|  |
| --- |
| 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 № 24**

Выполнил: студент 2 курса

группы \_\_\_\_\_\_\_\_\_\_

шифр БСБО-04-19

Стенькин Даниил Сергеевич  
*(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 № 24.**

**Реализация связи элементов линейного списка: Библиотека классов**

**Способ организации линейного связанный список: Стек**

**Алгоритм сортировки: Бинарная вставка**

**Теория о сортировках.**

Сортировка бинарными вставками. Задача заключается в следующем: есть часть массива, которая уже отсортирована, и требуется вставить остальные элементы массива в отсортированную часть, сохранив при этом упорядоченность. Для этого на каждом шаге алгоритма мы выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированной части массива, до тех пор пока весь набор входных данных не будет отсортирован.

Метод выбора очередного элемента из исходного массива осуществляется по алгоритму целочисленного двоичного поиска. Двоичный(бинарный) поиск заключается в том, что на каждом шаге множество объектов делится на две части и в работе остаётся та часть множества, где находится искомый объект. Или же, в зависимости от постановки задачи, мы можем остановить процесс, когда мы получим первый или же последний индекс вхождения элемента.



**Листинг программы с расчетами**

using System;

using System.Collections.Generic;

namespace Alg\_lab\_1

{

class Program

{

public static bool temp\_level = false;

//очистка стека до выбранной позиции(включительно)

static void clearto(Stack<int> stack, int pos) //4+2n

{

//проходит с самого верха стека до нужного элемента

for (int i = 0; i <= pos; i++) //2+=2+2n

{

stack.Pop();

}

}

static void set(Stack<int> stack, int a, int pos) //3n2+13n

{

//создаётся копия стека

Stack<int> temp = new Stack<int>(stack.ToArray()); //3

//очистка до выбранной позиции

clearto(stack, pos); //4+2n

//на место выбранной позиции вставляется выбранное число

stack.Push(a); //1

//цикл начинается с позиции выше чем выбранная, и заканчивается самым верхним значением

for (int i = pos - 1; i >= 0; i--) //3+=3+(n-1)(14+3n)

{

temp\_level = true; //1

//из копии мы добавляем в стек значения, которые прежде затёрли, и при помощи get получаем нужное значение на нужной позиции

stack.Push(get(temp, i)); //11+3n

}

}

//позволяет получить значение на выбранной позиции

static int get(Stack<int> stack, int pos) //2+1+3+3+1+2+2+3(n-1)=11+3n

{

//заготовка для копии стека

Stack<int> temp;

//if необходим, когда в этот метод отправляется КОПИЯ, а не ОРИГИНАЛЬНЫЙ СТЕК.

if (temp\_level) //1

{

//если отправилась копия, то она снова перевораичвается

Stack<int> temp2 = new Stack<int>(stack.ToArray()); //3

temp = new Stack<int>(temp2.ToArray()); //3

temp\_level = false; //1

}

else

{

//если отправился оригинальный стек, то просто создаётся копия

temp = new Stack<int>(stack.ToArray());

}

//проход с самой верхней позиции вплоть до позиции на 1 выше выбранной

for (int i = temp.Count - 1; i > pos; i--) //2+=2+3(n-1)

{

//удаление

temp.Pop();

}

//после прохождения цикла наверху оказывается нужное значение. Peek возвращает его

return temp.Peek(); //2

}

public static void sort(Stack<int> stack)

//1+2+2+

{

//Для удобства храним размер стека в отдельной переменной

int length = stack.Count; //2

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

{

//current - нынешний элемент

int current = get(stack, i); //11+3n+1

//находим, в какую позицию вставим значение current

int insertedPosition = findPositionN(stack, 0, i - 1, current);

//1+5+log2n(18+3n)

//цикл проходит начиная с элемента, который левее выбранного. т.к мы рассматриваем стек, удобно представить что рассматриваем начиная с элемента который ВЫШЕ

//и вплоть до позиции на которой мы вставляем значение current

for (int j = i - 1; j >= insertedPosition; j--) //3+ =3 + =3in2+16in+14i+3

{

//стек: значение выбранного в этом цикле элемента присваивается элементу который ниже его

//массив: значение выбранного в этом цикле элемента присваивается элементу который правее его

set(stack, get(stack, j), j + 1); //11+3n+1+3n2+13n

}

//мы перекопировали все значения, тем самым сделав СДВИГ, который позволяет нам без потерь значений стека спокойно вставить current на выбранную позицию

set(stack, current, insertedPosition); //3n2+16n+14

}

}

//Алгоритм бинарного поиска

public static int findPositionN(Stack<int> stack, int start, int end, int current) //4+log2n(4+12+3n+2)+1=5+log2n(18n+3)

{

//start - это всегда 0, а end - это число выше(левее) чем current

//Работает до тех пор, пока область поиска не "схлопнется" и не будет промежутка, из-за которго мы и размышляем, какую позицию выбрать

while (start <= end) //log2n(4+12+3n+2)

{

//узнаем индекс середины

int mid = start + (end - start) / 2; //4

//меньше или больше середины

if (current < get(stack, mid)) //1+11+3n

{

//если меньше середины, то радиус поиска уменьшается вплоть до области от начала до элемента ПЕРЕД серединным

end = mid - 1; //2

}

else

{

//если больше середины, то область поиска уменьшается вплоть до области от элемента ПРАВЕЕ середины до конца(т.е до элемента левее current)

start = mid + 1; //2

}

}

//в итоге значение start и станет тем самым положением которое мы ищем

return start; //1

}

static void Main(string[] args)

{

Random rnd = new Random();

Stack<int> numbers = new Stack<int>();

int n = 50;

int[] rands = new int[1400];

Console.WriteLine($"N\_op: is {N\_op}");

for (int i = 0; i < 1400; i++)

{

rands[i] = rnd.Next(0, 1000);

}

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

{

for (int j = 0; j < n; j++)

{

numbers.Push(rands[j]);

}

N\_op = 0;

DateTime date = DateTime.Now;

sort(numbers);

DateTime date2 = DateTime.Now;

Console.WriteLine($"\nНомер сортировки: {i}, отсортировано элементов: {n}, время сортировки (ms): {(date2-date).TotalMilliseconds}, Количество операций: {N\_op}");

clearto(numbers, n-1);

n += 150;

}

Console.ReadKey(); }

}

}

F(n) = 1+2+2+ =

= (3 n^4)/2 + (25 n^3)/2 + 34 n^2 + (3 n^2 log(n))/log(2) + 44 n + (18 n log(n))/log(2) + 5

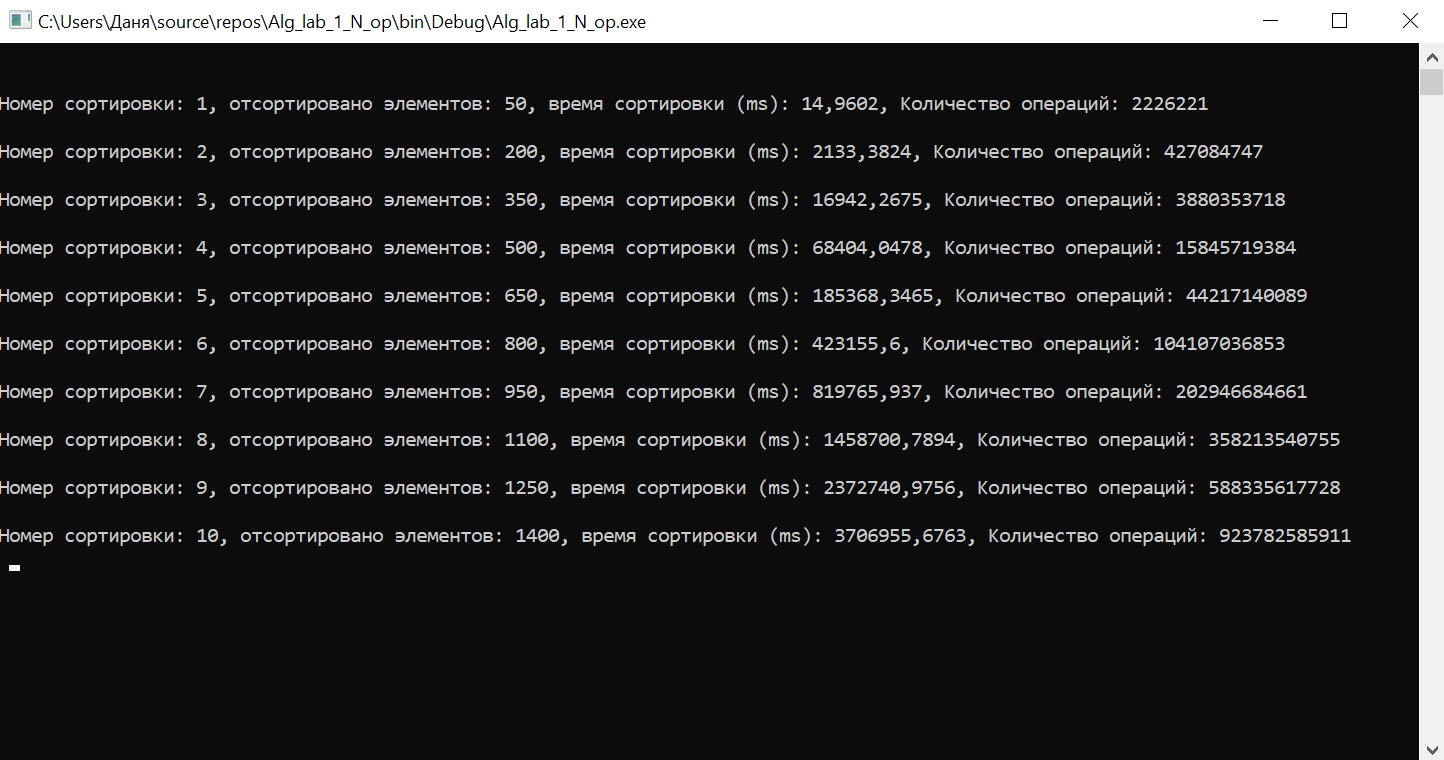
O(F(n)) = n4

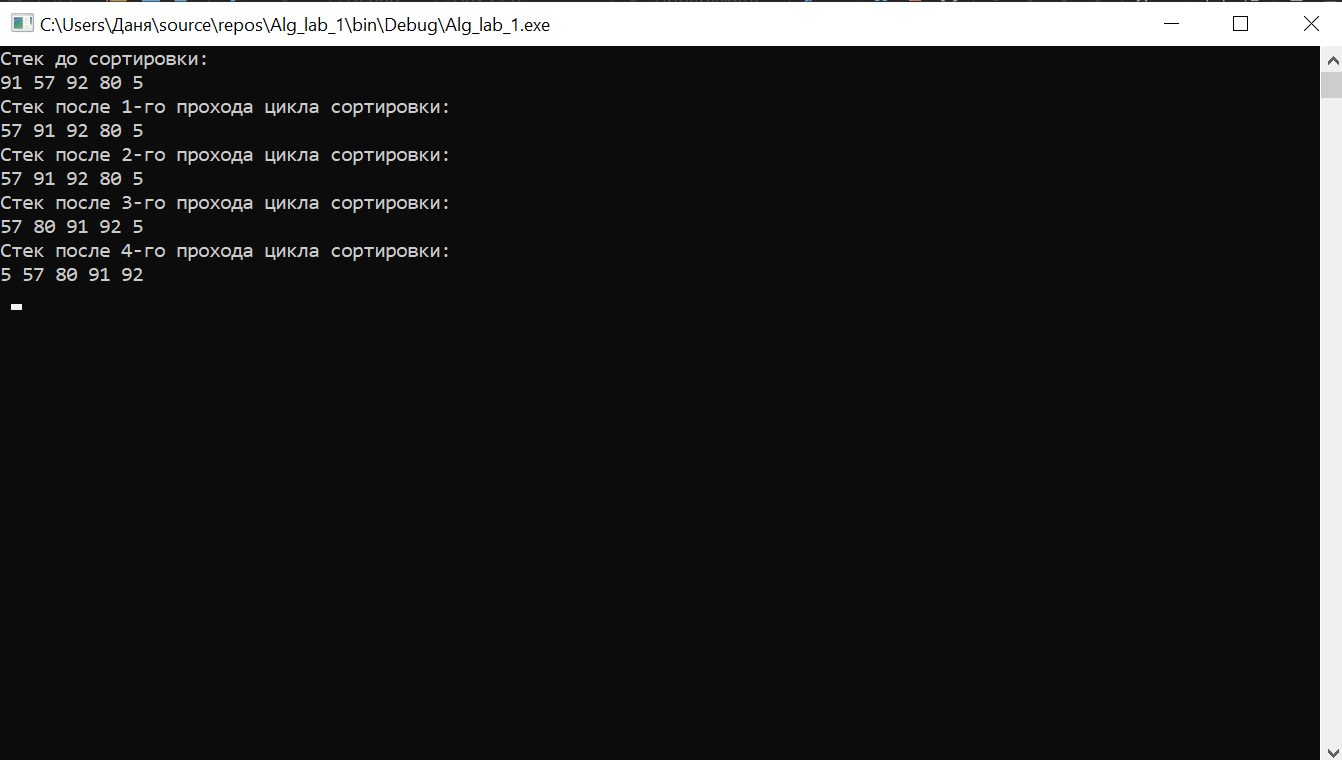
**Таблица результата экспериментов и графики зависимостей**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 50 | 11 072 113 | 6 250 000 | 0,014 | 2 226 221 |
| 200 | 2 502 313 586 | 1 600 000 000 | 2,133 | 427 084 747 |
| 350 | 23 052 651 968 | 15 006 250 000 | 16,942 | 3 880 353 718 |
| 500 | 95 327 827 035 | 62 500 000 000 | 68,404 | 15 845 719 384 |
| 650 | 271 218 534 328 | 178 506 250 000 | 185,368 | 44 217 140 089 |
| 800 | 620 840 450 280 | 409 600 000 000 | 423,155 | 104 107 036 853 |
| 950 | 1 232 534 240 459 | 814 506 250 000 | 819,765 | 202 946 684 661 |
| 1100 | 2 212 865 563 385 | 1 464 100 000 000 | 1458,7 | 358 213 540 755 |
| 1250 | 3 686 625 072 630 | 2 441 406 250 000 | 2372,74 | 588 335 617 728 |
| 1400 | 5 796 828 418 097 | 3 841 600 000 000 | 3706,955 | 923 782 585 911 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 790 865 242,285 | 446 428 571,429 | 4,974 | 2,807 |
| 1 173 142 796,824 | 750 117 205,813 | 5,859 | 3,746 |
| 1 360 680 673,339 | 885 742 533,349 | 5,941 | 3,867 |
| 1 393 600 184,715 | 913 689 257,938 | 6,016 | 3,944 |
| 1 463 135 677,832 | 962 983 093,091 | 6,134 | 4,037 |
| 1 467 170 304,688 | 967 966 820,669 | 5,963 | 3,934 |
| 1 503 521 424,382 | 993 585 051,814 | 6,073 | 4,013 |
| 1 517 012 108,991 | 1 003 701 926,373 | 6,178 | 4,087 |
| 1 553 741 696,364 | 1 028 939 643,619 | 6,266 | 4,150 |
| 1 563 770 916,587 | 1 036 322 264,500 | 6,275 | 4,159 |

**Скриншоты работы программы**





**Выводы**

В результате экспериментов стало ясно, что временная сложность O(F(n)) алгоритма возрастает по экспоненте. Несмотря на то, что алгоритм бинарного поиска имеет сложность O(logn), операции по перемещению элементов стека для последующей вставки нужного элемента в подходящую позицию затрачивают большое количество ресурсов, из-за чего алгоритм сортировки бинарными вставками можно считать малоэффективным для большого количества элементов.

**Литература**

1. Структуры данных и алгоритмы. Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман. – М.: Издательский дом «Вильямс», 2000

2. Д. Кнут. Искусство программирования для ЭВМ.

3. <https://habr.com/ru/post/415935/>

4.neerc.ifmo.ru/wiki/index.php?title=Сортировка\_вставками

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

using System;

using System.Collections.Generic;

namespace Alg\_lab\_1\_N\_op

{

class Program

{

public static ulong N\_op=0;

//когда создаётся копия стека, то она становится перевернутой

//temp\_level используется в методе get() когда достаётся значение из копии стека. Помогает решить путаницу с копией копии стека, которая по сути равна стеку, а нам это не нужно

public static bool temp\_level = false;

//очистка стека до выбранной позиции(включительно)

static void clearto(Stack<int> stack, int pos)

{

//проходит с самого верха стека до нужного элемента

N\_op+=4;

for (int i = 0; i <= pos; i++)

{

N\_op+=3;

stack.Pop();

}

}

static void set(Stack<int> stack, int a, int pos)

{

N\_op+=3;

//создаётся копия стека

Stack<int> temp = new Stack<int>(stack.ToArray()); N\_op+=3;

//очистка до выбранной позиции

clearto(stack, pos);

//на место выбранной позиции вставляется выбранное число

stack.Push(a);N\_op++;

//цикл начинается с позиции выше чем выбранная, и заканчивается самым верхним значением

N\_op+=3;

for (int i = pos - 1; i >= 0; i--)

{

N\_op+=4;

temp\_level = true;

//из копии мы добавляем в стек значения, которые прежде затёрли, и при помощи get получаем нужное значение на нужной позиции

stack.Push(get(temp, i));

}

}

//позволяет получить значение на выбранной позиции

static int get(Stack<int> stack, int pos)

{

N\_op+=2;

//заготовка для копии стека

Stack<int> temp;

//if необходим, когда в этот метод отправляется КОПИЯ, а не ОРИГИНАЛЬНЫЙ СТЕК.

if (temp\_level)

{

N\_op++;

//если отправилась копия, то она снова перевораичвается

Stack<int> temp2 = new Stack<int>(stack.ToArray()); N\_op+=3;

temp = new Stack<int>(temp2.ToArray()); N\_op+=3;

temp\_level = false;N\_op++;

}

else

{

//если отправился оригинальный стек, то просто создаётся копия

temp = new Stack<int>(stack.ToArray()); N\_op+=4;

}

//проход с самой верхней позиции вплоть до позиции на 1 выше выбранной

N\_op+=3;

for (int i = temp.Count - 1; i > pos; i--)

{

N\_op+=3;

//удаление

temp.Pop();

}

//после прохождения цикла наверху оказывается нужное значение. Peek возвращает его

N\_op+=2;

return temp.Peek();

}

public static void sort(Stack<int> stack)

{

N\_op++;

//Для удобства храним размер стека в отдельной переменной

int length = stack.Count; N\_op+=2;

N\_op+=2;

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

{

N\_op+=2;//при итерации

//current - нынешний элемент

int current = get(stack, i); N\_op++;

//находим, в какую позицию вставим значение current

int insertedPosition = findPositionN(stack, 0, i - 1, current); N\_op+=2;

//цикл проходит начиная с элемента, который левее выбранного. т.к мы рассматриваем стек, удобно представить что рассматриваем начиная с элемента который ВЫШЕ

//и вплоть до позиции на которой мы вставляем значение current

N\_op+=3;

for (int j = i - 1; j >= insertedPosition; j--)

{

//стек: значение выбранного в этом цикле элемента присваивается элементу который ниже его

//массив: значение выбранного в этом цикле элемента присваивается элементу который правее его

set(stack, get(stack, j), j + 1); N\_op++;

N\_op+=2;

}

//мы перекопировали все значения, тем самым сделав СДВИГ, который позволяет нам без потерь значений стека спокойно вставить current на выбранную позицию

set(stack, current, insertedPosition);

}

}

//Алгоритм бинарного поиска

public static int findPositionN(Stack<int> stack, int start, int end, int current)

{

N\_op+=4;

//start - это всегда 0, а end - это число выше(левее) чем current

//Работает до тех пор, пока область поиска не "схлопнется" и не будет промежутка, из-за которго мы и размышляем, какую позицию выбрать

while (start <= end)

{

//узнаем индекс середины

int mid = start + (end - start) / 2; N\_op+=4;

//меньше или больше середины

if (current < get(stack, mid))

{

//если меньше середины, то радиус поиска уменьшается вплоть до области от начала до элемента ПЕРЕД серединным

end = mid - 1; N\_op+=3;

}

else

{

//если больше середины, то область поиска уменьшается вплоть до области от элемента ПРАВЕЕ середины до конца(т.е до элемента левее current)

start = mid + 1; N\_op+=3;

}

}

//в итоге значение start и станет тем самым положением которое мы ищем

N\_op++;

return start;

}

public static void print(Stack<int> stack)

{

//с самого верхнего значения до самого нижнего

for (int i = 0; i < stack.Count; i++)

{

Console.Write(get(stack, i) + " ");

}

Console.WriteLine();

}

static void Main(string[] args)

{

Random rnd = new Random();

Stack<int> numbers = new Stack<int>();

int n = 50;

int[] rands = new int[1400];

Console.WriteLine($"N\_op: is {N\_op}");

for (int i = 0; i < 1400; i++)

{

rands[i] = rnd.Next(0, 1000);

}

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

{

for (int j = 0; j < n; j++)

{

numbers.Push(rands[j]);

}

N\_op = 0;

DateTime date = DateTime.Now;

sort(numbers);

DateTime date2 = DateTime.Now;

Console.WriteLine($"\nНомер сортировки: {i}, отсортировано элементов: {n}, время сортировки (ms): {(date2-date).TotalMilliseconds}, Количество операций: {N\_op}");

clearto(numbers, n-1);

n += 150;

}

Console.ReadKey();

}

}

}