|  |
| --- |
| 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 № 32**

Выполнил: студент 1 курса

группы БСБО-01-22

шифр 22Б0668

Кирьянов В.А.  
*(фио студента)*

Проверил:

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Москва 2023 г.

**Задание на лабораторную работу № 1.**

В рамках лабораторной работы №1 требуется программно реализовать (с помощью указателей (однонаправленный/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 № 68.**

**Реализация связи элементов линейного списка:** указатели

**Способ организации линейного связанный список:** стек

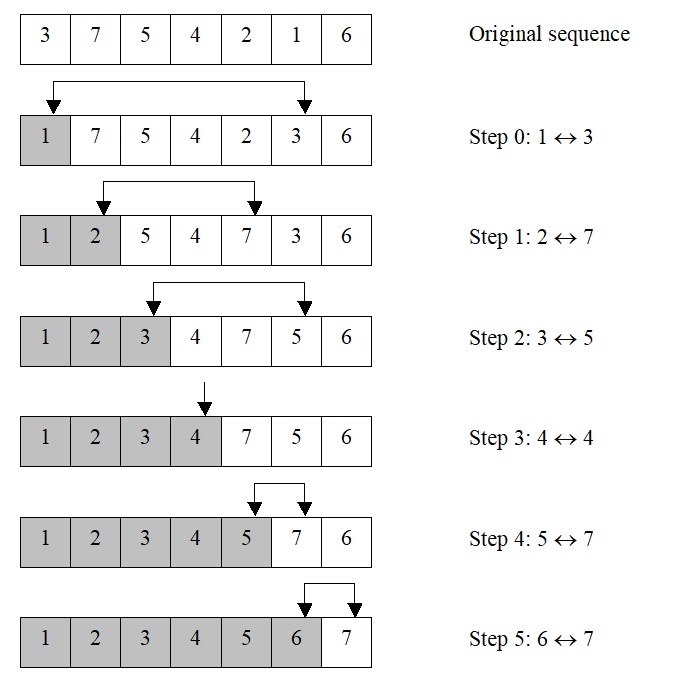
**Алгоритм сортировки:** сортировка простым выбором

**Теория о сортировках.**

**Сортировка простым выбором.**

Сортировка выбором — [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8). На массиве из *n* элементов имеет время выполнения в худшем, среднем и лучшем случае [O](https://ru.wikipedia.org/wiki/O-%D0%BD%D0%BE%D1%82%D0%B0%D1%86%D0%B8%D1%8F)(*n*2), предполагая что сравнения делаются за постоянное время. Общая идея алгоритма состоит в следующем:

* Находим индекс минимального значения в текущем списке
* Производим обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции)
* Теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы



**Листинг программы с расчетами.**

**Element.cs**

namespace algorithms

{

public class Element

{

public int Value = 0;

public Element? Next;

public Element(int value, Element? next = null) // 2

{

this.Value = value; // 1

// указатель на следующий элемент в стеке

this.Next = next; // 1

}

}

}

**Stack.cs**

namespace algorithms

{

public class Stack

{

public int length;

private Element? head;

public Stack() // 1

{

// изначально голова стека равняется null

head = null; // 1

Application.N\_OP += 1;

}

public bool isEmpty() // 1

{

Application.N\_OP += 1;

return length == 0; // 1

}

// если передан сразу экземпляр класса Element

public void Push(Element elem) // 4

{

elem.Next = head; // 2

head = elem; // 1

length++; // 1

Application.N\_OP += 4;

}

/\* Выталкиваем верхний элемент из стека, со стороны головы \*/

public Element? Pop() // 6

{

// если стек не пустой

Application.N\_OP += 2;

if (!isEmpty()) // 2

{

// сохраняем выталкиваемый элемент

Element result = head; // 1

// головой теперь становится тот элемент, который хранился в head.Next

head = head.Next; // 2

// уменьшаем счетчик длины стека

length--; // 1

Application.N\_OP += 4;

return result;

}

else

{

// если стек пустой, то кидаем исключение в юзера

throw new Exception("Stack is empty.");

}

}

/\* Получение элемента стека по индексу \*/

public Element Get(int index) // 27n + 12

{

// если индекс попадает в диапазон стека

Application.N\_OP += 4;

if ((length - 1 >= index) && (index >= 0)) // 4

{

Stack tmp\_stack = new Stack(); // 2

Application.N\_OP += 4;

// записываем все элементы стека до искомого индекса (не включая) во временный стек

for (int cnt = 0; cnt < index; cnt++) // 2 + = 13n+2

{

tmp\_stack.Push(Pop()); // 1 + 4 + 6 = 11

Application.N\_OP += 13;

}

// сохраняем искомый элемент

Element result = head; // 1

Application.N\_OP += 4;

// возвращаем всё как было

while (!tmp\_stack.isEmpty()) // 3 + = 14n+3

{

Push(tmp\_stack.Pop()); // 4 + 1 + 6 = 11

Application.N\_OP += 14;

}

return result;

}

throw new Exception("Index out of range.");

}

// вставка, если передан сразу экземпляр класса Element

public void Set(int index, Element value) // 26n + 15

{

Application.N\_OP += 3;

if ((index >= 0) && (index <= length)) // 3

{

Stack tmp\_stack = new Stack(); // 2

Application.N\_OP += 4;

for (int cnt = 0; cnt < index; cnt++) // 2 + = 13n+2

{

tmp\_stack.Push(Pop()); // 1 + 6 + 4 = 11

Application.N\_OP += 13;

}

value.Next = head; // 2

Push(value); // 4

Application.N\_OP += 8;

while (tmp\_stack.length != 0) // 2 + = 13n+2

{

Push(tmp\_stack.Pop()); // 4 + 1 + 6 = 11

Application.N\_OP += 13;

}

return;

}

throw new Exception("Index out of range.");

}

/\* Красивый вывод стека с указанием головы \*/

public string Show()

{

string result = "HEAD -> { ";

if (head != null)

{

Element current\_element = head;

do

{

result += current\_element.Value.ToString() + ", ";

current\_element = current\_element.Next;

}

while (current\_element != null);

return result.Substring(0, result.Length - 2) + " }";

}

return result + " }";

}

/\* Перегрузка оператора индексации \*/

public Element this[int index]

{

get

{

Application.N\_OP += 2;

return Get(index); // 27n+12+2 = 27n+14

}

set

{

Application.N\_OP += 3;

Set(index, value); // 26n+15+3 = 26n+18

}

}

}

}

**SimpleChoice.cs**

namespace algorithms

{

public class SimpleChoice

{

public Stack stack;

public SimpleChoice(Stack stack) // 1

{

this.stack = stack; // 1

Application.N\_OP += 1;

}

public Stack Sort() // 54n3 + 194n2 + 90n + 6

{

int x = 0; // 1

int y = 0; // 1

int min\_index = 0; // 1

Application.N\_OP += 6;

for (x = 0; x < stack.length - 1; x++) // 3 + = 54n3 + 194n2 + 94n + 3

{

min\_index = x; // 1

Application.N\_OP += 4;

// В этом цикле мы находим индекс минимального элемента на отрезке от stack[x+1] до stack[stack.length - 1]

for (y = x + 1; y < stack.length; y++) // 3 + = 54n2 + 30n + 3

{

Application.N\_OP += 3;

if (stack[y].Value < stack[min\_index].Value) // 3 + 2\*(27n+12)

{

min\_index = y; // 1

Application.N\_OP += 1;

}

Application.N\_OP += 3;

}

// если элемент уже и так стоит на своём месте, то перепрыгиваем на следующую итерацию

Application.N\_OP += 1;

if (min\_index == x) // 1

{

Application.N\_OP += 1;

continue; // 1

}

Element current\_x = stack[x]; // 27n+12+1 = 27n+13

Element current\_min = stack[min\_index]; // 27n+12+1 = 27n+13

Application.N\_OP += 2;

/\* Удаляем из стека stack[min\_index] (чтобы далее на его место вставить значение stack[x]) \*/

Stack tmp\_stack = new Stack(); // 2

Application.N\_OP += 5;

// Добираемся до элемента stack[min\_index] сквозь предшествующие ему элементы (записывая все значения во временный стек, чтоб не потерять)

for (int i = 0; i < min\_index+1; i++) // 3 + = 15n+3

{

tmp\_stack.Push(stack.Pop()); // 12

Application.N\_OP += 15;

}

// избавляемся от головы временного стека, то бишь элемента stack[min\_index]

tmp\_stack.Pop(); // 7

Application.N\_OP += 9;

// и возвращаем всё остальное в сортируемый стек

while (tmp\_stack.length != 0) // 2 + = 14n+2

{

stack.Push(tmp\_stack.Pop()); // 12

Application.N\_OP += 14;

}

/\* Удаляем из стека stack[x] (чтобы далее на его место вставить значение stack[min\_index]) \*/

tmp\_stack = new Stack(); // 2

Application.N\_OP += 5;

// Добираемся до элемента stack[x] сквозь предшествующие ему элементы (записывая все значения во временный стек, чтоб не потерять)

for (int i = 0; i < x+1; i++) // 3 + = 15n+3

{

tmp\_stack.Push(stack.Pop()); // 12

Application.N\_OP += 15;

}

// избавляемся от головы временного стека, то бишь элемента stack[x]

tmp\_stack.Pop(); // 7

Application.N\_OP += 9;

// и возвращаем всё остальное в сортируемый стек

while (tmp\_stack.length != 0) // 2 + = 14n+2

{

stack.Push(tmp\_stack.Pop()); // 12

Application.N\_OP += 14;

}

/\* Вставляем stack[min\_index] на бывшее место stack[x]

И stack[x] на бывшее место stack[min\_index] \*/

// ПРИСВАИВАЕМ ИМЕННО В ТАКОМ ПОРЯДКЕ, В ИНОМ СЛУЧАЕ current\_x ВСТАНЕТ НЕ НА ТО МЕСТО И ВСЁ СЛОМАЕТСЯ НАФИГ :)

stack[x] = current\_min; // 26n+15

stack[min\_index] = current\_x; // 26n+15

/\* Таким образом мы свапнули местами stack[x] и stack[min\_index]

В этом и состоит суть простой сортировки вставкой :) \*/

Application.N\_OP += 4;

}

return stack;

}

}

}

**Program.cs**

public class Application

{

public static long N\_OP = 0;

static void Main(string[] args)

{

Stopwatch timer = new Stopwatch();

Random rand = new Random();

for (int i = 1; i <= 10; i++)

{

Stack stack = new Stack();

for (int j = 0; j < i\*300; j++)

{

stack.Push(new Element(rand.Next(1000)));

}

N\_OP = 0;

SimpleChoice sc = new SimpleChoice(stack);

timer.Restart();

sc.Sort();

timer.Stop();

Console.WriteLine($"Номер сортировки: {i} Количество отсортированных элементов: {i \* 300} Время сортировки (m:s.ms): {timer.Elapsed.ToString().Substring(3)} Количество операций (N\_Op): {N\_OP}");

}

}

}

F(n)=

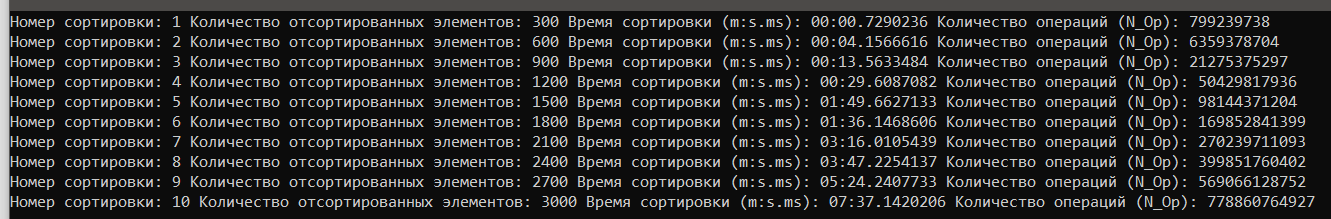
O(F(n)) = n3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 300 | 1475488206 | 27000000 | 0.729 | 799239738 |
| 600 | 11733896406 | 216000000 | 4.157 | 6359378704 |
| 900 | 39523224606 | 729000000 | 13.563 | 21275375297 |
| 1200 | 93591472806 | 1728000000 | 29.609 | 50429817936 |
| 1500 | 182686641006 | 3375000000 | 109.663 | 98144371204 |
| 1800 | 315556729206 | 5832000000 | 96.147 | 169852841399 |
| 2100 | 500949737406 | 9261000000 | 196.011 | 270239711093 |
| 2400 | 747613665606 | 13824000000 | 227.226 | 399851760402 |
| 2700 | 1064296513806 | 19683000000 | 324.241 | 569066128752 |
| 3000 | 1459746282006 | 27000000000 | 457.142 | 778860764927 |

**Таблица результата экспериментов и графики зависимостей**

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 2023989308.642 | 37037037.037 | 1.84612 | 0.033782 |
| 2822683763.772 | 51960548.472 | 1.84513 | 0.033965 |
| 2914047379.341 | 53749170.537 | 1.85770 | 0.034264 |
| 3160912992.874 | 58360633.591 | 1.85588 | 0.034265 |
| 1665891330.768 | 30776104.976 | 1.86141 | 0.034388 |
| 3282023663.827 | 60657118.787 | 1.85782 | 0.034336 |
| 2555722573.764 | 47247348.363 | 1.85372 | 0.034270 |
| 3290176588.973 | 60838108.315 | 1.86972 | 0.034573 |
| 3282424227.060 | 60704846.087 | 1.87025 | 0.034588 |
| 3193200979.140 | 59062610.742 | 1.87421 | 0.034666 |

**Скриншот работы программы:**

****

**Выводы.**

По результатам экспериментов было установлено, что графики C1, C2 от N имеют линейную зависимость на отрезке [1, 4], обратную линейную на отрезке [4, 5], линейную на отрезке [5, 6], обратную линейную на отрезке [6, 7], линейную на отрезке [7, 8], а на отрезке [8, 10] и вовсе не зависят от количества элементов. Графики C3, C4 от N имеют линейную зависимость от количества элементов.

**Литература:**

1. Левитин А. В. Глава 4. Метод декомпозиции: Быстрая сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 174—179. — 576 с.2. Д. Кнут. Искусство программирования для ЭВМ.
2. Структуры данных и алгоритмы. Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман. – М.: Издательский дом «Вильямс», 2000
3. Д. Кнут. Искусство программирования для ЭВМ.