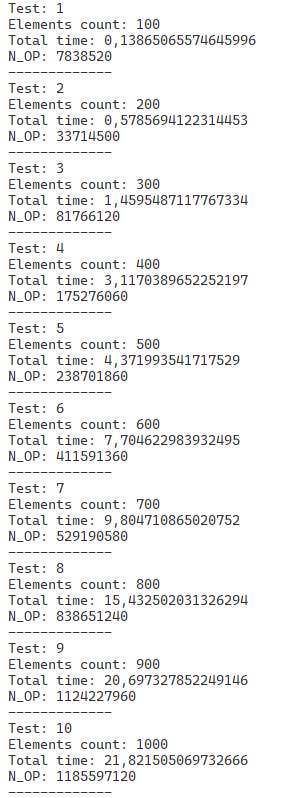
            print(f"N\_OP: {struct.n\_op}")

            print("-------------")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 100 | 16004391 | 66439 | 0,138650656 | 7838520 |
| 200 | 73517086 | 305754 | 0,578569412 | 33714500 |
| 300 | 177962199 | 740594 | 1,459548712 | 81766120 |
| 400 | 332231805 | 1383017 | 3,117038965 | 175276060 |
| 500 | 538346040 | 2241446 | 4,371993542 | 238701860 |
| 600 | 797862760 | 3322375 | 7,704622984 | 411591360 |
| 700 | 1112051243 | 4631093 | 9,804710865 | 529190580 |
| 800 | 1481982959 | 6172068 | 15,43250203 | 838651240 |
| 900 | 1908585153 | 7949163 | 20,69732785 | 1124227960 |
| 1000 | 2392675189 | 9965784 | 21,82150507 | 1185597120 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 115429610,4 | 479179,5721 | 2,04176186 | 0,008475906 |
| 127067011,3 | 528465,9734 | 2,180577676 | 0,009068924 |
| 121929605,9 | 507412,7887 | 2,176478463 | 0,009057464 |
| 106585708 | 443695,7657 | 1,895477368 | 0,007890507 |
| 123135140,8 | 512682,8413 | 2,255307271 | 0,009390149 |
| 103556366,3 | 431218,3394 | 1,93848277 | 0,008072023 |
| 113420095,6 | 472333,5046 | 2,101419196 | 0,008751277 |
| 96029986,34 | 399939,5528 | 1,767102806 | 0,007359517 |
| 92214085,17 | 384067,1038 | 1,697685186 | 0,007070775 |
| 109647578,5 | 456695,5511 | 2,018118253 | 0,008405709 |

**Таблица результата экспериментов и графики зависимостей**

**Скриншот работы программы:**

**Выводы.**

В результате данной работы был реализован класс Queue для работы с очередью на массиве. Также были реализованы базовые операции над ней и рассчитана их алгоритмическая сложность.

По результатам экспериментов было установлено, что алгоритм сортировки Шелла для очереди на массиве имеет зависимость от числа элементов

**Литература:**

1. Д. Кнут. Искусство программирования. Том 3. Сортировка и поиск, 2-е изд.