|  |
| --- |
| 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 № 80**

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 № 80.**

**Реализация связи элементов линейного списка: Библиотека классов**

**Способ организации линейного связанный список: Стек**

**Алгоритм сортировки: Пузырковая**

**Теория о сортировках.**

**Сортировка простыми обменами**, **сортировка пузырьком** (англ. *bubble sort*) — простой алгоритм сортировки. Для понимания и реализации этот алгоритм — простейший, но эффективен он лишь для небольших массивов.

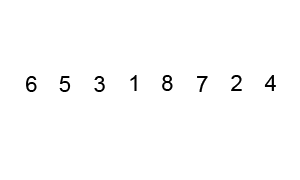


Рисунок 1. Иллюстрация алгоритма сортировки пузырьком

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется перестановка элементов. Проходы по массиву повторяются N − 1 {\displaystyle N-1} раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает - массив отсортирован. При каждом проходе алгоритма по внутреннему циклу очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции, как пузырёк в воде - отсюда и название алгоритма).

**Листинг программы с расчетами.**

from typing import Generic, List, TypeVar

VT = TypeVar("VT")

class Stack(Generic[VT]):

    \_stack: List[VT]

    \_size: int

    \_n\_op: int

    def \_\_init\_\_(self, max\_size: int = 10000):

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

        self.\_n\_op = 0

        self.\_size = 0

    def push(self, el: VT):  # 5

        self.\_stack[self.size] = el  # 3

        self.\_size += 1  # 2

        self.\_n\_op += 5

    def pop(self) -> VT:  # 9

        if self.is\_empty():  # 2

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

        el = self.\_stack[self.\_size - 1]  #  5

        self.\_size -= 1  # 2

        self.\_n\_op += 9

        return el

    def is\_empty(self) -> bool:  # 2

        return self.\_size == 0

    @property

    def n\_op(self) -> int:

        return self.\_n\_op

    @property

    def size(self) -> int:  # 1

        return self.\_size

    @property

    def top(self) -> VT:  # 1

        return self.\_stack[self.\_size - 1]

def print\_stack(stack: Stack[VT]) -> None:

    buffer = Stack[VT](max\_size=stack.size)

    elements = []

    for \_ in range(stack.size):

        el = stack.pop()

        elements.append(el)

        buffer.push(el)

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

    for \_ in range(buffer.size):

        stack.push(buffer.pop())

def seek(stack: Stack[VT], pos: int) -> VT:  # 28n - 22

    buffer = Stack[VT](max\_size=stack.size)  # 2

    stack\_size = stack.size  # 2

    for \_ in range(stack\_size - pos - 1):  # (n - 1) \* (

        buffer.push(stack.pop())  # 14

    # ) = 14n - 14

    el = stack.top  # 2

    for \_ in range(stack\_size - pos - 1):  # (n - 1) \* (

        stack.push(buffer.pop())  # 14

    # ) = 14n - 14

    stack.\_n\_op += buffer.n\_op

    return el

def push\_by\_pos(stack: Stack[VT], el: VT, pos: int) -> None:  # 28n + 9

    buffer = Stack[VT](max\_size=stack.size)  # 2

    stack\_size = stack.size  # 2

    for \_ in range(stack\_size - pos):  # n \* (

        buffer.push(stack.pop())  # 14

    # ) = 14n

    stack.push(el)  # 5

    for \_ in range(stack\_size - pos):  # n \* (

        stack.push(buffer.pop())  # 14

    # ) = 14n

    stack.\_n\_op += buffer.n\_op

def pop\_by\_pos(stack: Stack[VT], pos: int) -> VT:  # 28n - 15

    buffer = Stack[VT](max\_size=stack.size)  # 2

    stack\_size = stack.size  # 2

    for \_ in range(stack\_size - pos - 1):  # (n - 1) \* (

        buffer.push(stack.pop())  # 14

    # )= 14n - 14

    el = stack.pop()  # 9

    for \_ in range(stack\_size - pos - 1):  # (n - 1) \* (

        stack.push(buffer.pop())  # 14

    # ) = 14n - 14

    stack.\_n\_op += buffer.n\_op

    return el

def swap(stack: Stack[VT], pos1: int, pos2: int) -> None:  # 112n - 12

    temp = pop\_by\_pos(stack, pos1)  #  28n - 15

    push\_by\_pos(stack, pop\_by\_pos(stack, pos2 - 1), pos1)  # 56n - 6

    push\_by\_pos(stack, temp, pos2)  # 28n + 9

def reverse(stack: Stack[VT]) -> Stack[VT]:  # 14n + 2

    out = Stack[VT](max\_size=stack.size)  # 2

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

        out.push(stack.pop())  # 14

    # ) = 14n

    out.\_n\_op += stack.n\_op

    return out

def slice\_(stack: Stack[VT], l: int = 0, r: int = -1) -> Stack[VT]:  # 62n + 18

    stack\_size = stack.size  # 2

    slice\_stack = Stack[VT](max\_size=stack.size)  # 2

    buffer = Stack[VT](max\_size=stack.size)  # 2

    if r == -1:  # 1

        r = stack.size - 1  # 3

    for \_ in range(stack\_size - r - 1):  # (n - 1) \* (

        buffer.push(stack.pop())  # 14

    # ) = 14n - 14

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

        slice\_stack.push(stack.top)  # 6

        buffer.push(stack.pop())  # 14

    # ) = 20n + 20

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

        stack.push(buffer.pop())  # 14

    # ) = 14n

    slice\_stack.\_n\_op += buffer.n\_op

    slice\_stack.\_n\_op += stack.n\_op

    return reverse(slice\_stack)  # 14n + 2

def bubble\_sort(queue: Stack[VT]) -> Stack[VT]:

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

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

            if seek(queue, i) < seek(queue, j): # 56n - 44

                swap(queue, i, j)  # 112n - 12

        # ) = 168n^2 - 56n + 1

    # ) = 168n^3 - 56n^2 + n + 1

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 = Stack[int]()

        for \_ in range(20):

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

        bubble\_sort(struct)

        print\_stack(struct)

    elif sys.argv[1] == "tests":

        tests = 10

        step = 100

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

            struct = Stack[int]()

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

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

            start\_time = time.time()

            bubble\_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 |
| 50 | 20860050 | 125000 | 0,072573662 | 3197822 |
| 100 | 167440101 | 1000000 | 0,583374262 | 27276364 |
| 150 | 565740151 | 3375000 | 1,945456028 | 90327994 |
| 200 | 1341760201 | 8000000 | 4,272730589 | 201672568 |
| 250 | 2621500251 | 15625000 | 8,745773792 | 415672198 |
| 300 | 4530960301 | 27000000 | 14,96338105 | 691673404 |
| 350 | 7196140351 | 42875000 | 24,85098124 | 1137710938 |
| 400 | 10743040401 | 64000000 | 37,39787912 | 1710316592 |
| 450 | 15297660451 | 91125000 | 51,29157543 | 2362472206 |
| 500 | 20986000501 | 125000000 | 72,97471738 | 3276261052 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 287432788,7 | 1722387,942 | 6,523205482 | 0,039089105 |
| 287020034,9 | 1714165,443 | 6,138651801 | 0,036661778 |
| 290800790,6 | 1734811,762 | 6,263176297 | 0,037363832 |
| 314028739,5 | 1872338,972 | 6,653161678 | 0,039668261 |
| 299744804 | 1786577,194 | 6,306652847 | 0,037589716 |
| 302803242,5 | 1804405,028 | 6,550722169 | 0,039035764 |
| 289571678,6 | 1725283,988 | 6,325104304 | 0,037685319 |
| 287263359,7 | 1711326,992 | 6,281316834 | 0,037419973 |
| 298248987,7 | 1776607,547 | 6,47527637 | 0,038571882 |
| 287579058,3 | 1712922,016 | 6,405472631 | 0,038153248 |

**Таблица результата экспериментов и графики зависимостей**

**Скриншот работы программы:**

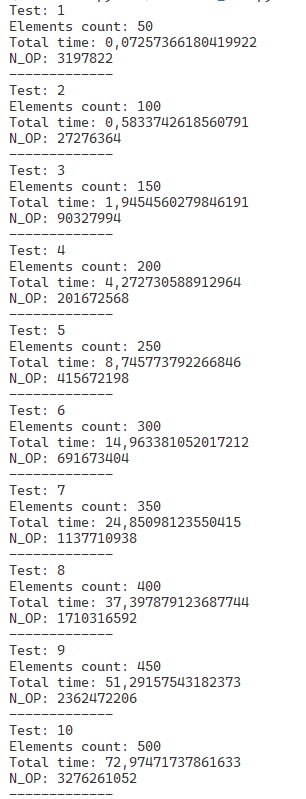


Рисунок 3. Пример вывода программы при выполнении сортировки

**Выводы.**

В данной работе был реализован класс Stack для удобной работы со стеком из библиотеки классов (list в языке python). Так же были реализованы базовые операции над этой структурой и вспомогательные операции для сортировки.

По результатам экспериментов было установлено, что алгоритм сортировки **пузырьком** для **стека из библиотеки классов** имеет **кубическую** зависимость от числа элементов.

**Литература:**

* Левитин А. В. Глава 3. Метод грубой силы: Пузырьковая сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006.