|  |
| --- |
| 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 № 55**

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 № 55.**

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

**Способ организации линейного связанный список: Очередь**

**Алгоритм сортировки: Пирамидальная сортировка**

**Теория о сортировках.**

**Пирамидальная сортировка** (англ. *Heapsort*, «Сортировка кучей») — алгоритм сортировки, работающий в худшем, в среднем и в лучшем случае (то есть гарантированно) за O ( n log ⁡ n ) {\displaystyle O(n\log n)} операций при сортировке n {\displaystyle n} элементов. Количество применяемой служебной памяти не зависит от размера массива (то есть O ( 1 ) {\displaystyle O(1)} ). Может рассматриваться как усовершенствованная сортировка пузырьком, в которой элемент всплывает (min-heap) / тонет (max-heap) по многим путям.

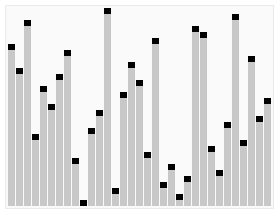


Рисунок 1. Визуализация работы алгоритма

**Алгоритм**

Сортировка пирамидой использует бинарное сортирующее дерево. Сортирующее дерево — это такое дерево, у которого выполнены условия:

1. Каждый лист имеет глубину либо d {\displaystyle d} d, либо d − 1 {\displaystyle d-1} d - 1, d {\displaystyle d} d — максимальная глубина дерева.
2. Значение в любой вершине не меньше (другой вариант — не больше) значения её потомков

Удобная структура данных для сортирующего дерева — такой массив Array, что Array[0] — элемент в корне, а потомки элемента Array[i] являются Array[2i+1] и Array[2i+2].

Алгоритм сортировки будет состоять из двух основных шагов:

1. Выстраиваем элементы массива в виде сортирующего дерева.

Этот шаг требует операций

2. Будем удалять элементы из корня по одному за раз и перестраивать дерево. То есть на первом шаге обмениваем Array[0] и Array[n-1], преобразовываем Array[0], Array[1], … , Array[n-2] в сортирующее дерево. Затем переставляем Array[0] и Array[n-2], преобразовываем Array[0], Array[1], … , Array[n-3] в сортирующее дерево. Процесс продолжается до тех пор, пока в сортирующем дереве не останется один элемент. Тогда Array[0], Array[1], … , Array[n-1] — упорядоченная последовательность.

Этот шаг требует O ( n log ⁡ n ) {\displaystyle O(n\log n)} операций.

**Листинг программы с расчетами.**

from queue import SimpleQueue

N\_OP = 0

def print\_queue(queue: SimpleQueue):

    elems = []

    for \_ in range(queue.qsize()):

        el = queue.get()

        elems.append(el)

        queue.put(el)

    print(", ".join(map(str, elems)))

def rotate(queue: SimpleQueue): # 2

    global N\_OP

    queue.put(queue.get()) # 2

    N\_OP += 2

def seek(queue: SimpleQueue, pos: int): # 2pos + 3 + 2n - 2pos - 2 = 2n + 1

    global N\_OP

    for \_ in range(pos): # 2pos

        rotate(queue) # 2

    el = queue.get() # 2

    queue.put(el) # 1

    for \_ in range(queue.qsize() - pos - 1): # 2n - 2pos - 2

        rotate(queue) # 2

    N\_OP += queue.qsize() \* 2 + 1

    return el

def pop\_by\_pos(queue: SimpleQueue, pos: int): # 2pos + 2 + 2n - 2pos = 2n + 2

    global N\_OP

    for \_ in range(pos): # 2pos

        rotate(queue) # 2

    el = queue.get() # 2

    for \_ in range(queue.qsize() - pos): # 2n - 2pos

        rotate(queue) # 2

    N\_OP += queue.qsize() \* 2 + 2

    return el

def push\_by\_pos(queue: SimpleQueue, el, pos: int): # 2 + 2pos + 1 + 2n - 2pos - 2 = 2n + 1

    global N\_OP

    if pos >= queue.qsize(): # 2

        queue.put(el) # 1

        N\_OP += 3

        return

    for \_ in range(pos): # 2pos

        rotate(queue) # 2

    queue.put(el) # 1

    for \_ in range(queue.qsize() - pos - 1): # 2n - 2pos - 2

        rotate(queue) # 2

    N\_OP += queue.qsize() \* 2 + 1

def swap(queue: SimpleQueue, pos1: int, pos2: int): # 8n + 7

    temp = pop\_by\_pos(queue, pos1) # 2n + 3

    push\_by\_pos(queue, pop\_by\_pos(queue, pos2 - 1), pos1) # 2n + 2 + 2n + 1 = 4n + 3

    push\_by\_pos(queue, temp, pos2) # 2n + 1

def heapify(queue: SimpleQueue, n: int, i: int): # 7 + 8n + 10 + 8n + 7 + log(n) = 16n + log(n) + 14

    largest = i # 1

    l = 2 \* i + 1 # 3

    r = 2 \* i + 2 # 3

    if l < n and seek(queue, i) < seek(queue, l): # 4n + 4

        largest = l # 1

    if r < n and seek(queue, largest) < seek(queue, r): # 4n + 4

        largest = r # 1

    if largest != i: # 1

        swap(queue, i, largest) # 8n + 7

        heapify(queue, n, largest) # ~log(n)

def heap\_sort(queue: SimpleQueue): #32n^2 + nlog(n) + 21n + 1

    n = queue.qsize() # 1

    for i in range(n // 2 - 1, -1, -1): # (n // 2) \* (16n + log(n) + 14) = 8n^2 + nlog(n)/2 + 7n

        heapify(queue, n, i)

    for i in range(n-1, 0, -1): # n \* (8n + 7 + 16n + log(n) + 14) = 24n^2 + nlog(n) + 14n

        swap(queue, 0, i)

        heapify(queue, i, 0)

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

    import time

    from random import randint

    tests = 10

    step = 200

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

        N\_OP = 0

        queue = SimpleQueue[int]()

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

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

        start\_time = time.time()

        heap\_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}")

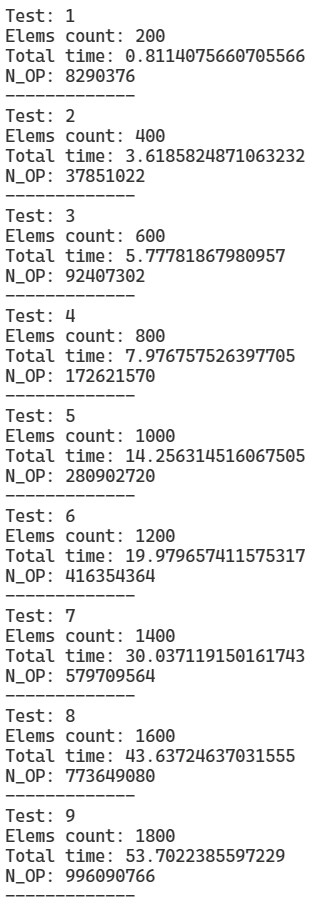
        print(f"N\_OP: {N\_OP}")

        print("-------------")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 200 | 1285729,771 | 40000 | 0,7128 | 8240908 |
| 400 | 5131858,542 | 160000 | 3,1212 | 38416584 |
| 600 | 11538138,29 | 360000 | 5,8203 | 92556004 |
| 800 | 20504516,08 | 640000 | 8,1740 | 172608754 |
| 1000 | 32030966,78 | 1000000 | 13,6066 | 279979104 |
| 1200 | 46117475,58 | 1440000 | 21,0608 | 416666246 |
| 1400 | 62764032,7 | 1960000 | 30,1814 | 580023198 |
| 1600 | 81970631,17 | 2560000 | 51,7764 | 774039492 |
| 1800 | 103737265,8 | 3240000 | 66,4470 | 996889822 |
| 2000 | 128063932,6 | 4000000 | 71,2460 | 1246322838 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 1803773,529 | 56116,72278 | 0,156017974 | 0,004853834 |
| 1644194,074 | 51262,335 | 0,133584458 | 0,004164868 |
| 1982395,803 | 61852,48183 | 0,124661154 | 0,003889537 |
| 2508504,537 | 78297,03939 | 0,118791867 | 0,003707807 |
| 2354075,727 | 73493,74568 | 0,114404848 | 0,003571695 |
| 2189730,475 | 68373,47109 | 0,110682053 | 0,003456003 |
| 2079560,017 | 64940,65882 | 0,108209521 | 0,003379175 |
| 1583165,905 | 49443,37575 | 0,105899805 | 0,003307325 |
| 1561203,151 | 48760,6664 | 0,104060914 | 0,003250108 |
| 1797489,439 | 56143,50279 | 0,102753419 | 0,003209441 |

**Таблица результата экспериментов и графики зависимостей**

**Скриншот работы программы:**

**Выводы.**

По результатам экспериментов было установлено, что пирамидальный алгоритм сортировки (Heap sort) для очереди из библиотеки классов зависимость от числа элементов

**Литература:**

1. Левитин А. В. Глава 6. Метод преобразования: Пирамиды и пирамидальная сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006.