|  |
| --- |
| 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 № 97**

Выполнил: студент 2 курса

группы БИСО-03-19

шифр 19Б1397

Решетников Д. Д.

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 № 97.**

**Реализация связи элементов линейного списка: Указатели**

**Способ организации линейного связанный список: Дек**

**Алгоритм сортировки: Пирамидальная сортировка**

**Теория о сортировках**

Пирамидальная сортировка (или сортировка кучей, Heap Sort) — это метод сортировки сравнением, основанный на такой структуре данных как двоичная куча. Она похожа на сортировку выбором, где сначала ищется максимальный элемент и помещается в конец. После этого операция повторяется для следующих элементов.

**Алгоритм пирамидальной сортировки:**

1. Построение кучи из входных данных. После построения максимальный элемент будет храниться в корне кучи.
2. Замена максимального элемента на последний элемент кучи и уменьшение её размера на 1.
3. Преобразование полученного дерева в кучу с новым корнем.
4. Повторение вышеуказанных шагов до тех пор, пока размер кучи больше 1.

**Как построить кучу?**

Процедура преобразования в кучу (далее процедура HeapifyDeque) может быть применена к узлу, только если его дочерние узлы уже преобразованы. Таким образом, преобразование должно выполняться снизу вверх.

Пример:

Входные данные: 4, 10, 3, 5, 1

4(1)

/ \

10(2) 3(3)

/ \

5(4) 1(5)

Числа в скобках представляют индексы в представлении данных в виде массива.

Применение процедуры HeapifyDeque к индексу 2:

4(1)

/ \

10(2) 3(3)

/ \

5(4) 1(5)

Применение процедуры HeapifyDeque к индексу 1:

10(1)

/ \

5(2) 3(3)

/ \

4(4) 1(5)

Процедура HeapifyDeque вызывает себя рекурсивно для создания кучи сверху вниз.

**Листинг программы с расчетами.**

|  |
| --- |
| #include <iostream> |
|  | #include <string> |
|  | #include <cstdlib> |
|  | #include <ctime> |
|  | #include <chrono> |
|  | using namespace std; |
|  | bool IsEmpty = true; //1 |
|  | long long N\_op = 0; |
|  | struct Item |
|  | { |
|  | int item; |
|  | Item\* next; |
|  | Item\* prev; |
|  | }; |
|  | //3 |
|  | struct Deque |
|  | { |
|  | int AllSize = 0; |
|  | Item\* LeElem = NULL; |
|  | Item\* RiElem = NULL; |
|  | }; |
|  | //23 + 19 + (n - 1)(12 + 19 + 2) + 14 = 23 + 33n |
|  | void GetIndexValue(Deque\* el, int nodeIndex, int& findedValue) |
|  | { |
|  | if (nodeIndex != 1 and nodeIndex <= el->AllSize) |
|  | { |
|  | Item\* findValue = el->LeElem; |
|  | Deque forValue; |
|  | Item\* forValueFinder; |
|  | forValue.RiElem = new Item; |
|  | forValue.RiElem->next = nullptr; |
|  | forValue.RiElem->prev = nullptr; |
|  | forValue.LeElem = forValue.RiElem; |
|  | forValueFinder = forValue.RiElem; |
|  | N\_op += 23; |
|  | for (int i = 1; i < nodeIndex; i++) |
|  | { |
|  | if (i != 1) { |
|  | forValue.RiElem = new Item; |
|  | forValue.RiElem->prev = forValueFinder; |
|  | forValueFinder->next = forValue.RiElem; |
|  | forValue.RiElem->next = nullptr; |
|  | N\_op += 12; |
|  | } |
|  | forValue.RiElem->item = el->LeElem->item; |
|  | findValue = el->LeElem->next; |
|  | delete el->LeElem; |
|  | el->LeElem = findValue; |
|  | findValue = el->LeElem->next; |
|  | N\_op += 19; |
|  | if (i != 1) { |
|  | forValueFinder = forValueFinder->next; |
|  | N\_op += 2; |
|  | } |
|  | } |
|  | forValue.RiElem->next = el->LeElem; |
|  | el->LeElem->prev = forValue.RiElem; |
|  | findedValue = el->LeElem->item; |
|  | el->LeElem = forValue.LeElem; |
|  | N\_op += 14; |
|  | } |
|  | else if (nodeIndex == 1 and nodeIndex <= el->AllSize) { |
|  | findedValue = el->LeElem->item; |
|  | N\_op += 7; |
|  | } |
|  | } |
|  | //18 + 2 + 7 + 15 + 19 + (n - 1)(7 + 15 + 19) + 7 + 3 + 18 + 18 + (n - 1)(3 + 20 + 18) + 21 + 5 + 15 + 17 + (n - 1)(5 + 20 + 17) + 14 + (n - 1)7 = 48 + 131n |
|  | void SwapNodes(Deque\* el, int node1, int node2) |
|  | { |
|  | if ((node1 <= el->AllSize && node2 <= el->AllSize) && node1 > node2) { |
|  | int memory1 = 0, memory2 = 0; |
|  | int currentSize = el->AllSize; |
|  | Item\* helpBegin = NULL; |
|  | Item\* helpSwap; |
|  | Deque helpSecDeq; |
|  | Item\* helpSwapSec = helpSecDeq.LeElem; |
|  | N\_op += 17; //17 |
|  | if (node2 == 1) { |
|  | memory2 = el->LeElem->item; |
|  | N\_op += 5; //5 |
|  | for (int i = node2; i < node1; i++) |
|  | { |
|  | helpSecDeq.RiElem = new Item; |
|  | N\_op += 5; //5 |
|  | if (helpSecDeq.AllSize == 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = nullptr; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | helpSecDeq.LeElem = helpSecDeq.RiElem; |
|  | helpBegin = helpSecDeq.RiElem; |
|  | N\_op += 20; //20 |
|  | } |
|  | else if (helpSecDeq.AllSize != 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = helpSwapSec; |
|  | helpSwapSec->next = helpSecDeq.RiElem; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | N\_op += 20; //20 |
|  | } |
|  | helpSwap = el->LeElem; |
|  | helpSwap = el->LeElem->next; |
|  | delete el->LeElem; |
|  | el->LeElem = helpSwap; |
|  | el->LeElem->prev = nullptr; |
|  | helpSwap = el->LeElem->next; |
|  | helpSecDeq.AllSize++; |
|  | el->AllSize--; |
|  | N\_op += 23; //23 |
|  | } |
|  | memory1 = el->LeElem->item; |
|  | el->LeElem->item = memory2; |
|  | el->LeElem->prev = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->next = el->LeElem; |
|  | el->LeElem = helpBegin; |
|  | helpSecDeq.AllSize = 0; |
|  | el->AllSize = currentSize; |
|  | el->LeElem->item = memory1; |
|  | N\_op += 26; //26 |
|  | } |
|  | else if (node2 != 1) |
|  | { |
|  | N\_op += 2; //2 |
|  | for (int i = 1; i < node2; i++) |
|  | { |
|  | helpSwap = el->LeElem; |
|  | helpSecDeq.RiElem = new Item; |
|  | N\_op += 7; //7 |
|  | if (helpSecDeq.AllSize == 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = nullptr; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.LeElem = helpSecDeq.RiElem; |
|  | helpBegin = helpSecDeq.RiElem; |
|  | N\_op += 15; //15 |
|  | } |
|  | else if (helpSecDeq.AllSize != 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = helpSwapSec; |
|  | helpSwapSec->next = helpSecDeq.RiElem; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | N\_op += 15; //15 |
|  | } |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | helpSecDeq.AllSize++; |
|  | helpSwap = el->LeElem->next; |
|  | delete el->LeElem; |
|  | el->LeElem = helpSwap; |
|  | helpSwap = el->LeElem->next; |
|  | el->AllSize--; |
|  | N\_op += 19; //19 |
|  | } |
|  | memory2 = el->LeElem->item; |
|  | el->LeElem->prev = nullptr; |
|  | N\_op += 7; //7 |
|  | for (int i = node2; i < node1; i++) |
|  | { |
|  | helpSecDeq.RiElem = new Item; |
|  | N\_op += 3; //3 |
|  | if (helpSecDeq.AllSize == 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = nullptr; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | helpSecDeq.LeElem = helpSecDeq.RiElem; |
|  | N\_op += 18; //18 |
|  | } |
|  | else if (helpSecDeq.AllSize != 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = helpSwapSec; |
|  | helpSwapSec->next = helpSecDeq.RiElem; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | N\_op += 20; //20 |
|  | } |
|  | helpSwap = el->LeElem; |
|  | helpSwap = el->LeElem->next; |
|  | helpSwap->prev = nullptr; |
|  | delete el->LeElem; |
|  | el->LeElem = helpSwap; |
|  | helpSwap = el->LeElem->next; |
|  | helpSecDeq.AllSize++; |
|  | el->AllSize--; |
|  | N\_op += 18; //18 |
|  | } |
|  | memory1 = el->LeElem->item; |
|  | el->LeElem->item = memory2; |
|  | el->LeElem->prev = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->next = el->LeElem; |
|  | helpSecDeq.AllSize = 0; |
|  | el->LeElem = helpBegin; |
|  | el->AllSize = currentSize; |
|  | N\_op += 21; //21 |
|  | for (int i = 1; i < node2; i++) |
|  | { |
|  | helpSecDeq.RiElem = new Item; |
|  | N\_op += 5; //5 |
|  | if (helpSecDeq.AllSize == 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = nullptr; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | N\_op += 15; //15 |
|  | } |
|  | else if (helpSecDeq.AllSize != 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = helpSwapSec; |
|  | helpSwapSec->next = helpSecDeq.RiElem; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | N\_op += 20; //20 |
|  | } |
|  | helpSwap = el->LeElem; |
|  | helpSwap = el->LeElem->next; |
|  | delete el->LeElem; |
|  | el->LeElem = helpSwap; |
|  | helpSwap = el->LeElem->next; |
|  | helpSecDeq.AllSize++; |
|  | el->LeElem->prev = nullptr; |
|  | N\_op += 17; //17 |
|  | } |
|  | el->LeElem->item = memory1; |
|  | el->LeElem->prev = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->next = el->LeElem; |
|  | el->LeElem = helpBegin; |
|  | N\_op += 13; //13 |
|  | } |
|  | } |
|  | else if ((node1 <= el->AllSize && node2 <= el->AllSize) && node1 < node2) { |
|  | int memory1 = 0, memory2 = 0; |
|  | int currentSize = el->AllSize; |
|  | Item\* helpBegin = NULL; |
|  | Item\* helpSwap; |
|  | Deque helpSecDeq; |
|  | Item\* helpSwapSec = helpSecDeq.LeElem; |
|  | N\_op += 17; //17 |
|  | if (node1 == 1) { |
|  | memory1 = el->LeElem->item; |
|  | N\_op += 5; //5 |
|  | for (int i = node1; i < node2; i++) |
|  | { |
|  | helpSecDeq.RiElem = new Item; |
|  | N\_op += 5; //5 |
|  | if (helpSecDeq.AllSize == 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = nullptr; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | helpSecDeq.LeElem = helpSecDeq.RiElem; |
|  | helpBegin = helpSecDeq.RiElem; |
|  | N\_op += 20; //20 |
|  | } |
|  | else if (helpSecDeq.AllSize != 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = helpSwapSec; |
|  | helpSwapSec->next = helpSecDeq.RiElem; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | N\_op += 20; //20 |
|  | } |
|  | helpSwap = el->LeElem; |
|  | helpSwap = el->LeElem->next; |
|  | delete el->LeElem; |
|  | el->LeElem = helpSwap; |
|  | el->LeElem->prev = nullptr; |
|  | helpSwap = el->LeElem->next; |
|  | helpSecDeq.AllSize++; |
|  | el->AllSize--; |
|  | N\_op += 23; //23 |
|  | } |
|  | memory2 = el->LeElem->item; |
|  | el->LeElem->item = memory1; |
|  | el->LeElem->prev = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->next = el->LeElem; |
|  | el->LeElem = helpBegin; |
|  | helpSecDeq.AllSize = 0; |
|  | el->AllSize = currentSize; |
|  | el->LeElem->item = memory2; |
|  | N\_op += 26; //26 |
|  | } |
|  | else if (node1 != 1) |
|  | { |
|  | N\_op += 2; //2 |
|  | for (int i = 1; i < node1; i++) |
|  | { |
|  | helpSwap = el->LeElem; |
|  | helpSecDeq.RiElem = new Item; |
|  | N\_op += 7; //7 |
|  | if (helpSecDeq.AllSize == 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = nullptr; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.LeElem = helpSecDeq.RiElem; |
|  | helpBegin = helpSecDeq.RiElem; |
|  | N\_op += 15; //15 |
|  | } |
|  | else if (helpSecDeq.AllSize != 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = helpSwapSec; |
|  | helpSwapSec->next = helpSecDeq.RiElem; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | N\_op += 15; //15 |
|  | } |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | helpSecDeq.AllSize++; |
|  | helpSwap = el->LeElem->next; |
|  | delete el->LeElem; |
|  | el->LeElem = helpSwap; |
|  | helpSwap = el->LeElem->next; |
|  | el->AllSize--; |
|  | N\_op += 19; //19 |
|  | } |
|  | memory1 = el->LeElem->item; |
|  | el->LeElem->prev = nullptr; |
|  | N\_op += 7; //7 |
|  | for (int i = node1; i < node2; i++) |
|  | { |
|  | helpSecDeq.RiElem = new Item; |
|  | N\_op += 3; //3 |
|  | if (helpSecDeq.AllSize == 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = nullptr; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | helpSecDeq.LeElem = helpSecDeq.RiElem; |
|  | N\_op += 18; //18 |
|  | } |
|  | else if (helpSecDeq.AllSize != 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = helpSwapSec; |
|  | helpSwapSec->next = helpSecDeq.RiElem; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | N\_op += 20; //20 |
|  | } |
|  | helpSwap = el->LeElem; |
|  | helpSwap = el->LeElem->next; |
|  | helpSwap->prev = nullptr; |
|  | delete el->LeElem; |
|  | el->LeElem = helpSwap; |
|  | helpSwap = el->LeElem->next; |
|  | helpSecDeq.AllSize++; |
|  | el->AllSize--; |
|  | N\_op += 18; //18 |
|  | } |
|  | memory2 = el->LeElem->item; |
|  | el->LeElem->item = memory1; |
|  | el->LeElem->prev = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->next = el->LeElem; |
|  | helpSecDeq.AllSize = 0; |
|  | el->LeElem = helpBegin; |
|  | el->AllSize = currentSize; |
|  | N\_op += 21; //21 |
|  | for (int i = 1; i < node1; i++) |
|  | { |
|  | helpSecDeq.RiElem = new Item; |
|  | N\_op += 5; //5 |
|  | if (helpSecDeq.AllSize == 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = nullptr; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | N\_op += 15; //15 |
|  | } |
|  | else if (helpSecDeq.AllSize != 0) |
|  | { |
|  | helpSecDeq.RiElem->next = nullptr; |
|  | helpSecDeq.RiElem->prev = helpSwapSec; |
|  | helpSwapSec->next = helpSecDeq.RiElem; |
|  | helpSwapSec = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->item = el->LeElem->item; |
|  | N\_op += 20; //20 |
|  | } |
|  | helpSwap = el->LeElem; |
|  | helpSwap = el->LeElem->next; |
|  | delete el->LeElem; |
|  | el->LeElem = helpSwap; |
|  | helpSwap = el->LeElem->next; |
|  | helpSecDeq.AllSize++; |
|  | el->LeElem->prev = nullptr; |
|  | N\_op += 17; //17 |
|  | } |
|  | el->LeElem->item = memory2; |
|  | el->LeElem->prev = helpSecDeq.RiElem; |
|  | helpSecDeq.RiElem->next = el->LeElem; |
|  | N\_op += 11; //11 |
|  | while (el->LeElem->prev != nullptr) { |
|  | el->LeElem = el->LeElem->prev; |
|  | N\_op += 7; //7 |
|  | } |
|  | N\_op += 3; //3 |
|  | } |
|  | } |
|  | else if (node1 > el->AllSize || node2 > el->AllSize) { |
|  | cout << "There are only " << el->AllSize << " elements" << endl; |
|  | N\_op += 6; //6 |
|  | } |
|  | } |
|  | //9 \* 2 + 5(23 + 33n) + 4 \* 2 + 6 + 7 + 48 + 131n + log(n) = 1202 + 296n + log(n) |
|  | //1 + 3 + 3 + 5 + 1 + 5 + 1 + 1 + 7 + 3 + log(n) = 30 + log(n) - without deque |
|  | void HeapifyDeque(Deque\* el, int& n, int& i) |
|  | { |
|  | int largest = i; |
|  | int largEl = 0, leftEl = 0, rightEl = 0; |
|  | int leftBr = 2 \* i; |
|  | int rightBr = 2 \* i + 1; |
|  | N\_op += 9; //9 |
|  | GetIndexValue(el, i, largEl); //23 + 33n |
|  | GetIndexValue(el, leftBr, leftEl); //23 + 33n |
|  | GetIndexValue(el, rightBr, rightEl); //23 + 33n |
|  | N\_op += 9; //9 |
|  | if (leftBr <= n && leftEl > largEl) { |
|  | largest = leftBr; |
|  | GetIndexValue(el, largest, largEl); //23 + 33n |
|  | N\_op += 4; //4 |
|  | } |
|  | if (rightBr <= n && rightEl > largEl) { |
|  | largest = rightBr; |
|  | GetIndexValue(el, largest, largEl); //23 + 33n |
|  | N\_op += 4; //4 |
|  | } |
|  | N\_op += 6; //6 |
|  | if (largest != i) |
|  | { |
|  | SwapNodes(el, i, largest); //48 + 131n |
|  | HeapifyDeque(el, n, largest); //log(n) |
|  | N\_op += 7; //7 |
|  | } |
|  | } |
|  | //4 + n/2(6 + 1202 + 296n + log(n)) + 2 + n(1202 + 296n + log(n) + 48 + 131n + 8) = 6 + (3724n + 1150n^2 + 3n^2 \* log(n))/2 |
|  | void HeapSort(Deque\* el) |
|  | { |
|  | int one = 1; |
|  | N\_op += 4; |
|  | for (int i = el->AllSize / 2; i >= 1; i--) { |
|  | HeapifyDeque(el, el->AllSize, i); //1202 + 296n + log(n) |
|  | N\_op += 6; |
|  | } |
|  | N\_op += 2; |
|  | for (int i = el->AllSize; i >= 1; i--) |
|  | { |
|  | HeapifyDeque(el, i, one); //1202 + 296n + log(n) |
|  | SwapNodes(el, one, i); //48 + 131n |
|  | N\_op += 8; |
|  | } |
|  | } |
|  | //4 + 20 = 24 |
|  | void CreateRightNode(Deque\* el) |
|  | { |
|  | Item\* helpR; |
|  | int num = rand() % 100 + 1; |
|  | N\_op += 4; |
|  | if (IsEmpty == false) { |
|  | helpR = el->RiElem; |
|  | el->RiElem = new Item; |
|  | helpR->next = el->RiElem; |
|  | el->RiElem->prev = helpR; |
|  | el->RiElem->next = nullptr; |
|  | el->RiElem->item = num; |
|  | el->AllSize++; |
|  | N\_op += 20; |
|  | } |
|  | else if (IsEmpty != false) { |
|  | el->RiElem = new Item; |
|  | el->RiElem->next = nullptr; |
|  | el->RiElem->prev = nullptr; |
|  | el->RiElem->item = num; |
|  | el->LeElem = el->RiElem; |
|  | el->AllSize++; |
|  | IsEmpty = false; |
|  | N\_op += 20; |
|  | } |
|  | } |
|  | //25 + 7 + 13 + (n - 1)(7 + 14 + 13) + 13 = 24 + 34n |
|  | void ShowAllNodes(Deque\* el) |
|  | { |
|  | if (el->AllSize != 1) { |
|  | Item\* secDeqWatch; |
|  | Deque secDeq; |
|  | secDeq.RiElem = new Item; |
|  | secDeq.RiElem->next = nullptr; |
|  | secDeq.RiElem->prev = nullptr; |
|  | secDeq.AllSize = el->AllSize; |
|  | Item\* deqWatch = el->LeElem; |
|  | secDeqWatch = secDeq.RiElem; |
|  | secDeq.LeElem = secDeq.RiElem; |
|  | int index = 1; |
|  | N\_op += 25; //25 |
|  | cout << endl << "Deque will be shown from left side" << endl; |
|  | while (index != el->AllSize) { |
|  | deqWatch = el->LeElem->next; |
|  | cout << index << " element is: " << el->LeElem->item << endl; |
|  | N\_op += 7; //7 |
|  | if (index != 1) |
|  | { |
|  | secDeq.RiElem = new Item; |
|  | secDeq.RiElem->prev = secDeqWatch; |
|  | secDeqWatch->next = secDeq.RiElem; |
|  | secDeq.RiElem->next = nullptr; |
|  | secDeqWatch = secDeqWatch->next; |
|  | N\_op += 14; //14 |
|  | } |
|  | secDeq.RiElem->item = el->LeElem->item; |
|  | delete el->LeElem; |
|  | el->LeElem = deqWatch; |
|  | deqWatch->prev = nullptr; |
|  | index++; |
|  | N\_op += 13; //13 |
|  | } |
|  | cout << index << " element is: " << el->LeElem->item << endl << endl; |
|  | secDeqWatch->next = el->LeElem; |
|  | el->LeElem->prev = secDeq.RiElem; |
|  | el->LeElem = secDeq.LeElem; |
|  | N\_op += 13; //13 |
|  | } |
|  | else if (el->AllSize == 1) { |
|  | cout << "1 element is: " << el->LeElem->item << endl << endl; |
|  | N\_op += 4; //4 |
|  | } |
|  | } |
|  | // (n - 1)(17 + 2) + n(17 + 2 + 2) + 5 = 40n - 14 |
|  | void DeleteAll(Deque\* el) |
|  | { |
|  | while (el->LeElem->next != nullptr) |
|  | { |
|  | Item\* helpL = el->LeElem; |
|  | helpL = el->LeElem->next; |
|  | delete el->LeElem; |
|  | el->LeElem = helpL; |
|  | helpL = helpL->next; |
|  | el->LeElem->prev = nullptr; |
|  | N\_op += 17; |
|  | if (el->AllSize != 0) { |
|  | el->AllSize--; |
|  | N\_op += 2; |
|  | } |
|  | N\_op += 2; |
|  | } |
|  | delete el->LeElem; |
|  | IsEmpty = true; |
|  | el->AllSize--; |
|  | N\_op += 5; |
|  | } |
|  | //3 + n(24 + 1) + 24 + 34n + 1 + 6 + (3724n + 1150n^2 + 3n^2 \* log(n))/2 + 1 + 24 + 34n + 1 + 40n - 14 + 1 + 1 = 48 + (3990n + 1150n^2 + 3n^2 \* log(n))/2 |
|  | int main() |
|  | { |
|  | srand(time(NULL)); |
|  | Deque node; |
|  | int size; |
|  | cout << "Type size of this deque: "; |
|  | cin >> size; |
|  | N\_op += 3; |
|  | for (int i = 1; i <= size; i++) |
|  | { |
|  | CreateRightNode(&node); //24 |
|  | N\_op++; |
|  | } |
|  | cout << "Current deque: " << endl; |
|  | ShowAllNodes(&node); //24 + 34n + 1 |
|  | auto start = chrono::system\_clock::now(); |
|  | HeapSort(&node); //6 + (624n + 1185n^2 + 3n^2 \* log(n))/2 |
|  | auto end = chrono::system\_clock::now(); |
|  | cout << "Sorted deque: " << endl; |
|  | ShowAllNodes(&node); // 24 + 34n + 1 |
|  | DeleteAll(&node); //40n - 14 + 1 |
|  | chrono::duration<double> elapsed = end - start; |
|  | cout << "Heap sort time: " << elapsed.count() << " seconds" << endl; |
|  | N\_op += 4; |
|  | cout << endl << "N\_op: " << N\_op << endl; |
|  | return 0; |
|  | } |

F(n) = 3 + n(24 + 1) + 24 + 34n + 1 + 6 + (3724n + 1150n^2 + 3n^2 \* log(n))/2 + 1 + 24 + 34n + 1 + 40n - 14 + 1 + 1 = 48 + (3990n + 1150n^2 + 3n^2 \* log(n))/2

O(n) = n^2

**Таблица результата экспериментов и графики зависимостей**

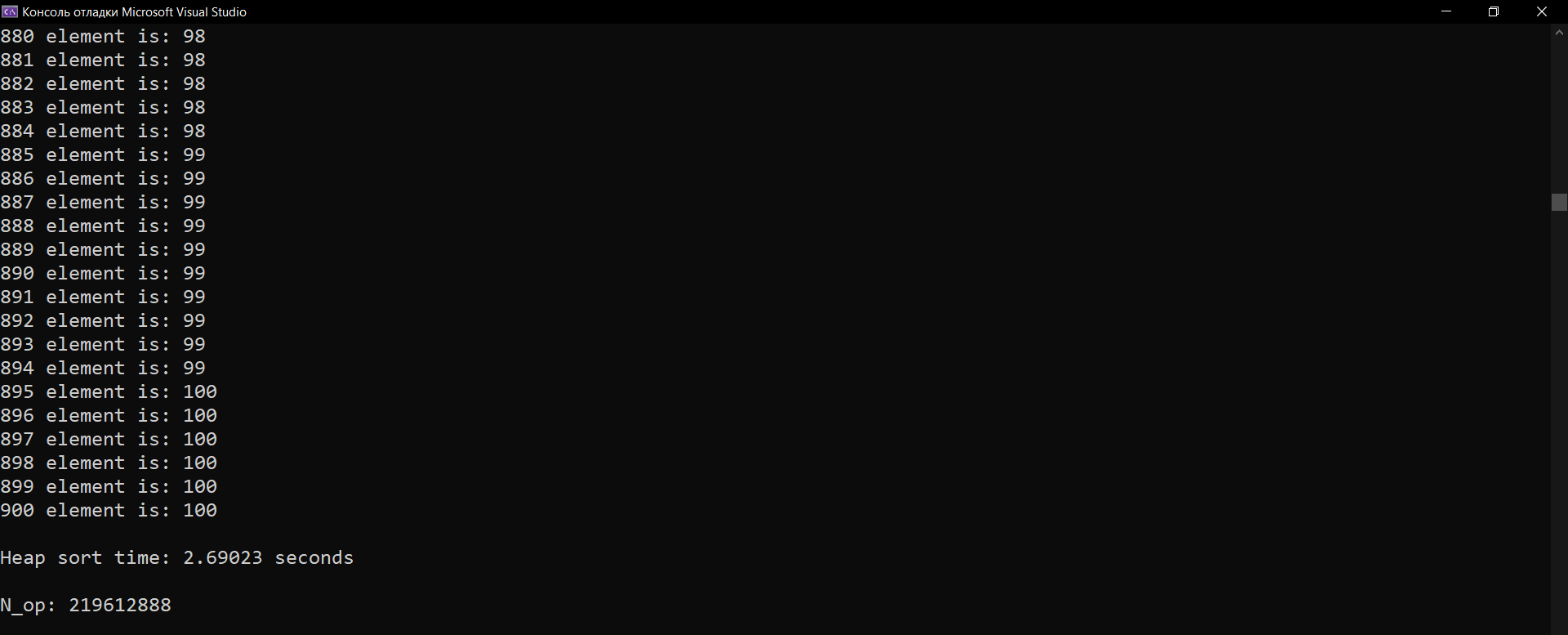
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Кол-во элементов** | **F(n)** | **O(F(n))** | **T(n) сек** | **N\_op** |
| 300 | 341419242 | 740593,6821 | 0,38061 | 24050510 |
| 600 | 1524087427 | 3322374,729 | 0,980155 | 95413748 |
| 900 | 3638387334 | 7949162,765 | 2,69023 | 219612888 |
| 1200 | 6732487164 | 14729498,91 | 4,63405 | 388479988 |
| 1500 | 10840029401 | 23739180,27 | 6,48589 | 599129328 |
| 1800 | 15986956555 | 35036651,06 | 9,37907 | 870738548 |
| 2100 | 22194404946 | 48669525,63 | 12,6763 | 1185277177 |
| 2400 | 29480216517 | 64677995,66 | 16,6919 | 1550688183 |
| 2700 | 37859831393 | 83096841,51 | 21,0029 | 1948032506 |
| 3000 | 47346859831 | 103956721,1 | 25,7943 | 2448539784 |

|  |  |  |  |
| --- | --- | --- | --- |
| **С1=F(n)/T(n)** | **С2=O(F(n))/T(n)** | **С3=F(n)/N\_op** | **С4=O(F(n))/N\_op** |
| 897031718,8 | 1945807,210 | 14,19592526 | 0,030793263 |
| 1554945317 | 3389642,178 | 15,97345727 | 0,034820713 |
| 1352444711 | 2954826,452 | 16,56727602 | 0,036196249 |
| 1452830065 | 3178536,898 | 17,33033199 | 0,037915721 |
| 1671324892 | 3660126,870 | 18,09297074 | 0,039622798 |
| 1704535370 | 3735621,022 | 18,36022603 | 0,040237855 |
| 1750858290 | 3839410,998 | 18,72507577 | 0,041061725 |
| 1766139056 | 3874813,272 | 19,01105383 | 0,041709221 |
| 1802600183 | 3956446,087 | 19,43490741 | 0,042656804 |
| 1835555135 | 4030220,672 | 19,33677375 | 0,042456619 |

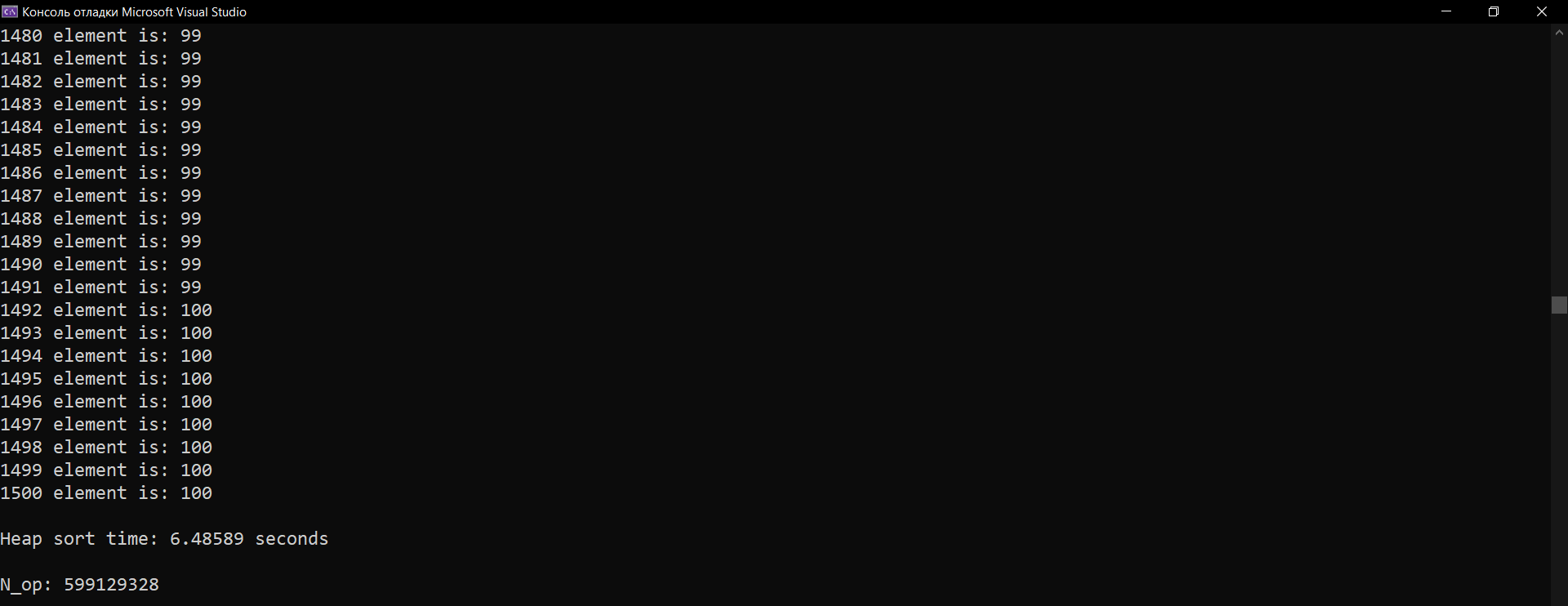
**Скриншоты работы программы:**

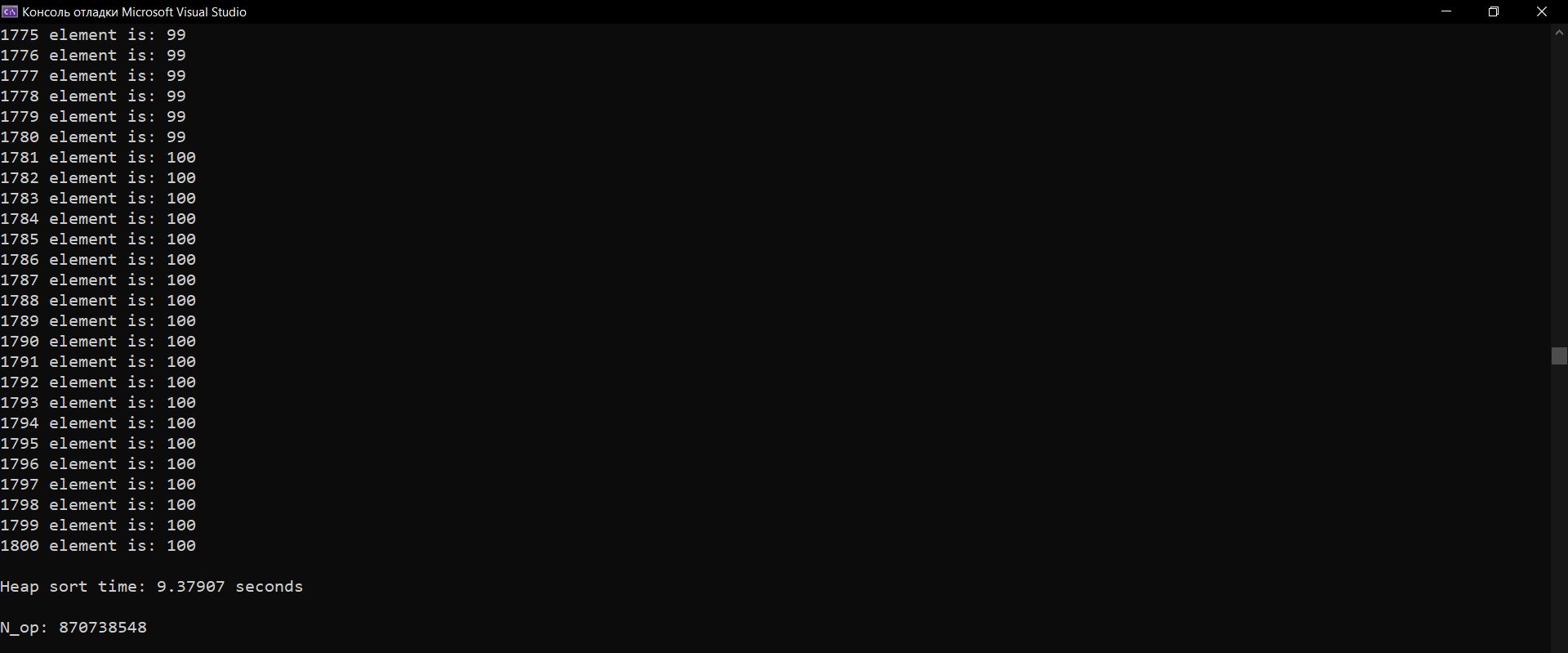
****

****

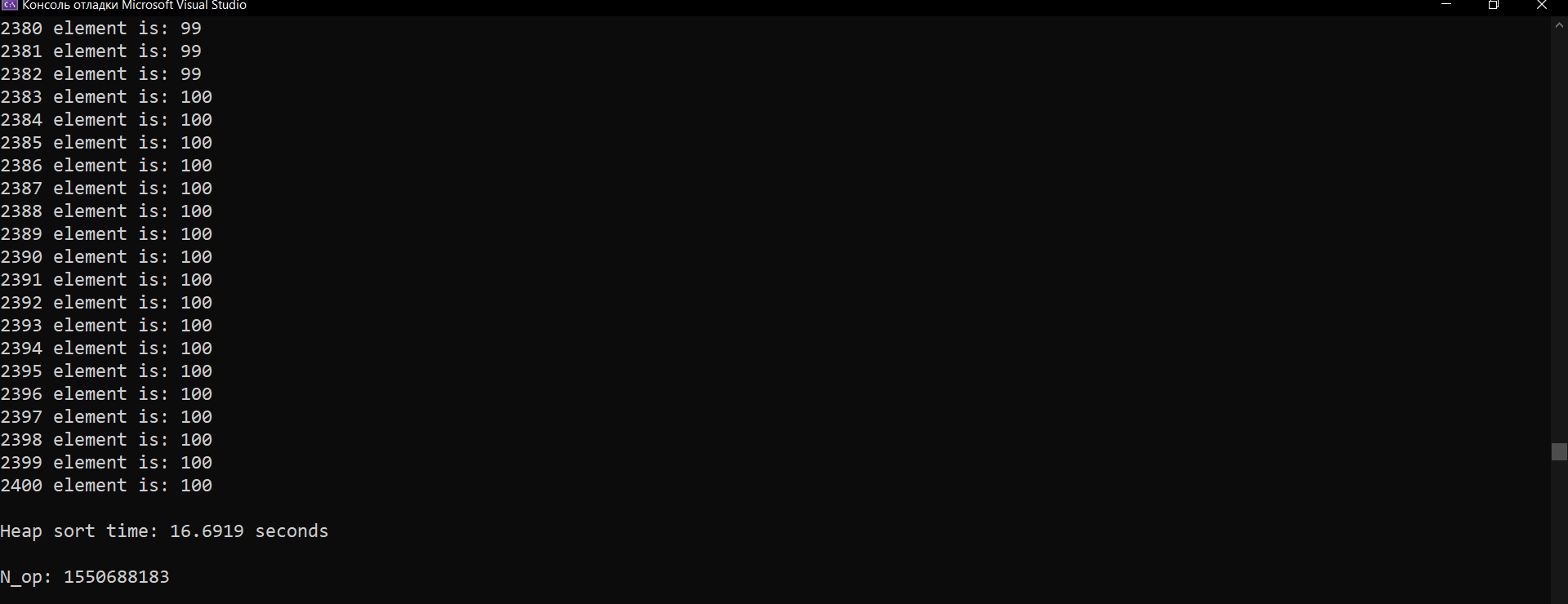
****

****

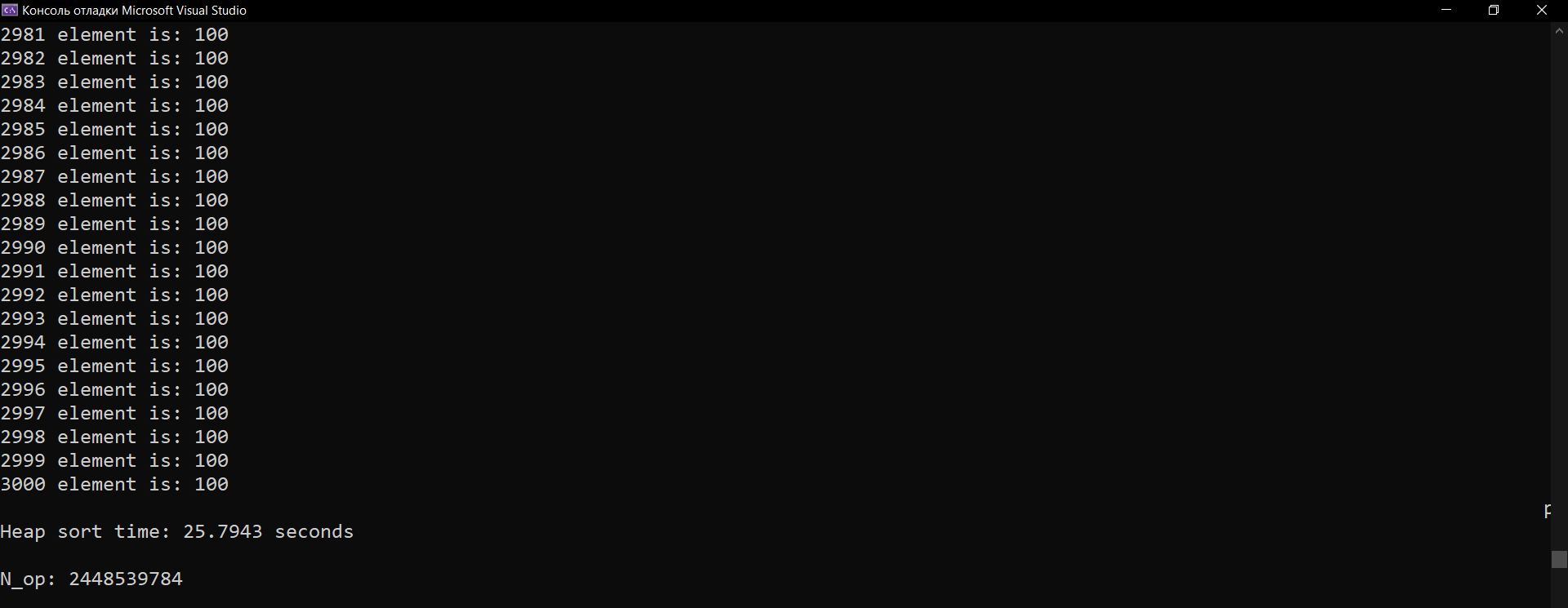
****

****

****

****

****

****

**Вывод:**

По результатам экспериментов было установлено, что графики C1, C2, C3, C4 от N имеют линейную зависимость от количества элементов.

**Источники:**

https://habr.com/ru/company/otus/blog/460087/