|  |
| --- |
| 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 № 50**

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 № 50.**

**Реализация связи элементов линейного списка: Указатели**

**Способ организации линейного связанный список: Очередь с головой и хвостом**

**Алгоритм сортировки: Пирамидальная**

**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.

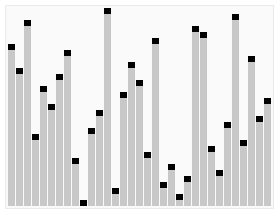


Рисунок 1. Визуализация работы алгоритма

**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]]

    \_tail: Optional[Node[VT]]

    \_size: int

    \_n\_op: int

    def \_\_init\_\_(self):

        self.\_head = None

        self.\_tail = None

        self.\_size = 0

        self.\_n\_op = 0

    def push(self, value: VT):  # 5

        node = Node(value)

        if self.is\_empty():  # 1

            self.\_head = node  # 1

            self.\_tail = node  # 1

        else:

            self.\_tail.next = node  # 2

            self.\_tail = node  # 1

        self.\_size += 1  # 1

        self.\_n\_op += 4

    def pop(self) -> VT:  # 6

        if self.is\_empty():  # 1

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

        node = self.\_head  # 1

        self.\_head = node.next  # 2

        if self.\_head is None:  # 1

            self.\_tail = None  # 1

        self.\_size -= 1  # 1

        self.\_n\_op += 6

        return node.value

    def is\_empty(self) -> bool: # 1

        return self.\_size == 0

    @property

    def head(self) -> VT: # 1

        return self.\_head.value

    @property

    def tail(self) -> VT: # 1

        return self.\_tail.value

    @property

    def size(self) -> int: # 1

        return self.\_size

    @property

    def n\_op(self) -> int: # 1

        return self.\_n\_op

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): # 11

    queue.push(queue.pop())

def seek(queue: Queue[VT], pos: int) -> VT: # 11n + 2

    for \_ in range(pos): # 11pos

        rotate(queue)

    el = queue.head # 2

    for \_ in range(queue.size - pos): # 11n - 11pos

        rotate(queue)

    return el

def pop\_by\_pos(queue: Queue[VT], pos: int) -> VT: # 11n + 6

    for \_ in range(pos): # 11pos

        rotate(queue)

    el = queue.pop() # 6

    for \_ in range(queue.size - pos): # 11n - 11pos

        rotate(queue)

    return el

def push\_by\_pos(queue: Queue[VT], el: VT, pos: int): # 11n - 3

    if pos >= queue.size: # 2

        queue.push(el) # 5

        return

    for \_ in range(pos): # 11pos

        rotate(queue)

    queue.push(el) # 6

    for \_ in range(queue.size - pos - 1): # 11n - 11pos - 11

        rotate(queue)

def swap(queue: Queue, pos1: int, pos2: int): # 44n + 6

    temp = pop\_by\_pos(queue, pos1) # 11n + 6

    push\_by\_pos(queue, pop\_by\_pos(queue, pos2 - 1), pos1) # 22n + 3

    push\_by\_pos(queue, temp, pos2) # 11n - 3

def heapify(queue: Queue, n: int, i: int): # (7 + 22n + 8 + 22n + 8 + 1 + 44n + 6) \* log(n) = 88n \* log(n) + 30log(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 + 11n + 2 + 1 + 11n + 2 = 22n + 7

        largest = l # 1

    if r < n and seek(queue, largest) < seek(queue, r): # 1 + 1 + 11n + 2 + 1 + 11n + 2 = 22n + 7

        largest = r # 1

    if largest != i: # 1

        swap(queue, i, largest) # 44n + 6

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

def heap\_sort(queue: Queue):

    # 1 + 44n^2 \* log(n) + 15n \* log(n) + 44n^2 + 6n + 88n^2 \* log(n) + 30n \* log(n) =

    # = 132n^2 \* log(n) + 45n \* log(n) + 21n + 1

    n = queue.size # 1

    for i in range(n // 2 - 1, -1, -1): # (n / 2) \* (88n \* log(n) + 30log(n)) = 44n^2 \* log(n) + 15n \* log(n)

        heapify(queue, n, i)

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

        swap(queue, 0, i) # 44n + 6

        heapify(queue, i, 0) # 88n \* log(n) + 30log(n)

    # ) = 44n^2 + 6n + 88n^2 \* log(n) + 30n \* log(n)

if \_\_name\_\_ == '\_\_main\_\_':

    import time

    from random import randint

    tests = 10

    step = 100

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

        N\_OP = 0

        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}")

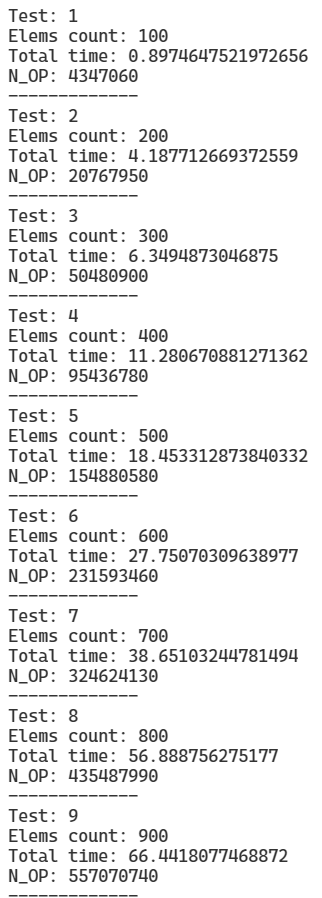
        print("-------------")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 100 | 8801888,523 | 66438,5619 | 0,8974 | 4347060 |
| 200 | 40432556,39 | 305754,2476 | 4,1877 | 20767950 |
| 300 | 97875756,1 | 740593,6821 | 6,3494 | 50480900 |
| 400 | 182722233,1 | 1383016,99 | 11,2806 | 95436780 |
| 500 | 296083112,5 | 2241446,071 | 18,4533 | 154880580 |
| 600 | 438815243,3 | 3322374,729 | 27,7507 | 231593460 |
| 700 | 611616748,9 | 4631093,445 | 38,651 | 324624130 |
| 800 | 815076950,7 | 6172067,961 | 56,8887 | 435487990 |
| 900 | 1049705844 | 7949162,765 | 66,4418 | 557070740 |
| 1000 | 1315952987 | 9965784,285 | 84,3188 | 703448370 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 9808210,969 | 74034,50178 | 2,024791129 | 0,015283562 |
| 9655074,716 | 73012,45256 | 1,946872772 | 0,014722409 |
| 15414961,43 | 116639,9474 | 1,938867098 | 0,01467077 |
| 16197917,94 | 122601,3679 | 1,914589251 | 0,014491446 |
| 16044995,34 | 121465,8663 | 1,911686491 | 0,014472092 |
| 15812763,04 | 119722,1954 | 1,894765264 | 0,014345719 |
| 15824086,02 | 119818,2051 | 1,884076667 | 0,014266017 |
| 14327572,1 | 108493,7424 | 1,87164048 | 0,014172763 |
| 15798877,27 | 119640,9905 | 1,884331322 | 0,014269575 |
| 15606875,18 | 118191,7234 | 1,870717231 | 0,014167044 |

**Таблица результата экспериментов и графики зависимостей**

**Скриншот работы программы:**



**Выводы.**

По результатам экспериментов было установлено, что алгоритм сортировки пирамидальной (кучей) для очереди с головой и хвостом 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.