|  |
| --- |
| 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 № 99**

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 №99.**

**Реализация связи элементов линейного списка: Указатели**

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

**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.

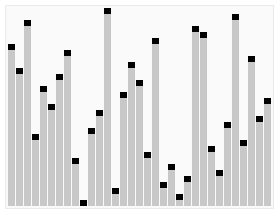


Рисунок 1. Визуализация работы алгоритма

**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 typing import Generic, TypeVar, Optional

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 heapify(queue: Queue, n: int, i: int):

    # (1 + 3 + 3 + 4n^2 + 20n + 5 + 1 + 4n^2 + 20n + 5 + 1 + 1 + 8n^2 + 36n + 4) \* log(n \* log(n)) =

    # = (16n^2 + 76n + 24) \* log(n \* log(n))

    largest = i # 1

    l = 2 \* i + 1 # 3

    r = 2 \* i + 2 # 3

    if l < n and seek(queue, i) < seek(queue, l): # 1 + 1 + 2n^2 + 10n + 1 + 2n^2 + 10n + 2 = 4n^2 + 20n + 5

        largest = l # 1

    if r < n and seek(queue, largest) < seek(queue, r): # 1 + 1 + 2n^2 + 10n + 1 + 2n^2 + 10n = 4n^2 + 20n + 5

        largest = r # 1

    if largest != i: # 1

        swap(queue, i, largest) # 8n^2 + 36n + 4

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

def heap\_sort(queue: Queue):

    # 1 + 8n^3 + 38n^2 + nlog(n \* log(n)) / 2 + 12n + 24n^3 + 112n^2 + nlog(n \* log(n)) + 28n =

    # 32n^3 \* log(n \* log(n)) + 150n^2 +  + 40n + 1

    n = queue.size # 1

    for i in range(n // 2 - 1, -1, -1): # (n / 2) \* (16n^2 + 76n + 24) \* log(n \* log(n)) ~= 8n^3 \* log(n \* log(n)) + 38n^2 + 12n

        heapify(queue, n, i)

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

        swap(queue, 0, i) # 8n^2 + 36n + 4

        heapify(queue, i, 0) # (16n^2 + 76n + log(n\*log(n)) + 24) \* log(n \* log(n))

    # ) ~= 24n^3 \* log(n \* log(n)) + 112n^2 + 28n

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()

        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}")

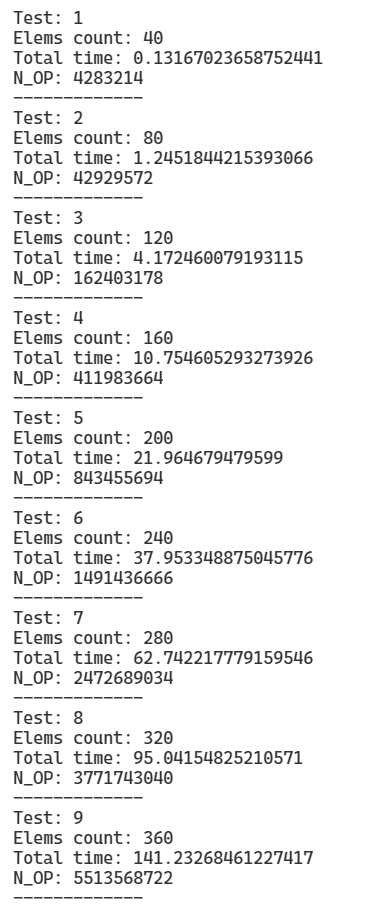
        print(f"N\_OP: {queue.n\_op}")

        print("-------------")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 40 | 5181272,59 | 154364,7372 | 0,1316 | 4283214 |
| 80 | 44550615,04 | 1362106,689 | 1,2451 | 42929572 |
| 120 | 156332060,21 | 4817726,85 | 4,1724 | 162403178 |
| 160 | 380314493,45 | 11764627,89 | 10,7546 | 411983664 |
| 200 | 757188963,80 | 23474405,09 | 21,9646 | 843455694 |
| 240 | 1328282210,21 | 41238519,04 | 37,9533 | 1491436666 |
| 280 | 2135409869,67 | 66363708,4 | 62,7422 | 2472689034 |
| 320 | 3220783464,08 | 100169083,2 | 95,0415 | 3771743040 |
| 360 | 4626946277,09 | 143984121,1 | 141,2326 | 5513568722 |
| 400 | 6396726544,12 | 199147204,5 | 196,9647 | 7698797394 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 39371372,26 | 1172984,325 | 1,209669325 | 0,036039464 |
| 35780752,58 | 1093973,728 | 1,037760522 | 0,031728867 |
| 37468138,29 | 1154665,624 | 0,962617001 | 0,029665225 |
| 35362960,36 | 1093915,895 | 0,923130033 | 0,028556054 |
| 34473150,61 | 1068738,11 | 0,897722274 | 0,027831225 |
| 34997805,47 | 1086559,51 | 0,890605844 | 0,027650198 |
| 34034666,77 | 1057720,456 | 0,863598229 | 0,026838679 |
| 33888180,05 | 1053950,992 | 0,853924414 | 0,02655777 |
| 32761177,5 | 1019482,196 | 0,839192637 | 0,026114506 |
| 32476512,51 | 1011080,688 | 0,830873475 | 0,025867313 |

**Таблица результата экспериментов и графики зависимостей**

**Скриншот работы программы:**

**Выводы.**

По результатам экспериментов было установлено, что алгоритм сортировки подсчетом для очереди на массиве 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.