|  |
| --- |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА – Российский технологический университет"**  **РТУ МИРЭА** |

Институт комплексной безопасности и специального приборостроения

Кафедра КБ-14 «Цифровые технологии обработки данных»

**ОТЧЕТ   
о выполнении лабораторной работы №1**

**«Реализация сортировки линейных структур данных»**

**по дисциплине   
«Алгоритмы и структуры данных»**

**Вариант № 1**

Выполнил: студент 2 курса

группы

шифр

*(фио студента)*

Проверил: *\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Москва 2020 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 № 1.**

**Реализация связи элементов линейного списка:** **Указатели**

**Способ организации линейного связанный список: Дек**

**Алгоритм сортировки**: **«Сравнение и подсчет**»

**Теория о сортировках.**

**Сортировка сравнением и подсчетом.**

Сортировка сравнением и подсчетом – алгоритм сортировки целых неотрицательных чисел. Данный алгоритм использует массив - счетчик длиной, равной диапазону среди различных неповторяющихся сортируемых объектов. Сортировка происходит в соответствии с индексами буфера, где индексом является сам элемент, а значением – количество этих повторяющихся элементов. Алгоритм работает за линейное время O(n+k), где n – длина исходного массива, k – массива - счетчика. Алгоритм особенно эффективен для чисел небольшого диапазона. Например, если требуется отсортировать большое количество чисел с небольшим разбросом среди сортируемых значений, поэтому для эффективной работы k не должно сильно быть больше, чем n.

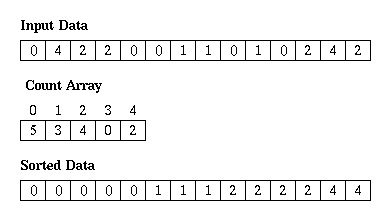


Рисунок 1. – Пример отсортированных данных.

Рисунок 1 иллюстрирует работу данного алгоритма. Первостепенно идет обход по исходному массиву в поисках максимального значения. После чего инициализируется счетчик – массив для подсчета количества каждого элемента. Длина счетчика равна единице, прибавленной к максимальному элементу. Так, в примере на рисунке максимальным элементом является четыре. Следовательно, длина счетчика равна пяти. Далее идет подсчет количество повторений каждого элемента в исходном массиве и присвоение его счетчику, где индекс – элемент, повторения которого считали, а значение – количество повторений. Например, в примере на рисунке 1 значение ноль повторяется пять раз, следовательно, нулевой элемент счетчика равен пяти, один повторяется три раза, значит, первый элемент равен трем и т.д. После завершения подсчета всех значений в исходный массив поочередно записываются индексы счетчика столько раз, сколько записаны повторения в значениях этих индексов. Так, если нулевой элемент равен пяти, то в массив запишется ноль пять раз, а если первый элемент равен 3, то один запишется 3 раза и т.д. На этом сортировка завершена

**Листинг программы с расчетами.**

import random

import time

#класс Node - узел дека

class Node:

    def \_\_init\_\_(self, next=None, prev=None, elem=None): //3

        #ссылка на следующий элемент

        self.next = next //1

        #ссылка на предыдущий элемент

        self.prev = prev //1

        #ссылка на самого себя

        self.elem = elem //1

#класс DoubleLinkedList - дек

class DoubleLinkedList:

    def \_\_init\_\_(self, head=None, tail=None, length=0): //3

        #количество операций

        self.n\_op = 0

        #ссылка на голову

        self.head = head //1

        #ссылка на хвост

        self.tail = tail //1

        #длина дека

        self.length = length //1

    #добавление элемента в конец дека

    def push\_right(self, elem): //16

        #проверка на добавление первого элемента в конец, если таковых не имеется

        if self.tail is None:  # length == 0 //1

            #добавление узла

            new\_node = Node(None, None, elem) //4

            #т.к элементов еще нет, голова будет присвоена нынешнему узлу

            self.head = new\_node //1

            #т.к элементов еще нет, хвост будет равен голове

            self.tail = self.head; //1

            #т.к в деке только один элемент, то длина всего дека равна 1

            self.length = 1 //1

        #в ином случае, если хотя бы 1 элемент присутствует

        else:

            #добавление узла

            new\_node = Node(None, self.tail, elem) //4

            #т.к был создан новый узел, хвост будет являться нынешним узлом

            self.tail.next = new\_node //1

            #глобальная ссылка на хвост  также будет указывать на новый узел

            self.tail = new\_node //1

            #т.к был создан новый узел, длина увеличивается на 1

            self.length += 1 //2

    #удаление элемента из конца дека

    def pop\_right(self): //11

        #проверка на наличие хотя бы 1 элемента, для удаления

        if self.tail is None:  # length == 0 //1

            raise Exception("Нечего удалять")

        #если элемент в деке только 1

        elif self.length == 1: //1

            #ссылки на хвост больше нет

            self.tail = None //1

            #ссылки на голову больше нет

            self.head = None //1

            #длина уменьшается на 1

            self.length -= 1 //2

        #иначе, если элементов в деке больше 1

        else:

            #ссылка на хвост, будет указывать на предыдущий элемент,после самого последнего, который мы удалим

            self.tail = self.tail.prev //1

            #если ссылка на хвост существует

            if self.tail is not None: //1

                #ссылки у последнего элемента на следующий нет

                self.tail.next = None //1

            #т.к был удален узел, длина уменьшается на 1

            self.length -= 1 //2

    #добавление элемента в начало дека

    def push\_left(self, elem): //16

        #проверка на добавление первого элемента в начало, если таковых не имеется

        if self.head is None: //1

            #добавление узла

            new\_node = Node(None, None, elem) //4

            #т.к элементов еще нет, голова будет присвоена нынешнему узлу

            self.head = new\_node //1

            #т.к элементов еще нет, хвост будет равен голове

            self.tail = self.head //1

            #т.к в деке только один элемент, то длина всего дека равна 1

            self.length = 1 //1

         #в ином случае, если хотя бы 1 элемент присутствует

        else:

            #добавление узла

            new\_node = Node(self.head, None, elem) //4

            #т.к был создан новый узел, голова будет являться нынешним узлом

            self.head.prev = new\_node //1

            #глобальная ссылка на голову  также будет указывать на новый узел

            self.head = self.head.prev //1

            #т.к был создан новый узел, длина увеличивается на 1

            self.length += 1 //2

    #удаление элемента из начала дека

    def pop\_left(self): //11

        #проверка на наличие хотя бы 1 элемента, для удаления

        if self.head is None:  # length == 0 //1

            raise Exception("Нечего удалять")

        #если элемент в деке только 1

        elif self.length == 1: //1

            #ссылки на хвост больше нет

            self.tail = None //1

            #ссылки на голову больше нет

            self.head = None //1

            #длина уменьшается на 1

            self.length -= 1 //2

        #иначе, если элементов в деке больше 1

        else:

            #ссылка на голову, будет указывать на следующий элемент, после самого первого, который мы удалим

            self.head = self.head.next //1

            #если ссылка но голову существует

            if self.head is not None: //1

                #ссылки у первого элемента на предыдущий нет

                self.head.prev = None //1

            #т.к был удален узел, длина уменьшается на 1

            self.length -= 1 //2

#получение элемента из любого узла дека

    def get(self,index):

        buff = DoubleLinkedList()

        gotten\_elem =None

        while self.length>=index:

             #если индекс дека равен длине

            if self.length==index:

                gotten\_elem=self.right\_elem()

            #получение самого последнего элемента дека

            right\_elem = self.right\_elem()

            #добавление этого элемента слева в буфер

            buff.push\_left(right\_elem)

            #удаление последнего узла из дека

            self.pop\_right()

        #[выгрузка элементов из буфера обратно в дек]

        #пока длина буфера не равна 0

        while buff.length!=0:

            #получение самого первого элемента буфера

            left\_elem = buff.left\_elem()

            #добавление этого элемента справа обратно в дек

            self.push\_right(left\_elem)

            #удаление первого узла из буфера

            buff.pop\_left()

        return gotten\_elem

    #установка элемента в любой узел дека

    def set(self,index,new\_elem):

        buff = DoubleLinkedList()

        while self.length>=index:

             #если индекс дека равен длине

            if self.length==index:

                self.pop\_right()

                self.push\_right(new\_elem)

            #получение самого последнего элемента дека

            right\_elem = self.right\_elem()

            #добавление этого элемента слева в буфер

            buff.push\_left(right\_elem)

            #удаление последнего узла из дека

            self.pop\_right()

        #[выгрузка элементов из буфера обратно в дек]

        #пока длина буфера не равна 0

        while buff.length!=0:

            #получение самого первого элемента буфера

            left\_elem = buff.left\_elem()

            #добавление этого элемента справа обратно в дек

            self.push\_right(left\_elem)

            #удаление первого узла из буфера

            buff.pop\_left()

    #возвращение длины списка

    def len(self): //1

        return self.length

    #возвращение самого первого элемента

    def left\_elem(self): //1

        return self.head.elem

    #возвращение самого последнего элемента

    def right\_elem(self): //1

        return self.tail.elem

    #возвращение количества операций

    def num\_of\_operations(self):

        return self.n\_op

    #вывод всех элементов дека

    def print\_list(self):

        #получение ссылки на голову

        current = self.head

        #пока узел существует

        while current:

            #печатается его элемент

            print(current.elem)

            #получение ссылки на следующий узел

            current=current.next

#поиск максимального элемента в деке

def max(dll): //

    #присвоение отрицательного значения для последующего поиска максимального значения

    max\_elem = -1 //1

    #инициализация буфера для отгрузки элементов из дека

    buff = DoubleLinkedList() //4

    #[поиск максимума]

    #пока длина дека не равна 0

    while dll.length!=0: //=

        #получение самого последнего элемента дека

        right\_elem = dll.right\_elem() //1+1

        #добавление этого элемента слева в буфер

        buff.push\_left(right\_elem) //16+1

        #удаление последнего узла из дека

        dll.pop\_right() //11

        #если последний элемент дека больше максимального

        if right\_elem> max\_elem: //1

            #максимальный элемент становится новым найденным

            max\_elem = right\_elem //1

    #[выгрузка элементов из буфера обратно в дек]

    #пока длина буфера не равна 0

    while buff.length!=0: //

        #получение самого последнего элемента буфера

        right\_elem = buff.right\_elem() //1+1

        #добавление этого элемента слева обратно в дек

        dll.push\_left(right\_elem) //16+1

        #удаление последнего узла из буфера

        buff.pop\_right() //11

    #возврат максимального элемента

    return max\_elem //1

#сортировка дека сравнением и подсчетом

def counting\_sort(dll): //5++

    #инициализация буфера для подсчета и сохранения элементов (буфер заполняется нулями от 0 до максимального элемента + 1)

    counting\_buff = [0 for i in range(max\_elem+1)] //1

    #инициализация буфера для отгрузки элементов из дека

    buff = DoubleLinkedList() //4

    #[сохранение элементов дека в буфер подсчета для сортировки]

    #пока длина дека не равна 0

    while dll.length!=0: //

        #получение самого последнего элемента дека

        right\_elem = dll.right\_elem() //1+1

        #сохранение и подсчет элементов в качестве индексов буфера подсчета

        counting\_buff[right\_elem]+=1 //1+1+1

        #добавление этого элемента слева в буфер

        buff.push\_left(right\_elem) //16+1

        #удаление последнего узла из дека

        dll.pop\_right() //11

    #пока длина буфера не равна 0

    while buff.length!=0: //

        #получение самого последнего элемента буфера

        right\_elem = buff.right\_elem() //1+1

        #добавление этого элемента слева обратно в дек

        dll.push\_left(right\_elem) //16+1

        #удаление последнего узла из буфера

        buff.pop\_right() //11

    #итерация от 0 до максимального элемента + 1 (максимальный элемент + 1,тк важно, чтобы итерация доходила до индекса максимального элемента)

    for i in range(max\_elem+1): //

            #итерация от 0 до количества повторений элемента, который является индексом

            for j in range(counting\_buff[i])://

                #удаление первого узла из дека

                dll.pop\_left() //11

                #добавление нового узла в дек справа

                dll.push\_right(i) //16+1

#заполнение дека рандомными целыми числами

def random\_filling(elem\_count):

    #итерация от 0 до нужного количества

    for i in range(elem\_count):

        #добавление нового узла в дек справа с элементом в диапазоне от 1 до 100

        dll.push\_right(random.randint(1,100))

elem\_count = 10000

count = 0

for i in range(10):

    count+=elem\_count

    #инициализация дека

    dll =DoubleLinkedList()

    #вызов метода для рандомного заполнения дека

    random\_filling(count)

    #начало отсчета сортировки

    start\_time = time.time()

    #вызов метода для поиска максимального элемента в деке

    max\_elem = max(dll) //3+

    #сортировка дека

    counting\_sort(dll) //2+5++

    #конец отсчета сортировки и последующий вывод количества элементов, затраченного времени и числа операций

    print(f"{i}) Count of elements: {dll.length};   Total time: {time.time() - start\_time} sec;   Num of operations: {dll.num\_of\_operations()};")

F(n)=3++2+5++=

=18+32n+30n+33n+30n+101\*3+101\*28n=2828n+125n+319=2953n+319

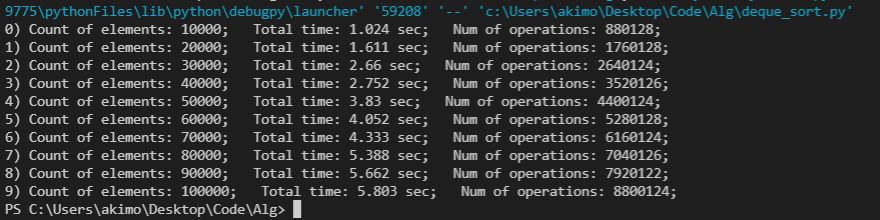
O(F(n))=n

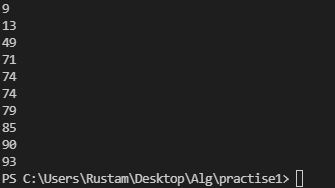
**Таблица результата экспериментов и графики зависимостей**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | F(N) | O(F(n)) | T(n) (сек) | N\_op |
| 15000 | 29530319 | 10000 | 1.024 | 880128 |
| 30000 | 59060319 | 20000 | 1.611 | 1760128 |
| 45000 | 88590319 | 30000 | 2.660 | 2640124 |
| 60000 | 118120319 | 40000 | 2.752 | 3520126 |
| 75000 | 147650319 | 50000 | 3.830 | 440124 |
| 90000 | 177180319 | 60000 | 4.052 | 5280128 |
| 105000 | 206710319 | 70000 | 4.333 | 6160124 |
| 120000 | 236240319 | 80000 | 5.388 | 7040126 |
| 135000 | 265770319 | 90000 | 5.662 | 7920122 |
| 150000 | 295300319 | 100000 | 5.803 | 8800124 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 28838202.148 | 9765.625 | 33.552 | 0.011 |
| 36660657.356 | 12414.649 | 33.555 | 0.011 |
| 33304631.203 | 11278.195 | 33.555 | 0.011 |
| 42921627.544 | 14534.884 | 33.556 | 0.011 |
| 38550997.128 | 13054.830 | 33.556 | 0.011 |
| 43726633.514 | 14807.502 | 33.556 | 0.011 |
| 47706051.004 | 16155.089 | 33.556 | 0.011 |
| 43845641.982 | 14847.810 | 33.556 | 0.011 |
| 46939300.424 | 15895.443 | 33.556 | 0.011 |
| 50887526.969 | 17232.466 | 33.556 | 0.011 |

**Скриншот работы программы:**

****

****

**Выводы.**

В данной работе был реализован дек с использованием указателей. В качестве алгоритма сортировки был использован алгоритм “Сравнение и подсчет”. По результатам экспериментов было установлено, что графики C1, C2, C3, C4 от N имеют линейную зависимость от количества элементов.

**Литература:**

1. Алгоритмы: построение и анализ. Кормен Т.Х., Лейзерсон Ч.И.

2. https://en.wikipedia.org/wiki/Counting\_sort

**Приложение 1. Применение счетчика операций N\_op.**

import random

import time

#класс Node - узел дека

class Node:

    def \_\_init\_\_(self, next=None, prev=None, elem=None):

        #ссылка на следующий элемент

        self.next = next

        #ссылка на предыдущий элемент

        self.prev = prev

        #ссылка на самого себя

        self.elem = elem

#класс DoubleLinkedList - дек

class DoubleLinkedList:

    def \_\_init\_\_(self, head=None, tail=None, length=0):

        #количество операций

        self.n\_op = 0

        #ссылка на голову

        self.head = head;  self.n\_op+=1

        #ссылка на хвост

        self.tail = tail;  self.n\_op+=1

        #длина дека

        self.length = length;  self.n\_op+=1

    #добавление элемента в конец дека

    def push\_right(self, elem):

        #проверка на добавление первого элемента в конец, если таковых не имеется

        if self.tail is None:  # length == 0

            self.n\_op+=1

            #добавление узла

            new\_node = Node(None, None, elem);  self.n\_op+=4

            #т.к элементов еще нет, голова будет присвоена нынешнему узлу

            self.head = new\_node;  self.n\_op+=1

            #т.к элементов еще нет, хвост будет равен голове

            self.tail = self.head;  self.n\_op+=1

            #т.к в деке только один элемент, то длина всего дека равна 1

            self.length = 1;  self.n\_op+=1

        #в ином случае, если хотя бы 1 элемент присутствует

        else:

            self.n\_op+=1

            #добавление узла

            new\_node = Node(None, self.tail, elem);  self.n\_op+=4

            #т.к был создан новый узел, хвост будет являться нынешним узлом

            self.tail.next = new\_node;  self.n\_op+=1

            #глобальная ссылка на хвост  также будет указывать на новый узел

            self.tail = new\_node;  self.n\_op+=1

            #т.к был создан новый узел, длина увеличивается на 1

            self.length += 1;  self.n\_op+=2

    #удаление элемента из конца дека

    def pop\_right(self):

        #проверка на наличие хотя бы 1 элемента, для удаления

        if self.tail is None:  # length == 0

            self.n\_op+=1

            raise Exception("Нечего удалять")

        #если элемент в деке только 1

        elif self.length == 1:

            self.n\_op+=1

            #ссылки на хвост больше нет

            self.tail = None;  self.n\_op+=1

            #ссылки на голову больше нет

            self.head = None;  self.n\_op+=1

            #длина уменьшается на 1

            self.length -= 1;  self.n\_op+=2

        #иначе, если элементов в деке больше 1

        else:

            self.n\_op+=1

            #ссылка на хвост, будет указывать на предыдущий элемент,после самого последнего, который мы удалим

            self.tail = self.tail.prev;  self.n\_op+=1

            #если ссылка на хвост существует

            if self.tail is not None:

                self.n\_op+=1

                #ссылки у последнего элемента на следующий нет

                self.tail.next = None;  self.n\_op+=1

            #т.к был удален узел, длина уменьшается на 1

            self.length -= 1;  self.n\_op+=2

    #добавление элемента в начало дека

    def push\_left(self, elem):

        #проверка на добавление первого элемента в начало, если таковых не имеется

        if self.head is None:

            self.n\_op+=1

            #добавление узла

            new\_node = Node(None, None, elem);  self.n\_op+=4

            #т.к элементов еще нет, голова будет присвоена нынешнему узлу

            self.head = new\_node;  self.n\_op+=1

            #т.к элементов еще нет, хвост будет равен голове

            self.tail = self.head;  self.n\_op+=1

            #т.к в деке только один элемент, то длина всего дека равна 1

            self.length = 1;  self.n\_op+=1

         #в ином случае, если хотя бы 1 элемент присутствует

        else:

            self.n\_op+=1

            #добавление узла

            new\_node = Node(self.head, None, elem);  self.n\_op+=4

            #т.к был создан новый узел, голова будет являться нынешним узлом

            self.head.prev = new\_node;  self.n\_op+=1

            #глобальная ссылка на голову  также будет указывать на новый узел

            self.head = self.head.prev;  self.n\_op+=1

            #т.к был создан новый узел, длина увеличивается на 1

            self.length += 1;  self.n\_op+=2

    #удаление элемента из начала дека

    def pop\_left(self):

        #проверка на наличие хотя бы 1 элемента, для удаления

        if self.head is None:  # length == 0

            self.n\_op+=1

            raise Exception("Нечего удалять")

        #если элемент в деке только 1

        elif self.length == 1:

            self.n\_op+=1

            #ссылки на хвост больше нет

            self.tail = None;  self.n\_op+=1

            #ссылки на голову больше нет

            self.head = None;  self.n\_op+=1

            #длина уменьшается на 1

            self.length -= 1;  self.n\_op+=2

        #иначе, если элементов в деке больше 1

        else:

            self.n\_op+=1

            #ссылка на голову, будет указывать на следующий элемент, после самого первого, который мы удалим

            self.head = self.head.next;  self.n\_op+=1

            #если ссылка но голову существует

            if self.head is not None:

                self.n\_op+=1

                #ссылки у первого элемента на предыдущий нет

                self.head.prev = None;  self.n\_op+=1

            #т.к был удален узел, длина уменьшается на 1

            self.length -= 1;  self.n\_op+=2

#получение элемента из любого узла дека

    def get(self,index):

        buff = DoubleLinkedList()

        gotten\_elem =None

        while self.length>=index:

             #если индекс дека равен длине

            if self.length==index:

                gotten\_elem=self.right\_elem()

            #получение самого последнего элемента дека

            right\_elem = self.right\_elem()

            #добавление этого элемента слева в буфер

            buff.push\_left(right\_elem)

            #удаление последнего узла из дека

            self.pop\_right()

        #[выгрузка элементов из буфера обратно в дек]

        #пока длина буфера не равна 0

        while buff.length!=0:

            #получение самого первого элемента буфера

            left\_elem = buff.left\_elem()

            #добавление этого элемента справа обратно в дек

            self.push\_right(left\_elem)

            #удаление первого узла из буфера

            buff.pop\_left()

        return gotten\_elem

    #установка элемента в любой узел дека

    def set(self,index,new\_elem):

        buff = DoubleLinkedList()

        while self.length>=index:

             #если индекс дека равен длине

            if self.length==index:

                self.pop\_right()

                self.push\_right(new\_elem)

            #получение самого последнего элемента дека

            right\_elem = self.right\_elem()

            #добавление этого элемента слева в буфер

            buff.push\_left(right\_elem)

            #удаление последнего узла из дека

            self.pop\_right()

        #[выгрузка элементов из буфера обратно в дек]

        #пока длина буфера не равна 0

        while buff.length!=0:

            #получение самого первого элемента буфера

            left\_elem = buff.left\_elem()

            #добавление этого элемента справа обратно в дек

            self.push\_right(left\_elem)

            #удаление первого узла из буфера

            buff.pop\_left()

    #возвращение длины списка

    def len(self):

        self.n\_op+=1

        return self.length

    #возвращение самого первого элемента

    def left\_elem(self):

        self.n\_op+=1

        return self.head.elem

    #возвращение самого последнего элемента

    def right\_elem(self):

        self.n\_op+=1

        return self.tail.elem

    #возвращение количества операций

    def num\_of\_operations(self):

        return self.n\_op

    #вывод всех элементов дека

    def print\_list(self):

        #получение ссылки на голову

        current = self.head

        #пока узел существует

        while current:

            #печатается его элемент

            print(current.elem)

            #получение ссылки на следующий узел

            current=current.next

#поиск максимального элемента в деке

def max(dll):

    #присвоение отрицательного значения для последующего поиска максимального значения

    max\_elem = -1;  dll.n\_op+=1

    #инициализация буфера для отгрузки элементов из дека

    buff = DoubleLinkedList(); dll.n\_op+=4

    #[поиск максимума]

    #пока длина дека не равна 0

    while dll.length!=0:

        dll.n\_op+=1

        #получение самого последнего элемента дека

        right\_elem = dll.right\_elem();  dll.n\_op+=2

        #добавление этого элемента слева в буфер

        buff.push\_left(right\_elem);  dll.n\_op+=2

        #удаление последнего узла из дека

        dll.pop\_right();  dll.n\_op+=1

        #если последний элемент дека больше максимального

        if right\_elem> max\_elem:

            dll.n\_op+=1

            #максимальный элемент становится новым найденным

            max\_elem = right\_elem;  dll.n\_op+=1

    #[выгрузка элементов из буфера обратно в дек]

    #пока длина буфера не равна 0

    while buff.length!=0:

        dll.n\_op+=1

        #получение самого последнего элемента буфера

        right\_elem = buff.right\_elem();  dll.n\_op+=2

        #добавление этого элемента слева обратно в дек

        dll.push\_left(right\_elem);  dll.n\_op+=2

        #удаление последнего узла из буфера

        buff.pop\_right();  dll.n\_op+=1

    #возврат максимального элемента

    return max\_elem;  dll.n\_op+=1

#сортировка дека сравнением и подсчетом

def counting\_sort(dll):

    #инициализация буфера для подсчета и сохранения элементов (буфер заполняется нулями от 0 до максимального элемента + 1)

    counting\_buff = [0 for i in range(max\_elem+1)];  dll.n\_op+=1

    #инициализация буфера для отгрузки элементов из дека

    buff = DoubleLinkedList();  dll.n\_op+=4

    #[сохранение элементов дека в буфер подсчета для сортировки]

    #пока длина дека не равна 0

    while dll.length!=0:

        dll.n\_op+=1

        #получение самого последнего элемента дека

        right\_elem = dll.right\_elem();  dll.n\_op+=2

        #сохранение и подсчет элементов в качестве индексов буфера подсчета

        counting\_buff[right\_elem]+=1;  dll.n\_op+=3

        #добавление этого элемента слева в буфер

        buff.push\_left(right\_elem);  dll.n\_op+=2

        #удаление последнего узла из дека

        dll.pop\_right();  dll.n\_op+=1

    #пока длина буфера не равна 0

    while buff.length!=0:

        dll.n\_op+=1

        #получение самого последнего элемента буфера

        right\_elem = buff.right\_elem();  dll.n\_op+=2

        #добавление этого элемента слева обратно в дек

        dll.push\_left(right\_elem);  dll.n\_op+=2

        #удаление последнего узла из буфера

        buff.pop\_right();  dll.n\_op+=1

    #итерация от 0 до максимального элемента + 1 (максимальный элемент + 1,тк важно, чтобы итерация доходила до индекса максимального элемента)

    for i in range(max\_elem+1):

            dll.n\_op+=1

            #итерация от 0 до количества повторений элемента, который является индексом

            for j in range(counting\_buff[i]): #сколько раз потвторяется

                dll.n\_op+=1

                #удаление первого узла из дека

                dll.pop\_left();  dll.n\_op+=1

                #добавление нового узла в дек справа

                dll.push\_right(i);  dll.n\_op+=2

#заполнение дека рандомными целыми числами

def random\_filling(elem\_count):

    #итерация от 0 до нужного количества

    for i in range(elem\_count):

        #добавление нового узла в дек справа с элементом в диапазоне от 1 до 100

        dll.push\_right(random.randint(1,100))

elem\_count = 10000

count = 0

Fn\_list = Tn\_list = OFn\_list = N\_op\_list=[]

with open("text.txt",mode='w') as f:

        f.write(" ")

for i in range(10):

    count+=elem\_count

    #инициализация дека

    dll =DoubleLinkedList()

    #вызов метода для рандомного заполнения дека

    random\_filling(count)

    #начало отсчета сортировки

    start\_time = time.time()

    #вызов метода для поиска максимального элемента в деке

    max\_elem = max(dll); dll.n\_op+=3

    #сортировка дека

    counting\_sort(dll); dll.n\_op+=2

    #конец отсчета сортировки и последующий вывод количества элементов, затраченного времени и числа операций

    print(f"{i}) Count of elements: {dll.length};   Total time: {round(time.time() - start\_time,3)} sec;   Num of operations: {dll.num\_of\_operations()};")