|  |
| --- |
| 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 № 53**

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 № 53.**

**Реализация связи элементов линейного списка: Массив**

**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.

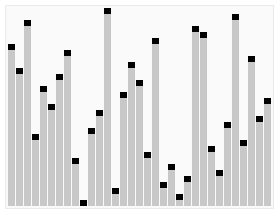


Рисунок 1. Визуализация работы алгоритма

**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

from typing import Generic, List, TypeVar

VT = TypeVar("VT")

class Queue(Generic[VT]):

    \_queue: List[VT]

    \_nop: int

    def \_\_init\_\_(self):

        self.\_queue = []

        self.\_nop = 0

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

        self.\_nop += self.size + 2

        self.\_queue = self.\_queue.copy()

        self.\_queue.append(0)

        # Имитация работы со списком как с массивом.

        # Создание массива на элемент больше и копирование элементов

        self.\_queue[-1] = value

    def pop(self) -> VT:  # n + 2

        self.\_nop += self.size + 2

        if self.is\_empty():

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

        el = self.\_queue[0]

        self.\_queue = self.\_queue[1:]  # создание нового списка на элемент меньше

        return el

    def is\_empty(self) -> bool:  # 1

        return len(self.\_queue) == 0

    @property

    def size(self) -> int:  # 1

        return len(self.\_queue)

    @property

    def tail(self) -> VT:  # 1

        return self.\_queue[-1]

    @property

    def head(self) -> VT:  # 1

        return self.\_queue[0]

    @property

    def n\_op(self) -> int:

        return self.\_nop

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 rotate(queue: Queue):  # 2n + 4

    queue.push(queue.pop())

def seek(

    queue: Queue, pos: int

):  # 2n \* pos + 4pos + 4n + 8 + 2n^2 + 8n - 2n \* pos - 4pos - 2n - 4 = 2n^2

    for \_ in range(pos):  # pos \* (2n + 4) = 2n \* pos + 4pos

        rotate(queue)

    el = queue.pop()  # n + 2

    queue.push(el)  # n + 2

    for \_ in range(

        queue.size - pos - 1

    ):  # (n - pos - 1) \* (2n + 4) = 2n^2 + 8n - 2n \* pos - 4pos - 2n - 4

        rotate(queue)

    return el

def pop\_by\_pos(queue: Queue, pos: int):  # 2n^2 + 3n - 2

    for \_ in range(pos):  # 2n \* pos + 4pos

        rotate(queue)

    el = queue.pop()  # n + 2

    for \_ in range(queue.size - pos):  # (n - pos - 1) \* (2n + 4)

        rotate(queue)

    return el

def push\_by\_pos(queue: Queue, el, pos: int):  # 2n^2 + 3n - 2

    if pos >= queue.size:  # 1

        queue.push(el)  # n + 2

        return

    for \_ in range(pos):  # 2n \* pos + 4pos

        rotate(queue)

    queue.push(el)  # n + 2

    for \_ in range(queue.size - pos - 1):  # (n - pos - 1) \* (2n + 4)

        rotate(queue)

def swap(queue: Queue, pos1: int, pos2: int):  # 6n^2 + 9n - 7

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

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

    push\_by\_pos(queue, temp, pos2)  # 2n^2 + 3n - 2

def heapify(queue: Queue, n: int, i: int):  #  14n^2 + nlog(n) + 9n + 9

    largest = i  # 1

    l = 2 \* i + 1  # 3

    r = 2 \* i + 2  # 3

    if l < n and seek(queue, i) < seek(queue, l):  # 4n^2 + 3

        largest = l  # 1

    if r < n and seek(queue, largest) < seek(queue, r):  # 4n^2 + 3

        largest = r  # 1

    if largest != i:  # 1

        swap(queue, i, largest)  # 6n^2 + 9n - 7

        heapify(queue, n, largest)  # ~= nlog(n)

def heap\_sort(queue: Queue):  # 27n^3 + 2n^2 \* log(n) + 23n^2 + 7n

    n = queue.size  # 1

    for i in range(

        n // 2 - 1, -1, -1

    ):  # (n // 2) \* (

        heapify(queue, n, i) # 14n^2 + nlog(n) + 9n + 9

    # ) = 7n^3 + n^2log(n) + 5n^2 + 5n

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

        swap(queue, 0, i)  #  6n^2 + 9n - 7

        heapify(queue, i, 0)  # 14n^2 + nlog(n) + 9n + 9

    # ) = 20n^3 + n^2log(n) + 18n^2 + 2n

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

    import time

    from random import randint

    tests = 10

    step = 30

    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()

        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: {queue.n\_op}")

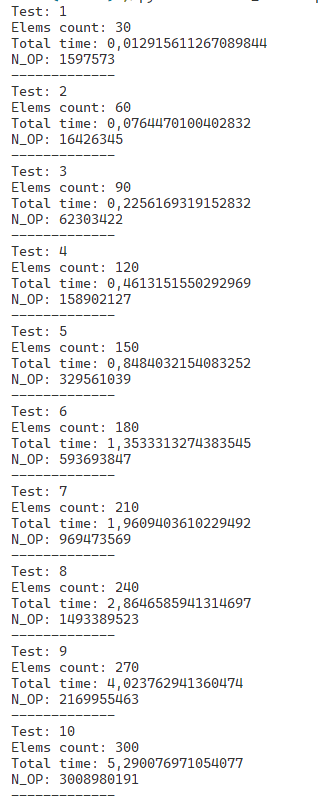
        print("-------------")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 30 | 752569 | 27000 | 0,012915611 | 1597573 |
| 60 | 5928023 | 216000 | 0,07644701 | 16426345 |
| 90 | 19901589 | 729000 | 0,225616932 | 62303422 |
| 120 | 47047920 | 1728000 | 0,461315155 | 158902127 |
| 150 | 91741474 | 3375000 | 0,848403215 | 329561039 |
| 180 | 158356602 | 5832000 | 1,353331327 | 593693847 |
| 210 | 251267590 | 9261000 | 1,960940361 | 969473569 |
| 240 | 374848680 | 13824000 | 2,864658594 | 1493389523 |
| 270 | 533474083 | 19683000 | 4,023762941 | 2169955463 |
| 300 | 731517982 | 27000000 | 5,290076971 | 3008980191 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 58268153,38 | 2090493,391 | 0,471070066 | 0,016900636 |
| 77544205,93 | 2825486,568 | 0,360885071 | 0,013149608 |
| 88209641,71 | 3231140,473 | 0,319430107 | 0,011700802 |
| 101986505,1 | 3745812,339 | 0,296081124 | 0,010874618 |
| 108134283,8 | 3978061,302 | 0,278374757 | 0,010240895 |
| 117012440,7 | 4309365,993 | 0,266731081 | 0,009823245 |
| 128136273,2 | 4722734,145 | 0,259179412 | 0,009552607 |
| 130852828,7 | 4825705,942 | 0,251005297 | 0,009256795 |
| 132580892,7 | 4891689,766 | 0,245845637 | 0,009070693 |
| 138281160,3 | 5103895,491 | 0,243111598 | 0,00897314 |

**Таблица результата экспериментов и графики зависимостей**

**Скриншот работы программы:**



**Выводы.**

По результатам экспериментов было установлено, что пирамидальный алгоритм сортировки (Heap sort) для очереди на массиве имеет 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.