Федеральное агентство связи

Ордена трудового Красного Знамени

федеральное государственное

бюджетное образовательное учреждение

высшего образования

«Московский технический университет связи и информатики»

Кафедра информатики

Отчет по лабораторной работе №1 по СиАОД

«Методы поиска, хеширование данных и задача про 8 ферзей»

Выполнил: студент группы БВТ1903

Клычёв Д.А

Проверил: Павликов А.

Москва, 2021

# **1. Задание**

# **2. Ход работы**

Язык программирования, используемый для выполнения работы: C#, версия 9.0.

Для выполнения поставленных задач было создано решение в среде разработки MVS2019, включающее проект Lab1, исполняемый код которого представлен в классах Lab2.cs, листинг представлен ниже:

#define bin

#define tree

#define fib

#define int

#define hash

#define chess

using System;

using System.Diagnostics;

using System.Collections.Generic;

namespace Lab2

{

internal class Lab2

{

public static void Run2Lab()

{

Random rnd = new Random();

Stopwatch sw = new Stopwatch();

#region Task\_1

#if bin

#region BinarySearch

Console.WriteLine("Введите размерность массива для бинарного поиска или 0 для перехода к следующему заданию.");

var size = int.Parse(Console.ReadLine());

while (0 != size)

{

Console.WriteLine("\nВведите искомый элемент.");

var key\_bin = int.Parse(Console.ReadLine());

var mas = new int[size];

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

{

mas[i] = rnd.Next(-1000, 1001);

}

Array.Sort(mas);

sw.Start();

var start = DateTime.Now;

var binary = BinarySearch(mas, key\_bin);

var end = DateTime.Now;

sw.Stop();

Console.WriteLine($"\nЭлемент {key\_bin} присутствует в массиве - {binary}.");

Console.WriteLine($"Затраченное время на выполнение: {/\*sw.Elapsed\*/start - end}.");

sw.Reset();

sw.Start();

start = DateTime.Now;

Array.BinarySearch(mas, key\_bin);

end = DateTime.Now;

sw.Stop();

Console.WriteLine($"\nЗатраченное время на выполнение встроенным алгоритмом класса Array: {start - end}.");

sw.Reset();

Console.WriteLine("Введите размерность массива для повторной генерации или 0 для перехода к следующему заданию.");

size = int.Parse(Console.ReadLine());

}

#endregion

#endif

#if tree

#region BinaryTree

Console.WriteLine("Введите размерность дерева или 0 для перехода к следующему заданию.");

var size\_tree = int.Parse(Console.ReadLine());

while (size\_tree != 0)

{

Console.WriteLine("Введите искомый элемент.");

var key\_tree = int.Parse(Console.ReadLine());

var set = new HashSet<int>();

for (int i = 0; i < size\_tree; i++)

{

set.Add(rnd.Next(-1000, 1001));

}

sw.Start();

var start = DateTime.Now;

var contains\_set = set.Contains(key\_tree);

var end = DateTime.Now;

sw.Stop();

Console.WriteLine($"\nЭлемент {key\_tree} присутствует в множестве - {contains\_set}.");

Console.WriteLine($"Затраченное время на выполнение встроенным методом класса HashSet: {/\*sw.Elapsed\*/start - end}.");

sw.Reset();

int[] arr = new int[set.Count];

//int[] arr = { 5,8,11,3,-2,4,10,17};

set.CopyTo(arr);

BinaryTree tree = new BinaryTree(arr);

sw.Start();

start = DateTime.Now;

var contains\_tree = tree.HasValue(key\_tree);

end = DateTime.Now;

sw.Stop();

Console.WriteLine($"\nЭлемент {key\_tree} присутствует в дереве - {contains\_tree}.");

Console.WriteLine($"Затраченное время на выполнение методом класса BinaryTree: {/\*sw.Elapsed\*/start - end}.");

sw.Reset();

Console.WriteLine("\nВведите элемент для добавления.");

var adding = int.Parse(Console.ReadLine());

tree.Add(adding);

start = DateTime.Now;

contains\_tree = tree.HasValue(adding);

end = DateTime.Now;

Console.WriteLine($"\nЭлемент {adding} присутствует в дереве после добавления - {contains\_tree}.");

Console.WriteLine($"Затраченное время на выполнение методом класса BinaryTree: {/\*sw.Elapsed\*/start - end}.");

Console.WriteLine("\nВведите элемент для удаления.");

int deleting = int.Parse(Console.ReadLine());

tree.Delete(deleting);

start = DateTime.Now;

contains\_tree = tree.HasValue(deleting);

end = DateTime.Now;

Console.WriteLine($"\nЭлемент {deleting} присутствует в дереве после удаления - {contains\_tree}");

Console.WriteLine($"Затраченное время на выполнение методом класса BinaryTree: {/\*sw.Elapsed\*/start - end}");

/\*Console.WriteLine(set.Count);

Console.WriteLine(tree.Count);\*/

sw.Reset();

Console.WriteLine("\nВведите размерность массива для повторной генерации или 0 для перехода к следующему заданию.");

size\_tree = int.Parse(Console.ReadLine());

}

#endregion

#endif

#if fib

#region Fibonacchi

Console.WriteLine("Введите размерность массива для поиска Фибоначчи или 0 для перехода к следующему заданию.");

var size\_fib = int.Parse(Console.ReadLine());

while (size\_fib != 0)

{

Console.WriteLine("Введите искомый элемент.");

var key\_fib = int.Parse(Console.ReadLine());

var set = new HashSet<int>(size\_fib);

for (int i = 0; i < size\_fib; i++)

{

set.Add(rnd.Next(-1000, 1001));

}

var arr = new int[set.Count];

//int[] arr = { 1, 2, 6, 14, 18, 40, 55, 91, 114, 225, 335, 556, 667, 889, 668, 44458, 88889595, 989849841, 989849843 };

set.CopyTo(arr);

Array.Sort(arr);

sw.Start();

var start = DateTime.Now;

var contains\_fib = Fib(arr, key\_fib);

var end = DateTime.Now;

sw.Stop();

Console.WriteLine($"\nЭлемент {key\_fib} присутствует в массиве - {contains\_fib}.");

Console.WriteLine($"Затраченное время на выполнение поиском Фиббоначи: {/\*sw.Elapsed\*/start - end}.");

sw.Reset();

Console.WriteLine("\nВведите размерность массива для повторной генерации или 0 для перехода к следующему заданию.");

size\_fib = int.Parse(Console.ReadLine());

}

#endregion

#endif

#if int

#region Interpolation

Console.WriteLine("Введите размерность массива для интерполяционного поиска или 0 для перехода к следующему заданию.");

var size\_interpol = int.Parse(Console.ReadLine());

while (size\_interpol != 0)

{

Console.WriteLine("Введите искомый элемент.");

var key\_interpol = int.Parse(Console.ReadLine());

var set = new HashSet<int>(size\_interpol);

for (int i = 0; i < size\_interpol; i++)

{

set.Add(rnd.Next(-1000, 1001));

}

//var arr = new int[set.Count];

//set.CopyTo(arr);

int[] arr = { 1, 2, 6, 14, 18, 40, 55, 91, 114, 225, 335, 556, 667, 889, 668, 44458, 88889595, 9898491, 9898443 };

Array.Sort(arr);

sw.Start();

var start = DateTime.Now;

var contains\_int = Interpolation(arr, key\_interpol);

var end = DateTime.Now;

sw.Stop();

Console.WriteLine($"\nЭлемент {key\_interpol} присутствует в массиве - {contains\_int}.");

Console.WriteLine($"Затраченное время на выполнение интерполяционным поиском: {/\*sw.Elapsed\*/start - end}.");

sw.Reset();

Console.WriteLine("\nВведите размерность массива для повторной генерации или 0 для перехода к следующему заданию.");

size\_interpol = int.Parse(Console.ReadLine());

}

#endregion

#endif

#endregion

#region Task 2

#if hash

#region Simple

Console.WriteLine("Введите размерность массива для генерации массива для простого хеширования или 0 для перехода к следующему заданию.");

var size\_simp\_hash = long.Parse(Console.ReadLine());

while (size\_simp\_hash != 0)

{

var set = new HashSet<int>((int)size\_simp\_hash);

while (set.Count != size\_simp\_hash)

{

set.Add(rnd.Next(-(int)size\_simp\_hash, (int)size\_simp\_hash));

}

var arr = new int[set.Count];

set.CopyTo(arr);

Console.WriteLine($"Введите искомый элемент");

int item = int.Parse(Console.ReadLine());

sw.Start();

var start = DateTime.Now;

SimpleHash hash = new SimpleHash(arr);

var end = DateTime.Now;

sw.Stop();

Console.WriteLine($"Затраченное время на хеширование: {/\*sw.Elapsed\*/start - end}.");

sw.Reset();

sw.Start();

start = DateTime.Now;

int index = hash.IndexOf(item);

end = DateTime.Now;

sw.Stop();

Console.WriteLine($"Индекс искомого элемента в хеш-таблице: {index}");

Console.WriteLine($"Затраченное время на поиск: {/\*sw.Elapsed\*/start - end}.");

sw.Reset();

Console.WriteLine("\nВведите размерность массива для повторной генерации или 0 для перехода к следующему заданию.");

size\_simp\_hash = long.Parse(Console.ReadLine());

}

#endregion

#region Pseudo

Console.WriteLine("Введите размерность массива для генерации массива для хеширования на основе псеводослучайных чисел или 0 для перехода к следующему заданию.");

var size\_pseudo\_hash = long.Parse(Console.ReadLine());

while (size\_pseudo\_hash != 0)

{

var set = new HashSet<int>((int)size\_pseudo\_hash);

while (set.Count != size\_pseudo\_hash)

{

set.Add(rnd.Next(-(int)size\_pseudo\_hash, (int)size\_pseudo\_hash));

}

var arr = new int[set.Count];

set.CopyTo(arr);

Console.WriteLine($"Введите искомый элемент");

int item = int.Parse(Console.ReadLine());

sw.Start();

var start = DateTime.Now;

PseudoHash hash = new PseudoHash(arr);

var end = DateTime.Now;

sw.Stop();

Console.WriteLine($"Затраченное время на хеширование: {/\*sw.Elapsed\*/start - end}.");

sw.Reset();

sw.Start();

start = DateTime.Now;

int index = hash.IndexOf(item);

end = DateTime.Now;

sw.Stop();

Console.WriteLine($"Индекс искомого элемента в хеш-таблице: {index}");

Console.WriteLine($"Затраченное время на поиск: {/\*sw.Elapsed\*/start - end}.");

sw.Reset();

Console.WriteLine("\nВведите размерность массива для повторной генерации или 0 для перехода к следующему заданию.");

size\_simp\_hash = long.Parse(Console.ReadLine());

}

#endregion

#region Chains

Console.WriteLine("Введите размерность массива для генерации массива для хеширования цепочками или 0 для перехода к следующему заданию.");

var size\_chains\_hash = long.Parse(Console.ReadLine());

while (size\_chains\_hash != 0)

{

var set = new HashSet<int>((int)size\_chains\_hash);

while (set.Count != size\_chains\_hash)

{

set.Add(rnd.Next(-(int)size\_chains\_hash, (int)size\_chains\_hash));

}

var arr = new int[set.Count];

set.CopyTo(arr);

Console.WriteLine($"Введите искомый элемент");

int item = int.Parse(Console.ReadLine());

sw.Start();

var start = DateTime.Now;

ChainHash hash = new ChainHash(arr);

var end = DateTime.Now;

sw.Stop();

Console.WriteLine($"Затраченное время на хеширование: {/\*sw.Elapsed\*/start - end}.");

sw.Reset();

sw.Start();

start = DateTime.Now;

bool index = hash.HasValue(item);

end = DateTime.Now;

sw.Stop();

Console.WriteLine($"Искомый элемент присутствует в хеш-таблице: {index}");

Console.WriteLine($"Затраченное время на поиск: {/\*sw.Elapsed\*/start - end}.");

sw.Reset();

Console.WriteLine("\nВведите размерность массива для повторной генерации или 0 для перехода к следующему заданию.");

size\_simp\_hash = long.Parse(Console.ReadLine());

}

#endregion

#endif

#endregion

#region Task 3

#if chess

#region Chess

Console.WriteLine("Введите значение от 1 до 24 для вывода результата поиска или 0 для выхода.");

var key = int.Parse(Console.ReadLine());

while (key != 0)

{

sw.Start();

Console.WriteLine($"\nВарианты расстановки ферзей:\n");

var start = DateTime.Now;

Chess(key);

var end = DateTime.Now;

sw.Stop();

Console.WriteLine($"Затраченное время на выполнение: {/\*sw.Elapsed\*/start - end}.");

sw.Reset();

Console.WriteLine("\nВведите любое значение кроме 0 для вывода результата поиска или 0 для выхода.");

key = int.Parse(Console.ReadLine());

}

#endregion

#endif

#endregion

}

#region Task\_1

/// <summary>

/// Бинарный поиск

/// </summary>

/// <param name="source">Массив, в котором производится поиск элемента</param>

/// <param name="key">Искомый элемент</param>

/// <returns>True, если элемент присутствует, иначе False</returns>

static bool BinarySearch(int[] source, int key)

{

if (source.Length > 0)

{

if (source.Length == 1 && source[0] != key)

{

return false;

}

if (source.Length % 2 == 0)//если чётное количество элементов

{

if (source[source.Length / 2 - 1] == key)

{

return true;

}

else if (source[source.Length / 2 - 1] < key)

{

var cutted = new int[source.Length / 2];

Array.Copy(source, source.Length / 2, cutted, 0, source.Length / 2);

return BinarySearch(cutted, key);

}

else

{

var cutted = new int[source.Length / 2];

Array.Copy(source, cutted, source.Length / 2);

return BinarySearch(cutted, key);

}

}

else//если нечётное

{

if (source[source.Length / 2] == key)

{

return true;

}

else if (source[source.Length / 2] < key)

{

var cutted = new int[source.Length / 2];

Array.Copy(source, source.Length / 2 + 1, cutted, 0, cutted.Length);

return BinarySearch(cutted, key);

}

else

{

var cutted = new int[source.Length / 2 + 1];

Array.Copy(source, cutted, source.Length / 2 + 1);

return BinarySearch(cutted, key);

}

}

}

else return false;

}

/// <summary>

/// Бинарное дерево

/// </summary>

public class BinaryTree

{

private Node \_root = null;

public int Count { get; private set; }

public BinaryTree(int[] arr)

{

Count = 0;

foreach (int value in arr)

{

Add(value);

}

}

/// <summary>

/// Функция добавления значения в дерево.

/// </summary>

/// <param name="val">Добавляемое значение</param>

public void Add(int val)

{

var nod = new Node(val);

if (\_root == null)

{

\_root = nod;

Count++;

}

else if (val == \_root.\_value)

{

return;

}

else if (nod.\_value < \_root.\_value)

{

if (\_root.AddLeft(nod))

{

Count++;

}

else

{

return;

}

}

else

{

\_root.AddRight(nod);

Count++;

}

}

/// <summary>

/// Проверяет, есть ли нода с указанным значением в дереве

/// </summary>

/// <param name="key">Интересующее значение</param>

/// <returns>true если есть, иначе false</returns>

public bool HasValue(int key)

{

if (Count == 0)

{

return false;

}

else if (\_root.HasValue(key))

{

return true;

}

else

{

return false;

}

}

/// <summary>

/// Удаляет из дерева указанный элемент

/// </summary>

/// <param name="val">Удаляемыйэлемент</param>

public void Delete(int val)

{

int iterator = 0;

int childs = 0;

int[] buffer;

Node del = null;

if (\_root.HasValue(val, ref del))

{

if (del.HasLeft())

{

del.CountLeftChilds(ref childs);

}

if (del.HasRight())

{

del.CountRightChilds(ref childs);

}

buffer = new int[childs];

del.WriteValues(buffer, ref iterator);

del.BreakLinks();

foreach (int value in buffer)

{

Add(value);

}

Count--;

}

}

/// <summary>

/// Представляет составную часть дерева, храняющую ссылки на потомков и своё значение.

/// </summary>

protected class Node

{

public int? \_value;

Node \_uppernode = null;

Node \_left = null;

Node \_right = null;

public Node(int val)

{

\_value = val;

}

internal void WriteValues(int[] buffer, ref int iterator)

{

if (HasLeft())

{

buffer[iterator] = (int)\_left.\_value;

iterator++;

\_left.WriteValues(buffer, ref iterator);

}

if (HasRight())

{

buffer[iterator] = (int)\_right.\_value;

iterator++;

\_right.WriteValues(buffer, ref iterator);

}

}

/// <summary>

/// Проверяет наличие потомка слева, если его нет - присоединяет, если есть - выполняет проверку дальше

/// </summary>

/// <param name="nod">Присоединяемый нод</param>

public bool AddLeft(Node nod)

{

if (nod.\_value < this.\_value && !HasLeft())//если ноды нет

{

this.\_left = nod;

nod.\_uppernode = this;

return true;

}

else//иначе проверяем далее по цепочке

{

if (nod.\_value == \_left.\_value)

{

return false;

}

else if (\_left.\_value > nod.\_value)

{

\_left.AddLeft(nod);

return true;

}

else

{

\_left.AddRight(nod);

return true;

}

}

}

/// <summary>

/// Выполняет проверку наличия потомка справа

/// </summary>

/// <returns>true если есть, иначе</returns>

internal bool HasRight()

{

if (\_right != null)

{

return true;

}

else

{

return false;

}

}

/// <summary>

/// Выполняет проверку потомка слева

/// </summary>

/// <returns>true если есть, иначе</returns>

internal bool HasLeft()

{

if (\_left != null)

{

return true;

}

else

{

return false;

}

}

/// <summary>

/// Проверяет наличие потомка справа, если его нет - присоединяет, если есть - выполняет проверку дальше

/// </summary>

/// <param name="nod">Присоединяемый нод</param>

public bool AddRight(Node nod)

{

if (nod.\_value > this.\_value && !HasRight())//если ноды нет

{

this.\_right = nod;

nod.\_uppernode = this;

return true;

}

else//иначе проверяем далее по цепочке

{

if (nod.\_value == \_right.\_value)

{

return false;

}

else if (\_right.\_value > nod.\_value)

{

\_right.AddLeft(nod);

return true;

}

else

{

\_right.AddRight(nod);

return true;

}

}

}

/// <summary>

/// Выполняет проверку значения текущего экземпляра.

/// </summary>

/// <param name="key"></param>

/// <returns></returns>

internal bool HasValue(int key, ref Node deletion)

{

bool flag = false;

if (\_value == key)

{

deletion = this;

return true;

}

if (HasLeft())

{

flag = \_left.HasValue(key, ref deletion);

}

if (HasRight() && !flag)

{

flag = \_right.HasValue(key, ref deletion);

}

return flag;

}

/// <summary>

/// Выполняет проверку значения текущего экземпляра.

/// </summary>

/// <param name="key"></param>

/// <returns></returns>

internal bool HasValue(int key)

{

bool flag = false;

if (\_value == key)

{

return true;

}

if (HasLeft())

{

flag = \_left.HasValue(key);

}

if (HasRight() && !flag)

{

flag = \_right.HasValue(key);

}

return flag;

}

/// <summary>

/// Считает количество левых потомков нода

/// </summary>

/// <param name="childs">Счётчик потомков, возвращает значение по ссылке</param>

internal void CountLeftChilds(ref int childs)

{

childs++;

if (\_left.HasLeft())

{

\_left.CountLeftChilds(ref childs);

}

if (\_left.HasRight())

{

\_left.CountRightChilds(ref childs);

}

}

/// <summary>

/// Считает количество правых потомков нода

/// </summary>

/// <param name="childs">Счётчик потомков, возвращает значение по ссылке</param>

internal void CountRightChilds(ref int childs)

{

childs++;

if (\_right.HasLeft())

{

\_right.CountLeftChilds(ref childs);

}

if (\_right.HasRight())

{

\_right.CountRightChilds(ref childs);

}

}

/// <summary>

/// Разрывает ссылки между удаляемым элементом и его корнем.

/// </summary>

internal void BreakLinks()

{

if (\_uppernode.HasLeft() && \_left == this)

{

\_uppernode.\_left = null;

\_left = \_right = \_uppernode = null;

}

else

{

\_uppernode.\_right = null;

\_left = \_right = \_uppernode = null;

}

}

}

}

/// <summary>

/// Поиск Фибоначчи

/// </summary>

/// <param name="arr"></param>

/// <param name="key"></param>

/// <returns></returns>

public static bool Fib(int[] arr, int key)

{

int i, p, q;

bool breaker = false;

Init(arr);

int n = arr.Length;

while (!breaker)

{

if (i < 0)

{

IndexUp();

}

else if (i >= n)

{

IndexDown();

}

else if (arr[i] == key)

{

return true;

}

else if (key < arr[i])

{

IndexDown();

}

else if (key < arr[i])

{

IndexUp();

}

}

return false;

long FibNumber(int k)

{

long first = 0;

long second = 1;

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

{

long temp = second;

second += first;

first = temp;

}

return first;

}

void IndexUp()

{

if (p == 1)

{

breaker = true;

}

i += q;

p -= q;

q -= p;

}

void IndexDown()

{

if (q == 0)

{

breaker = true;

}

i -= q;

int temp = q;

q = p - q;

p = temp;

}

void Init(int[] arr)

{

breaker = false;

int k = 0;

int n = arr.Length;

while (FibNumber(k + 1) < n + 1)

{

k += 1;

}

int m = (int)(FibNumber(k + 1) - (n + 1));

i = (int)(FibNumber(k) - m);

p = (int)(FibNumber(k - 1));

q = (int)(FibNumber(k - 2));

}

}

public static bool Interpolation(int[] arr, int key)

{

int mid;

int i = 0;

int j = arr.Length - 1;

while (arr[i] < key && arr[j] > key)

{

if (arr[j] == arr[i])

break;

mid = i + ((key - arr[i]) \* (j - i)) / (arr[j] - arr[i]);

if (arr[mid] < key)

i = mid + 1;

else if (arr[mid] > key)

j = mid - 1;

else

return true;

}

if (arr[i] == key || arr[j] == key) return true;

else return false; // Not found

}

#endregion

#region Task 2

class SimpleHash

{

int?[] table;

public SimpleHash(int[] arr)

{

table = new int?[arr.Length];

foreach (var item in arr)

{

AddItem(item);

}

}

void Delete(int item)

{

var index = IndexOf(item);

if (index != -1)

{

table[index] = null;

}

}

public int IndexOf(int item)

{

return Array.IndexOf(table, item);

}

void AddItem(int item)

{

var index = Hash(item);

if (table[index] == null)

{

table[index] = item;

}

else

{

Rehash(item, index);

}

}

int Hash(int item)

{

return Math.Abs(item % table.Length);

}

void Rehash(int item, int index)

{

int i = 1;

bool flag = true;

int new\_index;

while (flag)

{

new\_index = (index + i) % table.Length;

if (table[new\_index] == null)

{

table[new\_index] = item;

flag = false;

}

else

{

++i;

}

}

}

}

class PseudoHash

{

int?[] table;

public PseudoHash(int[] arr)

{

table = new int?[arr.Length];

foreach (var item in arr)

{

AddItem(item);

}

}

void Delete(int item)

{

var index = IndexOf(item);

if (index != -1)

{

table[index] = null;

}

}

public int IndexOf(int item)

{

return Array.IndexOf(table, item);

}

void AddItem(int item)

{

var index = Hash(item);

if (table[index] == null)

{

table[index] = item;

}

else

{

Rehash(item, index);

}

}

int Hash(int item)

{

return Math.Abs(item % table.Length);

}

void Rehash(int item, int index)

{

Random rnd = new Random();

bool flag = true;

int new\_index;

while (flag)

{

new\_index = (index + rnd.Next(1, table.Length)) % table.Length;

if (table[new\_index] == null)

{

table[new\_index] = item;

flag = false;

}

}

}

}

class ChainHash

{

List<int>[] links;

public ChainHash(int[] arr)

{

links = new List<int>[arr.Length];

foreach (var item in arr)

{

AddItem(item);

}

}

void Delete(int item)

{

if (HasValue(item))

{

var hash = Hash(item);

var index = links[hash].IndexOf(item);

links[hash].RemoveAt(index);

}

}

public bool HasValue(int item)

{

var res = links[Hash(item)]?.Exists(i => i == item);

return res ?? false;

}

void AddItem(int item)

{

var index = Hash(item);

if (links[index] == null)

{

links[index] = new List<int>();

links[index].Add(item);

}

else

{

links[index].Add(item);

}

}

int Hash(int item)

{

return Math.Abs(item % links.Length);

}

}

#endregion

#region Task 3

class Figure

{

public int Val { get; private set; }

public int NextStartPos { get; internal set; }

public bool Placed { get; internal set; } = false;

public Figure(int val, int pos)

{

Val = val;

NextStartPos = pos;

}

}

static void Chess(int size = 8)

{

var field = new int[size, size];

int counter = 1;

Figure first\_in\_row = null;

Stack<Figure> to\_place = new Stack<Figure>(size);

Stack<Figure> placed = new Stack<Figure>(size);

for (int i = size; i > 0; i--)

{

to\_place.Push(new Figure(i, 0));

}

bool[,] free = new bool[size, size];

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

{

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

{

free[i, j] = true;

}

}

string res;

long condition = 0;

switch (size)

{

case 1:

Console.WriteLine(1);

return;

case 2:

case 3:

Console.WriteLine(0);

return;

case 4:

condition = (int)ChessSizes.Four;

break;

case 5:

condition = (int)ChessSizes.Five;

break;

case 6:

condition = (int)ChessSizes.Six;

break;

case 7:

condition = (int)ChessSizes.Seven;

break;

case 8:

condition = (int)ChessSizes.Standart;

break;

case 9:

condition = (int)ChessSizes.Nine;

break;

case 10:

condition = (int)ChessSizes.Ten;

break;

case 11:

condition = (int)ChessSizes.Eleven;

break;

case 12:

condition = (int)ChessSizes.Twelve;

break;

case 13:

condition = (int)ChessSizes.Thirteen;

break;

case 14:

condition = (int)ChessSizes.Fourteen;

break;

case 15:

condition = (int)ChessSizes.Fifteen;

break;

case 16:

condition = (int)ChessSizes.Sixteen;

break;

case 17:

condition = (int)ChessSizes.Seventeen;

break;

case 18:

condition = (int)ChessSizes.Eighteen;

break;

case 19:

condition = (long)ChessSizes.Nineteen;

break;

case 20:

condition = (long)ChessSizes.Twenty;

break;

case 21:

condition = (long)ChessSizes.TwentyOne;

break;

case 22:

condition = (long)ChessSizes.TwentyTwo;

break;

case 23:

condition = (long)ChessSizes.TwentyThree;

break;

case 24:

condition = (long)ChessSizes.TwentyFour;

break;

}

while (counter < condition)

{

CheckPosition();

}

void ToString(int[,] field)

{

res = "";

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

{

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

{

if (field[i, j].ToString().Length == 1)

{

res += " " + field[i, j] + "|";

}

else

res += field[i, j] + "|";

}

res += "\n";

}

}

void CheckPosition()

{

if (to\_place.Count > 0)//если в стеке ещё есть неразмещённые фигуры

{

var fig = to\_place.Pop();

for (int i = fig.NextStartPos; i < size; i++)

{

if (IsFree(i, fig.Val - 1))//если клетка не закрыта пересечением

{

PlaceFig(fig, i);//размещаем фигуру на поле

break;

}

}

if (!fig.Placed)

{

fig.NextStartPos = 0;

to\_place.Push(fig);

RemoveFig();//убираем последнюю выставленную обратно в очередь

}

}

else

{

ToString(field);

Console.WriteLine($"{counter}:\n{res}");

RemoveFig();

++counter;

}

}

void RemoveFig()

{

var returned = placed.Pop();

returned.Placed = false;

field[returned.NextStartPos - 1, returned.Val - 1] = 0;

to\_place.Push(returned);

first\_in\_row = returned;

FreeField();

OccupeField();

}

void OccupeField()

{

foreach (var item in placed)

{

FillHorisVert(item.NextStartPos - 1, item.Val - 1);

FillDiags(item.NextStartPos - 1, item.Val - 1);

}

}

void FreeField()

{

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

{

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

{

free[i, j] = true;

}

}

}

bool IsFree(int row, int col)

{

if (free[row, col])//если свободна

{

return true;

}

return false;

}

void FillHorisVert(int row, int col)

{

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

{

free[i, col] = false;

free[row, i] = false;

}

}

void FillDiags(int row, int col)

{

int row\_now = row, col\_now = col;

while (row\_now >= 0 && row\_now < size && col\_now >= 0 && col\_now < size)

{

free[row\_now, col\_now] = false;

--col\_now;

--row\_now;

}

row\_now = row;

col\_now = col;

while (row\_now >= 0 && row\_now < size && col\_now >= 0 && col\_now < size)

{

free[row\_now, col\_now] = false;

++col\_now;

++row\_now;

}

row\_now = row;

col\_now = col;

while (row\_now >= 0 && row\_now < size && col\_now >= 0 && col\_now < size)

{

free[row\_now, col\_now] = false;

++row\_now;

--col\_now;

}

row\_now = row;

col\_now = col;

while (row\_now >= 0 && row\_now < size && col\_now >= 0 && col\_now < size)

{

free[row\_now, col\_now] = false;

--row\_now;

++col\_now;

}

}

void PlaceFig(Figure fig, int row\_placed)

{

field[row\_placed, fig.Val - 1] = fig.Val;

fig.Placed = true;

fig.NextStartPos = row\_placed + 1;

placed.Push(fig);

FillHorisVert(row\_placed, fig.Val - 1);

FillDiags(row\_placed, fig.Val - 1);

}

}

enum ChessSizes : long

{

Four = 3,

Five = 11,

Six = 5,

Seven = 41,

Standart = 93,

Nine = 353,

Ten = 725,

Eleven = 2681,

Twelve = 14201,

Thirteen = 737111,

Fourteen = 365597,

Fifteen = 2279185,

Sixteen = 14772513,

Seventeen = 95815105,

Eighteen = 666090625,

Nineteen = 4968057849,

Twenty = 39029188885,

TwentyOne = 314666222713,

TwentyTwo = 2691008701645,

TwentyThree = 24233937684441,

TwentyFour = 227514171973737,

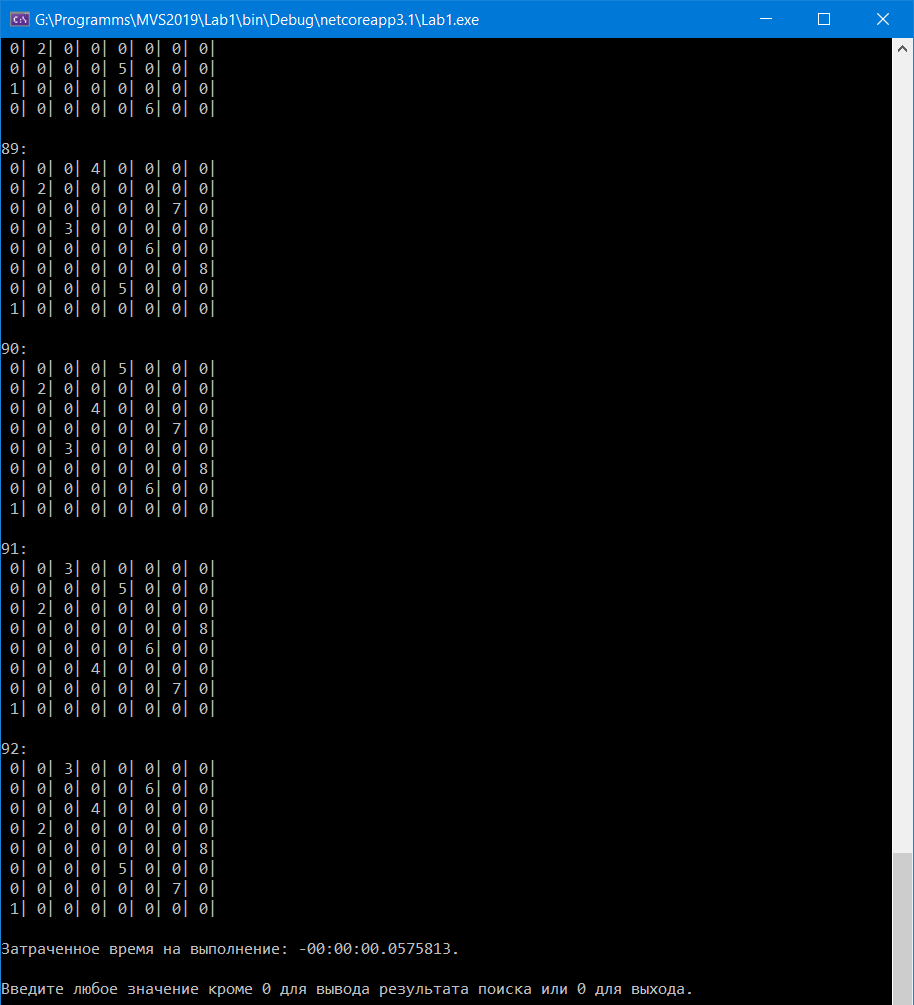
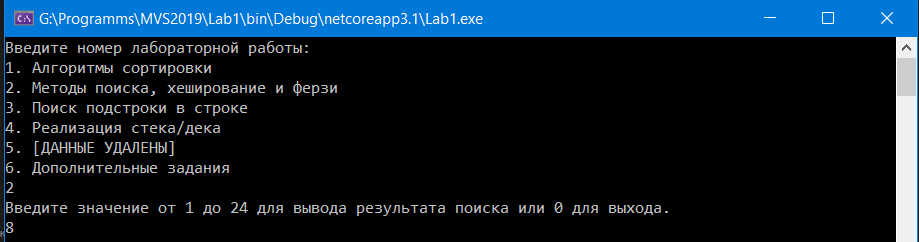
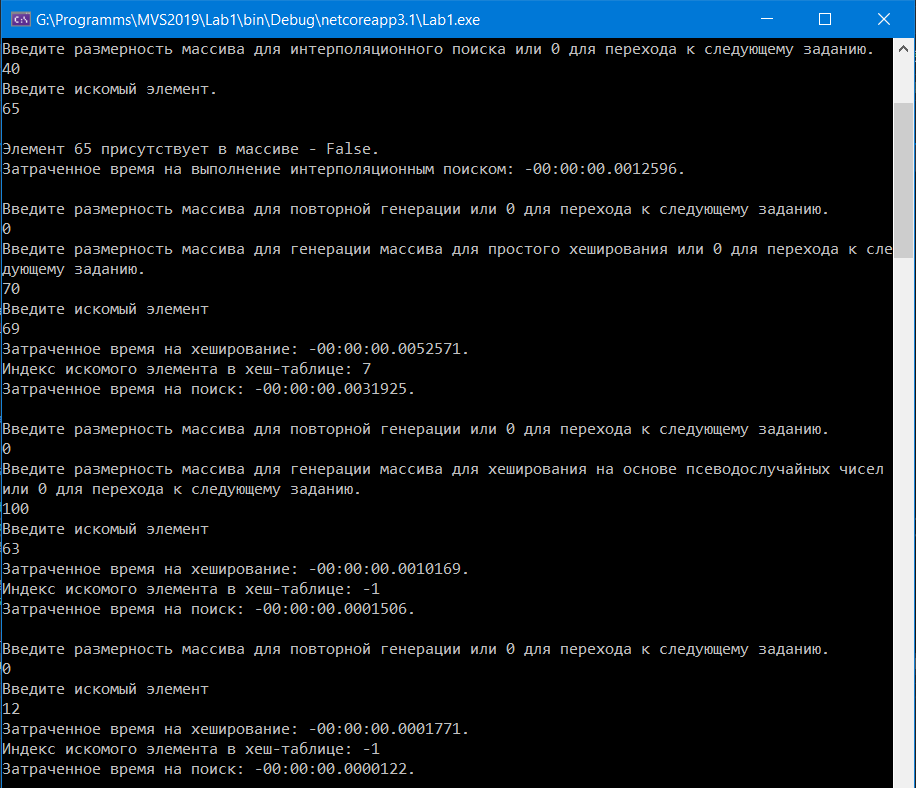
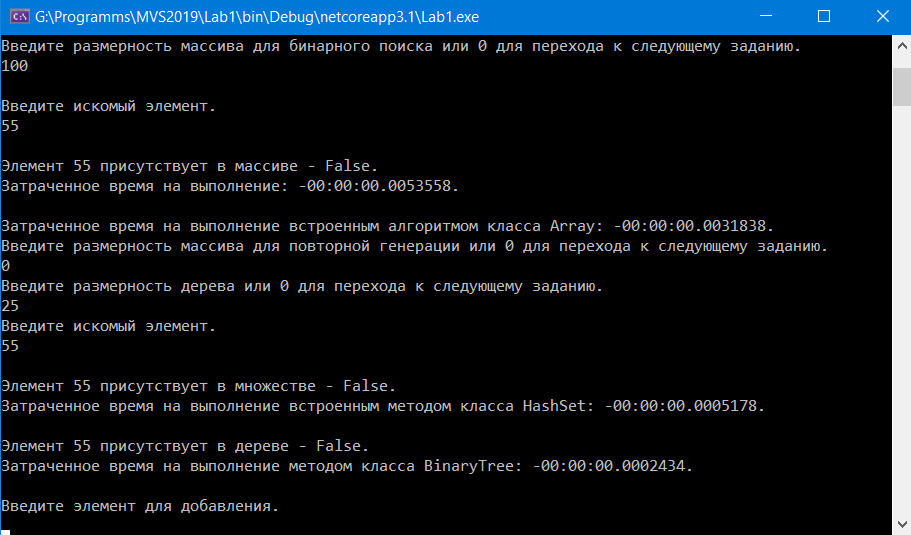
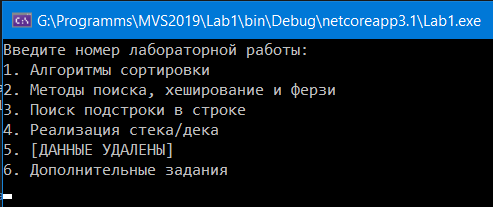
}

#endregion

}

}

Результаты выполнения программы представлены на рисунках ниже:

 **3. Вывод**

Анализируя полученное время выполнения реализованных алгоритмов в сравнении со встроенными аналогами можно сделать вывод о незначительных различиях при небольших размерах входных данных и достаточно большИх различиях при бОльших размерах(1000-1200 элементов).

Кроме того, были применены различные алгоритмы хеширования и рехеширования данных, показывающие хорошее время поиска заданных элементов в множестве, основанном на хеш-таблицах. Наилучшим алгоритмом по времени из реализованных является рехеширование методом цепочек.

Была реализована программа для решения задачи о 8 ферзях для размерностей доски [1;24], показывающая хороший результат по времени для размерностей [1;12]. При размере доски более 12 поиск всевозможных вариантов теоретически возможен и производится, но сопряжён со слишком большими временными затратами из-за слишком большого количества возможных вариантов решения задачи.