|  |
| --- |
| 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 практической работы №1**

**«Реализация сортировки линейных структур данных»**

**по дисциплине   
«Программная реализация нелинейных структур»**

**Вариант № 37**

Выполнил: студент 2 курса

группы \_\_\_\_\_\_\_\_\_\_

шифр БСБО-04-19

Семак Руслан Дмитриевич  
*(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 линенйый 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 (конструктур, 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 № 37.**

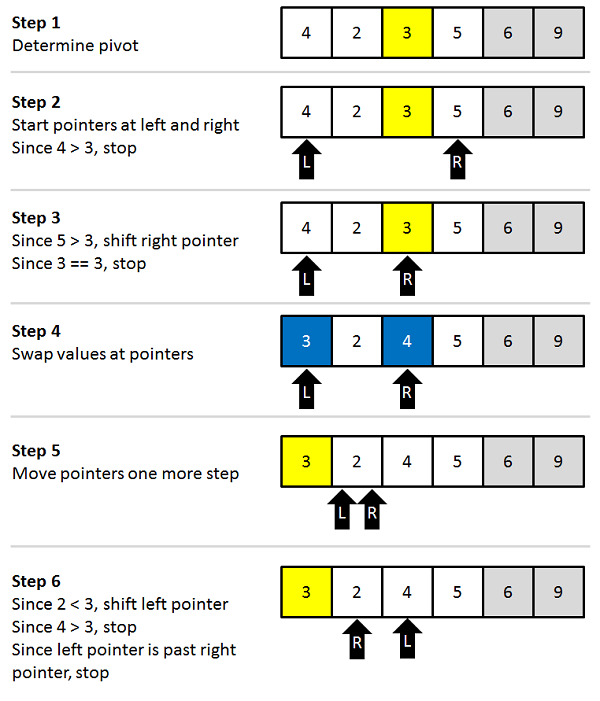
**Реализация связи элементов линейного списка: Массив**

**Способ организации линейного связанный список: Очередь**

**Алгоритм сортировки: Быстрая сортировка с медианным элементом**

**Теория о сортировках.**

Быстрая сортировка относится к парадигме «Разделяй и властвуй». Суть сортировки с медианным элементом заключается в следующем: Находится медианный элемент, два цикла меняют местами элементы, которые больше/меньше медианного. Затем оба цикла встречаются в некоей точке, которая далее делит массив на два сабмассива, затем рекурсивно вызываются их сортировки. Деление общей задачи на подзадачи поменьше происходит до того момента, пока сабмассивы не будут состоять из 2-3 элементов. Стек вызовов закрывается и возвращается отсортированный массив.



**Листинг программы с расчетами**

class Queue1

{

int[] storage;

int head\_pos\_cur;

int head\_pos\_max;

int size = 0;

public Queue1(int n) // 8 + 3n

{

storage = new int[n]; // 3

for (int i = 0; i < n; i++) { storage[i] = 0;} // 2+=2+3n

head\_pos\_max = n - 1; // 2

head\_pos\_cur = n; // 1

}

public void AddLast(int item) // 7

{

storage[head\_pos\_cur - 1] = item; // 4

size++; // 1

if (head\_pos\_cur != 0) // 1

{

head\_pos\_cur--; // 1

}

}

public int GetFirst() // 20+13n

{

int toReturn = storage[head\_pos\_max]; // 2

for (int i = head\_pos\_max; i > 0; i--) // 2+=2+13n

{

storage[i] = storage[i - 1]; // 4

if (i < (head\_pos\_max + 1 - size) + 1) // 4

{

storage[i] = 0; // 2

}

if (i == 1) { storage[0] = 0;} // 3

}

size--; // 1

head\_pos\_cur++; // 1

return toReturn; // 1

}

public int Peek() // НЕ ИСПОЛЬЗУЕТСЯ В ПРОГРАММЕ. СОЗДАН ПО УСЛОВИЮ АТД.

{

return storage[head\_pos\_max];

}

public int Length() { return head\_pos\_max + 1; } // 1

public void Show() // НЕ ИСПОЛЬЗУЕТСЯ В АЛГОРИТМЕ СОРТИРОВКИ

{

Console.Write("Size: " + size + " ");

foreach (int x in storage)

{

Console.Write(x + " ");

}

Console.WriteLine();

}

public int[] ToArray() // 1

{

return storage;

}

public bool isEmpty() // НЕ ИСПОЛЬЗУЕТСЯ В АЛГОРИТМЕ СОРТИРОВКИ

{

foreach (int x in storage)

{

if (x != 0) { return false; } //

return true;

}

return true;

}

// Sorting block

static internal void StartQSort(ref Queue1 ex) // 2

{

QSort(ex.ToArray());

}

static void Swap(ref int a, ref int b) // 5

{

int tmp = a;

a = b;

b = tmp;

}

static internal void QSort(int[] array, int firstIndex = 0, int lastIndex = -1)

{

if (lastIndex < 0) // 1

{

lastIndex = array.Length - 1; // 3

}

if (firstIndex >= lastIndex) // 1

{

return; // 1

}

int middleIndex = (lastIndex - firstIndex) / 2 + firstIndex, currentIndex = firstIndex; // 5

Swap(ref array[firstIndex], ref array[middleIndex]); // 7

for (int i = firstIndex + 1; i <= lastIndex; ++i)

{ // 2+ 11) = 2 + 11n

if (array[i] <= array[firstIndex]) // 3

{

Swap(ref array[++currentIndex], ref array[i]); // 8

}

}

Swap(ref array[firstIndex], ref array[currentIndex]); // 7

QSort(array, firstIndex, currentIndex - 1); // nlog2n

QSort(array, currentIndex + 1, lastIndex);

}

static internal void Clear(Queue1 q) //

{

q.storage = new int[0];

q.head\_pos\_cur = 0;

q.head\_pos\_max = 0;

q.size = 0;

}

}

F(n) = 8 + 3n + 7 + 20 + 13n + 2 + 2 + 5 + 18 + 7 + nlog2n + 2 + 11n=

= 71 + 27n + nlog2n

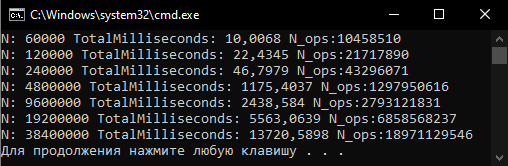
O(F(n)) = nlog2n

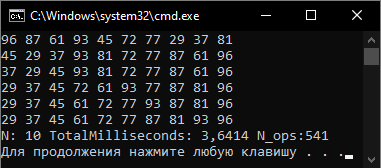
**Таблица результата экспериментов и графики зависимостей**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (мсек) | N\_op |
| 3000 | 84071 | 34653 | 0,9982 | 380871 |
| 60000 | 1680071 | 952380 | 16,9502 | 10455404 |
| 120000 | 3360071 | 2024760 | 18,9783 | 19950201 |
| 240000 | 6720071 | 4289520 | 57,8493 | 44609784 |
| 4800000 | 134400071 | 106536000 | 1010,3597 | 1192063203 |
| 9600000 | 268800071 | 222672000 | 2204,0262 | 2814648386 |
| 19200000 | 537600071 | 464544000 | 4914,1074 | 6882333702 |
| 38400000 | 1075200071 | 967488000 | 11953,9534 | 18998314911 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 84222.6 | 34715.49 | 0.22 | 0.0910 |
| 99118.06 | 56186.95 | 0.16 | 0.0911 |
| 177048.05 | 106688.16 | 0.17 | 0.1015 |
| 116165.12 | 74149.9 | 0.15 | 0.0962 |
| 133022.0 | 105443.64 | 0.11 | 0.0894 |
| 121958.66 | 101029.65 | 0.1 | 0.0791 |
| 109399.33 | 94532.73 | 0.08 | 0.0675 |
| 89945.14 | 80934.56 | 0.06 | 0.0509 |

**Скриншоты работы программы**





**Вывод**

В результате экспериментов стало ясно, что временная сложность O(F(n)) алгоритма возрастает по логарифмической функции. Можно заметить, что среднее и лучшее время работы алгоритма сортировки является n\*logn. Что делает его в большинстве случаев самым эффективным по времени. Выигрывать другие алгоритмы по эффективность он начинает уже когда количество входных элементов исчисляется тысячами. До этой точки он работает как и другие алгоритмы, может даже хуже.

**Литература**

1 Грокаем алгоритмы. Бхаргава Адитья. – М.: Питер, 2019

2. https://en.wikipedia.org/wiki/Quicksort

3. https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/qsort?view=msvc-160

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

class Queue1

{

int[] storage; // 0 <=> null <=> None <=> non-init <=> пусто

int head\_pos\_cur; // Текущая позиция головки

int head\_pos\_max; // Максимальная позиция головки

int size = 0; // Текущее кол-во элементов

public Queue1(int n)

{

Program.N\_op += 11;

storage = new int[n];

for (int i = 0; i < n; i++) { storage[i] = 0; Program.N\_op += 3; } // Заполняем пустыми значениями

head\_pos\_max = n - 1;

head\_pos\_cur = n; // При обращении к головке используется "-1" => ставим текущую на ='n'

}

public void AddLast(int item) // Добавить в конец очереди

{

if (head\_pos\_cur == 0 & storage[0] == 0) // Если мы в конце очереди и конец пустой

{ // Без этого условия можно было бы изменять конец заполненной очереди

storage[head\_pos\_cur - 1] = item;

size++;

Program.N\_op += 7;

}

if (head\_pos\_cur != 0)

{

storage[head\_pos\_cur - 1] = item;

head\_pos\_cur--;

size++;

Program.N\_op += 6;

}

}

public int GetFirst() // Забрать из начала очереди

{

int toReturn = storage[head\_pos\_max]; // Макс, тк очередь двигается к правому краю

Program.N\_op += 3;

// Сдвиг на 1 вправо всех других элементов и головки

for (int i = head\_pos\_max; i > 0; i--)

{

storage[i] = storage[i - 1]; Program.N\_op += 3;

if (i < (head\_pos\_max + 1 - size) + 1)

{

storage[i] = 0;

Program.N\_op += 6;

}

if (i == 1) { storage[0] = 0; Program.N\_op += 3; }

}

// Очередь полностью сдвинулась на 1 вправо, т.е. [0] пусто

Program.N\_op += 4;

size--;

head\_pos\_cur++;

return toReturn;

}

public int Peek() // Нужно, чтобы соответстсвовать АТД

{ // Возвращает первого из очереди без удаления

return storage[head\_pos\_max];

}

public int Length() { return head\_pos\_max + 1; } // Возвращает длину

public void Show() // Выводит

{

Console.Write("Size: " + size + " ");

foreach (int x in storage)

{

Console.Write(x + " ");

}

Console.WriteLine();

}

public int[] ToArray() // Возвращает массив

{

return storage; Program.N\_op += 1;

}

public bool isEmpty() // Метод проверки на пустоту. Для соответствия АТД.

{

foreach (int x in storage)

{

if (x != 0) { return false; } // Нули = пустота

return true;

}

return true;

}

// Sorting block

static internal void StartQSort(ref Queue1 ex) // Метод для запуска сортировки

{

Program.N\_op += 1;

QSort(ex.ToArray());

}

static void Swap(ref int a, ref int b)

{

Program.N\_op += 5;

int tmp = a;

a = b;

b = tmp;

}

static internal void QSort(int[] array, int firstIndex = 0, int lastIndex = -1)

{

Program.N\_op += 3;

if (lastIndex < 0)

{

Program.N\_op += 4;

lastIndex = array.Length - 1;

}

if (firstIndex >= lastIndex)

{

Program.N\_op += 2;

return;

}

int middleIndex = (lastIndex - firstIndex) / 2 + firstIndex, currentIndex = firstIndex;

Swap(ref array[firstIndex], ref array[middleIndex]);

Program.N\_op += 9;

for (int i = firstIndex + 1; i <= lastIndex; ++i)

{

if (array[i] <= array[firstIndex])

{

Program.N\_op += 9;

Swap(ref array[++currentIndex], ref array[i]);

}

}

Program.N\_op += 12;

Swap(ref array[firstIndex], ref array[currentIndex]);

QSort(array, firstIndex, currentIndex - 1);

QSort(array, currentIndex + 1, lastIndex);

}

static internal void Clear(Queue1 q) // Инкапсулированное обнуление для соответствия АТД

{

q.storage = new int[0];

q.head\_pos\_cur = 0;

q.head\_pos\_max = 0;

q.size = 0;

}

}